Metasm

Yoann Guillot

Résumé Cet article introduit Metasm, un framework d'assemblage-désassemblage multiplateformes scriptable. Ses principales caractéristiques :
- intégration de nombreuses briques (assembleur, désassembleur, linker, ...) ;
- license libre ;
- écrit en Ruby, un langage très dynamique ;
- générique et facilement extensible ;
- vaste champ d'application : compilation, exploitation de faille, injection de code... ;
- utilisable à bas ou haut niveau, de la gestion d'instruction à la gestion d'exécutables entiers.

1 Introduction

Metasm est un framework qui permet au programmeur d'interagir avec des blocs de code exécutables, c'est à dire du code compilé.

Il est constitué d'un ensemble de primitives de bases de gestion du code machine pour différentes architectures, à partir desquelles il propose des algorithmes de plus haut niveau, qui peuvent faire abstraction des spécificités liées à une architecture matérielle donnée.

Le framework inclut en outre des primitives de gestion de différents formats de fichiers exécutables.

Il permet de transformer les données brutes (code source, programme binaire) en différents objets manipulables aisément par le programmeur, et propose plusieurs algorithmes de haut niveau, notamment une fonctionnalité d'assembleur/linker, et de désassembleur.

Ce framework est intégralement écrit en Ruby[1], un langage de script dynamique, orienté objet. Il n'a aucune dépendance externe, ce qui laisse un contrôle total sur chacune des briques qui le compose, et garantit une portabilité maximale.

Il est publié sous licence LGPL[2].

Le principe de développement est la modularité, ce qui permet d'obtenir un programme qui peut être adapté à des tâches très variées en écrivant juste le code nécessaire à cette tâche.

2 Principes

La fonction de base du framework est de transformer les données brutes (code source, code machine, fichier exécutable) en différents objets Ruby qui seront manipulables par le programmeur. Ces objets sont autant que possible similaires d'une architecture à l'autre.

Metasm fonctionne sur le principe de la délégation, c'est à dire que les différentes opérations sont exécutées dans un objet générique, qui va au besoin appeler des méthodes sur des objets plus spécifiques qu'il référence, et ainsi de suite, récursivement. Par exemple pour l'assemblage de code exécutable, l'algorithme encode les données, les séquences d'alignement et les labels, mais il

[1] `http://ruby-lang.org/`
[2] `http://www.gnu.org/licences/lgpl.html`

repose sur l'instance du processeur pour encoder chaque instruction. Le processeur étant libre d'utiliser d'autres objets plus spécifiques pour certaines sous-tâches, comme l'encodage d'un argument particulier.

Le framework est également écrit de manière modulaire, en décomposant chaque tâche en sous-tâches indépendantes, de telle sorte que la réalisation d'une opération proche d'une opération qui existe déjà dans le framework soit réalisable en surchargeant juste la méthode nécessaire. Par exemple si l'on souhaite désassembler un exécutable Windows dont la section d'imports est modifiée, il suffira de dériver la classe `PE` (qui gère ces exécutables) et de redéfinir la fonction `decode_imports`, tout en réutilisant le code existant pour les opérations standard, comme le décodage du header. Cela permet, comme on dit, de *capitaliser sur ses assets*.

Metasm offre deux fonctionnalités relativements indépendantes, qui sont la gestion de code exécutable et la gestion de formats de fichiers exécutables. Les exceptions à ce découplage sont liées à la fonctionnalité de `linker`, notamment l'encodage de la section `.plt` des exécutables `ELF`, qui est une section de code exécutable générée par le compilateur, et l'encodage du header `MZ` des exécutables `PE/COFF` qui est en fait un petit exécutable MSDOS 16 bits lui aussi généré par le linker.

Les primitives de base du framework sont les 4 opérations qui permettent de transformer un code source en une représentation sous forme d'objets Ruby, de transformer ces objets en sequence binaire, et vice-versa.

Le nom attribué à chacune est :
- parse,
- encode,
- decode,
- render.

Ces fonctionnalités sont disponibles à différents niveaux : on peut encoder un entier, une instruction, une section de code, un programme.

Le principe de modularité s'applique pour ces fonctions, de sorte qu'il soit possible par exemple de réécrire un parseur d'instructions sans avoir à modifier l'encodeur.

2.1 Parsing

Le parsing est la phase de transformation d'un texte descriptif (code source) en objets Ruby. Cela s'applique principalement à la génération d'instructions.

Le parseur actuel est assez souple au niveau syntaxique. Il supporte :
- les macros de type assembleur (`foo macro bar ... endm`), avec support des labels internes,
- les macros de type C (`#define bla`, `#ifdef ... #elif ... #end`, `#include "toto"`),
- les commentaires type asm/C/C++ (`foo ; bar`, `foo // bar`, `foo /* bar */`),
- les entiers en différentes bases (`0x42, 28, 43h, 0b1100_0101, 0777`),
- les chaînes de caractères avec séquences d'échappement (`"foo\x42 bar\n"`),
- la définition recursive de séquences de données (`db 42 dup(21 dup(?), "kikoo\0", 28 dup (label_rouge - label_bleue))`)
- certains pseudo-labels (`$` qui est l'adresse de début de l'instruction courante, `$$` qui est l'adresse de début de la section courante)

Il est possible d'inclure des fichiers C, afin d'importer des prototypes de fonctions, des constantes du preprocesseur, des structures.

Le support de High Level Assembly[3] (`.if eax > 42 jmp eax .endif`) est laissé à la discrétion de chaque processeur, il est implémenté pour architecture Intel.

Il existe également quelques pseudo-instructions permettant de spécifier les symboles à importer du système, les symboles à exporter, des directives d'alignement, la répartition des instructions dans différentes sections (`.text, .data, foobar`) et les permissions de ces dernières (read, write, execute, discardable...).

Le parseur utilise la syntaxe Intel pour les instructions IA32, un parseur GAS sera probablement écrit dans le futur.

La phase de parsing est responsable de la traduction d'un code source en séquences d'objets de ces 3 types, répartis dans différentes sections :

- labels, qui nomment un emplacement particulier ;
- datas, qui contiennent des données arbitraires, initialisées ou non ;
- instructions, c'est à dire un nom d'opcode et une séquence d'arguments dont le type dépend du processeur.

On peut également trouver quelques objets exotiques, comme des directives d'alignement.

2.2 Encodage

L'encodage consiste à transformer les objets Ruby (qu'ils soient issus du décodage ou du parsing, ou qu'ils soient créés à la main par le programmeur) en séquence binaire compréhensible par la machine.

En pratique, cette opération aboutit quasi systématiquement à la création d'un objet `Encoded-Data`, qui est une séquence binaire augmentée de certaines meta-informations, comme une association $label \mapsto offset$ et des informations de relocation (par exemple : les octets 6 à 9 doivent être fixés avec la valeur de $12 + 4 * addr\_foobar$ encodée comme un entier 32 bits non signé little-endian).

On trouvera une primitive d'encodage dans la majorité des objets Metasm, qu'il s'agisse d'un entier, d'une instruction, d'une section ou d'un format d'exécutable.

Une instruction est susceptible de renvoyer différents encodages possibles (par exemple en IA32 il est souvent permis d'encoder un entier soit sur 8 soit sur 32 bits), c'est alors la fonction d'encodage de la section qui est chargée de résoudre cette incertitude et de choisir la forme à utiliser.

L'encodage correspond également, pour les formats d'exécutables, à la phase d'édition de liens, c'est à dire que l'encodeur va déterminer à quelle adresse il va charger les différentes sections, résoudre les relocations existantes et générer les informations de relocation à inclure dans l'exécutable ; l'encodeur est en fait responsable de la mise en forme de toutes les informations que l'on trouve dans l'objet abstrait programme, comme les tables d'import/export, les stubs d'appel de fonctions importées, etc. Lors de cette phase, toutes les valeurs arbitraires (timestamps, flags, checksums, ...) peuvent être choisies par le programmeur, elles seront par défaut initialisées à une valeur raisonnable. Il est également possible de spécifier différents comportements pour le linker, qui dépendent du format exact utilisé (on peut par exemple fusionner ou non les sections d'un fichier `PE`, construire ou non les tables de sections et de segments pour un fichier `ELF`, ...)

L'encodage de fichier exécutable se traduit par la production d'une séquence binaire « pure », mais en général l'assemblage final se contente d'appeler une série de sous-fonctions qui vont construire les différentes tables (import, export, relocs, ...), il est ainsi assez simple d'interférer avec le processus pour obtenir un linker réalisant certaines opérations de manière spécifique.

[3] `http://webster.cs.ucr.edu/AsmTools/HLA/`

2.3 Décodage

Le décodage est l'opération inverse de l'encodage : il s'agit donc de transformer une séquence binaire en objet Ruby, ou un objet Ruby en sa préimage par rapport à l'encodage.

Pour un fichier exécutable, il s'agit donc de renvoyer un programme abstrait contenant toutes les informations extraites du fichier : imports, exports, point d'entrée, sections encodées avec relocations et exports, ainsi que toutes les options nécessaires pour pouvoir réencoder le programme à l'identique.

Le décodage d'une instruction correspond lui à la transformation d'une séquence d'octets en une instruction, c'est à dire un nom d'opcode et une séquence d'objets représentant les arguments. Comme cette transformation perd de l'information, le décodage d'instruction va en fait renvoyer un autre objet, qui en plus de contenir l'instruction décodée, va également fournir un pointeur vers l'opcode exact rencontré, ainsi que d'autres informations, comme la taille du bloc décodé, afin de permettre l'affichage des données originales ayant aboutit à cette instruction (chose impossible autrement, car certaines instructions ont plusieurs formes d'encodage possible, notamment sur architecture complexe comme IA32).

Il n'existe pas vraiment de notion de décodage pour une section. Cette fonctionnalité est remplacée par une méthode de désassemblage au niveau du programme, qui va suivre le flot d'exécution à partir d'un point d'entrée spécifié par le programmeur. Ce désassemblage aboutit à la construction d'un graphe de blocs d'instructions (un bloc étant une séquence d'instructions qui s'exécute de manière continue, notamment sans branchements). Le désassemblage utilise une technique de backtracking générique qui se base sur un pseudo-émulateur basique fourni par l'instance du processeur pour résoudre la majorité des branchements, ainsi que (bientôt ;)) sur les prototypes des fonctions importées et l'ABI[4] du système d'exploitation pour suivre le plus exactement possible le flot d'exécution.

Il est alors possible d'afficher ces blocs sous forme d'un code source dont l'assemblage devrait redonner le binaire original. Pour cette partie, il reste du travail à faire, notamment au niveau de la gestion des segments de données.

2.4 Affichage

L'affichage est la dernière facette des fonctionnalités de Metasm.

Il est responsable de la représentation des informations, notamment l'affichage du code source pour les instructions.

Cette notion d'affichage est réalisée de manière récursive : par exemple une instruction IA32 `mov eax, dword ptr [ebx + 42]` va répondre à la requête d'affichage par une liste constituée d'une chaîne (le nom de l'opcode), un objet de type *registre*, et un objet de type *modrm* (c'est le nom des indirections mémoire sous IA32). Le registre va lui-même répondre à la requête d'affichage par une chaîne, le *modrm*, lui, va renvoyer un registre et une constante numérique, et ainsi de suite.

Ce mode de fonctionnement qui peut sembler complexe au premier abord à pour but de permettre une intégration simple dans une interface graphique avec options contextuelles : il est en effet facile d'afficher le résultat final, et l'algorithme réalisant l'affichage connait précisément la position de chacun des constituants de l'instruction, et peut donc réagir différemment suivant que l'utilisateur clique sur « mov » ou sur « eax ».

[4] Application Binary Interface, ce qui définit les conventions d'appels de fonctions, les registres à conserver, etc

Chaque objet qui répond à la requête d'affichage répond également à une requête de contexte, qui doit renvoyer une liste de transformations qui lui est applicable (par exemple pour le *modrm*, changer le type de donnée pointée en `byte ptr`, rajouter un sélecteur de segment, ...)

Notons que ceci reste à l'état de projet, en l'absence de réelle d'interface graphique pour le moment.

Le but est d'obtenir à terme une interface fournissant le même type de fonctionnalités que celle d'IDA[5], le désassembleur interactif de DataRescue (*enfin, un jour peut-être :)*).

3 Les classes

L'ensemble du code est encapsulé dans le module `Metasm`, chacune des classes définies ici en fait partie.

Les deux classes centrales de Metasm sont les classes `EncodedData`, qui représentent une séquence d'octets couplée à plusieurs méta-informations, et `Expression`, qui est utilisée dès que l'on parle de valeur numérique constante.

3.1 EncodedData

Cette classe est utilisée pour représenter toutes les chaînes binaires qui vont être ou ont été mappées dans la mémoire de l'ordinateur : instruction compilée, section assemblée, données.

EncodedData contient une séquence binaire donnée, accessible via son accesseur `data`. C'est une chaîne standard Ruby.

EncodedData possède également un attribut `virtsize` qui représente la taille de ce segment en mémoire. Les octets situés apres la fin de `data` ont une valeur non définie ; ils sont utilisés pour représenter entre autres des données non initialisées. Il est possible de remplir cet espace vacant à l'aide de la méthode `fill` qui accepte en argument une taille et la valeur d'un octet qui sera répété autant de fois que nécessaire. `virtsize` est toujours au moins égal à la taille de `data`.

Les deux autres méta-informations incluses sont `export` qui est une liste de correspondances *nom* $\mapsto$ *offset*, et `reloc` qui est une liste de correspondances *offset* $\mapsto$ *relocation*. Une relocation comprend une `target`, qui est une `Expression` arbitraire, un `type` (signé 32 bits, ...) et une `endianness`. Les relocations peuvent être résolues en passant un hash à la méthode `fixup`, qui va alors parcourir la liste des relocations, calculer la valeur de chaque `target` en fonction du hash fourni, et si cette valeur est numérique, l'encoder dans les octets définis.

Les objets EncodedData ont une fonction de concaténation, qui se charge de mettre à jour la taille virtuelle de l'objet, de remplir les vides si nécessaire, de fixer les offsets des exports et des relocations etc. Elle est accessible par la méthode, `<<`, qui accepte en argument un autre `EncodedData`, une `String` ou un entier représentant le code ASCII d'un octet à concaténer.

Un `EncodedData` possède également un pointeur `ptr` modifiable librement, et une méthode associée `get_byte` qui renvoie la valeur de l'octet pointé par `ptr`, définie à 0 pour les données non initialisées. Cette méthode incrémente `ptr`, et est utilisée par les fonctions de décodage.

3.2 Expression

Cette classe représente une expression arithmétique arbitraire, avec variables, sous forme d'arbre.

[5] `http://www.datarescue.com/idabase/`

Une `Expression` possède un opérateur sous forme de `Symbol` (`:+`, `:-`, `:/`, `:<<`, etc), une valeur à droite (qui peut être une autre `Expression`), et une valeur à gauche (qui peut également être une `Expression`, ou `nil` dans le cas d'un opérateur unaire).

La classe possède un constructeur particulier, afin de faciliter cette structure interne : pour représenter $1 * (2 + 3)$, on utilisera `Expression[1, :*, [2, :+, 3]]`, où chaque sous-tableau représente une nouvelle Expression.

La particularité de cette classe réside dans son support basique du calcul formel pour les additions, par l'intermédiaire de la méthode `reduce`, qui renvoie une nouvelle `Expression` ayant une forme canonique définie (pour les membres de type additif), cette méthode peut également renvoyer un entier si la valeur numérique est calculable. La forme canonique transforme les soustractions en additions de l'opposé, regroupe les termes numériques dans le membre droit, et simplifie les variables qui peuvent l'être ($a + b + (-a) \mapsto +b$). Les sous-arbres purement numériques sont évalués et remplacés par leur valeur, ces opérations étant effectuées par Ruby (donc avec le support transparent des bignums).

Il est possible d'attribuer une valeur aux variables utilisées dans une `Expression` (cette valeur étant soit arbitraire, soit numérique, soit une autre variable, soit une `Expression`, soit n'importe quel objet) au moyen de la méthode `bind` qui prend en argument un hash de type *variable* $\mapsto$ *valeur*. `Bind` renvoie une nouvelle `Expression`, et `bind !` modifie l'expression courante *in situ*.

Le parseur de la classe gère la priorité des opérateurs avec les mêmes règles qu'en langage C.

Le décodeur prend en argument un `EncodedData` dont le champs `ptr` pointe sur l'offset à décoder, ainsi que le type et l'endianness de la valeur à lire. Si le pointeur correspond à une relocation dont le type et l'endianness correspondent à la demande, sa cible est renvoyée par le décodeur. Une méthode `decode_imm` permet de décoder l'entier pointé en ignorant les éventuelles relocations, dans ce cas la valeur retournée est un entier.

L'encodage d'une `Expression` donne soit un `EncodedData` contenant la valeur numérique si celle-ci existe, soit un `EncodedData` virtuel (membre `data` vide), mais pour lequel est définie une relocation (dont la `target` est l'`Expression` elle-même).

3.3 Instruction/Opcode/DecodedInstruction

Ces classes représentent respectivement une instruction particulière, une classe d'instructions, et une instruction décodée. Elles sont génériques, c'est-à-dire qu'une instruction IA32 et une instruction MIPS sont le même objet. Seul l'interprétation de leurs champs change.

Un `Opcode` est constitué d'un nom, d'une liste symbolique d'arguments (dont les valeurs autorisées dépendent du CPU), d'une représentation binaire (soit un entier pour les architectures à taille d'instruction fixe, soit d'un tableau d'octets pour les autres), d'un ensemble de champs (ex : *le numéro du registre de destination est codé sur les bits 5, 6 et 7*) et d'un ensemble de propriétés. Certaines propriétés sont communes à toutes les architectures (comme `:setip` qui indique que l'instruction déroute le flot d'exécution), d'autres sont spécifiques au CPU.

Une `Instruction` est elle définie par un nom, une liste d'arguments concrets dont le type dépend de l'architecture (exemple : registre eax), et d'une liste de préfixes.

Une `DecodedInstruction` contient simplement une `Instruction`, un `Opcode` et un entier donnant le nombre d'octets lus pour décoder cette instruction. Cette classe existe pour éviter de perdre de l'information au niveau du désassemblage, dans le cas où il existe plusieurs façons d'encoder une instruction, afin de permettre l'affichage classique « sequence binaire / instruction » (ex : `\x90 nop`)

3.4 CPU

C'est la classe principale pour chaque architecture supportée.

Un CPU contient une liste d'opcodes, et des méthodes responsables de la gestion des instructions (parse, encode, etc). La plupart des architectures définissent également des classes internes, notamment pour la gestion de certains arguments (registres, indirections mémoire), qui peuvent avoir leurs propres routines d'encodage / décodage.

Un CPU posséde également une endianness, et un mode de fonctionnement (par exemple un processeur IA32 peut être en mode 16 bits ou 32 bits, voire 64 avec EM64T).

Pour le parsing, un processeur accepte tout nom d'opcode valide, lit ensuite tous les arguments qu'il trouve (en s'arrêtant s'il rencontre un nouveau nom d'opcode valide), et ensuite la liste d'opcodes est parcourue pour déterminer si l'instruction lue est encodable, par vérification de sa signature. Si ce n'est pas le cas, une exception est levée.

Pour l'encodage, l'ensemble des opcodes dont la signature correspond à celle de l'instruction sont utilisés, et le CPU renvoie une liste d'`EncodedData` correspondant à cette instruction. Un même opcode peut également produire plusieurs EncodedData, notamment si l'opcode dispose d'un champ autorisant différentes tailles de constantes, et si l'`Expression` reçue en paramètre ne peut être calculée. Dans ce cas toutes les possibilités, différenciables par l'intermédiaire du champ `type` de leur relocations, sont retournées à l'appelant.

Lors du décodage d'une instruction, le CPU va tout d'abord rechercher quel opcode est utilisé en comparant les données au masque binaire défini par chaque opcode. Une table de précalcul est utilisée pour optimiser cette recherche : la table liste tous les opcodes susceptibles de convenir en fonction du premier octet de données. Si aucun opcode ne correspond, la sequence est interprété comme un préfixe, et si cela ne donne rien non plus, une exception est levée.

Une fois l'`Opcode` déterminé, le CPU passe au décodage des arguments pour fournir une `Instruction` complète. Ces deux objets sont regroupés au sein d'une `DecodedInstruction`, avec la taille du texte lu pour décoder l'instruction.

Le processeur est également chargé de l'émulation des instructions pour le backtracking et pour la détermination de la cible d'un changement du flot d'exécution.

3.5 Section

Une section contient principalement un `EncodedData`, auquel elle rajoute quelques attributs, notamment un nom, une liste de permissions mémoire, une adresse de base, et une liste d'`Instruction`

/ `Data`, résultat du parsing d'un fichier source ou du décodage et de la transformation en source de l'`EncodedData`.

Son rôle est de faire la liaison entre une liste d'instructions et l'`EncodedData` correspondant.

Sa principale fonction réside dans l'encodage, où elle est notamment chargée de résoudre les ambiguïtés liées aux différentes possibilités d'encodage d'une même instruction. L'algorithme en question est décrit dans la partie 4.1.

3.6 Program

`Program` est une représentation abstraite d'un programme informatique. C'est généralement la classe racine dans un script Metasm.

Un `Program` contient une liste de `Sections`, une référence vers un processeur, et une liste d'imports et d'exports. Ces deux derniers éléments correspondent aux tables du même type que l'on retrouve dans les fichiers exécutables, ils représentent les interactions avec le reste du système d'exploitation.

Les exports sont une série d'associations *nom exporté* $\mapsto$ *nom de label* dans une des sections ; les imports sont une association *nom de bibliothèque* $\mapsto$ *liste de fonctions à importer*, chaque fonction importée pouvant être associée à un label qui correspond à un stub généré par le linker de sorte à être exécutable (un import est simplement l'adresse d'un pointeur).

Cet objet est responsable de toutes les propriétés globales du programme. Il est par exemple capable de créer un nom de label unique.

Au niveau du parsing, il est chargé de gérer les sections et leurs permissions, et les pseudo-labels (`$` et `$$`).

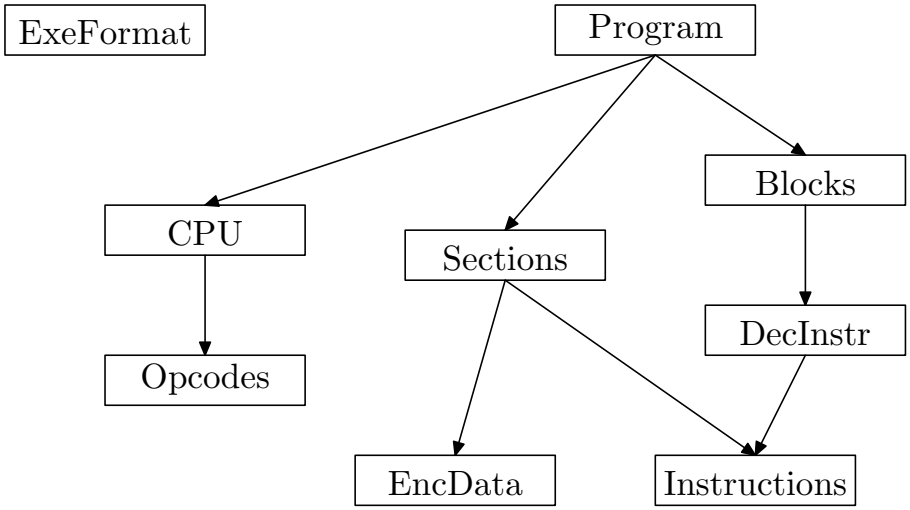
Pour le décodage, il implémente la fonction de désassemblage : à partir d'un point d'entrée, il va décoder chaque instruction, en tentant de déterminer la cible des opcodes qui déroutent le flot d'exécution avec l'aide du processeur courant. C'est également dans le `Program` que sont stockés les prototypes de fonctions lues depuis les headers C. Enfin il est responsable de la transformation du graphe de `DecodedInstruction`s en code source « linéaire » dans les `Section`s.

3.7 ExeFormat

C'est la classe dont dérivent les formats de fichiers exécutables. Ces classes sont chargées de transformer un `Program` en fichier exécutable (encodage) et vice-versa. Les deux plus grosses classes, COFF et ELF, supportent également la gestion de fichiers « objet », qui peuvent ensuite (théoriquement) être liés à d'autres objets pour former un exécutable.

3.8 Diagramme

Voici un diagramme représentant les relations entre les classes (une flèche indique qu'un objet d'une classe contient une référence vers des objets de la classe pointée)



4 Algorithmes

4.1 Résolution des conflits d'instructions

Lorsque plusieurs encodages sont possibles pour une instruction, cet algorithme est chargé d'en choisir un. Il est utilisé dans la méthode d'assemblage d'une section, et est surchargeable au besoin. Il choisit la version la plus courte possible capable d'encoder l'information donnée.

Cet algorithme est nécessaire en particulier pour résoudre les ambiguités au niveau de l'encodage des sauts dans l'architecture IA32 : l'instruction `jmp` a deux formes, soit la forme `short` qui s'encode sur deux octets mais qui ne permet des sauts qu'à plus ou moins 127 octets de distance, soit la forme `near` qui permet des sauts arbitraires mais qui s'encode en 5 octets.

Le CPU, qui encode les instructions, n'a pas de visibilité sur autre chose que l'instruction qu'il encode, il ne peut donc pas connaître la valeur de ce saut et choisir la forme la plus appropriée. C'est pourquoi il va renvoyer les deux formes possibles, chacune avec une information de relocation contenant l'information sur la taille du champ disponible pour encoder la distance du saut, et donc les valeurs extrêmes applicables dans chaque cas.

Lors du parsing de l'instruction, l'usage veut que l'on spécifie l'intruction comme `jmp target`. Or la véritable forme de l'instruction est `jmp target - addr_after_jmp` où `addr_after_jmp` est l'adresse du premier octet suivant l'instruction encodée de saut. Il s'agit donc d'évaluer la distance entre deux labels.

Ce probléme est résolu de manière générique grace à l'algorithme suivant :

Chaque élément (instruction, data) de la section est encodé dans l'ordre, en conservant les choix multiples quand ils se présentent. L'algorithme travaille en effet directement sur les `EncodedData` et non au niveau du source.

Le pire binding possible est alors contruit pour la section, en attribuant à chaque label rencontré l'offset correspondant au choix le plus long pour chaque ambiguïté, de manière à maximiser toutes

les différences entre offsets. Ce binding est ensuite utilisé pour réduire chaque `target` de relocation. On choisit alors en fonction du résultat de cette réduction :

- Si le choix est non numérique, c'est que la relocation dépend d'un label qui n'est pas défini dans la section. Dans ce cas, on prend l'intruction la plus courte parmi toutes celles dont le type de relocation est le plus grand (c'est-à-dire que si l'on a le choix entre une instruction de 3 octets avec une relocation 8 bits, une de 5 octets avec une relocation 32 bits et une de 7 octets avec une relocation 32 bits, on va choisir la deuxième, qui est la plus courte parmi celles offrant une relocation 32 bits).
- Si le choix est numérique, on sélectionne l'instruction la plus courte parmi celles dont les relocations permettent d'encoder la valeur calculée.

Une fois toutes les ambiguïtés levées de cette manière, on recalcule le binding réel, et on peut fixer les relocations résolues à leur valeur définitive.

Cet algorithme n'est pas optimal, mais en pratique il donnera quasiment toujours la meilleure solution, et il a l'avantage d'être générique et de se faire en une passe.

4.2 Résolution des cibles de sauts, backtracking

Cet algorithme est chargé de déterminer les adresses où le flot d'exécution est détourné par certaines instructions, lors du désassemblage d'un programme.

Dans la liste d'opcodes d'un processeur, les instructions qui modifient le flot d'exécution sont marquées avec le flag `:setip` (*set instruction pointer*). Lorsque le désassembleur rencontre une telle instruction, il passe le relai au processeur afin de déterminer la cible du saut. Cette valeur peut être soit numérique (dans ce cas la cible est connue), soit une expression, dont certaines variables sont des symboles spéciaux qui représentent la valeur d'un registre à cet instant. Une nouvelle forme d'expression, l'`Indirection` peut également apparaître à cette étape : elle indique alors que la cible du saut est la valeur pointée par le pointeur de l'indirection, qui est une `Expression`.

Pour déterminer les valeurs possibles pour cette Expression/Indirection, le programme va alors remonter le flot d'exécution ayant amené à ce point, et à chaque étape va passer l'instruction rencontrée à la fonction d'émulation du processeur, en demandant spécifiquement de résoudre l'expression recherchée. Le processeur renvoie alors une nouvelle valeur, qui peut être :

- Numérique. Dans ce cas la recherche est terminée, la cible est connue.
- Une Indirection dont le pointeur est numérique. Dans ce cas l'adresse est connue, il est donc possible de trouver la valeur de la cible. La recherche est terminée.
- `nil` : Le backtracking est impossible (instruction non émulée par exemple), la recherche s'arrête.
- Une autre Expression/Indirection : là la recherche continue, mais en résolvant la valeur de cette nouvelle Expression.

Cette résolution est récursive, c'est à dire que l'on peut connaître la valeur d'une Indirection dont le pointeur est une Expression faisant intervenir une Indirection etc.

Lors du backtracking, le processeur peut arriver à un branchement, où plusieurs blocs pointent vers le bloc courant. Dans ce cas la recherche se poursuit dans chacune des branches.

Cet algorithme marche bien dans de nombreuses situations, il est notamment capable de retrouver intelligemment les adresses de retour de procédures sans devoir assumer qu'une instruction de type `call` va retourner dans tous les cas.

Il faut cependant noter qu'il ralentit considérablement l'analyse du code, et est donc à réserver à l'étude de petites sections exécutables.

De plus, il existe plusieurs situations où il est inefficace. Ce peut être quand l'émulateur ne remplit pas son rôle : dans ce cas le saut doit être marqué comme « non résolu ». Une situation similaire est le cas où la valeur du saut est une valeur dans un tableau, dont l'index est borné mais n'a pas de valeur exacte : c'est le cas de certaines constructions de type `switch() case :` en C, qui se traduisent par une séquence similaire à celle-ci :

```
mov ebx, variable_intracable
jmp [jump_table + 4*ebx]
```

Dans ce cas une solution pourrait être de reconnaître l'accès à un tableau et de marquer chaque entrée de celui-ci comme des pointeurs de code, mais la question des limites inférieures et supérieures des index de ce tableau reste posée.

Enfin la plus mauvaise situation est celle où l'algorithme donne une mauvaise réponse. Cette situation est notamment possible dans le cadre d'alias mémoire : plusieurs pointeurs pointent sur la même adresse, un d'eux est utilisé comme adresse de saut, et un autre est utilisé pour un accès en modification. Par exemple :

```
mov [eax], adresse ; initialisation du pointeur
mov ebx, eax ; copie du pointeur
add dword ptr [ebx], 42 ; modification du pointeur
jmp [eax]
```

Dans cette situation, le module de backtracking va tracer la valeur pointée par `eax`, et trouver `adresse`, alors que la cible du saut sera réellement `adresse+42`.

Une solution serait de marquer tous les accès en écriture à la mémoire, et de vérifier lorsqu'on passe par une indirection lors d'un branchement que l'adresse pointée n'est pas modifiée, mais cette solution risque d'être lourde à l'usage. Cela sera tout de même très utile dans l'analyse automatique de petites sections de code obfusqué.

5 Applications

Voici quelques exemples d'utilisation de Metasm :

5.1 MetasmShell

Il s'agit d'un assembleur interactif : on rentre une ou plusieurs instructions à l'invite de commande, et celles-ci sont immédiatement assemblées et affichées sous forme hexadécimale. Les métadonnées sont également affichées.

Cette application fonctionne en construisant pour chaque ligne reçue en entrée un programme, qu'elle assemble et dont elle affiche le contenu compilé.

Son source est très simple, et est un bon exemple d'usage du framework.

En voici une version simplifiée :

```
require 'metasm'

cpu = Metasm::Ia32.new
prog = Metasm::Program.new cpu
line = gets
prog.parse line
prog.encode
edata = prog.sections[0].encoded
puts edata.inspect, edata.reloc.inspect, edata.export.inspect
```

5.2 Metasploit

Metasploit[6] est un framework open source d'exploitation de vulnérabilités, dont la version 3 est également écrite en Ruby. Les shellcodes inclus sont sous forme compilée, sous forme hexadécimale dans des chaînes statiques, ce qui les rend difficiles à modifier.

Metasm remplace très avantageusement ces shellcodes par une version qui fait de la compilation à la volée, rendant obsolète les techniques de customisation (actuellement si l'on veut changer l'adresse IP utilisée par un payload de type *connectback* il faut aller patcher les octets à la main dans le shellcode, à un offset difficile à déterminer). Ici on peut soit modifier dynamiquement le code source et le réassembler, soit le compiler en gardant les méta-données, et fixer l'adresse IP au moyen d'une relocation grâce aux mécanismes de base de Metasm.

L'utilisation de ces mêmes méta-données rend beaucoup plus souple et robuste les conventions d'interaction entre les différents stages des exploits.

On peut également utiliser les primitives de manipulation de code pour générer de manière automatique des shellcodes répondant à des contraintes très précises, par exemple quand l'espace disponible pour l'exécution du shellcode est partagé avec une structure de données du programme exploité dont certains membres doivent avoir une valeur précise : on peut générer un shellcode qui n'utilise que les « trous » de cette structure, sans avoir à coder directement en hexadécimal à la main.

Une intégration plus poussée permetrait également de mettre en place un générateur polymorphique de code, mais cette fonctionnalité n'était pas encore réalisée au moment de l'écriture de cet article.

Enfin une application très intéressante serait la création d'un format exécutable de type shellcode possédant des fonctionnalités de linker dynamique, ce qui permettrait par exemple d'écrire normalement un shellcode qui fait des appels d'API Windows, mais de telle sorte que la phase de linkage rajoute des mécanismes comme l'évasion de détection par HIPS, ce qui est très difficile et fastidieux avec l'architecture actuelle de Metasploit.

5.3 Slipfest

Slipfest[7] est un programme open source et libre de test de logiciels HIPS. Il a été réécrit en utilisant Metasm, de maniére à rendre son utilisation beaucoup plus flexible, le transformant en une sorte de petit débuggueur / injecteur interactif. Il permet notamment l'injection de code arbitraire pour la mise en place de hooks sur des fonctions de bibliothèques, réalisé de manière très robuste

[6] `http://www.metasploit.com/`
[7] `http://slipfest.cr0.org/`

grâce à l'utilisation du désassembleur durant l'étape de substitution d'instructions nécessaire à l'installation du hook.

5.4 Transcompilateur

Une autre application possible de Metasm est la transformation d'un programme ELF en un binaire Windows ou vice-versa (en utilisant tout de même une bibliothèque de compatibilité comme Cygwin[8] ou Wine[9]), sans nécessiter l'accès au source du binaire, simplement en transformant les fichiers en leur équivalent sous l'autre OS, et en rajoutant éventuellement une couche de normalisation de l'ABI pour les appels de fonctions externes.

Ceci n'est à l'heure actuelle qu'une voie de recherche envisagée.

6 Conclusion

Pour conclure, on peut dire que Metasm est un outil extrêmement puissant dans la gestion du code machine, pour lequel il n'existe que peu d'équivalents. Sans lui il faudrait recourir à l'utilisation de nombreuses briques disparates, une pour la gestion d'ELF ou du PE, une pour les instructions Intel, etc. L'intégration de toutes ces facettes dans un même framework et le fait que celui-ci soit écrit dans un langage hautement dynamique lui confère des avantages indéniables.

7 Remerciements

Je remercie Julien Tinnès et Raphaël Rigo pour leur soutien quotidien, Philippe Biondi pour le SSTIC, et Benoît Merlet et Jordan Augé pour leur relecture attentive ; ainsi que Laurent Butti pour avoir le premier utilisé Metasm dans Metasploit. Merci également à France Télécom R&D, entre autres pour son soutien de la recherche en sécurité informatique.

[8] `http://www.cygwin.com/`
[9] `http://www.winehq.com/`