Analyse de Programmes par Traçage

Wadie Guizani, Jean-Yves Marion, Daniel Reynaud

Analyse de Programmes par Traçage

Wadie Guizani, Jean-Yves Marion, Daniel Reynaud
Université de Nancy - LORIA
Campus Scientifique - BP 239
54506 Vandoeuvre-lès-Nancy Cedex
{guizaniw|marionjy|reynaudd}@loria.fr

March 24, 2010

Abstract

L'analyse de programmes sans code source, regroupant la recherche de vulnérabilités et l'analyse de logiciels malveillants, relève actuellement plus de la programmation système que de l'algorithmique. Par conséquent, ces champs de recherche ne bénéficient pas des avancées en analyse de code et en optimisation.

Nous proposons un cadre formel basé sur l'analyse de traces permettant de raisonner sur les propriétés algorithmiques des programmes binaires. Nous avons appliqué ce cadre à l'analyse des programmes auto-modifiants comme par exemple les malwares compressés ou chiffrés, et nous avons validé son utilité par l'expérience en testant une implémentation sur un grand nombre d'échantillons collectés sur des honeypots.

Les traces présentent un intérêt évident pour l'ingénierie inverse : faciles à collecter de manière générique avec différents outils (même statiques) et plus fiables que le désassemblage, elles permettent de raisonner avec exactitude sur les effets d'un programme sur le système, les flux de données ou encore les multiples chemins d'un exécutable. Elles sont donc une manière simple et élégante d'unifier les différentes branches de l'analyse de programmes.

Introduction

Les programmes auto-modifiants sont particulièrement intéressants du fait de la nature fondamentale du concept d'auto-référence et de ses conséquences sur la calculabilité. En effet, les auto-modifications sont gênantes pour l'analyse de programmes car le listing du programme dépend alors du temps et des entrées. On peut également remarquer que tout programme peut facilement être transformé en programme auto-modifiant à l'aide d'un packer. Il en découle que les packers sont rencontrés très fréquemment durant l'analyse de malware: le packing est simple à appliquer, il rend l'analyse statique inopérante et

il change la signature du binaire. L'utilisation de packers est suspecte mais pas malicieuse par nature, car ils peuvent être utilisés pour des raisons légitimes comme la compression de code. De plus, l'analyse statique ne peut révéler la présence que de packers connus [14] et ne peut généralement pas révéler le code packé.

Dans ce papier, nous exploitons l'analyse de traces pour détecter automatiquement les packers inconnus, et nous proposons une méthode pour détecter les techniques de protection dynamiques comme le déchiffrement de code, la vérification d'intégrité et les techniques anti-virtualisation. Nous avons ensuite validé cette technique en la testant sur un grand nombre de malwares dans un environnement distribué.

Contributions

Nous apportons les contributions suivantes :

- un modèle théorique des programmes auto-modifiants (Section 1)
- une taxonomie des comportements auto-référents
- une définition claire de ce qu'est une couche de code
- un prototype basé sur l'instrumentation dynamique de code
- une architecture distribuée pour l'analyse de code dynamique basé sur un cluster de machines virtuelles (Section 2)
- le résultat d'une expérience à grande échelle sur les malwares capturés par un honeypot (Section 3)

Cas d'utilisation

Le modèle que nous utilisons pour les programmes auto-modifiants est très générique et n'est pas spécifique aux malwares. Par conséquent, il peut être utilisé dans des scénarios très différents. Le prototype que nous avons implémenté peut être vu comme un système de classification : il attribue un score à des binaires inconnus basé sur leur utilisation de techniques d'auto-modifications et de blindage de code.

1 Analyse par TraceSurfer

TraceSurfer est notre prototype pour l'analyse de logiciel malveillant, basé sur l'instrumentation dynamique de binaire (en particulier Pin [19]). Il est capable de reconstruire les couches des programmes auto-modifiants et de détecter des mécanismes de protection comme des relations entre ces couches.

Dans un premier temps, nous allons passer en revue la littérature du domaine (section 1.1), introduire le typage de la mémoire (section 1.2), les couches de code (section 1.3) et finalement les mécanismes de protection (section 1.4).

1.1 Etat de l'art

Nos travaux sur les couches de code peuvent être vus comme une généralisation de la technique bien connue pour l'unpacking automatique [4, 15]. Le principe de cette méthode est de noter chaque écriture en mémoire lors de l'exécution (ou de l'émulation) de la cible, et de vérifier à chaque étape si le pointeur d'instruction fait partie de l'ensemble des adresses écrites. Si c'est le cas, cela signifie que l'instruction pointée a été générée dynamiquement. L'unpacking peut alors se poursuivre en déchargeant une copie de la mémoire (dump), et en reconstruisant un exécutable valide à partir de cette copie.

De nombreuses implémentations sont basées sur ce modèle [13, 9], en particulier Renovo [17], VxStripper [16], Saffron [21], Azure [22], et des implémentations basées sur Bochs [4, 8].

Nous essayons d'étendre cette technique, sans toutefois avoir pour but de faire de l'unpacking (ce qui évite les difficultés liées au dump de la mémoire et à la reconstruction de l'exécutable [11]). Nous raffinons le procédé pour trouver du code dynamique en hiérarchisant la mémoire en niveaux et en notant également les lectures en mémoire. Cela nous permet ensuite de définir les mécanismes de protection (par exemple le déchiffrement ou la vérification d'intégrité) comme des motifs sur les accès mémoire.

1.2 Typage de la mémoire

Nous considérons la trace d'un programme comme étant une suite arbitrairement grande d'instructions i_1, ..., i_x, ..., i_{max}. Si nous connaissons précisément les effets de chaque instruction sur la mémoire, alors nous connaissons l'état de la mémoire à chaque étape.

Nous associons à chaque adresse mémoire m à l'étape x (c'est-à-dire après l'exécution de l'instruction i_x) un niveau d'exécution $Exec(m, x)$, de lecture $Read(m, x)$ et d'écriture $Write(m, x)$.

Initialement, pour tout m, nous avons $Exec(m, 0) = Read(m, 0) = Write(m, 0) = 0$. Nous mettons ensuite ces niveaux à jour après l'exécution de chaque instruction en fonction de ses effets sur la mémoire.

A l'étape $x + 1$, pour mettre à jour $Exec(\_, x + 1)$ (resp. $Read$, $Write$) étant donné i_{x+1} et $Exec(\_, x)$ (resp. $Read$, $Write$), nous appliquons la règle d'exécution:

- *Règle d'exécution:* si i_{x+1} est à l'adresse $m_{i_{x+1}}$, alors :

$$Exec(m_{i_{x+1}}, x + 1) \leftarrow Write(m_{i_{x+1}}, x) + 1$$

  Cette règle signifie que du code de niveau d'écriture k a un niveau d'exécution de $k + 1$.

Ensuite nous appliquons les règles ci-dessous, selon i_{x+1}:

- *Règle de lecture:* si i_{x+1} lit l'adresse m' (comme par exemple l'instruction `mov eax, m'`), alors :

  $$Read(m', x+1) \leftarrow Exec(m_{i_{x+1}}, x+1)$$

  Cette règle signifie qu'une adresse lue par une instruction de niveau k a un niveau de lecture k

- *Règle d'écriture:* si i_{x+1} écrit l'adresse m' (comme par exemple l'instruction `mov m', eax`), alors :

  $$Write(m', x+1) \leftarrow Exec(m_{i_{x+1}}, x+1)$$

  Cette règle signifie qu'une adresse écrite par une instruction de niveau k a un niveau d'écriture k.

Remarque : la règle d'exécution est toujours appliquée *après* l'exécution de l'instruction, donc on n'a pas besoin de tenir compte de la manière dont le contrôle a été transferré (saut direct, saut indirect, transfert à l'instruction suivante, ou transfert asynchrone). Dans certains cas, les trois règles (exécution, lecture, écriture) sont appliquées, comme par exemple avec l'instruction `xor m, m`.

1.3 Construction des couches de code

Au cours d'une exécution, une même adresse peut avoir des niveaux différents à deux instants différents. Une couche de code peut être vue comme l'ensemble des adresses mémoires ayant eu le même niveau au cours de l'exécution.

Ainsi, on définit $\mathbf{R}_k$ (resp. $\mathbf{W}_k$, $\mathbf{X}_k$) l'ensemble des adresses mémoires qui ont eu le niveau de lecture (resp. écriture, exécution) k durant l'exécution :

$\mathbf{R}_k = \{m \mid \exists x\, Read(m, x) = k\}$

$\mathbf{W}_k = \{m \mid \exists x\, Write(m, x) = k\}$

$\mathbf{X}_k = \{m \mid \exists x\, Exec(m, x) = k\}$

On peut ainsi définir *la couche de code k* comme le triplet $(\mathbf{R}_k, \mathbf{W}_k, \mathbf{X}_k)$.

La construction des ensembles $\mathbf{R}_k$, $\mathbf{W}_k$ et $\mathbf{X}_k$ peut se faire avec une complexité $O(max.log(max))$, où max est le nombre d'instructions exécutées par le programme. Le calcul peut être fait à chaud (c'est-à-dire au cours de l'exécution du programme), ou à froid si l'on a extrait une trace.

1.4 Mécanismes de protection

Une fois les couches de code reconstruites, il est possible de détecter des motifs couramment utilisés pour la protection des programmes contre la rétro-ingénierie.

Par exemple, un programme auto-modifiant va se traduire en un programme qui exécute du code de niveau k' qui a été écrit au niveau $0 < k < k'$. On construit l'ensemble *Self*(k, k') des adresses écrites au niveau k et exécutées au niveau k' de la manière suivante :

$$\textit{Self}(k, k') =_{\textit{dfn}} \mathbf{W}_k \cap \mathbf{X}_{k'},\ 0 < k < k'$$

Un programme auto-modifiant est donc un programme tel que $\cup_{k<k'} \textit{Self}(k, k')$ n'est pas vide.

Nous introduisons maintenant divers mécanismes de protection que l'on peut détecter avec TraceSurfer :

- *Auto-modification en aveugle* : la couche k modifie en aveugle la couche k' si des instructions de k' ont été écrites mais pas lues par la couche k :

$$\textit{Blind}(k, k') =_{\textit{dfn}} \textit{Self}(k, k') \setminus \mathbf{R}_k \neq \emptyset$$

- *Déchiffrement* : la couche k déchiffre la couche k' si des instructions de k' ont été à la fois lues et écrites par la couche k :

$$\textit{Decrypt}(k, k') =_{\textit{dfn}} \textit{Self}(k, k') \cap \mathbf{R}_k \neq \emptyset$$

- *Vérification d'intégrité* : la couche k vérifie l'intégrité de la couche k' si elle lit des instructions de la couche k' qui n'ont pas été modifiées par des couches entre k et k':

$$\textit{Check}(k, k') =_{\textit{dfn}} \mathbf{R}_k \cap \mathbf{X}_{k'} \setminus \cup_{k'' \in [\![k, k']\!]} \mathbf{W}_{k''} \neq \emptyset$$

- *Ecrasement de code* : la couche k est écrasée par la couche k' si des instructions de k ont été écrites par la couche k' avec $k < k'$:

$$\textit{Scrambled}(k, k') =_{\textit{dfn}} \mathbf{X}_k \cap \mathbf{W}_{k'} \neq \emptyset,\ 0 < k < k'$$

Dans tous les cas, on dit qu'une trace vérifie le mécanisme de protection A, $A \in \{\textit{Blind}, \textit{Decrypt}, \textit{Check}, \textit{Scrambled}\}$, si $\cup_{k,k'} A(k, k')$ n'est pas vide (c'est-à-dire s'il existe au moins deux couches vérifiant le motif).

L'algorithme que nous utilisons pour détecter les mécanismes de protection fonctionne en temps $O(n^2.m)$, où n est le nombre de couches et m est la borne supérieure de la taille des vagues.

1.5 Autres analyses

En plus des mécanismes de protection basés sur les couches de code (les protections algorithmiques), on peut aussi s'intéresser aux aspects plus techniques de l'anti-rétro-ingénierie, comme par exemple les techniques destinées à détecter les debuggers ou la virtualisation.

TraceSurfer est capable de détection l'utilisation de quelques techniques d'anti-virtualisation courantes comme RedPill [23] et ses variantes, ainsi que le canal caché de VMware utilisé dans ScoopyNG [18].

Un aspect intéressant de l'analyse à chaud est que l'instrumentation dynamique de binaire permet de modifier le comportement du programme. Notamment, si on détecte l'utilisation de la technique RedPill, on peut changer la sortie de l'instruction utilisée (SIDT, SLDT, SGDT...) afin de faire échouer le test. On peut donc améliorer artificiellement la transparence de la virtualisation, même s'il existe de nombreux autres moyens de détecter la présence d'un hyperviseur [10], certains étant virtuellement impossible à contrer de manière efficace [1, 12].

En résumé, nous sommes actuellement capables de détecter et contrer les techniques de détection d'hyperviseur suivantes :

- RedPill et ses variantes (SIDT, SLDT, SGDT)
- STR
- le canal caché de VMware

Nous considérons l'ajout des détections suivantes :

- d'autres techniques de détection de la virtualisation (pour VirtualPC, VirtualBox, Xen...)
- techniques anti-debugger
- techniques anti-sandbox (il s'agit de contre-mesures pour certains services d'analyse en ligne, comme Anubis [2], Joebox [6], CWSandbox [24], etc.)

1.6 Implementation

TraceSurfer est actuellement basé sur l'instrumentation dynamique de binaire afin de récolter les traces d'exécution. Il est composé d'environ 620 lignes de C++, qui sont un plugin pour Pin (un pintool). L'instrumentation dynamique de binaire offre certains avantages, comme un contrôle complet sur le binaire et un développement rapide. En revanche, certains inconvénients sont non négligeables, en particulier des problèmes de transparence [5] et un ralentissement parfois massif de l'exécution.

On pourrait envisager effectuer les mêmes calculs sans se baser sur l'instrumentation dynamique, en utilisant plutôt un émulateur logiciel comme QEMU [3], ce qui aiderait à améliorer la transparence de l'analyse au prix d'un ralentissement encore plus élevé.

En plus des problèmes de transparence, notre prototype souffre des limitations suivantes :

- pas de support des binaires 64 bits

- pas de support pour l'instrumentation en mode noyau
- support limité des applications avec plusieurs threads

2 Protocole expérimental

Pour évaluer l'efficacité de l'outil, nous l'avons testé sur des dizaines de milliers de logiciels potentiellement malveillants. Le but de cette expérience est de détecter quels échantillons sont protégés, le type de protection utilisé (packing, anti-virtualisation, vérification d'intégrité...), et le passage à l'échelle sur un grand volume de binaires.

2.1 Sélection des échantillons

Les échantillons ont été collectés sur le téléscope du Laboratoire de Haute Sécurité du Loria. Nous avons employé l'utilitaire `file` pour éliminer les binaires qui n'étaient pas du type exécutable DOS/Win32. Ce filtrage a retourné 59.554 binaires sur un nombre initial de 62.498 (voir section 3.1). Nous avons ensuite envoyé ces échantillons sur le cluster pour exécution.

2.2 Exécution sur le cluster

Nous avons utilisé un cluster composé de 12 noeuds tournant sous Ubuntu 8.04, chaque noeud disposant de deux Quad Core Xeon L5420 et de 16 Go de RAM. Ce cluster devait répondre aux contraintes suivantes :

- Isolation : on doit être capable d'exécuter des logiciels malveillants sur le cluster sans le compromettre, et sans faire de victimes collatérales
- Passage à l'échelle : l'architecture comporte X noeuds avec Y machines virtuelles par noeud, il doit être aisé d'ajouter ou remplacer des noeuds ou des machines virtuelles
- Automatisation : la présence d'un humain ne doit pas être nécessaire durant l'analyse

Pour la contrainte d'isolation, nous avons employé une couche de virtualisation avec des snapshots figés, et une déconnexion physique du réseau. Nous avons déployé les machines virtuelles avec VMware Server 2.0, chaque machine virtuelle disposant d'une carte Ethernet virtuelle en mode HostOnly afin de pouvoir communiquer avec son noeud hôte. Chaque machine virtuelle est un Windows XP Pro X64 avec Pin et un serveur SSH pré-installés. On utilise l'API VIX de VMware pour démarrer et éteindre les machines virtuelles.

La machine virtuelle a été déployée deux fois sur chacun des 12 noeuds. On utilise ensuite un programme qui distribue les échantillons sur les 24 machines virtuelles. Le fonctionnement de ce programme est simple : pour chaque

Fichiers sur le honeypot	62.498	
Fichiers exécutables	59.554	
Malwares selon Kaspersky	58.089	97,54%
Malwares selon ESET NOD32	57.685	96,86%

Table 1: Ensemble de départ

échantillon, une machine virtuelle est démarrée et l'échantillon est envoyée via une connection SFTP. On lance ensuite TraceSurfer sur le binaire avec SSH, et lorsque l'analyse est terminée (exécution terminée ou timeout), on récupère le rapport avec SFTP. Enfin, on éteint la machine virtuelle, ce qui efface toutes les modifications effectuées par l'échantillon sur le système invité. La synchronisation entre les noeuds du cluster est effectuée avec un mécanisme de lockfile qui permet d'éviter plusieurs analyses du même échantillon.

3 Résultats

3.1 Une vue d'ensemble des échantillons

Le tableau 1 donne une vision d'ensemble de notre ensemble de départ.

Nous avons lancé deux anti-virus du commerce sur les 59.554 fichiers exécutables. Les taux de détection de 97,54% et de 96,86% confirment l'intuition que les exécutables sont essentiellement des logiciels malveillants.

TraceSurfer a fourni un rapport pour 48.404 (81,28%) de ces fichiers en 34 heures et 10 minutes, avec un timeout de 90 secondes pour chaque échantillon. Sur ces binaires, 13.409 (27,70%) ont été stoppés à cause du timeout.

L'analyse a échoué sur 18,72% des échantillons pour plusieurs raisons possibles :

- l'instrumentation échoue sur certains binaires protégés de manière agressive
- certains binaires ne sont pas supportés (rootkits, exécutables 64 bits)
- certains fichiers PE sont cassés, ou plantent même sans instrumentation
- certains exécutables sont écrits pour une version spécifique de Windows et ne marcheront pas sur nos machines virtuelles

La section 3.2 comporte une analyse plus détaillée de la sortie de TraceSurfer.

Finalement, nous avons lancé pefile [7] avec approximativement 2.600 signatures sur notre ensemble de départ. Le résultat était assez remarquable : aucun packer n'était détecté dans 97,08% des échantillons.

Couche max	Binaires	Proportion
1	318	0,66%
2	4.184	8,64%
3	516	1,07%
4	589	1,22%
5	42.455	87,71%
6	86	0,18%
7	41	0,08%
8	92	0,19%
9	10	0,02%
10	38	0,08%
11	32	0,07%
12	40	0,08%
14	2	0,00%
15	1	0,00%

Table 2: Analyse des couches de code

3.2 Analyse détaillée

Le tableau 2 montre le nombre de couches de code trouvées par TraceSurfer. Les proportions dans ce tableau sont basées sur les 48.404 fichiers pour lesquels TraceSurfer a pu fournir un rapport.

On remarquera que :

- 99,34% des fichiers analysés utilisent du code généré dynamiquement (c'est-à-dire au moins 2 couches de code). Ils ne sont pas nécessairement packés par un produit courant, mais ils utilisent tous au moins une forme d'auto-modification.
- le nombre de couches est relativement bas (pas plus de 15)
- 87,71% des échantillons analysés utilisent 5 couches. Ce résultat était surprenant, l'hypothèse de départ était de trouver un pic à 2 couches, puisque la plupart des packers simples utilisent seulement 2 couches. Ce biais est sans doute dû à la configuration particulière du honeypot.

Le tableau 3a montre les différents mécanismes de protection détectés, comme définis dans la section 1.4.

Le tableau 3b montre les différentes technique d'anti-virtualisation utilisées. On remarque que très peu d'échantillons utilisent ces techniques (0,15%), et la plupart utilisent la technique RedPill (SIDT), ce qui confirme l'intuition.

Protection utilisée	Binaires	Proportion
Déchiffrement (*Decrypt*(k, k'))	44.046	91,00%
Auto-modif. aveugle (*Blind*(k, k'))	43.805	90,50%
Vérif. d'intégrité (*Check*(k, k'))	42.665	88,14%
Ecrasement (*Scrambled*(k, k'))	601	1,24%

(a) Détection des mécanismes de protection de code

Anti-virtualisation utilisée	Binaires	Proportion
Au moins une	71	0,15%
SIDT	65	0,13%
SLDT	0	0,00%
SGDT	0	0,00%
STR	0	0,00%
Canal caché VMware	14	0,03%

(b) Détection des techniques d'anti-virtualisation

Table 3: Analyse des échantillons du honeypot

3.3 Analyse de fichiers sains

Nous avons également lancé TraceSurfer sur un petit ensemble de programmes sains (dont on peut supposer qu'ils ne sont pas malveillants). Nous avons sélectionné 467 fichiers .exe uniques sur une installation fraîche de Windows XP. TraceSurfer a fourni un rapport pour 388 de ces fichiers (83,08%) dans les mêmes conditions que les échantillons du honeypot en 30 minutes environ.

Le tableau 4a montre que du code dynamique a été trouvé dans 66 de ces 388 fichiers (17,01%). Il semble s'agir du compilateur just-in-time de l'environnement .NET, ce qui correspond donc au résultat attendu (seulement 2 couches et pas de technique de protection).

Comme attendu, aucune technique d'anti-virtualisation n'a été détectée.

4 Limitations

Notre approche a les limitations suivantes :

- le mécanisme de *timeout* est un problème typique pour les outils d'analyse dynamique. Une solution potentielle serait d'utiliser un algorithme d'exploration de chemins multiples [20], mais il s'agit d'un problème avancé.
- la *transparence* de l'outil de traçage : notre modèle travaille sur des traces au niveau CPU afin d'être le plus générique possible. Cependant, l'instrumentation dynamique impose de modifier lourdement les binaires tracés, et ces modifications peuvent être détectées (ce qui risque

Couches	Binaires	Proportion
1	322	82,99%
2	66	17,01%

(a) Analyse des couches de code

Protection utilisée	Binaires	Proportion
Déchiffrement (*Decrypt*(k, k'))	61	15,72%
Auto-modif. aveugle (*Blind*(k, k'))	1	0,26%
Vérif. d'intégrité (*Check*(k, k'))	0	0,00%
Ecrasement (*Scrambled*(k, k'))	0	0,00%

(b) Détection des mécanismes de protection

Anti-virtualisation utilisée	Binaires	Proportion
Au moins une	0	0,00%

(c) Détection des techniques d'anti-virtualisation

Table 4: Analyse de fichiers sains

de changer le comportement du binaire). Une solution possible serait d'utiliser un autre mécanisme d'extraction des traces, comme un émulateur complet.

- écritures via le *noyau* : comme nous instrumentons uniquement le code en mode utilisateur, les zones mémoires écrites par le noyau ne sont pas visibles dans notre trace. On risque donc de manquer du code dynamique généré à l'aide d'appels système. Une solution pourrait être de surveiller explicitement les appels systèmes qui écrivent dans l'espace mémoire du processus ou d'utiliser un émulateur complet.
- les *labels* associés à un mécanisme de protection particulier sont donnés à titre indicatif et peuvent ne pas correspondre à l'intuition. Par exemple, un compilateur just-in-time va généralement apparaître sous la forme du pattern "déchiffrement de code", alors qu'un compilateur just-in-time ne travaille évidemment pas par déchiffrement. Les labels correspondent à des comportements probables dans le cadre de l'analyse de logiciels malveillants.

Conclusion

Nous avons défini un cadre théorique pour l'analyse de programmes avec du code généré dynamiquement, et nous avons défini une taxonomie des comportements auto-modifiants dans ce cadre. Nous avons également développé un prototype pour l'analyse de logiciels malveillants basé sur les traces, qui peut automatiquement détecter l'utilisation de mécanismes de protection avancés.

Enfin, nous avons distribué cette implémentation sur un cluster qui peut actuellement analyser environ 1.400 binaires par heure. Dans les versions futures, nous envisageons d'ajouter des détections pour d'autres comportements suspects, comme les techniques anti-sandbox par exemple.

Un axe de recherche futur serait d'utiliser TraceSurfer pour générer automatiquement des signatures comportementales et d'implémenter des techniques de détection efficaces basées sur ces signatures.

Remerciements

Nous souhaitons remercier Vincent Mussot pour son implémentation des mesures anti-anti-virtualisation dans TraceSurfer et pour son travail sur la transparence de Pin.

References

[1] Edgar Barbosa. Blue pill detection. In *SyScan*, 2007.

[2] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. Ttanalyze: A tool for analyzing malware, 2006.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, page 41, Berkeley, CA, USA, 2005. USENIX Association.

[4] Lutz Boehne. Pandora's bochs: Automatic unpacking of malware. 2008.

[5] Derek Bruening. Efficient, transparent, and comprehensive runtime code manipulation, 2004. Ph.D. Thesis, MIT.

[6] Stefan Bühlmann. Extending joebox - a scriptable malware analysis system, 2008. Bachelor's Thesis, University of Applied Sciences Northwestern Switzerland.

[7] Ero Carrera. 4 x 5: Reverse engineering automation with python. In *Black Hat USA*, 2007.

[8] Ero Carrera. Malware - behavior, tools, scripting and advanced analysis. In *HITB*, 2008.

[9] Silvio Cesare. Security applications for emulation. In *Ruxcon*, 2008.

[10] Peter Ferrie. Attacks on virtual machine emulators. In *AVAR Conference*, 2006.

[11] Peter Ferrie. Anti-unpacker tricks. 2008.

[12] Eric Filiol. Formal model proposal for (malware) program stealth. In *Virus Bulletin*, 2007.

[13] Tobias Graf. Generic unpacking – how to handle modified or unknown pe compression engines? In *Virus Bulletin*, 2005.

[14] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. A study of the packer problem and its solutions. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 98–115, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] Sebastien Josse. Secure and advanced unpacking using computer emulation. *Journal in Computer Virology*, 3, 2007.

[16] Sebastien Josse. Analyse et detection dynamique de code viraux dans un contexte cryptographique, 2009. Ph.D. Thesis, Ecole Polytechnique.

[17] Min Gyung Kang, Heng Yin, and Pongsin Poosankam. Renovo: A hidden code extractor for packed executables. In *5th ACM Workshop on Recurring Malcode*, 2007.

[18] Tobias Klein. Scoopyng - the vmware detection tool, 2008. `http://www.trapkit.de/research/vmm/scoopyng/index.html`.

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Kim Hazelwood, and Vijay Janapa Reddi. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation (PLDI)*, 2005.

[20] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[21] Danny Quist and Valsmith. covert debugging, circumventing software armoring techniques. In *Black Hat USA*, 2007.

[22] Paul Royal. Alternative medicine: The malware analyst's bluepill. In *Black Hat USA*, 2008.

[23] Joanna Rutkowska. Red pill... or how to detect vmm using (almost) one cpu instruction, 2004. `http://www.invisiblethings.org/papers/redpill.html`.

[24] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *Security and Privacy, IEEE*, 5(2):32–39, March-April 2007.