Comparaison fondée sur graphes des objets exécutables (version française)

Thomas Dullien[1] and Rolf Rolles[2]

[1] Ruhr-Universitaet Bochum
thomas.dullien@sabre-security.com
[2] University of Technology in Florida
rolf.rolles@sabre-security.com

Résumé Une méthode pour construire un isomorphisme optimal entre les ensembles des instructions, les ensembles des blocs basiques et les ensembles des fonctions de deux objets exécutables différents mais de façon similaire est présenté. Cet isomorphisme peut être utilisé pour la transcription des informations récupérées entre des codes désassemblés, des changements de récupération fait par les mises à jour de sécurité et des détecteurs de voleurs de codes.

Les applications les plus intéressantes dans le secteur de la sécurité sont les analyses de code malveillant qui sont capables de réduire l'analyse d'une famille de trojans ou de virus en analysant les différences entre les variantes et en réutilisant les détails des vulnérabilités corrigées quand le vendeur des patchs de sécurité refuse de dévoiler des informations.

Une charpente (*framework*) implantant la méthode décrite est présentée avec des dates empiriques sur la performance dans l'analyse des variantes multiples des mêmes malwares et dans la récupération des détails de vulnérabilité des mises à jour de sécurité.

1 Introduction

Alors que des programmes qui comparent des versions différentes de la même classe de code de source sont largement répandus depuis des années, l'accent a peu été mis sur l'importance de démasquer et d'analyser le changement entre deux versions du même objet exécutable.

Sans une routine automatique pour trouver les changements du code source et transcrire les résultats d'analyse entre les codes désassemblés des objets exécutables concernés, la partie chargée de l'analyse des changements est désavantagée : chaque tour d'ingénierie inverse nécessite une duplication massive du travail fait pendant les tours d'analyse précédents alors que peu de travail est nécessaire pour changer le code source et recompiler.

Aussi bien les auteurs de code malveillant type virus avec langage de haut niveau que les vendeurs de logiciels essaient de profiter de cette asymétrie : ils veulent gagner du temps, d'un côté pour permettre aux clients d'appliquer les patchs avant que les attaquants puissent créer des outils d'attaque automatiques, de l'autre pour permettre à leurs codes malveillants de se diffuser avant que les signatures de détection ne soient disponibles.

Cet article présente une nouvelle approche pour résoudre cette problématique : soient deux objets exécutables A et A', une application bijective entre les fonctions dans A et A' est construite par une amélioration itérative de l'isomorphisme d'un graphe partiel du graphe d'appels de l'objet exécutable. Une fois que l'application est générée, une application bijective entre les blocs basiques d'une paire de fonctions f, f' est construite en utilisant la même amélioration itérative de l'isomorphisme d'un graphe partiel du flow-graph des fonctions. Finalement, pour une paire de blocs basiques donnée, β, β' un isomorphisme des instructions est généré en utilisant la séquence des instructions comme graphe spécial et en procédant comme mentionné plus haut.

2 travail préliminaire

L'analyse et la classification automatique des changements des codes source ont été étudié bien souvent auparavant dans la littérature, et une énumération de tous les articles pertinents est hors du cadre de cette présentation. La plupart des recherches sont fondées sur la condition que le code source est une séquence de lignes, et sur l'application d'un algorithme de comparaison de séquences.

Le problème de déterminer des fonctions dans deux objets exécutables pour former des paires a été étudié dans [11][12], bien que concentrés sur la réutilisation des informations du *profiling* qui permettent la supposition de symboles disponibles pour les deux objets exécutables. D'autres études se sont concentrées sur la distribution efficace des patchs binaires. Les deux approches, même en trouvant des différences entre deux binaires, ne sont pas capables de traiter des optimisations agressives basées sur des informations de catégorisation. Dans le cas d'un changement des allocations de registres ou de l'ordre des instruction, beaucoup d'informations superflues sont générées. Une approche *bytecode-centric* pour déterminer les sections de code JAVA similaires est étudiée dans [9].

Une autre approche à la comparaison de binaires qui utilise aussi les isomorphismes des graphes est discutée dans [13]. En prenant les points d'entrée des objets exécutables comme point de départ, les blocs basiques sont déterminés un par un en fonction des instructions qu'ils contiennent. Quand une détermination est impossible, c'est qu'un changement a dû être effectué. Toutefois, à cause des comparaisons actuelles d'instructions, un nombre significatif d'emplacements est identifié de façon erronée comme changé : l'article mentionne qu'environ 3-5 % de toutes les instructions change entre deux versions du même objet exécutable. En outre, l'algorithme présenté réagit très faiblement lorsque le graphe d'appels a une connectivité basse ou lorsque des changements importants sont présents dans l'ordre des instructions. Ce travail est une extension directe des méthodes et codes présentés dans [10].

3 Analyses structurelle de code

En comparant des variantes différentes du même objet exécutable (ou juste deux objets exécutables arbitraires qui partagent une quantité suffisante de

code), il faut résoudre le problème des codes source similaires construits à des représentations peut-être complètement différentes au niveau assembleur. Un certain nombre de changements communs qui se produisent entre deux variantes d'un objet exécutable sont :

1. *Allocation de registre différent* ;
   dépendant des paramètres d'optimisation du compilateur et des changements du code, des registres différents seront assignés à des instructions identiques.
2. *Réorganisation d'instructions* :
   dépendant du modelage du compilateur pour le *pipeling* du CPU, des instructions individuelles seront réorganisées.
3. *Inversion de branche* :
   dans plusieurs cas, le compilateur essaie d'optimiser l'alignement des blocs basiques par une inversion de la condition d'une branche et par un échange des deux blocs basiques auxquels cette branche peut mener.

Évidement un nombre important de changements majeurs peuvent se produire. L'observation principale sur laquelle sont fondées les méthodes présentées dans cet article, est que le graphe d'appels d'un objet exécutable reste en majeure partie identique, même compilé avec un compilateur différent et pour une architecture différente.

Au lieu de se concentrer aux instructions du niveau assembleur obtenues via le code désassemblé, l'approche présentée se concentre sur les propriétés structurelles de l'objet exécutable, et surtout sur l'abstraction essentielle des fonctions et des blocs basiques, ainsi que leurs relations entre eux.

3.1 Notations

Dans cet article sont utilisés certains termes de la théorie des graphes, expliquées ci-après.

La notation $\mathfrak{P}(S)$ symbolise l'ensemble des parties d'un ensemble S donné.

Chaque fois que le mot graphe est utilisé, il réfère à un graphe probablement d'orientation cyclique qui se compose d'un ensemble de sommets et d'un ensemble d'arcs. Une simple lettre majuscule est utilisée pour décrire un graphe, et les index à la lettre sont utilisés si l'ensemble de sommets ou d'arcs y est relié.

Par conséquent le graphe G consiste en l'ensemble de sommets $G^n := \{G_1^n, \ldots, G_m^n\}$ et en l'ensemble d'arcs $G^e := \{G_1^e, \ldots, G_k^e | G_i^e \in G^n \times G^n\}$.

Pour l'utilisation suivante, nous définissons les fonctions

$$\begin{aligned} \text{up} &: G^n \to \mathfrak{P}(G^n) \\ \text{down} &: G^n \to \mathfrak{P}(G^n) \end{aligned}$$

qui associe un sommet donné G_i^n respectivement au sous-ensemble de G^n contenant ses parents directs (antécédents) et au sous-ensemble de G^n des enfants directs de G_i^n.

3.2 Un objet exécutable comme graphe d'un graphe

Nous traitons un objet exécutable comme graphe d'un graphe. Cela veut dire qu'un objet exécutable est vu comme un graphe orienté multi arcs A qui possède toutes les fonctions des codes désassemblés comme sommets et les relations d'appels entre ces fonctions comme arcs.

Chaque sommet $A_i^n \in A^n$ est lui-même un graphe, dont les sommets sont des blocs basiques individuels dans les codes désassemblés et les arcs représentent leurs relations de branches. De tels graphes sont habituellement appelés *control flow graphs* ou tout court *cfg*.

Chaque bloc basique en lui-même (cela veut dire chaque sommet dans le graphe représenté par A_i^n) est aussi un graphe, bien que d'une forme très simple : une séquence d'instructions de niveau assemblé.

3.3 Dépister les informations

Pour déterminer les graphes d'un objet exécutable, un bon code désassemblé du code binaire est nécessaire. Le standard industriel pour les codes désassemblés est [7], surtout grâce à ses excellentes capacités multi architecturelles en combinaison avec une interface de programmation qui permet de dénicher les informations nécessaires sans avoir connaissance du CPU concerné ou du code assemblé. Cela facilite l'implémentation d'algorithmes décrits une fois seulement tout en permettant de l'appliquer à des objets exécutables venant d'architectures différentes.

Demandes indirectes et problèmes de code désassemblé Dans plusieurs cas, la création d'un graphe d'appels complet (qui représente toutes les relations possibles entre les différentes fonctions) à base d'un code binaire n'est pas triviale. Spécialement des appels sub-fonctionnelles par des tableaux (très commun par exemple dans le code C++ qui utilise des méthodes virtuelles) sont dures à résoudre statiquement.

Dans l'approche présentée, ces demandes indirectes (non résolues statiquement) sont simplement ignorées et traitées comme une instruction ordinaire au niveau assembleur. Le risque est d'avoir des séquences non connectées du graphe d'appels dans lesquelles aucun point fixe n'est généré et qui mène à la sous-séquence du graphe qui n'est pas reconnue exactement. Grâce au grand nombre de propriétés différentes utilisables pour générer des points fixes (voir section 4), cela ne pose en réalité pas de problème en pratique.

4 Recherche de convergence structurale

L'idée principale de l'approche présentée est la suivante. Soient deux objets exécutables donnés, et les graphes A et B correspondants. Il s'agit maintenant de déterminer les points fixes. Pour cela, on utilise deux éléments (respectivement de A^n et B^n qui représentent le même élément dans les deux objets exécutables.

Ces points fixes sont utilisés pour générés d'autres points fixes itérativement jusqu'à ce que la convergence ne puisse être améliorée. Une fois le maximum de fonctions atteint, nous pouvons déterminer les blocs basiques identiques de la même manière. Comme nous avons déjà un isomorphisme qui nous permet de déterminer que deux fonctions sont reliées, il nous reste à déterminer les sommets de deux *cfg*s pour identifier les points fixes et à propager cette information pour déterminer de plus en plus de sommets.

Une fois que nous avons atteint le niveau des blocs basique, nous traitons deux blocs basiques convergents de nouveau comme un graphe (de forme très simple), et générons un isomorphisme de la même manière.

4.1 Sélecteurs

Un *selector* (sélecteur) n'est essentiellement qu'une classification qui, un sommet $A_i^n \in A^n$ d'un graphe et un ensemble de sommets dans un autre graphe donnés, reprend soit un élément de l'ensemble donné ou de l'ensemble vide, p.e.

$$s : A^n \times \mathfrak{P}(B^n) \rightarrow B^n \cup \{\emptyset\}$$

Le rôle du sélecteur est de sélectionner un seul sommet de l'ensemble de sommets donné, le plus similaire avec A_i^n, ou si plus d'un candidat avec la même similarité existe, de ne rien sélectionner du tout.

Il est intuitivement clair que la probabilité qu'un sélecteur choisisse un ensemble vide augmente proportionnellement à la taille de l'ensemble utilisé.

4.2 Propriétés

Une *propriété* π définit une classification des sommets de deux graphes A et B :

$$\pi(A, B) \rightarrow (A'^n, B'^n) \text{ avec } A'^n \subset A^n \text{ et } B'^n \subset B^n$$

L'objectif est la réduction de la taille des ensembles utilisés par le sélecteur pour augmenter la probabilité du sélecteur de choisir un résultat non vide.

4.3 Isomorphisme de graphe via points fixes et propagation

Génération de points fixes Pour un sélecteur s donné, on peut générer un isomorphisme de graphe approximatif $p : A^n \rightarrow B^n$ à partir d'un isomorphisme initial p_1. Puis, en utilisant celui-ci, on itère pour en générer des versions améliorées.

L'isomorphisme initial $p_1 : A^n \rightarrow B^n$ est simplement généré par la définition $p_1(x) \rightarrow s(x, B^n)$.

Cette construction simple peut être améliorée significativement si un certain nombre de propriétés est disponible. Soit $\Pi = \{\pi_1, \ldots, \pi_j\}$ un ensemble de propriétés. Un isomorphisme initial amélioré peut être généré de la façon suivante :

for $\pi \in \Pi$ **do**
  $(K, L) \leftarrow \pi(A, B)$;
  for $x \in K$ **do**
    define $p_1(x) \rightarrow s(x, L)$
  end
end

Propagation des points fixes La classification initiale p_1, donnée, les isomorphismes améliorés suivants p_i peuvent maintenant être générés itérativement :

Input : p_{n-1}, s, A, B
Result : p_n
$S \leftarrow \{x \in A^n | p_{n-1}(x) \neq \emptyset\}$;
for $x \in S$ S **do**
  P $\leftarrow$ `up` (x);
  K $\leftarrow$ `up` ($p_{n-1}(x)$);
  for $y \in P$ **do**
    if $s(y, K) \neq \emptyset$ **then**
      define $p_n(y) \rightarrow s(y, K)$
    end
  end
end

L'algorithme mentionné ci-dessus détermine des sommets pour lesquels p_{n-1} dispose d'une classification utile, puis examine seulement les ensembles de sommets qui sont des parents directs d'un sommet et leur image sous p_{n-1}. Vu que ces ensembles sont significativement moins complexes que les ensembles examinés avant, les chances pour s de déterminer un résultat non vide augmentent. L'algorithme mentionné ci-dessus peut clairement fonctionner avec *down* au lieu de *up*, et les meilleurs résultats sont atteints en alternant entre les deux.

4.4 Multiplication de petits nombres premiers (SSP, small primes product)

Un des changements le plus commun entre deux blocs basiques dans deux objets exécutables est le changement de l'ordre des instructions. Un algorithme pour rapidement déterminer si deux blocs basiques (ou même deux fonctions) ont les mêmes instructions (mais probablement dans un ordre différent) est d'une importance énorme : il peut être utilisé directement pour générer des points fixes initiaux supplémentaires.

Le problème Pour expliquer le problème plus précisément :

Soit $\mathcal{A} := \{\alpha_1, \ldots, \alpha_m\}$ un alphabet avec m éléments différents, et soit $\mathbb{S}_n$ le groupe de permutations à n éléments. Soient deux mots de longueur n, notés $\mathfrak{a}, \mathfrak{b} \in \mathcal{A}^n$, une permutation $\sigma \in \mathbb{S}_n$ existe si $\sigma(\mathfrak{a}) = \mathfrak{b}$. Nous notons $\mathfrak{a}_k$ la k-ième lettre d'un mot $\mathfrak{a}$. .

Une première solution Soit $P_m := \{3, \ldots, \rho_m\}$ l'ensemble des m premiers nombres premiers possibles. En plus considérons la classification :

$$\tau : \mathcal{A} \to P_m, \quad \tau(\alpha_i) = \rho_i$$

qui associe un petit nombre premier unique à chaque élément dans l'alphabet. Nous pouvons calculer le produit de toutes les lettres dans $\mathfrak{a}, \mathfrak{b}$ et vérifier que

$$\prod_{i=1}^{n} \tau(\mathfrak{a}_i) = \prod_{i=1}^{n} \tau(\mathfrak{b}_i)$$

La condition ci-dessus est équivalente à l'existence d'un σ avec $\sigma(\mathfrak{a}) = \mathfrak{b}$ à cause du caractère unique de la décomposition de nombres premiers et de la commutativité de la multiplication.

Ajuster à la réalité : arithmétique en mod 2^{64} Malheureusement, l'arithmétique des grands nombres est très complexe, et la méthode décrite ci-dessus n'est pas vraiment applicable dans des applications réelles. Si nous *limitons* les calculs ci-dessus à mod 2^{64}, nous pouvons utiliser la multiplication en registre normale de notre processeur x86. Cela évite les calculs coûteux d'arithmétique des grands nombres, tout en risquant de prétendre de façon erronée que un σ avec $\sigma(\mathfrak{a}) = \mathfrak{b}$ existe.

Quantifier le risque exact est très difficile parce que cela dépend des probabilités de l'accomplissement d'un certain α. Nous pouvons tout de même calculer une limite supérieure pour le risque d'une fausse correspondance sous l'hypothèse que tous les $\alpha \in \mathcal{A}$ possèdent une probabilité identique.

Pour chaque mot $\mathfrak{c}$ donné, l'inéquation suivante tient :

$$\prod_{i=1}^{n} \tau(\mathfrak{c}_i) \leq p_m^n \tag{1}$$

L'algorithme proposé prétendra de façon erronée que σ avec $\sigma(\mathfrak{a}) = \mathfrak{b}$ existe si et seulement si

$$\prod_{i=1}^{n} \tau(\mathfrak{a}_i) = k2^{64} + c \tag{2}$$

$$\prod_{i=1}^{n} \tau(\mathfrak{b}_i) = j2^{64} + c \tag{3}$$

avec $k \neq j, c < 2^{64}, \mathfrak{a} \neq \mathfrak{b}$. Nous supposons sans perte de généralité que $k > j$. Vu les hypothèses ci-dessus, il devient évident qu'il y a un maximum de $k - 1$ qui vaut de $\prod_{i=1}^{n} \tau(\mathfrak{b})$ ce qui est une équation suffisante (3). De (1), on déduit que :

$$k \leq \frac{p_m^n}{2^{64}}$$

Le nombre total l de mots $\mathfrak{c}_{(1)}, \ldots, \mathfrak{c}_{(l)} \in \mathcal{A}^n$ pour lesquels on a

$$\prod_{i=1}^{n} \tau(\mathfrak{c}_{(1),1}) \neq \cdots \neq \prod_{i=1}^{n} \tau(\mathfrak{c}_{(l),1})$$

est donné par $l = \binom{n+m-1}{n}$ comme nous pouvons maximiser le produit ci-dessus par une simple combinaison avec répétions.

Nous pouvons alors prétendre que la chance ρde deux mots $\mathfrak{a} \neq \mathfrak{b}$ choisi par hasard qui remplissent

$$\prod_{i=1}^{n} \tau(\mathfrak{a}_i) \equiv \prod_{i=1}^{n} \tau(\mathfrak{b}_i) \qquad \mod 2^{64}$$

est plus petite que ou égale à

$$(\frac{p_m^n}{2^{64}} - 1) \binom{n+m-1}{n}^{-1} = (\frac{p_m^n}{2^{64}} - 1) \frac{(m-1)!n!}{(n+m-1)!}$$

Il faut garder en mémoire qu'il s'agit d'une limite supérieure rudimentaire qui pourrait être améliorée significativement. Une telle amélioration est hors du cadre de ce papier.

La conclusion importante à faire est qu'utiliser la méthode proposée pour un alphabet de 100 éléments est définitivement sûre pour des mots plus courts que 14 lettres, et très probable pour une extensibilité au-delà

SPP et similarité de code Notre amélioration utilise des SPP pour identifier des séquences d'instructions avec des mnemonics similaires. Nous utilisons l'index du code désassemblé assigné pour chaque mnemonic pour le cataloguer dans un tableau avec des petits nombres premiers, et pour calculer le résultat comme *unsigned long long*. Cela est fait pour les deux, la fonction et le niveau blocs basique.

4.5 Exemples pour sélecteurs et propriétés

Propriété générale On peut imaginer différentes propriétés. Dans notre exemple, nous avons utilisé certaines propriétés différentes pour les graphes A, B avec des conséquences acceptables, et les meilleurs résultats en combinant tous mentionnés ci-dessus.

Toutes les classifications sont de la forme

$$\pi_1 : (A^n, B^n) \rightarrow (\{A_i^n, \ldots, A_k^n\}, \{B_j^n, \ldots, B_l^n\})$$

avec certains critères que les A_i^n, B_i^n doivent satisfaire. Nous ne mentionnerons ici que quelques critères, et leur signification.

1. *Sommets de k-in-degré / Sommets de k-out-degré*

$$\sharp(\mathrm{up}(A_i^n)) = k \text{ and } \sharp(\mathrm{up}(B_i^n)) = k$$

   Cela veut dire que nous choisissons des sommets pour quels l'in-degré est exactement k. En replaçant *up* par *down*, on détermine tous les sommets avec out-degré d'exactement k. Remarquons qu'en sélectionnant un k de zéro, tous les sommets d'origine (ou alternativement de feuille) seront déterminés.
2. *Sommets récursifs*

$$A_i^n \in \mathrm{up}(A_i^n) \text{ and } B_i^n \in \mathrm{up}(B_i^n) = 0$$

   Des sommets qui renvoient à eux-mêmes sont sélectionnés, et qui sélectionnent que des fonctions qui s'appellent récursivement eux-mêmes.

Propriétés spécifiques à des graphes d'appels La plupart des propriétés ne sont spécifiées pour des graphes abstraits mais utilisent le code assembleur concerné pour spécifier des propriétés comme les suivantes :

1. *Nom identique*
   Clairement la propriété plus évidente : Beaucoup de sommets dans les graphes d'appels d'une application ont des noms, soit de l'information de debug qui est disponible ou à cause de l'information d'import/export dans l'objet exécutable.
2. *Références de chaîne de caractères identique*
   Des sommets dans un graphes d'appels peuvent être sélectionnés par des chaînes de caractères communes, indiquant des fonctions qui toutes contiennent une référence au même chaîne de caractère.
3. *SPP identique*
   Des sommets dans des graphes d'appels peuvent être sélectionnés par des SPP communs.

Propriétés spécifiques au CFG's Les deux propriétés, référence de chaîne de caractères identique et SPP identique, peuvent être appliquées directement au CFG's. Pour compléter, la propriété suivante s'est montrée utile :

1. *Appel de sous-fonctions identiques*
   Des sommets dans les CFG peuvent contenir des appels de sous-fonction.
   Au moment où des isomorphismes de CFG sont calculés, un bon isomorphisme p_c pour le graphe d'appels est déjà disponible. Pour y réussir, un graphe d'appels sélectionne des sommets qui appellent des sous-fonctions qui sont identiques dans p_c.

Propriétés spécifiques pour le niveau d'instruction Les propriétés des instructions identiques du graphe de niveau d'instruction (qui n'est essentiellement qu'une séquence), de la référence de chaîne de caractère identique et de demandes de sous-fonction identiques sont utilisées.

Un sélecteur pour le graphe d'appels Nous associons un triplet avec chaque sommet du graphe d'appels. Ce triplet consiste en nombre de blocs basiques dans la fonction, le nombre d'arcs pour les raccorder sous forme de CFG, et le nombre de demandes de sous-fonctions se trouvant dans les blocs basiques.

Le sélecteur pour les sommets des graphes d'appels fonctionne facilement de la façon suivante : le triplet est interprété sous forme de simples vecteurs dans l'espace euclidien, et la distance euclidienne entre le touple de chaque élément des ensembles mis à disposition et le touple des éléments mises à disposition est calculé. Quand un seul touple avec une distance minimale est trouvé, le sélecteur détermine le sommet associé.

Plus formellement :

$$s_c(x, A) := \begin{cases} a \text{ si } \exists a \in A, \forall b \in A, b \neq a, |x - a| < |x - b| \\ \emptyset \text{ sinon} \end{cases}$$

Un sélecteur pour le CFG Dans le cas des *cfg*'s nous travaillons de nouveau avec des triplets de nombres naturels. La construction de ce sélecteur est basée sur l'observation que pour des changements mineurs dans la fonction, les changements dans le *cfg* sont souvent localisés dans une région du graphe. Cela implémente que pour un bloc basique a, donné, le changement va être soit en dessous ou soit au-dessus de ce bloc. Cela implémente que soit le nombre de blocs basique sur le chemin le plus court vers a ou le nombre de blocs basiques à partir de a vers la fin de la fonction reste constant.

La deuxième observation était que la plupart des fonctions incluent une quantité significative de routines de recherche d'erreurs qui est représentée dans les cfg's par un chemin qui dépasse la plupart des fonctions et saute directement au sommet de sortie.

Nous associons alors un triplet avec chaque sommet dans le *cfg*. Ce triplet contient le nombre de blocs sur le chemin le plus court pour sortir de la fonction, du nombre de blocs sur le chemins le plus court du point d'entrée de fonction vers sa sortie, et du nombre de demandes de sous-fonction fait dans ce bloc basique.

Le désavantage de cette approche est qu'en insérant un bloc basique dans un cfg, la signature de tous les blocs qui sont dominés par le cfg peut être incurvée.

Pour résoudre ce problème, un sélecteur spécial est utilisé : il faut un paramètre δ spécial. La définition est plus ou moins la même que dans une situation d'un *calltree* :

$$s_c(x, A, \delta) :=$$

$$\begin{cases} a \text{ si } \exists a \in A, \forall b \in A, b \neq a, |x - (a + \delta)| < |x - (b + \delta)| \\ \emptyset \text{ else} \end{cases}$$

Pendant la propagation des points fixes comme elle est décrit dans 4.3.2, le paramètre δ est calculé à partir de la différence entre les deux signatures dans le point fixe.

Un sélecteur pour le niveau des instructions Pour générer l'isomorphisme du niveau d'instructions, nous prenons essentiellement la distance à l'entrée et la distance jusqu'à la sortie du bloc basique comme signature et appliquons le même algorithme décrit ci-dessus.

5 Les applications

La capacité de générer un isomorphisme jusqu'au niveau d'instructions nous offre beaucoup d'applications intéressantes.

5.1 Transcrire des commentaires pour les analyses des variantes de malware

Pour des raisons de démonstration, nous traitons deux exemplaires du trojan Bagle, en l'occurrence Bagle.X et Bagle.W. Une analyse précise de l'exemplaire Bagle.W était réalisée, avec un code désassemblé détaillé dans lequel toutes les fonctions étaient proprement nommées et la plupart de la base de données était commenté.

Nous avons ensuite généré un code désassemblé vierge du Bagle.X : aucun nom de fonction majeur n'était présent, et le code désassemblé était complètement non commenté.

Après avoir fait fonctionner notre amélioration d'algorithme décrite sur les deux codes désassemblés, toutes les fonctions exceptée 6 dans le code désassemblé vierge étaient associées avec succès avec leurs pendants dans le code désassemblé déjà analysé. En plus, seulement 3 fonctions ont changées de manière significative.

Des 1524 commentaires dans les fonctions déterminées, toutes exceptée 10 étaient transférées avec succès entre les codes désassemblés.

En tout, le temps d'analyser la variante Bagle.X était réduit à l'examen des trois fonctions changées et des six (très petites) fonctions non déterminées. Presque tous les noms de fonction et commentaires qui ont été générés pour la base de données d'avant pouvaient être réutilisés. La réalisation de notre analyse durait moins de 30 secondes.

5.2 Récupérer l'information de vulnérabilité

H323ASN1.DLL Après que le NISCC a publié des informations sur de vulnérabilités multiples dans des parsers H.323 , la question se posait : où se trouvait la faute en question dans le produit Microsoft ISA Server. Microsoft a refusé de publier des informations détaillées sur la vulnérabilité qu'ils ont corrigé. Selon le rapport NISCC, le problème était localisé dans le décodage de ASN.1.

Les deux versions de H323ASN1.DLL, celle avant et celle après l'implémentation du patch, ont été analysées, et un total de 8 fonctions changées (sur 1655) étaient détectées.

Le changement pouvait être classé dans deux catégories :

1. Introduisant des analyses de sens pour des valeurs sans preuve de confiance qui spécifient des mots à décoder d'un flux ASN.1
2. Introduisant des analyses de sens à priorité pour les demandes à ASN1PERDecZeroTableCharStringNoAlloc()

Dans le deuxième cas, un entier de 32-bit d'un flux ASN.1 est passé à un ASN1PERDecZeroTableCharStringNoAlloc() comme deuxième argument. La variante traitée du patch introduit une analyse de la valeur pour assurer que cet argument est plus petit que 129.

Une inspection plus précise d'un ASN1PERDecZeroTableCharStringNoAlloc() révèle que la fonction calcule la largeur de l'allocation de mémoire pour la valeur par avant sans preuve : un attaquant était capable de fixer cette valeur se sorte que le calcul donnait un résultat MAXUNIT et serait alors d'une largeur très petite. L'opération de copie de sub-séquences corrompait alors l'arbre binaire, permettant à un attaquant de gagner le contrôle dans le prochain tour de consolidation de l'arbre binaire. Au lieu de fixer le problème dans le fond (p.e. dans la librairie MSASN1.DLL), un test sur la valeur est ajoutée dans l'application appelée (H323ASN1.DLL).

Cette analyse montre à par conséquent que chaque demande à ASN1PERDecZeroTableCharStringNoAlloc() a besoin de faire un test sur la valeur de la taille avant que la demande soit passée.

Un court scan du système entier a conduite à voir si d'autres applications à part le serveur ISA utilise la fonction concernée d'une manière dangereuse. Deux autres instances ont été découvertes : la librairie propre à Windows du H.323 Multimédia Provider (qui permet aux applications d'arbitrage de facilement traiter des données H.323) et le Vidéo Conferencing Software Netmeeting de Microsoft. Aucune ne fait un test pertinent sur les fonctions en question.

Le résultat était que la mise à jour à H323ASN1.DLL fixait une erreur mais alarmait tout le monde avec la capacité d'analyser des patchs, et que deux autres vulnérabilités non exploitées se présentaient, sans être corrigées.

Microsoft a été contacté et le problème fixé quelques mois plus tard, dans le MS04-11.

L'analyse totale a duré moins que 3 heures, et la durée réelle pour appliquer l'algorithme était de moins de 5 minutes.

SSL/PCT Parser En avril 2004, Microsoft publiait une mise à jour pour SCHANNEL.DLL, la librairie responsable pour l'utilisation de la communication SSL. Selon leur bulletin de sécurité, ils corrigeaient un problème de sécurité qui permettait à des attaquants de prendre le contrôle complet d'un ordinateur utilisant un serveur SSL. Aucun détail technique n'était publié, à part que le problème se trouvait dans une partie de la librairie responsable pour parser les paquets PCT.

Seules 20 fonctions différentes étaient détectées au total, mais seulement une avec un nom qui prétendait qu'elle était concernée par le PCT parsing. Un examen de la fonction Pct1SrvHandleUniHello() démontrait que l'ancienne version prenait une chaîne de données, notait chaque caractère et l'ajoutait à la chaîne

de données originale. La nouvelle version était changée de telle manière qu'elle assurait que la chaîne de données combinée ne dépasse pas 32 caractères.

Détecter et comprendre la vulnérabilité (un vulgaire stack-smash avec réécriture EIP) a pris moins de 30 minutes. Ensuite, le code était généré pour atteindre l'emplacement approprié dans le binaire. Durant 5 minutes, EIP pouvait être modifié par une valeur arbitraire et 10 heures après le début de l'analyse, un programme était créé qui exploitait dignement la vulnérabilité.

6 Conclusion

Il est démontré que la non-divulgation d'information de vulnérabilité n'est pas une dissuasion suffisante pour les attaquants potentiels et que les mises à jour de sécurité peuvent être utilisées en relativement peu de temps (à l'aide des bons outils). Il est aussi démontré qu'une prudence énorme doit régner en publiant des mises à jour de sécurité et que les informations dans le patch doivent être révélées au public. Une résolution de bug incomplète peut causer plus de mal que de bien en dévoilant l'existence d'autres bugs (non fixés) avec le patch.

Le travail présenté évoque aussi que la pratique commune d'attendre une ou deux semaines après la publication d'un patch avant de l'installer est très dangereuse.

Il est également démontré que des analyses de codes binaires fondées sur des propriétés structurelles du code est un domaine prometteur de recherche car il permet d'analyser des codes exécutables sans le besoin d'abstraire dans un langage intermédiaire ou des outils d'analyse spécifiques.

Références

1. M. Goossens, F. Mittelbach and A. Samarin, *Der LaTeX-Begleiter*, Addison-Wesley, 1995.
2. A. V. Aho, R. Sethi and J. D. Ullmann, *Compilerbau*, 2nd Edition, Oldenburg Verlag, 1999.
3. Z. Wang, K. Pierce and S. McFarling, *BMAT - A Binary Matching Tool for Stale Profile Propagation*, The Journal of Instruction-Level Parallelism (JILP), Vol. 2, May 2000.
4. Z. Wang, K. Pierce and S. McFarling, *BMAT - A Binary Matching Tool*, 2nd ACM Workshop on Feedback-Directed Optimization, November 1999.
5. Z. Wang Division, *BMAT – A Binary Matching Tool* `http://citeseer.nj.nec.com/262615.html`
6. Pocket Soft Inc., *RTPatch – Software Update Tool*, `http://www.pocketsoft.com/whitepapers/whitepaper.html`
7. DataRescue, *IDA Pro Disassembler*, `http://www.datarescue.com/idabase`
8. B. S. Baker, U. Manber and R. Muth, *Compressing Differences of Executable Code*, ACMSIGPLAN Workshop on Compiler Support for System Software (WCSS), pp. 1–10, 1999, `citeseer.nj.nec.com/baker99compressing.html`

9. B. S. Baker and U. Manber, *Deducing Similarities in Java Sources from Bytecodes*, Proc. of Usenix Annual Technical Conf., pp. 179–190, 1998, `citeseer.nj.nec.com/baker98deducing.html`
10. Halvar Flake, *Structural Comparison of Executable Objects*, DIMVA, pp. 161–173, 2004.
11. D. S. Hirschberg, *Algoritms for the Longest Common Subsequence Problem*, Journ. of the ACM, Vol. 24 Nr 4, pp. 664–675, ACM Press, 1977.
12. J. W. Hunt and T. G. Szymanski, *A fast algorithm for computing longest common susequences*, Commun. ACM, Vol. 20 Nr 5, pp. 350–353, 1977, ACM Press.
13. T. Sabin, *Comparing binaries with graph isomorphisms*, `http://razor.bindview.com/publish/papers/comparing-binaries.html`