Désobfuscation de DRM par attaques auxiliaires

Camille Mougey et Francis Gabriel
cmougey@quarkslab.com
gg@quarkslab.com

QuarksLab

Résumé Cet article présente l'analyse d'une DRM ("Digital Right Management") très répandue et présentant des fonctionnalités assez avancées en matière d'obfuscation. Afin de comprendre son fonctionnement, les méthodes de rétro-ingénierie (ou *reverse engineering*) classiques s'étant avérées peu efficaces, le choix a été fait d'une approche centrée sur l'analyse de trace.

À cette fin, le framework "pTra" a été développé. Au travers de l'article, nous montrons comment cette approche a facilité l'analyse de portions de code lourdement obscurcies, entrelaçant plusieurs protections. Nous présenterons les mécanismes mis en oeuvre par ces protections, ainsi que les méthodes utilisées pour les contourner.

1 Introduction

De nos jours, l'obscurcissement de code est de plus en plus utilisée. Elle intervient dans différentes situations, comme la protection logicielle (jeux vidéo, systèmes anti copie), et a pour but de rendre plus difficile, plus longue - voire non rentable - l'analyse d'un programme ou d'un bout de programme.

Nous avons récemment étudié le code d'une DRM, soit une "Digital Right Management", autrement nommée GDN pour "gestion des droits numériques", largement répandue et utilisée.

1.1 Constats initiaux

Cette DRM interagissant avec le réseau, nous nous sommes attachés à observer le trafic, c'est-à-dire les entrées et les sorties de l'algorithme.

Il apparaît très rapidement que les relations entre les deux ne sont pas triviales à deviner :

- l'avalanche de modifications d'un bit en entrée est totale (l'ensemble des bits de la sortie sont impactés) ;
- l'entropie des données en entrée et en sortie est proche de 1. Des algorithmes de compression ou de chiffrement sont très certainement impliqués ;

– les algorithmes répandus (fonctions de hash, de chiffrement, de compression) ne permettent pas d'obtenir les mêmes résultats.

Ensuite, une analyse statique du code fait clairement apparaître des méthodes de protection, la plus flagrante étant le "code flattening".

Enfin, une brève analyse dynamique laisse penser que très peu de données intelligibles apparaissent en mémoire.

Avec ces différents éléments, nous avons été enclins à penser que cette DRM utilise des méthodes d'obscurcissement, une hypothèse validée par la suite.

Ainsi, cet article détaille les différentes protections appliquées au code de cette DRM. Attaquer ce type de code est un peu comme enlever successivement les pelures d'un oignon, chacune d'entre elles vous faisant pleurer un peu plus.

2 Première couche : Code flattening

2.1 Définition

L'aplatissement de code (code flattening) est une méthode destinée à réduire les informations apportées par la connaissance du graphe de flot de contrôle (*control flow graph, CFG*).

La Figure 1 montre un graphe original d'une fonction, tandis que la Figure 2 un graphe soumis à ce type de protection.

Ce type de protection a déjà été abordée, notamment dans un article de MISC [7] et a déjà fait l'objet d'une présentation l'an dernier à SSTIC [10].

Dans les grandes lignes, le cheminement entre les blocs irréductibles, les *basic blocs*, sera remplacé par la mise à jour d'un contexte.

Dans une boucle infinie, ce contexte est analysé, et indique quel est le prochain basic block à être exécuté.

Le CFG apparaît alors sous la forme d'un ou plusieurs *switch case*, à l'intérieur de boucles.

Afin de l'attaquer, deux angles ont été envisagés :

– analyser la manière dont la protection est faite, pour ensuite essayer de l'inverser, afin de retrouver le code original ou un code proche équivalent ;
– essayer de comprendre directement ce que fait le code, sans l'information du CFG.

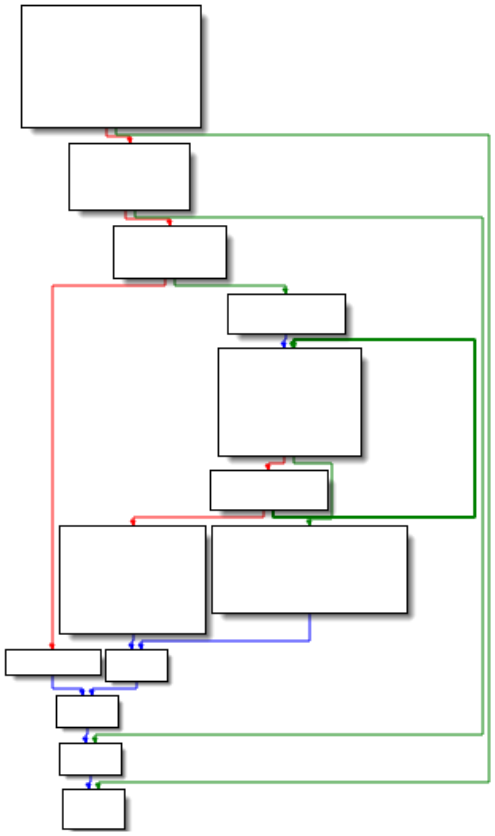


Figure 1. Un CFG normal

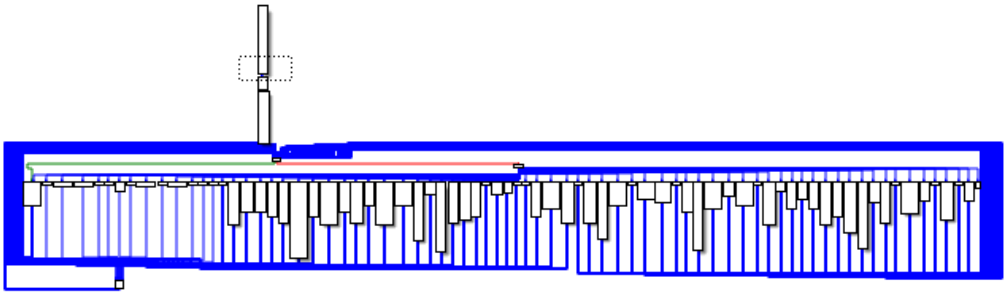


Figure 2. Un CFG ayant subi une transformation de type Code Flattening

2.2 Approche symbolique

L'analyse de la protection a l'avantage de capitaliser les recherches effectuées, et de permettre de les réutiliser si les mêmes outils d'obscurcissement ont été employés autre part. En revanche, il implique de rentrer très en détail dans la protection, gaspillant potentiellement des ressources inutilement.

La méthode de protection n'étant pas triviale, cette approche a dans un premier temps été laissée de côté.

Nous retravaillons actuellement dessus, notamment grâce à de l'analyse symbolique, de la détection de structures redondantes...

Cette méthode est très proche de ce qui a été présenté lors de la conférence SSTIC 2013 [10].

À l'heure de l'écriture de cet article, nous sommes capable de reconstruire de petites fonctions (de l'ordre d'une vingtaine de basic blocks dans leurs versions obscurcies).

Afin d'éviter une explosion combinatoire sur les grandes fonctions, nous essayons quelques heuristiques. Par exemple, la détection des boucles et de leurs variants dans le but de réaliser des exécutions concoliques.

Cependant, cette approche se heurte par exemple aux cas des obscurcissements de *switch case* et du calcul de leurs différents cas atteignables.

Ces travaux, incomplets, ne seront pas abordés dans cet article.

2.3 Approche par trace d'exécution

Une trace d'exécution est un suivi de l'évolution du contexte lors de l'exécution d'un programme. L'intérêt d'une trace lors de l'analyse d'un code aplati est d'obtenir un chemin possible du flot d'exécution. En revanche, cela implique que nous omettons d'autres chemins, et donc potentiellement d'autres fonctionnalités.

Le contexte d'exécution comprend le flot d'instructions, mais aussi le flot de données. La Figure 3 présente un exemple d'informations pouvant être récoltées.

En plus des informations concernant le flot de données (mise à jour des registres et accès mémoire), nous apprenons aussi que le saut n'a pas été pris [1].

1. C'est une information qui peut toujours être déduite de l'état des données avant l'exécution de l'instruction. Néanmoins, dans le cadre de l'analyse du code flattening, il est intéressant de l'avoir.

```
mov     eax, #0                         ; eax <- 0
lea     ebx, DWORD PTR [eax + eax]      ; ebx <- 0
mov     DWORD PTR[0x11223344], ebx      ; @32[0x11223344] <- 0
jz      0x1337beef                      ;
xor     ecx, ecx                        ; ecx <- 0, zf <- 0, nf <- 0...
```

Figure 3. Exemple de trace d'exécution

Afin d'utiliser au mieux cette approche, nous avons développé un framework nommé "pTra". Il vise a fournir des APIs permettant d'interagir avec la trace d'exécution d'un programme.

3 pTra

3.1 Présentation

"pTra" (Python TRace Analyser) est une plate-forme de traitement de trace d'exécution. Le but est de permettre de proposer des fonctionnalités (résumées sur le diagramme de cas d'utilisation de la Figure 4) avec une architecture modulaire, et d'être capable de passer à l'échelle. En effet, nous souhaitons être capable de gérer plusieurs centaines de millions d'instruction et autant d'accès mémoire.

De manière moins informelle, nous souhaitons traduire la possibilité de tester rapidement une idée d'exploitation de la trace.

Des exemples de ce qu'il est possible de faire sont illustrés dans les sections suivantes.

3.2 Réalisation

Architecture logicielle Le type de l'architecture logicielle est "en couches", représenté sur la Figure 5.

Les différentes couches sont :

- **la base de données** permet le stockage des informations à long terme ;
- **Database Access Object (DAO)** interface sur la base de données présentant une API constante. Cela permet de changer le type de base de données de manière transparente ;
- **les modèles** permettent d'instancier des objets représentant les éléments de la base de données, de les modifier et de les sauvegarder. Ils sont brièvement introduits dans le paragraphe suivant ;

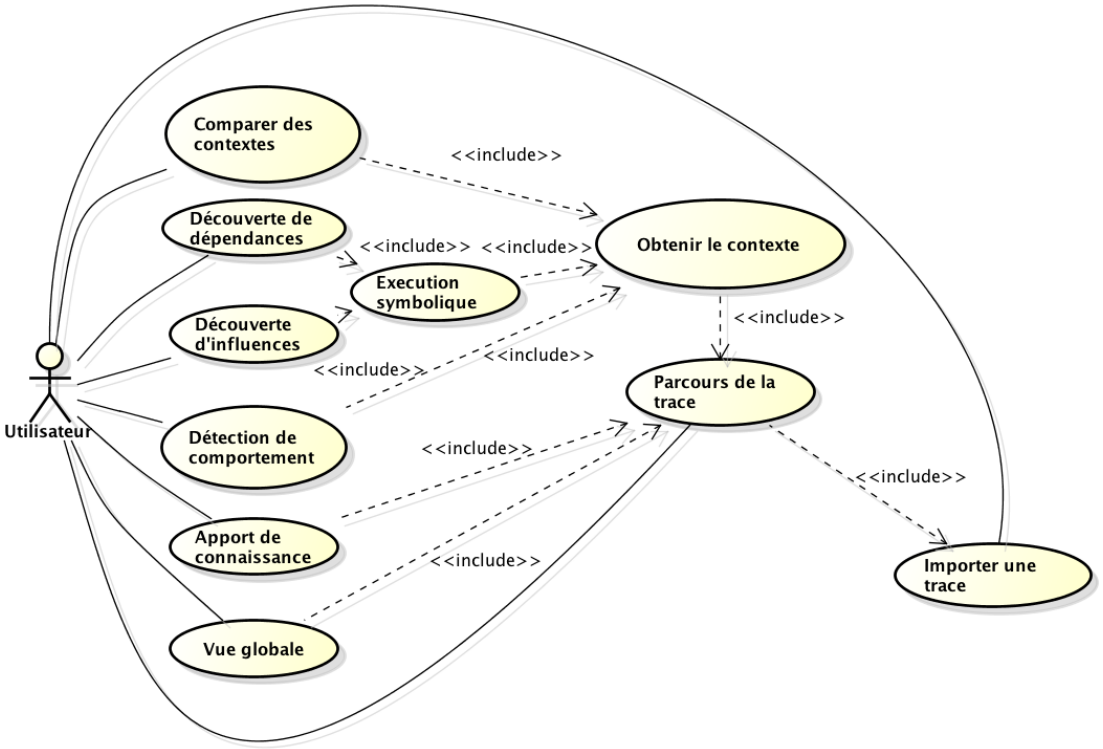


Figure 4. Cas d'utilisation de pTra

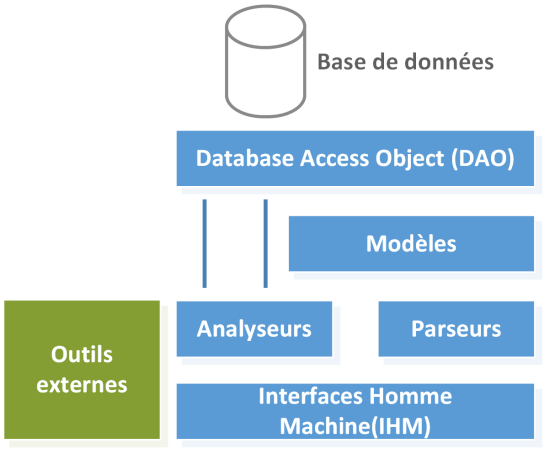


Figure 5. Architecture logicielle

– **les analyseurs** effectuent des opérations sur les éléments en base. C'est là que se trouve "l'intelligence" du système. Pour des raisons d'optimisation, ils peuvent dialoguer directement avec le DAO ;
– **les parseurs** traduisent une trace d'exécution écrite dans un format spécifique vers un ensemble d'instances des modèles ;
– **Interfaces Homme-Machine (IHM)** : interfaces avec l'utilisateur. Elles permettent de visualiser, de faire le lien et d'interagir avec les différents analyseurs ;
– **les outils externes** : les analyseurs, parseurs ainsi que les interfaces peuvent avoir recours à des outils externes afin de réaliser d'autres tâches, comme l'exécution symbolique ou le désassemblage.

Modèle de données La Figure 6 présente les modèles utilisés en base de données, pour la version à la date d'écriture de ce document. Ils ont tous, en plus de leurs champs spécifiques, un attribut d'identification unique "id".

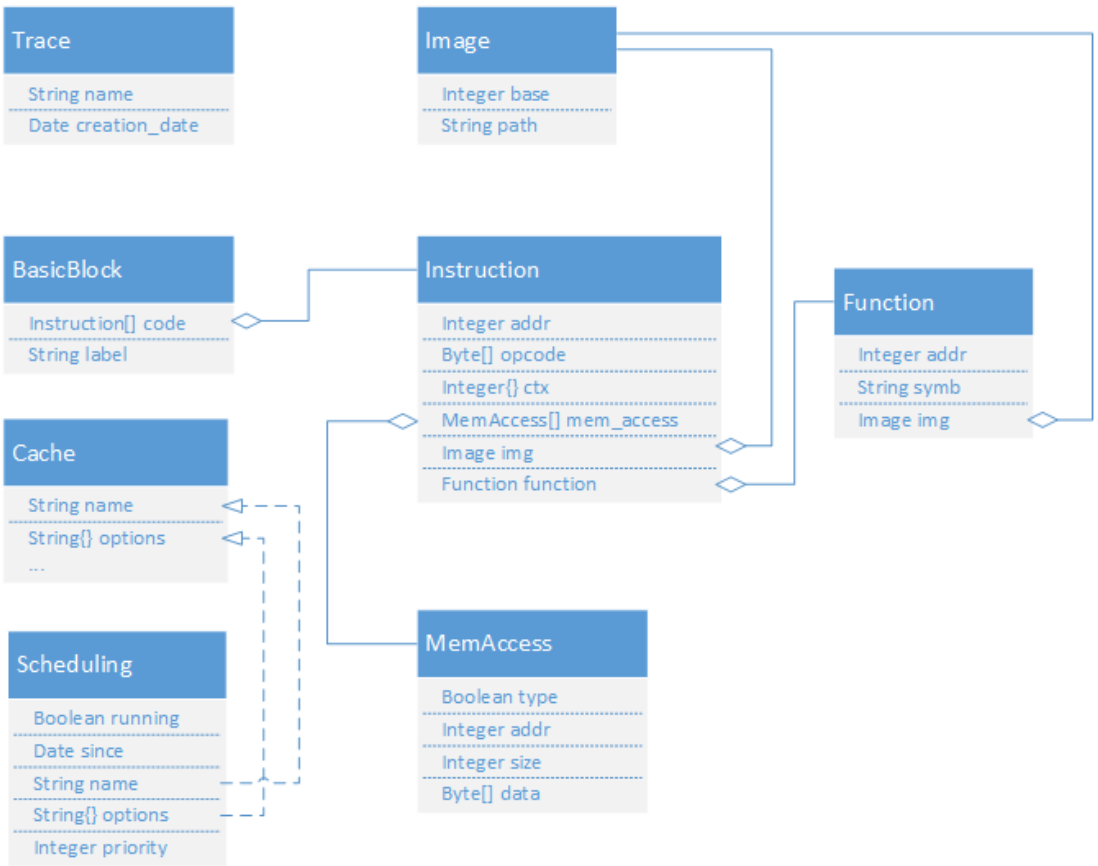


Figure 6. Modèles de données dans pTra

Voici la liste des différents modèles, au niveau de la trace d'exécution :

- **Trace** permet de conserver des méta-informations sur la trace comme le nom associé ainsi que la date d'import ;
- **Image** représente les librairies chargées dans la trace. Ce modèle va permettre de stocker le chemin d'une librairie (`path`) ainsi que l'adresse de base à laquelle elle a été chargée ;
- **BasicBlock** représente un ensemble d'instructions, et permet de les séparer en parties plus facilement traitables (de par leur nombre plus restreint). Un élément de BasicBlock représente ainsi un basic block dans la trace d'exécution ;
- **MemAccess** symbolise un accès vers la mémoire. Le type peut être soit une écriture, soit une lecture. L'accès sera fait à l'adresse `addr`, et un champ d'octets de taille `size` et de valeur `data` sera écrit/lu ;
- **Function** décrit une fonction dans la trace d'exécution, c'est-à-dire la cible d'une instruction d'appel de fonction. L'appel se fera vers l'adresse `addr` de la librairie `img`. Si les symboles sont renseignés, ils apparaissent dans `symb` ;
- **Instruction** représente une instruction dans la trace d'exécution. Il définit ainsi l'adresse `addr` de l'instruction dans la librairie `img`, ainsi que l'opcode `opcode` exécuté. Si un appel de fonction a été fait, il sera lié à cette instruction via `function`. De même, si un ou plusieurs accès mémoire sont faits, ils seront liés via `mem access`. Enfin, l'attribut `ctx` permet de connaître le contexte initial, avant l'exécution de l'instruction, c'est-à-dire l'état de l'ensemble des registres ;
- **Cache** permet de stocker les résultats d'opérations précédentes et de les réutiliser. L'attribut `name` désigne le type d'opération, `options` les différents paramètres. Le reste de la structure et des attributs dépend de l'opération ;
- **Scheduling** permet l'ordonnancement de tâches. Ses différents attributs sont `running` pour indiquer si la tâche est en train de tourner, `since` indiquant, si la tâche est en train d'être exécutée la date de début d'exécution, `name` et `options` les champs correspondants dans Cache et `priority` pour obtenir une file de priorité.

Nous avons choisi de ne représenter, par base de données, qu'une seule trace. Cela permet d'obtenir un gain de performance, grâce aux assertions suivantes :

- Si x et y sont dans la même table, $Id(y) > Id(x) \equiv$ l'événement décrit par x a eu lieu avant celui décrit par y. Autrement dit, l'axe des Ids représente l'axe temporel, pour une table donnée ;

– Si x et y sont dans la même table, il y a $Id(y) - Id(x)$ événements après l'événement décrit par x et avant l'événement décrit par y, pour une table donnée ;
– Quels que soient les éléments x et y, de deux tables quelconques, ils se rapportent à la même trace d'exécution.

Nous avons fait d'autres choix dans le but d'optimiser les performances. Tout d'abord, l'implémentation d'un cache, permettant de stocker des résultats et de limiter ainsi le nombre de requêtes vers la base de données.

Ensuite, l'utilisation d'un ordonnanceur permet de lancer des requêtes et de stocker les résultats en cache sur les temps d'inactivité de la base, comme les nuits ou les week-ends. Il est en effet possible de prévoir un ensemble de requêtes qui seront probablement effectuées par l'utilisateur dans son utilisation du framework.

Enfin, nous avons privilégié la rapidité de récupération du contexte par rapport à la place prise par la table d'instructions. En effet, pour chaque instruction, l'état détaillé des registres y est stocké. Il est donc possible de récupérer, en une seule requête, le contexte d'exécution d'une instruction. En revanche, il y a une perte d'espace, car il serait possible de ne stocker que les changements de contexte par rapport à l'instruction précédente (empiriquement, seuls un ou deux registres sont modifiés par l'exécution d'une instruction).

Implémentation L'implémentation du framework a été réalisée en Python. En effet, l'aspect performance des calculs n'est pas ici prioritaire ; c'est la rapidité des traitements mémoires, l'aspect modulaire et la facilité d'extension qui sont mis en avant. C'est dans cette optique que le choix s'est porté sur ce langage. Pour les parties de calcul, peu fréquentes mais sur lesquelles l'interpréteur Python n'est pas très efficace, nous utilisons l'interpréteur PyPy [5] [2].

Pour la base de données, le choix s'est porté sur MongoDB [2], car :
– c'est un système de base de données non relationnelles : il permet de modifier rapidement la structure de la base pendant la mise au point du framework, de stocker des structures différentes au sein d'une même table (appelée alors "document"), comme pour le Cache ;

2. Ce dernier utilisant des méthodes de compilation à la volée permettant d'optimiser, entre autres, les boucles

– il s'intègre très bien à Python, notamment grâce au DAO PyMongo [4].

Le passage a l'échelle est assuré par l'utilisation de MongoDB, ainsi qu'à une consommation mémoire indépendante de la quantité de données traitées, via l'utilisation massive d'itérateurs Python.

Pour mesurer le passage à l'échelle, nous utilisons la taille du fichier de trace d'exécution obtenu grâce à notre outil. Elle augmente linéairement en fonction de la taille de la trace d'exécution.

L'outil actuel permet de gérer des traces de plus de 30Go, la limite n'étant plus dans la capacité de l'outil à gérer des quantités de données, mais dans le temps nécessaire à l'exécution des actions. L'outil utilisé pour créer la trace d'exécution est basé sur Intel PIN (c'est un "pintool"). Il va permettre de récupérer la liste des librairies chargées, la liste des instructions exécutées et des changements de contextes associés (lectures/écritures mémoire, registres). Le résultat, sous forme de fichier texte, n'est pas compressé.

Concernant les données citées plus haut, une trace de 30Go représentera, en moyenne, 300 millions d'instructions exécutées (et environ la moitié d'accès mémoire). Enfin, les autres outils externes utilisés sont Miasm [6] pour l'exécution symbolique et Distorm [1] pour le désassemblage. Ils ont tout deux été remplacés par Miasm2.

4 Reconstruction d'un algorithme : RSA-OAEP

Une fois la trace d'exécution obtenue, et chargée en base de données, il s'agit maintenant de s'en servir pour comprendre ce que fait le programme.

Cette section présente comment nous avons pu, avec des approches différentes, comprendre qu'un des algorithmes exécutés était "RSA-OAEP"[3].

4.1 Détection de constante

Lorsqu'un analyste souhaite vérifier la présence d'algorithmes connus, comme par exemple des algorithmes de chiffrement symétrique ou des fonctions de hash, l'une de ses premières actions sera de se lancer à la recherche de constantes dites "magiques" (connues, publiques) dans le code du programme cible.

À titre d'exemple, nous pouvons citer le plugin KANAL de l'outil PEiD [3] qui réalise ce type de recherche de manière statique, et localise précisément ces informations dans le programme ciblé.

3. `http://fr.wikipedia.org/wiki/Optimal_Asymmetric_Encryption_Padding`

```
sha1_ctx->state[0] = 0x67452301;
sha1_ctx->state[1] = 0xefcdab89;
sha1_ctx->state[2] = 0x98badcfe;
sha1_ctx->state[3] = 0x10325476;
sha1_ctx->state[4] = 0xc3d2e1f0;
```

Figure 7. Les constantes d'initialisation SHA-1

Dans le cas où le code n'est pas protégé, cette opération est immédiate, étant donné que les constantes apparaissent en clair et au bon endroit, c'est-à-dire dans les fonctions qui vont initialiser (l'exemple de "SHA-1" dans la Figure 7) un algorithme cryptographique. L'analyse statique peut donc très bien s'appliquer à ce cas de figure. Il reste toutefois aisé de modifier le code afin que ces constantes soient calculées dynamiquement. En revanche, il est moins facile de modifier les algorithmes afin de leur permettre de fonctionner sans jamais utiliser ces constantes, car elles possèdent généralement des propriétés particulières.

Dans notre cas d'analyse, les constantes ne sont pas accessibles statiquement, mais modifiées, et leurs utilisations "dilués" dans les différentes couches de code obfusqué. Ainsi il n'est possible de les détecter qu'à l'exécution du code cible, code qui est bien souvent perdu sous plusieurs couches d'obscurcissement.

D'après les observations faites en 1.1, nous supposons que la DRM utilise a un moment ou un autre des fonctions de hash classiques, telles que MD5 ou SHA-1. Il est donc fort probable que ces constantes magiques apparaissent au moins une fois dans les registres du processeur.

Or, via la trace d'exécution, nous disposons d'un accès à l'ensemble des valeurs des registres pour chacune des instructions exécutées. Ainsi, en effectuant un parcours de cette base à la recherche de constantes connues d'une multitude d'algorithmes cryptographiques, nous avons pu mettre en évidence la présence des cinq constantes d'initialisation de l'algorithme de hash SHA-1 sur plusieurs zones de code très localisées. Cette faible distance entre les différentes constantes dans la trace nous permet d'éviter d'éventuels faux positifs, même s'il est très peu probable de tomber sur ces constantes en l'absence des algorithmes correspondants.

Le lecteur attentif remarquera aussi que cette approche ne perd pas en généralité par rapport à la recherche des constantes dans le programme : si elles sont présentes, en clair et utilisées, elles apparaîtront dans les registres/cases mémoires.

Après avoir confirmé qu'il s'agissait bien de l'algorithme voulu, en vérifiant notamment les entrées et sorties des fonctions correspondantes, il ne reste plus qu'à rajouter cette connaissance dans pTra pour mettre à jour le graphe des appels avec les noms adéquats, ici SHA-1_{INIT}.

De la même manière, nous avons pu détecter la présence d'un Mersenne Twister [4] dans le code de la DRM.

4.2 Obscurcissement des données

Différentiel mémoire

Une des méthodes de l'analyse dynamique pour comprendre ce que fait un code est l'observation des modifications qu'il engendre sur les données de la mémoire.

Il est possible de réaliser cela assez facilement, en partant de constats simples :

- si une donnée est utile et traitée, elle sera forcément lue ;
- les résultats des opérations seront écrits, soit en place, soit dans un autre endroit mémoire.

Notre base de données contenant l'ensemble des accès mémoires réalisés par chaque instruction, il est possible d'obtenir le contenu de la mémoire à un instant donné en remontant, pour chaque adresse mémoire ciblée, au dernier accès la concernant (lecture ou écriture).

Plus généralement, il est possible d'effectuer un différentiel mémoire entre deux instants de l'exécution.

Via une représentation adaptée - comme le montre la Figure 8 - nous avons accès rapidement aux données traitées, au moment de leur traitement, au type du traitement (lecture d'un *DWORD*...) auquel viennent s'ajouter quelques heuristiques telles que la détection de pointeurs et le calcul de l'entropie des blocs de données consécutifs.

Il reste alors à trouver les différentiels mémoires potentiellement intéressants. Quelques pistes sont :

- entre le début et la fin d'une fonction ;
- entre le début et la fin d'un tour de boucle ;
- les accès concernant les arguments d'une fonction, liste d'arguments obtenue en regardant les accès à la partie supérieure de la pile dans le cas d'un appel standard, ou aux registres dans le cas d'un appel rapide.

4. `http://en.wikipedia.org/wiki/Mersenne_twister`

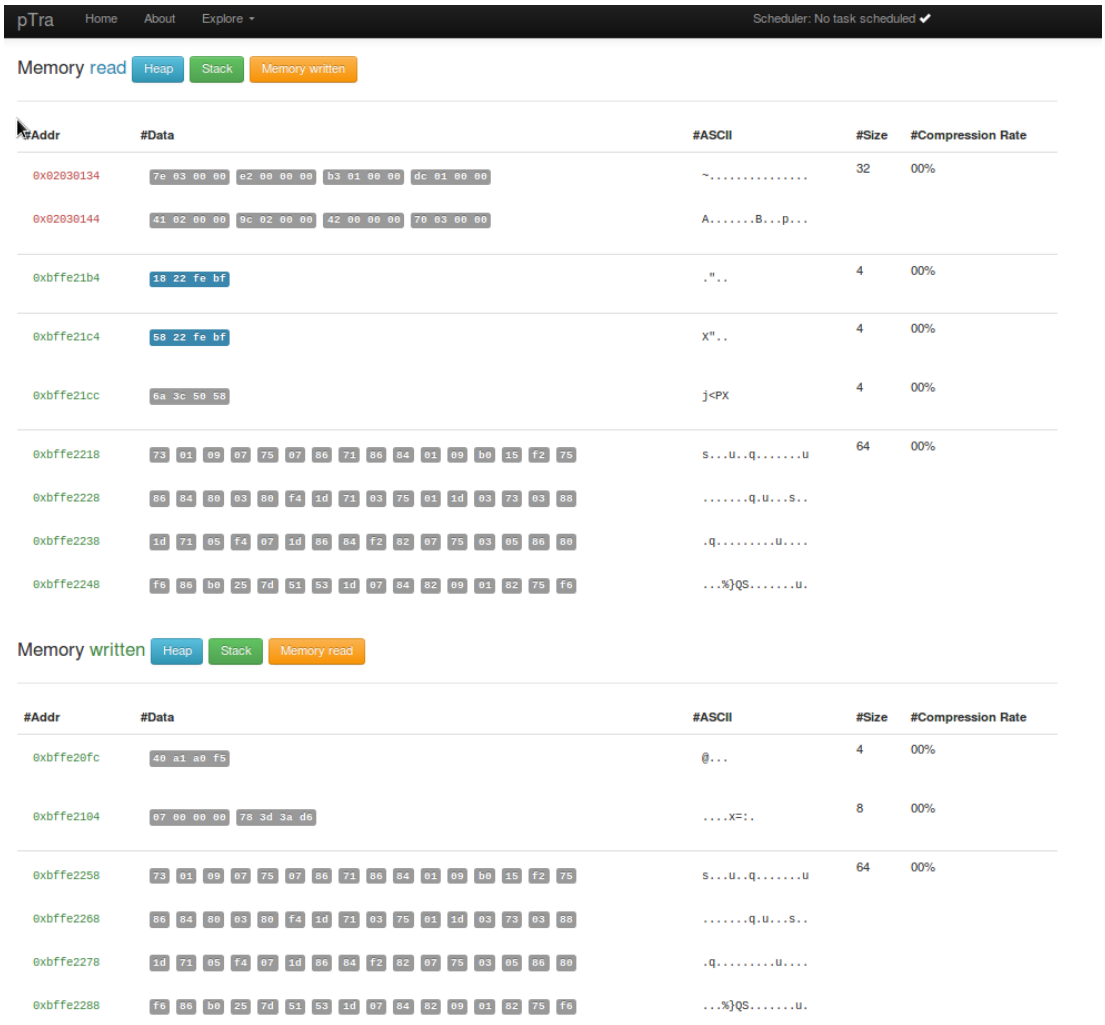


Figure 8. Aperçu du différentiel mémoire d'un `memcpy`

Mémoire transformée

En appliquant cet outil à notre cas d'étude, nous nous sommes rendu compte que les données passées en entrée de SHA-1, et plus généralement des fonctions, ainsi que celles en sortie, n'apparaissent pas en clair.

En remontant jusqu'à leur écriture, nous avons effectivement remarqué que ces données ont systématiquement été modifiées via l'application d'une fonction avant d'être stockées.

Le plus souvent, ces fonctions sont de la forme $f(x) : ax + b$ ou $f(x) : ax \oplus b$. Elles sont appliquées au niveau de l'octet, du word ou du dword.

Avant de réutiliser ces données, le programme va effectuer une nouvelle transformation, de la même famille que la première, lui permettant de retrouver les données initiales.

Cette protection, bien qu'assez simpliste, devient vite très embarrassante : il est très rare de connaître les données en clair au milieu du déroulement d'un algorithme.

En effet, quelques observations peuvent être faites sur cette protection :

- Il est très dur de savoir si les fonctions affines utilisées font partie de l'algorithme ou de la protection (il serait nécessaire de conserver l'ensemble des modifications effectuées sur une donnée) ;
- Il est tout à fait possible de stocker $f_1(a)$, puis $f_2(f_1(a))$ et d'appliquer enfin $f_3(f_2(f_1(a))) = a$, f_1, f_2, f_3 étant des fonctions affines, dans le but de casser la logique d'une fonction pour l'écriture et d'une autre pour la lecture ;
- La protection peut être réalisée de manière à varier suffisamment les bouts de code effectuant les fonctions, afin d'empêcher l'analyste d'identifier les motifs correspondants et de les inverser automatiquement.

En revanche, il est quand même possible d'extraire des informations :

- Sur un aspect pratique, les données se trouvant dans un même intervalle mémoire se doivent d'utiliser les mêmes fonctions d'obscurcissement. Sinon, il faudrait que le moteur d'obscurcissement soit capable de résoudre, à la compilation, l'ensemble des adresses que pourra atteindre une portion de code ;
- Grâce au point précédent, 2 octets distincts mais présents sur la même sortie représentent la même donnée (si l'obscurcissement est fait octet par octet).

Retour à notre cas : SHA-1

D'après ce qui a été développé en 4.2, nous savons que l'analyse mémoire des appels aux fonctions identifiées comme étant $SHA1\_update$ va laisser apparaître des fragments de la donnée d'entrée originale.

De plus, une fois ces morceaux identifiés, nous pouvons calculer le SHA-1 de ces données et le comparer avec le SHA-1 obscurci.

Supposons que la sortie du vrai SHA-1 soit :

```
AA 7B 15 07 B1 12 8D 15 45 D5 F2 47 E7 3F 15 9A C6 27 0B 1B
```

Et que celle obtenue en sortie du programme soit :

```
C6 6D A0 F2 9D D6 28 A0 3D 13 B1 C3 CD A9 A0 92 22 5B 00 34
```

On peut déjà remarquer que la répétition du 15 dans le premier hash correspond à une répétition du $A0$ dans le second.

Si, sur la base de ce qui a été observé précédemment, on fait l'hypothèse que la fonction de transformation est de l'une des formes introduites en 4.2, on obtient assez rapidement : $f(x) : x * 43 + 1C$

En plus de valider notre hypothèse de fonction de hash, nous pouvons maintenant obtenir une partie de l'état mémoire en clair et inverser les transformations des entrées et des sorties.

Il s'est avéré qu'en observant les différentes entrées de la fonction au cours de la trace, les données hashées étaient des certificats.

L'ordre ainsi que la présence de ces SHA-1 dans les certificats fait directement penser à une vérification de chaîne de certificats. La valeur du champ `Certificate Signature Algorithm` valant `SIG_RSA_SHA1` permet elle de déduire la présence de chiffrement RSA.

Le premier élément de la chaîne, le certificat racine, est directement en dur dans le binaire, dans sa forme transformée. Les autres certificats sont fournis par des étapes qui précèdent l'appel à la DRM.

Nous avons alors essayé d'identifier les données qui allaient être chiffrées ; il s'est très vite avéré que le code effectuant RSA est complètement dilué et très fortement obscurci.

Nous avons alors modifié le certificat racine, connaissant la transformation à appliquer pour qu'il reste compréhensible pour le programme, ainsi que les certificats de la chaîne afin d'annuler le chiffrement. Cela s'obtient directement en utilisant pour modulo la borne maximum de la taille ($0xFFFFF...$) et pour exposant le chiffre 1. Une fois les SHA-1 correspondants mis à jour dans les certificats, nous obtenons directement sur la sortie le message.

Pour information, l'annulation de l'exécution des trois RSA a permis de réduire le nombre d'instructions exécutées de quasiment 50 millions. Cela donne une idée de l'état de l'obscurcissement appliquée au programme.

4.3 Data slicing et reconstruction de fonction

De la connaissance du flot d'exécution et du flot de données, nous pouvons faire du :

- **data tainting** : trouver l'ensemble des éléments influencés par un élément donné ;
- **data slicing** : trouver l'ensemble des éléments ayant influencé un élément donné.

Intuitivement, le tainting avancera dans le temps, là où le slicing le remontera.

Data slicing Pour implémenter le data slicing, nous avons utilisé l'émulation symbolique fournie grâce au framework Miasm.

Par la suite, pour plus de commodité, notre représenterons "l'instruction ayant l'identifiant `id` n" par "l'instruction n". Ainsi, d'après ce qui a été décrit en 3.2, l'instruction $x = 17$ est exécutée dans le temps avant l'instruction $y = 255$ et nous extrapolerons par $x < y$.

L'algorithme est le suivant :

Pour obtenir la liste des dépendances et l'équation d'un élément k du contexte de l'instruction x, noté k_x, par rapport au contexte de l'instruction y (avec $y < x$ puisque c'est du slicing), nous procédons de la manière suivante :

1. Soit z l'instruction exécutant la dernière modification de l'élément k, tel que $z < x$ et $z \geq y$. Si un tel z ne peut être trouvé, alors l'élément k a pour équation $k_x = k_y$ et dépend seulement de k_y.
2. Sinon, on exécute symboliquement le code entre y et z, depuis le flot d'exécution, en ignorant les sauts.
3. L'équation de k_x est alors la même que k_z, et ses dépendances sont les éléments variables de l'équation.

Pour illustrer cet algorithme, nous allons prendre l'exemple du flot d'exécution de la Figure 3 (rappelé pour plus de lisibilité dans la Figure 9).

```
mov     eax, #0                          ; eax <- 0
lea     ebx, DWORD PTR [eax + eax]       ; ebx <- 0
mov     DWORD PTR[0x11223344], ebx       ; @32[0x11223344] <- 0
jz      0x1337beef                       ;
xor     ecx, ecx                         ; ecx <- 0, zf <- 0, nf <- 0...
```

Figure 9. Rappel de la Figure 3

Pour l'élément `edx` du contexte de l'instruction ligne 5, l'algorithme donne :

1. On recherche z, c'est-à-dire la dernière modification du registre `edx` entre la ligne 1 et la ligne 5 : il n'existe pas.
2. On a donc $edx_5 = edx_1$ et $dependances(edx_5) = dependances(edx_1)$.

Pour un exemple plus élaboré, intéressons nous au cas de la case mémoire $0x11223344$ du contexte de l'instruction ligne 5. Nous obtenons :

1. On recherche z : on obtient la ligne 3.
2. On exécute symboliquement les instructions entre la ligne 1 et la ligne 3, avec pour contexte initial $eax_1, ebx_1, ecx_1, \ldots, @32[0x11223344]_1 \ldots$.
3. On obtient $@32[0x11223344]_5 = ebx_3 = 2 * eax_2 = 0$.
4. On en déduit donc l'équation (la valeur 0), et une liste de dépendances vide.

Nous procédons de la même manière pour le data tainting : arrivé à l'instruction x, nous parcourons l'ensemble des dépendances des éléments du contexte pour trouver celles qui contiennent k_y : ce sont les éléments qui ont été influencés par notre élément k.

Le lecteur attentif remarquera que les dépendances implicites ne sont pas prises en compte : en ne tenant pas compte des sauts conditionnels, nous perdons les dépendances qui correspondent aux conditions de ces sauts.

Reconstruction du graphe de dépendance En appliquant le data slicing depuis une sortie provenant du contexte des entrées, il est donc théoriquement possible d'obtenir l'équation de transfert d'une fonction. Cependant, bien qu'applicable pour une petite fonction, le cas d'une fonction cryptographique entraînera des calculs suffisamment compliqués pour provoquer une explosion combinatoire, sans compter la lecture hasardeuse de l'équation de sortie.

Il reste cependant possible de tracer un graphe de dépendance, en appelant récursivement l'algorithme. En effet, nous avons constaté, empiriquement, que remonter en haut du basic block courant est souvent suffisant pour obtenir des équations avec peu de dépendances.

Depuis un élément racine, nous construisons ainsi un arbre de dépendance. Nous commençons par remonter en haut du basic block, ce qui nous donne de nouveaux noeuds. Ensuite, nous sautons directement à la dernière instruction ayant modifié l'élément, nous donnant le prochain

contexte de départ. Et ainsi de suite.
Cet arbre est représenté graphiquement sur la Figure 10 :

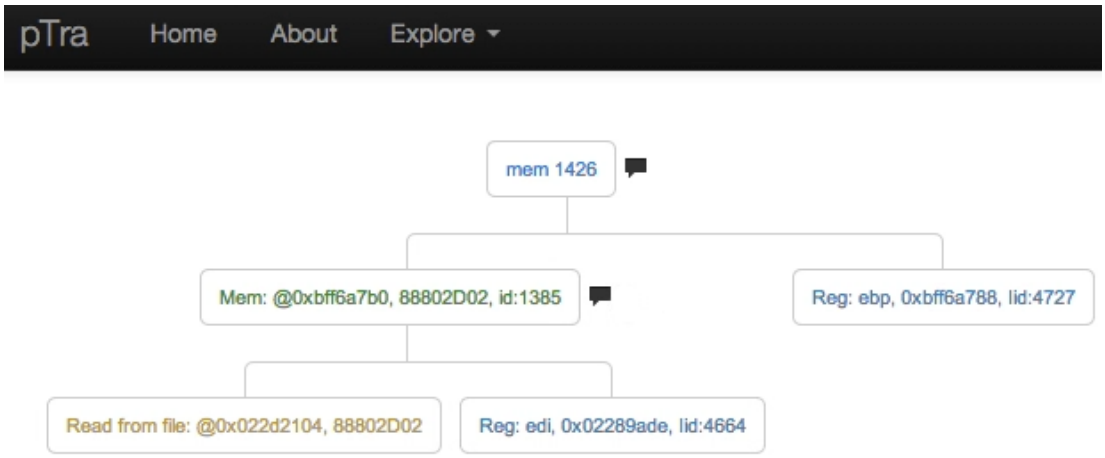


Figure 10. Exemple de data slicing (Vue Slicing du WebViewer)

1. On cherche les dépendances de l'accès mémoire numéro 1426.
2. L'accès dépend du registre `ebp` de l'instruction numéro 4727 et de l'accès mémoire numéro 1385 ; seul ce dernier semble intéressant, le registre `ebp` étant vraisemblablement présent pour indexer le déréférencement mémoire.
3. On applique alors récursivement l'algorithme sur l'accès mémoire 1385
   (a) Il dépend du registre `edi` de l'instruction 4664 ainsi qu'une valeur présente en dur dans le binaire.
   (b) Au passage, on remarque que la valeur est la même, qu'il s'agit donc par exemple d'un accès mémoire cherchant une valeur dans une table du binaire.

Retour au cas d'étude En exécutant du data slicing sur les données en entrée du RSA dont nous avons préalablement discuté (voir Section 4.2), nous avons pu tracer un graphe de dépendance entre différents éléments.

Dans ces éléments figurent des sorties de SHA-1, que nous savons maintenant identifier, des données provenant du générateur d'aléa (le Mersenne Twister) et des constantes.

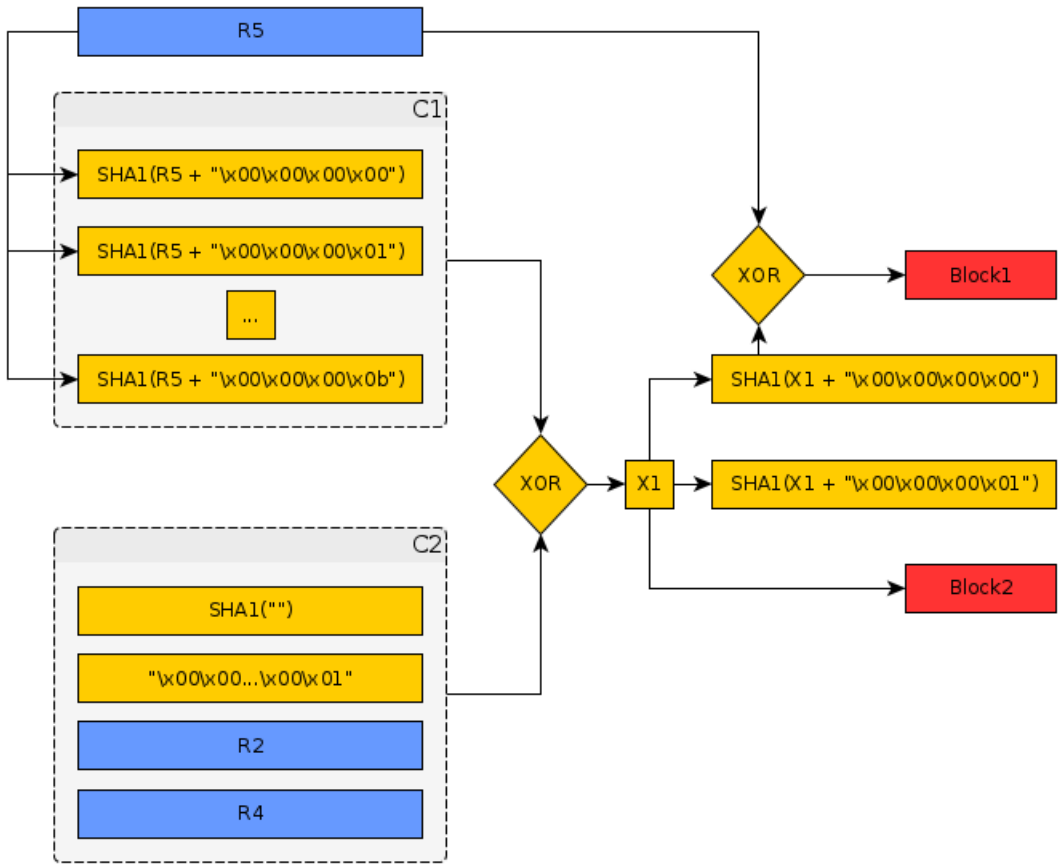


Figure 11. Graphe de dépendance RSA-OAEP

À l'aide de ces données nous avons pu reconstruire le graphique de la Figure 11. $R2$, $R4$ et $R5$ sont des nombres aléatoires.

Or cette manière de procéder correspond à l'algorithme RSA-OAEP [5] (rappelons ici que la sortie est chiffrée avec RSA). Une brève recherche sur l'Internet mondial nous permet de valider nos avancées : l'implémentation de la librairie OpenSSL présente sur le site "open source Apple" [6] correspond en tout point.

5. `https://fr.wikipedia.org/wiki/Optimal_Asymmetric_Encryption_Padding`
6. `http://www.opensource.apple.com/source/OpenSSL097/OpenSSL097-105/openssl/crypto/rsa/rsa_oaep.c`

5 Reconstruction d'une fonction de chiffrement "whiteboxée" : AES-CBC

À ce stade, nous avons une partie d'une des étapes de la DRM étudiée : RSA-OAEP. Dans cette section, nous nous intéressons à la construction d'une seconde partie de la sortie.

5.1 Quelques indices

Graphe de dépendance De la même manière que décrit en Section 4.3, nous établissons le graphe de dépendance de la partie de la sortie sujette de cette section. Ce graphe est représenté sur la Figure 12.

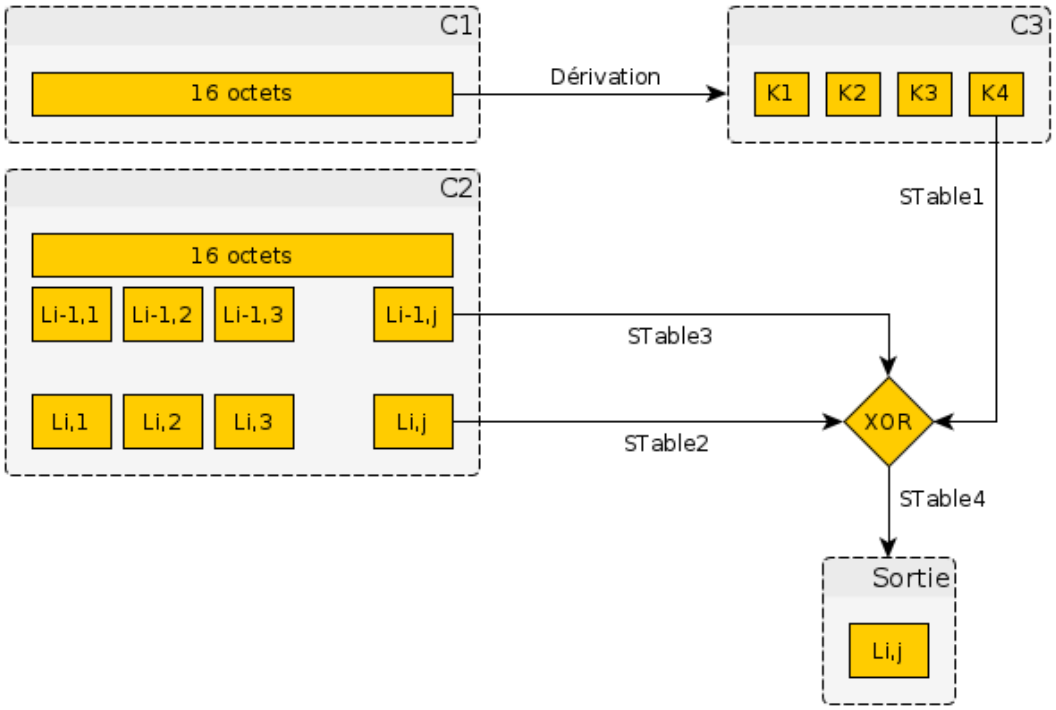


Figure 12. Graphe de dépendance WB-AES

Ce graphe présente une certaine ressemblance avec les algorithmes de chiffrement utilisant une dérivation de la clé ("key schedule").

Classe d'équivalence d'instructions Une pratique répandue de l'analyse dynamique est de poser un point d'arrêt sur une instruction traitant

une donnée afin d'identifier l'ensemble des données qui y transitent ; c'est typiquement le cas de l'étude d'un début de fonction ou d'un parseur.

Intuitivement, on comprend qu'il est fort probable que des éléments qui se ressemblent soient traités au même endroit, et que des éléments qui diffèrent ne le soient pas. Il est aussi probable que la première instruction qui lira une donnée soit dépendante du type de cette donnée.

En se basant sur ces deux hypothèses, nous avons établi la relation d'équivalence décrite informellement par :
Les données $d1$ et $d2$ sont équivalentes si et seulement si leur première lecture est faite par la même instruction. Deux instructions sont considérées comme étant les mêmes si et seulement si elles partagent la même adresse.

Le calcul des classes d'équivalence d'un bloc de données est alors très facile, via l'API de pTra. Il suffit de :

1. À partir de l'instant t où le bloc de données est en mémoire, trouver les instants après t (équivalents à trouver les instructions) correspondant aux premières lectures de chacune des données.
2. Pour chaque instruction ainsi obtenue, trier les données en fonction de l'adresse de l'instruction.

Ensuite, afin d'exploiter ces résultats, il s'agit de trouver un moyen de regrouper les données entre elles. Par exemple, si des données consécutives appartiennent à la même classe, on les regroupera par blocs. Mais on peut aussi regrouper les données consécutives présentant les même suites de classes. Par exemple, si les classes d'équivalences donnent :

```
Classe:  01  02  03  04  01  02  03  04  01  02  03  04  05
Donnée:  63  66  F5  F3  76  DC  B1  C1  F6  BC  4D  21  7E
```

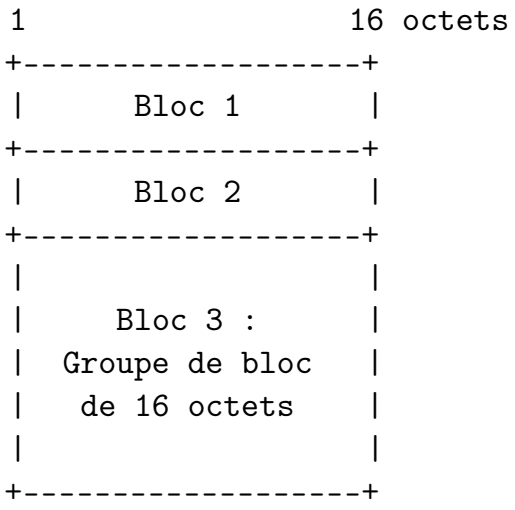
Il parait intéressant de regrouper en blocs les suites $1, 2, 3, 4$; il s'agit potentiellement de traitement effectués dans une boucle. On pourra donc présenter ce même jeu de données ainsi :

```
63  66  F5  F3
76  DC  B1  C1
F6  BC  4D  21
                7E
```

Cette astuce permet ainsi de dégager des structures dans des blocs de données, sans connaissances préliminaires.

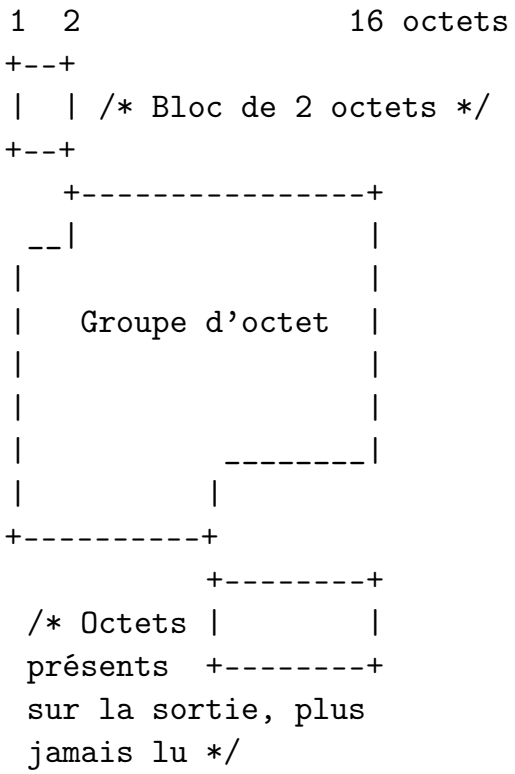
À l'aide du graphe de dépendance présenté en Section 5.1, nous avons isolé le bloc de données source des entrées.

L'application des classes d'équivalence sur ce bloc nous a clairement dégagé une structure (non recopiée pour des raisons de lisibilité) de la forme :



Le rapprochement avec les éléments du graphe est direct.

Nous avons aussi exécuté l'algorithme depuis la sortie immédiate de notre fonction inconnue. Cela a mis en évidence une structure de la forme :



Cela fait penser, comme nous le verrons par la suite, à un en-tête contenant la taille des données, suivi de ces dernières, pour enfin terminer avec du padding (et finir sur un multiple de 16).

Nous attirons l'attention du lecteur sur le fait que l'outil met en exergue l'absence de lecture, et donc de vérification du padding ; l'impact de cette absence est laissé en exercice.

Reconstruction de fonction Lorsque les fonctions ne sont pas trop compliquées, il est possible de les reconstruire à partir des équations obtenues par data slicing.

À l'aide d'un moteur de traduction du langage intermédiaire de Miasm vers un langage exécutable, comme du Python, il est même possible d'automatiser la tâche.

Par ce procédé, nous obtenons une fonction qui nous donne les mêmes sorties que le programme tracé en prenant les mêmes entrées ; c'est une forme "d'extraction" de code.

Quand les fonctions sont suffisamment courtes, nous effectuons des passes, manuelles cette fois, pour les clarifier, identifier les boucles, leurs compteurs ou leurs conditions, faire apparaître des tables de substitution, etc.

Il serait possible d'automatiser une partie de ces tâches, mais c'est une voie qui n'a pas été explorée.

En appliquant cette reconstruction au cas de la dérivation du bloc C1 du graphe (Figure 12) pour obtenir le bloc C3, nous obtenons la fonction :

```
def make_C3(inp):

        C3 = [inp]
        for i in xrange(10):
                tmp = []
                tmp.append(inp[0] ^ table1[(0x100*i)+inp[13]])
                tmp.append(inp[1] ^ table2[inp[14]])
                tmp.append(inp[2] ^ table2[inp[15]])
                tmp.append(inp[3] ^ table2[inp[12]])
                tmp.append(inp[4] ^ tmp[0])
                tmp.append(inp[5] ^ tmp[1])
                tmp.append(inp[6] ^ tmp[2])
                tmp.append(inp[7] ^ tmp[3])
                tmp.append(inp[8] ^ tmp[4])
                tmp.append(inp[9] ^ tmp[5])
                tmp.append(inp[10] ^ tmp[6])
                tmp.append(inp[11] ^ tmp[7])
                tmp.append(inp[12] ^ tmp[8])
                tmp.append(inp[13] ^ tmp[9])
                tmp.append(inp[14] ^ tmp[10])
                tmp.append(inp[15] ^ tmp[11])
                C3.append(tmp)
                inp = tmp

        return C3
```

Dans cette fonction, prenant en entrée un vecteur de 16 octets, `table1` et `table2` sont des tables de substitution d'octets : elle représente une fonction bijective de $GF(8)$ dans lui même.

Grâce aux indices obtenus par nos différentes observations, nous avons comparé cette fonction avec plusieurs fonctions de "key scheduling" connues ; la ressemblance avec la dérivation de clé d'AES est flagrante, comme le montre la Figure 13.

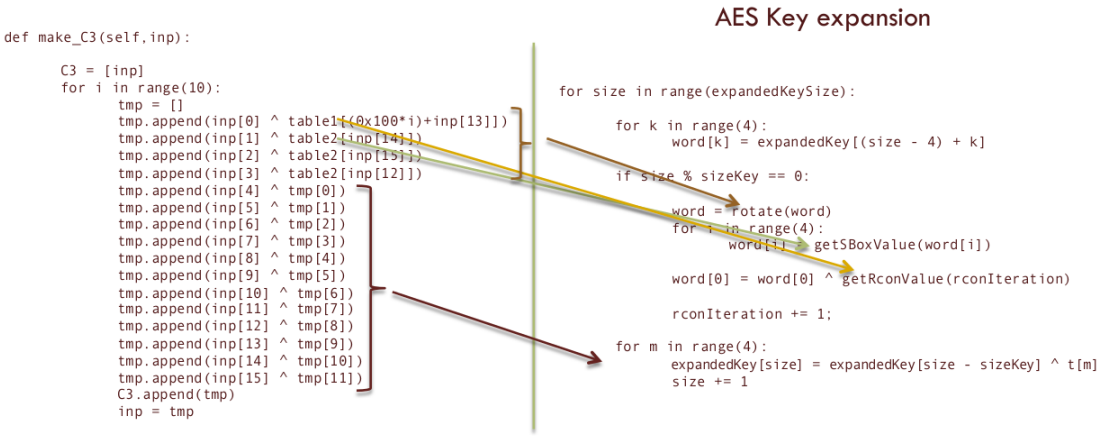


Figure 13. Comparaison `make_C3` et dérivation de clé AES

Avec le schéma de dépendance, nous pouvons préciser l'algorithme : il s'agit d'AES-CBC 128bits.

5.2 Identification d'une WhiteBox AES-CBC dynamique

L'algorithme étant identifié, nous essayons donc de reproduire les mêmes sorties à partir des mêmes entrées, sans résultat.

Il s'avère que l'étape de chiffrement est entièrement effectuée sur des états dérivés ; autrement dit, les données telles qu'elles se présentent dans l'algorithme AES standard n'apparaissent jamais dans cette fonction. Elles seront présentes sous une forme dérivée, transformée.

Nous avons nommé ce type de fonction "WhiteBox AES dynamique". En effet, elle est proche de ce qui s'appelle traditionnellement une WhiteBox AES (traité dans [8,9]) tout en permettant de prendre en entrée différentes clés.

Les intérêts d'un tel algorithme dans une DRM, alors qu'il peut être simplement extrait, sont :

- Une perte de temps pour l'analyste ;
- L'impossibilité de connaître ni l'entrée ni la sortie ;
- L'impossibilité de reproduire l'algorithme sur une autre plate-forme (dans le cas de l'interopérabilité, par exemple) ;
- L'impossibilité d'utiliser l'algorithme inverse s'il ne nous est pas fourni.

5.3 Résultat

Ce genre d'algorithme présente certaines propriétés mathématiques. En effet, si nous souhaitons que la transformation soit faite de bout en bout, il est donc nécessaire d'avoir des transformations qui soient homomorphes à l'opération XOR.

De plus, dans AES, les opérations XOR mélangent des données provenants de toutes les entrées.

Ces éléments réduisent considérablement les possibilités de fonctions de transformation. Le calcul effectif de ces fonctions ne rentre pas dans le cadre de cet article.

Finalement, cette approche nous a permis de mettre en évidence l'utilisation d'un algorithme AES-CBC 128bits et de connaître ses clés, ses vecteurs d'initialisation ainsi que ses messages. Ainsi, dans la DRM, l'algorithme intervient pour partager un secret avec le serveur.

6 Le moment écolo : les instructions équivalentes

6.1 Présentation

La dernière partie de la sortie de notre cas d'étude semble constituer de nombreux calculs. Ces calculs interviennent par exemple dans la mise à jour du contexte de l'aplatissement de code ou dans de longues transformations aboutissant sur la sortie.

Ils mettent en évidence une autre méthode de protection : le remplacement de parties de fonction par des équivalents bien moins intelligibles.

Par exemple, la fonction $f(x) = (16 * x + 16)\ mod\ 2^{32}$ pourrait être réécrite :
$f(x) = 129441535 - 1793574399 * (1584987567 * (3781768432 * x + 2881946191) - 4282621936)$, pour $x \in [0, 2^{32} - 1]$

Néanmoins, cette fonction serait aisément simplifiable à l'aide notamment de propagation de constantes. Un compilateur récent produira la même sortie que pour la fonction originale.

Obfuscations non simplifiées L'article [11] introduit de nombreuses équivalences entre opérations classiques et équations équivalentes composées d'opérations booléennes et arithmétiques.

C'est par exemple le cas de la comparaison signée entre deux éléments, dont le résultat est stocké dans le bit de poids fort de l'équation :

$$((x - y) \oplus ((x \oplus y) \wedge ((x - y) \oplus x)))(E)$$

L'article [12] introduit une notion plus élaborée : les équations MBAs, pour "Mixed Boolean Arithmetic". Il propose une méthode permettant d'en générer.

Après l'avoir implémentée, nous avons généré quelques exemples :

$$(x + y) \equiv ((x \wedge y) + (x \vee y))$$

$$(x + y) \equiv ((x \oplus y) + 2 \times (x \wedge y))$$

$$(x \oplus y) - y \equiv (x \wedge \neg y) - (x \wedge y)$$

L'article va plus loin, en introduisant notamment une méthode permettant de créer des fonctions complexes, n-aires, retournant toujours la même valeur (aussi appelées "prédicats opaques").

À la date d'écriture de cet article, ces équations MBAs ne sont pas simplifiées par les compilateurs. Elles ne sont pas non plus simplifiées par des programmes plus poussés, tel MatLab [7], Maple [8], Mathematica [9] ou encore Z3 [10].

Il s'avère que la combinaison d'opérations logiques / arithmétiques perd totalement les moteurs de simplification de ces outils.

Capitalisation Ces remplacements apparaissent tout au long du programme. Néanmoins, nous avons constaté que les mêmes éléments avaient tendance à apparaître plusieurs fois, seuls ou combinés avec d'autres.

Afin de ne pas perdre de temps à identifier et ré-analyser ces éléments, il est possible de capitaliser ces connaissances. Cela se traduit par l'ajout au sein de Miasm de nouvelles simplifications d'expression.

Si nous prenons l'exemple de la comparaison signée (6.1 (E)), nous devons :

7. http ://www.mathworks.fr/products/matlab/
8. http ://www.maplesoft.com/products/maple/
9. http ://www.wolfram.com/mathematica/
10. http ://z3.codeplex.com/

1. Identifier les expressions correspondantes à l'aide de la directive `MatchExpr` ;
2. Mettre à jour l'expression en la simplifiant ;
3. Indiquer à Miasm d'utiliser cette nouvelle simplification dans son moteur.

Cela se traduit par le code :

```
import miasm2.expression.expression as m2_expr
from miasm2.expression.expression_helper import expr_simp,
    expr_simp_cb

def check_slice_end_bit(expression):
    """Si @expression est le dernier bit d'un slice, retourne l'
        argument du slice;
    Sinon, retourne False"""

    if not isinstance(expression, m2_expr.ExprSlice):
        return False

    arg = expression.arg
    if expression.start != (arg.size - 1) or expression.stop != arg.
        size:
        return False

    return arg

# Jokers utilises pour remplaces n'importe quelle expression

jok1 = m2_expr.ExprId("jok1")
jok2 = m2_expr.ExprId("jok2")
jok3 = m2_expr.ExprId("jok3")


def expr_simp_inf_signed(expression):
    """Applique l'equivalence :
    ((x - y) ^ ((x ^ y) & ((x - y) ^ x))) [31:32] == x <s y"""

    arg = check_slice_end_bit(expression)
    if arg is False:
        return expression

    # Nous voulons jok3 = jok1 - jok2
    to_match = jok3 ^ ((jok1 ^ jok2) & (jok3 ^ jok1))
    # Canonisation de l'expression pour accelerer l'expression
        matching
    to_match_canon = to_match.canonize()

    # Recherche de correspondance
    result = MatchExpr(arg,
                       to_match_canon,
                       [jok1, jok2, jok3])

    # Si aucune correspondance n'a pu etre trouvee
```

```
    if (result is False) or (result == {}):
        return expression

    # @result est un dictionnaire : joker -> expression remplacee
    # On simplifie les deux parties avant de les comparer
    new_j3 = expr_simp(result[jok3])
    sub = expr_simp(result[jok1] - result[jok2])

    # Si jok3 = jok1 - jok2, c'est que l'on est dans le cas de l'
        equivalence
    if new_j3 == sub:
        # On remplace toute l'expression par l'operation "inferieur
            signee"
        return m2_expr.ExprOp(TOK_INF_SIGNED, result[jok1], result[
            jok2])
    else:
        return expression

# Indique au moteur de Miasm de prendre en compte cette nouvelle
    simplification
expr_simp_cb[m2_expr.ExprSlice].append(expr_simp_inf_signed)
```

Simplification des MBAs La construction des MBAs introduite dans l'article [12] repose sur la construction d'une matrice A et de son vecteur de combinaison associé v.

Ils permettent de construire une équation dans laquelle chaque élément $(x, y, x \oplus y \ldots)$ appartient à la base de l'ensemble. On peut alors écrire une matrice A composée de ces éléments, et chaque opération arithmétique est représentée par le signe de l'élément correspondant dans v.

Par exemple, l'équation $x + y - (x \oplus y)$ sera représentée par : $A = (f_1, f_2, f_3), v = (+1, +1, -1)$, avec $f_1 = x = (0, 0, 1, 1)$, $f_2 = y = (0, 1, 0, 1)$ et $f_3 = x \oplus y = (0, 1, 1, 0)$. On donne ici aussi une représentation des éléments de la base grâce à leur table de vérité.

Pour donner une expression valide, la combinaison linéaire des vecteurs colonnes de A avec pour coefficient les éléments de v doit valoir 0.

Ainsi, pour complexifier une expression, il suffit de partir de la matrice et du vecteur de combinaison associé et d'y ajouter de nouveaux vecteurs colonnes aléatoires avec des coefficients eux aussi aléatoires.

La complétion de cette matrice et de son vecteur pour atteindre une combinaison linéaire nulle permettra de traduire la construction d'une expression plus complexe.

De la même manière, partir de cette matrice et de ce vecteur et essayer de trouver la complétion qui permet d'atteindre la combinaison linéaire nulle en le moins d'ajout permet de trouver l'expression minimale correspondante.

Par exemple, si l'expression à simplifier est $x+\neg x-(x\wedge y)-(x\oplus y)+\neg y$, elle sera représentée par :

$$\begin{cases} A = \begin{matrix} 0\,1\,0\,0\,1 \\ 0\,1\,0\,1\,0 \\ 1\,0\,0\,1\,1 \\ 1\,0\,1\,0\,0 \end{matrix} \\ v = (+1,\ +1,\ \text{-1},\ \text{-1},\ +1) \end{cases}$$

Elle est équivalente au vecteur :

$$\begin{matrix} +2 \\ +0 \\ +1 \\ +0 \end{matrix}$$

Pour atteindre en un minimum d'ajout le vecteur nul, nous pouvons lui ajouter :

$$\begin{cases} A = \begin{matrix} 1\,1 \\ 0\,0 \\ 1\,0 \\ 0\,0 \end{matrix} \\ v = (\text{-1},\ \text{-1}) \end{cases}$$

Ce qui traduit la simplification :
$x + \neg x - (x \wedge y) - (x \oplus y) + \neg y - \neg y - \neg(x \vee y) = 0$
$x + \neg x - (x \wedge y) - (x \oplus y) + \neg y = \neg y + \neg(x \vee y)$

Même si cette méthode permet de simplifier automatiquemement certaines expressions, elle met en avant un problème : il est nécessaire d'étudier spécifiquement une méthode d'obscurcissement pour être en mesure de l'annuler.

La question est alors de savoir si l'investissement nécessaire est rentable dans l'analyse.

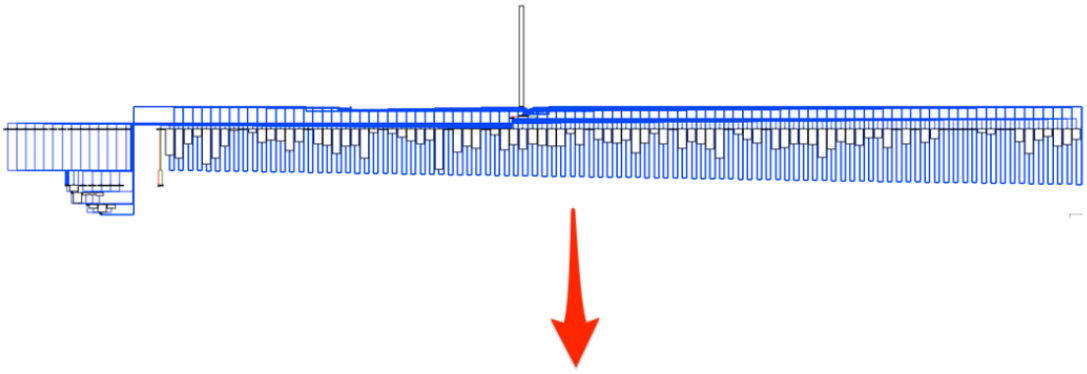
6.2 Version industrielle

Les derniers calculs effectués par la DRM impliquent des algorithmes de dérivation d'octets.

Ces algorithmes appliquent des niveaux d'obscurcissement très lourds en terme de performance.

Par exemple, la Figure 14 montre une fonction et sa fonction de transfert décrite en Python. La Figure 15 représente la même fonction de transfert

après identification des variables et application de passes classiques de simplification (propagation de constantes, etc.) pour finir sur sa fonction équivalente, un XOR.



```
int f(int x) {
    result = (0xed*((((((((((((((((((((((((- (((((((0xe5*x + 0xF7)&0xFF + ( 0x0 << 8)&0xFFFFFFFF)*0xFFFFFE2(
0x55)&0xFE)+(((0xe5*x + 0xF7)&0xFF + ( 0x0 << 8)&0xFFFFFFFF)*0xED)+0xD6)&0xFF&0xFF + ( 0x0 << 8)&0xFFFFFFF
F)*0x2))+0xFF)&0xFE)+(((((((0xe5*x + 0xF7)&0xFF + ( 0x0 << 8)&0xFFFFFFFF)*0xFFFFFE26)+0x55)&0xFE)+(((0xe5*
x + 0xF7)&0xFF + ( 0x0 << 8)&0xFFFFFFFF)*0xED)+0xD6)&0xFF&0xFF + ( 0x0 << 8)&0xFFFFFFFF))*0xE587A503)
0xB717A54D)*0xAD17DB56)+0x60BA9824)&0xFFFFFF46)*0xA57C144B)+(((((((- (((((((0xe5*x + 0xF7)&0xFF + ( 0x0 <<
 8)&0xFFFFFFFF)*0xFFFFFE26)+0x55)&0xFE)+(((0xe5*x + 0xF7)&0xFF + ( 0x0 << 8)&0xFFFFFFFF)*0xED)+0xD6)&0xFF&
0xFF + ( 0x0 << 8)&0xFFFFFFFF)*0x2))+0xFF)&0xFE)+(((((((0xe5*x + 0xF7)&0xFF + ( 0x0 << 8)&0xFFFFFFFF)*
0xFFFFFE26)+0x55)&0xFE)+(((0xe5*x + 0xF7)&0xFF + ( 0x0 << 8)&0xFFFFFFFF)*0xED)+0xD6)&0xFF&0xFF + ( 0x0 <<
8)&0xFFFFFFFF))*0xE587A503)+0xB717A54D)*0xE09C02E7)+0xB5ED2776)((((((((((- (((((((0xe5*x + 0xF7)&0xFF +
( 0x0 << 8)&0xFFFFFFFF)*0xFFFFFE26)+0x55)&0xFE)+(((0xe5*x + 0xF7)&0xFF + ( 0x0 << 8)&0xFFFFFFFF)*0xED)+0xD6
)&0xFF&0xFF + ( 0x0 << 8)&0xFFFFFFFF)*0x2))+0xFF)&0xFE)+(((((((0xe5*x + 0xF7)&0xFF + ( 0x0 << 8)&0xFFFFFFFF
)*0xFFFFFE26)+0x55)&0xFE)+(((0xe5*x + 0xF7)&0xFF + ( 0x0 << 8)&0xFFFFFFFF)*0xED)+0xD6)&0xFF&0xFF + ( 0x0 <<
 8)&0xFFFFFFFF))*0xE587A503)+0xB717A54D)*0xAD17DB56)+0x60BA9824)&0xFFFFFF46)*0xA57C144B)+(((((((-
((((((((0xe5*x + 0xF7)&0xFF + ( 0x0 << 8)&0xFFFFFFFF)*0xFFFFFE26)+0x55)&0xFE)+(((0xe5*x + 0xF7)&0xFF +
( 0x0 << 8)&0xFFFFFFFF)*0xED)+0xD6)&0xFF&0xFF + ( 0x0 << 8)&0xFFFFFFFF)*0x2))+0xFF)&0xFE)+(((((((0xe5*x +
0xF7)&0xFF + ( 0x0 << 8)&0xFFFFFFFF)*0xFFFFFE26)+0x55)&0xFE)+(((0xe5*x + 0xF7)&0xFF + ( 0x0 << 8)
&0xFFFFFFFF)*0xED)+0xD6)&0xFF&0xFF + ( 0x0 << 8)&0xFFFFFFFF))*0xE587A503) ...
    return result;
}
```

Figure 14. Équation de transfert de la fonction à analyser

Le lecteur attentif notera que cette fonction, équivalente à un XOR avec 0x5C, pourrait être employée pour effectuer un HMAC. Dans cette DRM, c'est effectivement le cas. Il notera aussi le facteur de dégradation des performances entre une instruction XOR sur un octet et cette fonction.

Simplification La simplification n'a pas été automatique. Il a d'abord été nécessaire de trouver les dérivations appliquées à l'entrée et à la sortie. Il s'agit ici de fonctions affines, qu'il faut prendre en compte dans l'écriture de la fonction pour obtenir un comportement identifiable.

De manière générale, les simplifications suivantes peuvent être appliquées :

- identification des variables ;
- simplification et optimisation type compilateur, comme la propagation de constantes ;

Figure 15. Identification des variables puis de la fonction d'origine, ici un `XOR 0x5C`

- le changement d'ensemble. Dans l'exemple du XOR, il est possible de ne se rapporter qu'à des opérations sur des octets, et non plus des doubles words ;
- l'identification de sous-parties de la fonction. C'est cette dernière, qui à partir d'un ensemble de fonction simple (opération arithmétique / logique, affine) a permis de trouver la fonction XOR. L'identification peut-être faite grâce au calcul des images de points intéressants, ou à l'aide d'outil spécialisé tel que Z3.

7 Conclusion

Notre approche nous a permis d'analyser le code d'une DRM commerciale, à l'état de l'art de l'obscurcissement.

Il est toujours possible d'analyser et de comprendre un code, moyennant les ressources nécessaires. L'intérêt de ce que nous présentons à travers cet article est d'avoir des outils et des méthodes de plus dans la panoplie de l'analyste, pouvant être utilisés efficacement, même sur du code très renforcé. Nos travaux ont été validés, ainsi que les outils associés, dans d'autres cas d'utilisation, comme l'analyse de code malveillant ou la recherche de vulnérabilité.

L'obscurcissement est de plus un sujet d'actualité, et des solutions de plus en plus élaborées apparaissent. On pourra notamment penser à l'initiative publique d'LLVM-O [11], encore trop jeune pour être comparée avec des solutions telles que celles utilisées pour notre cas d'étude.

Même les appareils mobiles tels que les smartphones embarquent aujourd'hui suffisamment de ressources pour se permettre d'en gâcher une grande partie en *obfuscation*.

En résumé, notre approche n'est pas meilleure qu'une autre ; elle offre simplement un autre angle d'attaque, plus adapté à certaines situations.

8 Bonus

Un autre exemple de ce qu'il est possible de faire à partir d'une trace, mais différent de ce qui a été présenté dans cet article car basé sur la visualisation des données, est le graphe mémoire.

La Figure 16 présente une vue des accès mémoire à travers le temps, pour une fonction. On y voit clairement apparaître trois groupes d'adresses distinctes, correspondants aux zones mémoire du binaire et des rdatas, du tas ainsi que de la pile.

La fonction en question, une fois l'aplatissement annulé, est composée d'un très long basic block, comportant plusieurs milliers d'instructions.

Un agrandissement de la pile sur cette même zone met en évidence une structure de cycle dans les accès mémoire. Cet agrandissement est illustré sur la Figure 17, juxtaposé à son équivalent dans le tas.

Cette vue permet d'obtenir rapidement :

- le fait que des boucles aient été déroulées dans le code ;
- l'absence d'écriture dans les zones autres que la pile ;
- le nombre de boucles ;
- la délimitation des boucles, utile pour analyser seulement un tour de boucle ;
- les entrées et sorties, apparaissant à côté du reste.

Cette vue peut aussi être utile pour identifier un algorithme. En effet, l'enchaînement des accès mémoire agit un peu à la manière d'une signature. Elle ne sera pas affectée par l'utilisation de dérivation des données mémoire (Section 4.2) et sera seulement diluée / étendue dans le temps par de l'aplatissement de code.

11. https ://github.com/obfuscator-llvm/obfuscator/wiki

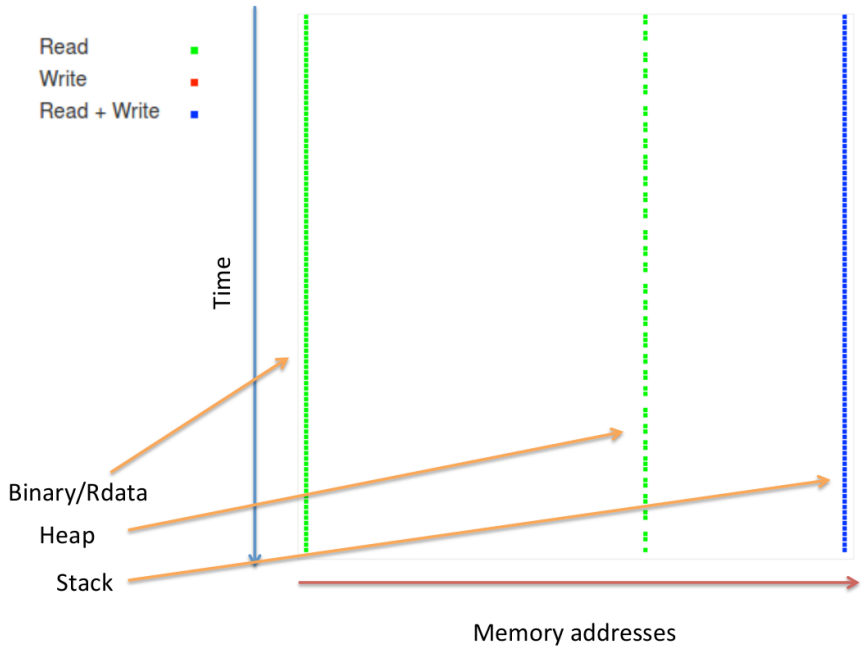


Figure 16. Représentation des accès mémoire en fonction du temps

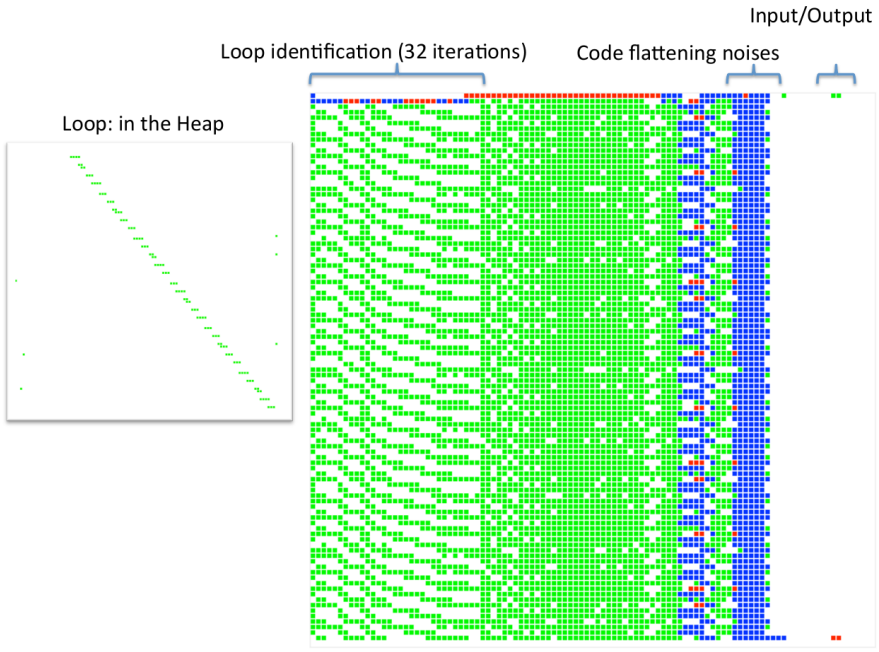


Figure 17. Agrandissement de la pile, détection de boucle

Remerciements

Nous tenons à remercier l'équipe de Quarkslab pour son aide et ses précieux conseils ainsi que l'équipe sécurité du CEA/DAM pour son soutien tout au long de cette étude.

Nous remercions tout spécialement F. Desclaux pour son travail et sa patience avec nos requêtes pour Miasm, ses conseils et son implication dans la rédaction de ce document.

Enfin, nous tenons aussi à remercier F. Raynal pour avoir rendu ce travail possible.

Références

1. Distorm. `https://code.google.com/p/distorm`.
2. Mongodb. `http://www.mongodb.org/`.
3. Peid. `http://www.aldeid.com/wiki/PEiD`.
4. Pymongo. `http://api.mongodb.org/python/current/`.
5. Pypy. `http://pypy.org/`.
6. F. Desclaux. Miasm. `https://code.google.com/p/miasm/`.
7. S. Josse G. Gueguen. Aplatissement de code. *MISC n°68 2013*.
8. J. A. Muir. A tutorial on white-box aes. 2013.
9. SysK. Practical cracking of white-box implementations. *Phrack 68*, 2012.
10. E. Vanderbeken. Mise à plat de graphes de flot de contrôle et exécution symbolique. *SSTIC 2013*.
11. Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
12. Yongxin Zhou, Alec Main, Yuan X. Gu, and Harold Johnson. Information hiding in software with mixed boolean-arithmetic transforms. In *Proceedings of the 8th International Conference on Information Security Applications*, WISA'07, pages 61–75, Berlin, Heidelberg, 2007. Springer-Verlag.