Mitigando a Evasão de Instrumentadores Binários Dinâmicos

Ailton Santos Filho[1], **Arthur Binda Alves** [1], **Isaque Vieira** [1], **Eduardo L. Feitosa**[1]

[1]Instituto de Computação – Universidade Federal do Amazonas (UFAM)
CEP 69.077-000 – Manaus – AM – Brasil

{assf,aba,ibv2,efeitosa}@icomp.ufam.ed.br

Abstract. *This paper presents countermeasures against the most recent evasion techniques in Dynamic Binary Instrumentation (DBI). They use features of the Intel Pin instrumentation tool to ensure the transparency of the analyzes. An evaluation of proposed countermeasures in comparison to state-of-the-art techniques was performed. The current limitations and the future of this work are also presented.*

Resumo. *Este artigo apresenta contramedidas contra as mais recentes técnicas de evasão (analysis-aware) em Instrumentação Binária Dinâmica (DBI). Elas utilizam recursos do instrumentador Intel Pin para garantir a transparência das análises. Uma avaliação das contramedidas propostas em comparação as tecnicas tidas como estado da arte foi realizada. As limitações existentes e o futuro desse trabalho também são apresentados.*

1. Introdução e Motivação

No atual estágio da "guerra" contra *malwares*, os *frameworks* de Instrumentação Binária Dinâmica (*Dynamic Binary Instrumentation*, DBI) têm sido soluções empregadas na análise de segurança de aplicações. Isto porque tais ferramentas permitem modificar ou adicionar código de máquina em programas ou aplicações e assim avaliar o desempenho e a corretude, detectando falhas em tempo de execução. Por isso, eles vêm sendo utilizados na detecção de *shellcode* [Mohaisen et al. 2015], *taint analysis* [Stamatogiannakis et al. 2014], análise de código auto-modificante [Giacobazzi et al. 2014], entre outros.

Infelizmente, à medida que os *framework* DBI ganharam popularidade, os *malwares* incorporaram técnicas para detectar quando estão sendo instrumentados (analisados ou avaliados). Assim, começaram a incorporar técnicas de evasão[1] em seus códigos, que os permitem reconhecer quando estão sendo executados em um ambiente real ou em um ambiente de instrumentação.

A ideia é que, usando *APIs* (*Application Programming Interface*) do sistema operacional, o *malware* seja capaz de perceber quando está sendo instrumentado e, assim, possa mudar seu comportamento ilegítimo. *Malwares* com esse perfil de funcionamento são conhecidos como aplicações conscientes de análise (*analysis-aware applications*) [Rodriguez et al. 2016]. Foi com base nesse conceito que Rodriguez et *al.* propuseram uma ferramenta (PinVMShield) para detectar técnicas de evasão

[1]Neste trabalho, o termo técnica de evasão engloba todos os métodos usados por *malwares* para evitar sua detecção, análise e compreensão.

 © 2017 Sociedade Brasileira de Computação

usadas por *malwares* conscientes da análise em ambientes *sandbox* ou virtualizados. A PinVMShield, baseada no instrumentador Pin [Intel 2016], é uma ferramenta capaz de enganar binários maliciosos e evitar que técnicas de detecção e evasão de VMs-*sandbox*, ditas *analisys-aware*, logrem êxito.

Recentemente, Sun et *al.* [Sun et al. 2016] demonstraram, em uma apresentação na Black Hat, entitulada "Break Out of The Truman Show", novas técnicas de evasão independentes das APIs do sistema operacional. Essas técnicas permitem que um *malware*, ao tomar ciência de que está sendo instrumentado, faça uso do *framework* DBI para executar ações maliciosas fora do contexto de análise do instrumentador, obviamente sem ser detectado. Entretanto, ao analisar as técnicas propostas por Sun et *al.*, e outras técnicas mais antigas, percebeu-se que algumas contramedidas podem ser tomadas. É neste contexto que este trabalho propõe contramedidas para evitar técnicas de evasão em *frameworks* DBI. Para tanto, empregando a ferramenta PVMShield como base, novas mitigações foram implementadas, incluindo para as técnicas de evasão propostas por Sun et *al.*.

As contribuições deste trabalho são duas. A primeira é a proposição e prova de técnicas anti-evasão de DBI, baseadas em duas das ideias de Sun et *al.*: *Code Cache* - região da memória onde os trechos de código instrumentados ficam armazenados - e *Thread Local Storage* (TLS) - que permite que *threads* de um mesmo processo tenham acesso a uma estrutura de dados comum, capazes de mitigar até os métodos de evasão do estado da arte. A segunda é o desenvolvimento de um protótipo para a defesa da transparência de instrumentadores binários dinâmicos com todo código fonte disponível[2].

2. Evadindo Instrumentadores Binários Dinâmicos

Dentre as técnicas de evasão apresentadas em [Sun et al. 2016], este trabalho foca em mitigar a detecção por *Code Cache* e por *Thread Local Storage*. A técnica de detecção por *Thread Local Storage* foi selecionada pela sua simplicidade de funcionamento, além de ser claramente uma técnica de evasão baseada no uso de APIs do Windows. Já a detecção por Code Cache foi escolhida devido ao seu impacto nas soluções que utilizam *frameworks* DBI, como o Pin, uma vez que permite a subversão do fluxo de execução dos instrumentadores, como mostrado por Sun et *al.*

2.1. Detecção por *Code Cache*

Quando *frameworks* DBI realizam análises em tempo de execução, não permitem que o código original da aplicação sendo instrumentada seja executado. Os trechos de código executados são os provenientes de regiões conhecidas como *Code Cache*, regiões especiais, alocadas pelos *frameworks* DBI, que contêm o código do programa alvo após ser modificado pelo instrumentador.

Sun et *al.* perceberam que os *Code Caches* do Pin possuem características diferentes em relação à outras páginas de memória, o que permite sua identificação. São elas: (i) As permissões da página de memória são sempre de execução, leitura e escrita; (ii) Os dados alocados no início de cada *Code Cache* são sempre valores hexadecimais pré-definidos (0xFEEDBEAF); (iii) O tamanho da página de memória é fixo em um valor pré-determinado (0x40000 é o valor padrão).

[2] `https://github.com/ailton07/PinVMShield`

© 2017 Sociedade Brasileira de Computação

Além disso, também perceberam que o Pin aloca *Code Caches* de acordo com sua necessidade no espaço de endereços do processo alvo, o que cria duas outras formas de detectar suas *Code Caches*. A primeira forma é executar intencionalmente uma sequência de instruções que ocorra apenas uma vez durante a execução do programa e checar todo o espaço de endereços do seu processo, a fim de contabilizar o número de ocorrências da sequência. Se houver mais de uma ocorrência, existe um *Code Cache*. Essa técnica foi nomeada como Detecção de *Code Cache* por código. A Figura 1 ilustra a prova feita por Sun et al. da detecção de *Code Cache*.

```
C:\Users\Wild Sator\Documents\Visual Studio 2013\Projects\btescape\DBI\codecachs
ch\Release>codecache_detect.exe

sigature function executed.
signature found @ 0x13c1038
memory search completed, signature count 1
```

```
C:\Users\Wild Sator\Documents\Visual Studio 2013\Projects\btescape\pin-2.14-7131
3-msvc12-windows>pin.exe -- ..\DBI\codecachsch\Release\codecache_detect.exe

sigature function executed.
signature found @ 0x1161038
signature found @ 0x18478b2
memory search completed, signature count 2

DBI Detected!!

DBI tool = PIN!
```

Figura 1. Exemplo de Detecção do *Code Cache*. Fonte: [Sun et al. 2016]

A segunda, nomeada de Detecção de *Code Cache* por dados, assume que a aplicação executaria intencionalmente uma instrução com um dado conhecido único no operando da instrução e checaria em todo o espaço de endereços do seu processo o número de ocorrências do dado. Caso haja mais de uma ocorrência, isso seria um indício de *Code Cache*. Provas de conceito das técnicas de detecção de *Code Cache* por código e por dados foram desenvolvidas pelos autores deste trabalho e encontram-se disponíveis[3] para uso. Também existe uma prova de conceito baseada na detecção das três características dos *Code Caches* enumaradas acima, que encontra-se disponível [4].

2.2. Detecção por Thread Local Storage

Thread Local Storage (TLS) é uma funcionalidade que permite que threads de um mesmo processo tenham acesso a uma estrutura de dados comum na forma de um vetor de até 1088 posições [Microsoft 2017]. É um recurso largamente utilizado para reduzir a necessicade do sincronismo entre threads de um programa. Sabendo que o Pin faz uso dessa funcionalidade, Sun et *al.* demonstraram que é possível uma aplicação sendo instrumentada determinar se está sob análise através de uma consulta e da contagem do número de *slots* sendo utilizados na TLS. Uma prova de conceito das técnicas de detecção por *Thread Local Storage* foi desenvolvida pelos autores deste trabalho e encontra-se disponível[5] para uso.

Na Figura 2, o terminal a esquerda representa a execução normal de uma aplicação e o terminal a direita a mesma aplicação sendo instrumentada pelo Pin.

[3] `https://github.com/ailton07/ActiveDetectionPinWithCodeCache`
[4] `https://github.com/ailton07/ActiveDetectionPinWithFEEDBEAF`
[5] `https://github.com/ailton07/TLS_DetectPin`

 © 2017 Sociedade Brasileira de Computação

```
TLS Slots:
[0]:    0x0
[1]:    0x0
[2]:    0x0
[3]:    0x0
[4]:    0x0
[5]:    0x0
[6]:    0x0
[7]:    0x0
[8]:    0x0
[9]:    0x0
[10]:   0x0
[11]:   0x0
[12]:   0x0
[13]:   0x0
[14]:   0x0
[15]:   0x0
[16]:   0x0
[17]:   0x0
[18]:   0x0
[19]:   0x0
[20]:   0x0
```

```
TLS Slots:
[0]:    0x0
[1]:    0x90100
[2]:    0x16c0010
[3]:    0x0
[4]:    0x0
[5]:    0x0
[6]:    0x0
[7]:    0x0
[8]:    0x0
[9]:    0x0
[10]:   0x0
DBI Detected!!

DBI tool = PIN!
```

Figura 2. Detecção do Pin via TLS. Fonte: [Sun et al. 2016]

3. Implementação

Este trabalho detecta e mitiga técnicas de evasão por *Code Cache* e por TLS. Seu desenvolvimento foi voltado para o instrumentador Intel PIN por ser o *framework* DBI com menor custo computacional imposto sobre a aplicação instrumentada [Luk et al. 2005] e na qual a *PinVMShield* foi empregada para viabilizar a implementação das contramedidas propostas.

3.1. Contramedida para Detecção por *Code Cache*

Em seu artigo, Sun et *al.* [Sun et al. 2016] apresentam uma série de técnicas de evasão de DBI por detecção de *Code Cache*. A primeira se foca em determinar se uma página de memória do processo é um *Code Cache* a partir de um conjunto pré-determinado de características. Em outras palavras, as permissões de página de memória são uma característica utilizada para confirmar a existência de um *Code Cache*. De forma prática, eles argumentam que a principal maneira de se obter o estado atual das permissões de uma página de memória é através da API do Windows *VirtualQuery*.

A contramedida proposta contra essa forma de detecção de *Code Cache* emprega a função *ReplaceWinAPI*, do PinVMShield, para alterar os retornos das chamadas a *VirtualQuery*, a fim de não retornar as informações reais de um *Code Cache*. Vale ressaltar que embora Sun et *al.* comentem que podem existir outras maneiras de se determinar as permissões de uma página de memória, essas permissões não são um critério suficiente por si só para identificar um *Code Cache*.

A segunda forma de verificar a presença de um *Code Cache* é através do tamanho da página de memória, empregando novamente a API *VirtualQuery* para determinar o tamanho da página de memória. A contramedida proposta também utiliza a função *ReplaceWinAPI* para impedir que o processo sendo instrumentado tome conhecimento do real tamanho da página de memória do *Code Cache*. O Pin fornece um conjunto APIs para mapeamento de suas *Code Caches*, entre elas destacam-se *TRACE_CodeCacheAddress* e *TRACE_CodeCacheSize*, que são utilizadas na contramedida.

A terceira forma é uma técnica de força bruta, com elevado custo computacional, que procura em todo o espaço de memória do processo por esses padrões de conteúdo que identificam um *Code Cache*. A contramedida proposta utiliza a API *INS_AddInstrumentFunction* do Pin para desenvolver um método que atua ativamente

© 2017 Sociedade Brasileira de Computação

sobre cada instrução do programa sendo instrumentado, a fim de impedir que sejam feitos acessos diretos ao espaços de memória dos *Code Caches*. É importante esclarecer que essa é uma solução de elevado custo computacional, assim como a técnica de evasão proposta por Sun et *al.*

Entretanto, sabendo do custo computacional desta técnica de evasão, os autores deste artigo propõem uma versão alternativa que não demanda o esforço computacional de procurar em todo o espaço de memória do processo. Para tanto, utiliza-se a API do Windows *VirtualQuery* para percorrer as páginas de memória, ao invés do espaço de memória, em busca do valor inicial do *Code Cache*. Contudo, neste caso, a contramedida proposta para a segunda técnica de Sun et *al.*, que utiliza a função *ReplaceWinAPI*, do PinVMShield, para impedir que a API do Windows *VirtualQuery* retorne páginas de memória de *Code Caches*, pode ser aplicada para inibir a evasão.

É importante destacar que o uso da função *ReplaceWinAPI* do PinVMShield também é a contramedida padrão para lidar com as questões apresentada na Seção 2.1: (i) das permissões da página de memória, descritas pelo campo *AllocationProtect* da estrutura *Memory Basic Information do Windows*; (ii) dos dados alocados no início de cada *Code Cache*, que são sempre os valores hexadecimais (0xFEEDBEAF, por exemplo); e (iii) do tamanho da página de memória é fixo em um valor pré-determinado (valor padrão 0x40000).

3.2. Contramedida para Detecção por Thread Local Storage

A detecção por *Thread Local Storage* (TLS) é uma técnica que detecta a presença dos *frameworks* DBI a partir da quantidade de posições sendo utilizadas no TLS. A princípio, a política utilizada por Sun et *al.* é de se um programa não utiliza a TLS, então não deveria haver nenhuma posição do TLS sendo utilizada. No entanto, foi observado que essa política pode acarretar falsos positivos.

Em seu trabalho, Sun et *al.* propuseram o uso das APIs TLS fornecidas pelo sistema operacional, sendo elas no Windows: *TlsAlloc*, *TlsGetValue*, *TlsSetValue* e *TlsFree*. Mais especificamente, a API *TlsGetValue* poderia ser utilizada para obter o valor de cada posição TLS e contabilizar quantas estão sendo utilizadas. A contramedida proposta utiliza a função da API *ReplaceWinAPI* do PinVMShield para garantir que durante as consultas ao TLS, a aplicação sendo instrumentada não seja capaz de obter os valores das posições utilizadas pelo Pin.

A alteração do retorno da API *TlsGetValue* pode ser detectada pela aplicação alvo da instrumentação, caso ela faça uso das APIs *TlsGetValue* e *TlsSetValue*. No entanto, se nem todos os espaços TLS estiverem em uso, é possível utilizar uma lógica de deslocamento de posições para garantir a transparência da técnica de mitigação. Por exemplo, caso o Pin esteja utilizando as cinco primeiras posições do TLS e a aplicação sob análise tente utilizar a primeira posição do TLS, basta mapear essa tentativa de uso para uma posição diferente das que o *framework* está utilizando.

4. Avaliação e Resultados Iniciais

Esta Seção apresenta e discute os resultados iniciais dos experimentos realizados. Para tanto, o protocolo experimental adotado e o ambiente de experimentação são detalhados. Em seguida, os resultados da efetividade de bloqueio (mitigação) das contramedidas

 © 2017 Sociedade Brasileira de Computação

propostas são discutidos. Por fim, uma análise do desempenho das contramedidas propostas em relação ao *PVMShield* é apresentada. Vale ressaltar que apenas o *PVMShield* é avaliado porque o trabalho de Sun et *al.* não possui contramedidas implementadas.

4.1. Ambientes de experimentação

Tanto os testes de bloqueio quanto às análises de desempenho foram realizados em um equipamento com Processador Intel Core i5-3230M 2.60 GHz, 4GB de memória RAM, sistema operacional Windows 7 Ultimate x64 based, compiladores Microsoft C/C++ Optimizing Compiler Version 15.00.30729.01 para x64 com Microsoft Visual Studio 2008, Microsoft C/C++ Optimizing Compiler Version 16.00.30319.01 para 80x86 com Microsoft Visual Studio 2010 e Pin 2.14 kit (71313) para MSVC10.

4.2. Eficácia

Para todas as técnicas de evasão de instrumentadores apresentadas na Seção 2 foram desenvolvidas provas de conceito funcionais capazes de detectar a presença do Pin, mesmo alvo de detecção do trabalho de Sun et *al.* Vale ressaltar que os códigos dessas provas de conceito estão disponíveis e estão listados neste artigo.

De forma direta, para cada técnica de detecção apontada, uma prova de conceito (contramedida implementada) e o *PinVMShield* (Rodriguez et *al.*) foi testada para verificar a capacidade de bloqueio. A Tabela 1 sumariza esses resultados, comprovando a eficácia das contramedidas em impedir que o Pin fosse detectado. O caractere *X* indica que a técnica de evasão não foi mitigada, enquanto o *OK* indica que a técnica de evasão foi mitigada com sucesso. Considerando que tanto o PinVMShield, quanto as técnicas de Sun et *al.* foram publicados em 2016, é esperado que o PinVMShield ainda não considerasse essas técnicas.

Tabela 1. Técnicas de Evasão impedidas pelo protótipo.

gray!50 **Técnica de Detecção**	**Pin**	**PinVMShield**	**Proposta**
Code Cache por Código proposta por Sun et al.	X	X	OK
Code Cache por Dado proposta por Sun et al.	X	X	OK
Code Cache por 0xFEEDBEAF proposta por Sun et al.	X	X	OK
Code Cache por Código alternativa	X	X	OK
Code Cache por Dado alternativa	X	X	OK
Code Cache por 0xFEEDBEAF alternativa	X	X	OK
Thread Local Storage	X	X	OK

4.3. Desempenho

Para medir os impactos das contramedidas propostas, comparando-as com a *PVMShield*, foi utilizado o SPEC CPU2006, uma suíte de *benchmarks*, aceita como padrão por indústrias do setor de computação, que se destina a testar exaustivamente o processador, sistema de memória e compilador de um sistema. Trabalhos como [Ferreira et al. 2014, Zeng 2015, Luk et al. 2005], este último dos próprios desenvolvedores do Pin, utilizam o SPEC CPU.

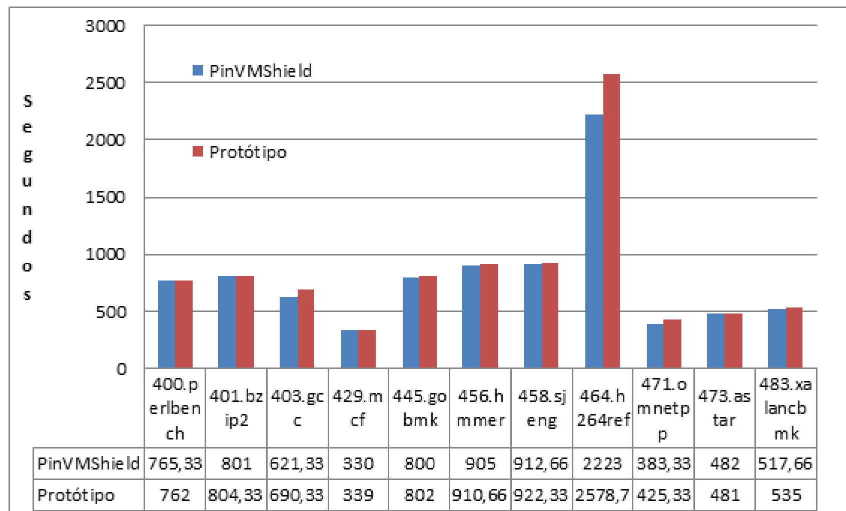
Foi utilizado o conjunto de componentes de testes SPECint, assim como em [Luk et al. 2005]. O SPECint2006 possui um conjunto de doze (12) aplicações de testes diferentes, que podem ser utilizadas para se obter métricas de desempenho. Ressalta-se

© 2017 Sociedade Brasileira de Computação

que apenas onze (11) são utilizáveis aqui, visto que a aplicação *462.libquantum* do SPEC não é aplicável em ambientes com Microsoft Visual C++, como o deste trabalho.

O SPEC foi configurado para: (i) utilizar o tamanho de entrada de dados de referência; (ii) utilizar o modo reportável que executa cada aplicação de teste três vezes; e (iii) executar em conjunto com o *PinVMShield* original sobre cada uma das onze aplicação de testes. Após isso, os experimentos foram repetidos com o SPEC configurado para executar cada aplicação de teste com as contramedidas criadas. Vale ressaltar que não foi incluída no teste de desempenho a contramedida da técnica de evasão sem uso de APIs do Windows, uma vez que os custos computacionais, da contramedida e da técnica de evasão, são impraticáveis no mundo real.

A Figura 3 apresenta a comparação dos tempos de execução para o *PinVMShield* e para as contramedidas propostas. Cada barra no histograma representa o tempo de execução de uma das abordagens para cada aplicação de teste do SPEC. Abaixo da barra tem-se o nome da aplicação e o tempo em segundos da execução.



	400.perlbench	401.bzip2	403.gcc	429.mcf	445.gobmk	456.hmmer	458.sjeng	464.h264ref	471.omnetpp	473.astar	483.xalancbmk
PinVMShield	765,33	801	621,33	330	800	905	912,66	2223	383,33	482	517,66
Protótipo	762	804,33	690,33	339	802	910,66	922,33	2578,7	425,33	481	535

Figura 3. Comparativo entre os tempos de execução do PinVMShiedl e do Prototipo.

Pode-se observar na Figura 3 que as diferenças do tempo de execução entre as abordagens são bem sensíveis. Para obter um comparativo se a diferença entre os tempos é significativa, fez-se um teste de média zero que forneceu que a média da diferença entre os tempos de execução é 24.211 segundos, com uma precisão de 95% e com o intervalo de confiança entre (-17.6593 ; 66.0815), o que nos diz que não há diferença sensível entre as duas abordagens. Para as aplicações *400.perlbench* e *473.astar* o tempo de execução do protótipo foi inferior ao tempo do PinVMShield, no entanto, não foi realizado nenhum procedimento para a otimização desses testes.

5. Andamento do Trabalho

O controle do acesso às informações do sistema continua sendo um ponto crítico para a defesa da transparência de ferramentas de análise, em especial os instrumentadores binários dinâmicos. Este trabalho, ainda em andamento, demonstrou possibilidades de combate à diferentes técnicas de evasão para DBI, inclusive as mais recentes.

 © 2017 Sociedade Brasileira de Computação

Foram desenvolvidas neste trabalho um conjunto de contramedidas (todas com prova de conceito e futuramente agregadas em uma ferramenta), tomando como base o *framework PinVMShield*, para diferentes técnicas de "analysis-aware". Foi mostrado que as contramedidas desenvolvidas são capazes de evitar a ação de técnicas emergentes de evasão de instrumentadores binários dinâmicos sem acarretar perdas consideráveis de desempenho.

No que tange à continuidade deste trabalho pretende-se: (i) organizar as implementações em uma única ferramenta; (ii) realizar um estudo comparativo com ferramenta *Arancino* [Polino et al. 2017], que possui funcionalidades semelhantes a *PinVMShield*; (iii) propor novas contramedidas para técnicas existentes, focando inicialmente nas técnicas levantadas por Sun et *al.* [Sun et al. 2016] não tratadas neste trabalho; (iv) desenvolvimento de novas técnicas de evasão de instrumentadores binários dinâmicos.

Referências

Ferreira, M. T., Santos Filho, A., and Feitosa, E. (2014). Controlando a Frequência de Desvios Indiretos para Bloquear Ataques ROP. In *Anais do XIV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - SBSeg 2014*. SBC.

Giacobazzi, R., Simon, A., and Zennou, S. (2014). Challenges in analysing executables: Scalability, self-modifying code and synergy (dagstuhl seminar 14241). In *Dagstuhl Reports*, volume 4.

Intel (2016). Pin 2.14 user guide. `https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/`.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN*, pages 190–200, New York, NY, USA. ACM.

Microsoft (2017). Thread local storage. `http://goo.gl/U0etlH`.

Mohaisen, A., Alrawi, O., and Mohaisen, M. (2015). Amal: High-fidelity, behavior-based automated malware analysis and classification. *Computers & Security*, 52:251–266.

Polino, M., Continella, A., Mariani, S., D'Alessio, S., Fontana, L., Gritti, F., and Zanero, S. (2017). Measuring and defeating anti-instrumentation-equipped malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 73–96.

Rodriguez, R. J., Gaston, I. R., and Alonso, J. (2016). Towards the detection of isolation-aware malware. *IEEE Latin America Transactions*, 14(2):1024–1036.

Stamatogiannakis, M., Groth, P., and Bos, H. (2014). Looking inside the black-box: capturing data provenance using dynamic instrumentation. In *International Provenance and Annotation Workshop*, pages 155–167. Springer.

Sun, K., Li, X., and Ou, Y. (2016). Break out of the truman show. In *BlackHat*. `goo.gl/vo63g8`.

Zeng, J. (2015). Bunary code reuse: A dynamic analysis based approac. Master's thesis, University of Texas, Texas.

© 2017 Sociedade Brasileira de Computação