Server-Side Dynamic Code Analysis

Wadie Guizani, Jean-Yves Marion, Daniel Reynaud-Plantey
Nancy University - LORIA
Campus Scientifique - BP 239
54506 Vandoeuvre-lès-Nancy Cedex (France)
{guizaniw,marionjy,reynaudd}_at_loria.fr

Abstract

The common use of packers is a real challenge for the anti-virus community. Indeed, a static signature analysis can usually only detect and sometimes remove known packers if a specific unpacking routine has been programmed manually. Generic unpacking does not solve the problem due to its limited effectiveness. Additionally, the important number of binaries to scan on a daily basis makes automated analysis necessary in order to protect information systems. In this context, we propose a taxonomy of self-modifying behaviors, a generic method to detect them in potentially malicious samples and a scalable architecture for the distributed analysis of a high volume of binaries.

Introduction

Self-modifying programs are particularly interesting because of the fundamental nature of self-reference and its consequences on computability. Indeed, self-modifications are very problematic for program analysis because the program listing depends on time. It is also worth noting that any normal program can be easily turned into a self-modifying program by using a packer. As a result, packers are commonly encountered during malware analysis: packing is easy and reliable, it makes static analysis harder and it changes the signature of the binary. The use of packers is suspicious but not malicious by nature, as they can be used for legitimate purposes such as code compression. Additionally, static analysis can only reveal the presence of known packers [15] and can not reveal the features of packed code.

In this paper, we address the problem of automatically detecting unknown or custom packers and we propose a method for dynamically detecting the use of run time code protections such as code decryption, integrity checking and anti-virtualization techniques. We also test this analysis on a large number of samples in a distributed environment.

Contributions

In this paper, our contributions are:

- a theoretical framework for modelling self-modifying programs (Section 1)
- a taxonomy of self-modifying behaviors
- a clear definition of code layers, or waves
- a prototype implementation using dynamic binary instrumentation (Subsection 1.4)
- an architecture for server-side analysis of binaries on a cluster of virtual machines (Section 2)
- the result of a large scale experiment on malware samples captured by a honeypot (Section 3)

Real-World Use

The framework we use to model self-modifying programs is very generic and not specific to malware. Therefore, it can be used in many different scenarios. The prototype we implemented with it can be seen as a scoring system: it takes unknown binaries and outputs a score (i.e. a warning level) based on their use of self-modifications and other code armouring techniques.

1 Analysis by TraceSurfer

TraceSurfer is our prototype implementation using dynamic binary instrumentation for malware analysis. This tool, based on Pin [20], can reconstruct the code

waves used in self-modifying programs and detect protection patterns based on these code waves. We are first going to introduce the work that we built upon (Subsection 1.1), and then introduce memory layering (Subsection 1.2), code waves (Subsection 1.3) and finally the code protection patterns (Subsection 1.4).

1.1 Related Work and Automatic Unpacking

Our work on code waves can be seen as a generalization of the well-known method for automatic unpacking [5, 16]. The principle of this method is to log every memory write during the execution of the target, usually within an emulator, and to log every instruction pointer. As soon as an instruction pointer corresponds to an address that has been previously written to, it means that dynamic code has been found. Then unpacking can be attempted by dumping the memory and rebuilding an executable from the memory dump. Numerous implementations have been based on this model [14, 10], including Renovo [18], VxStripper [17], Saffron [22], Azure [23], and Bochs-based implementations [5, 9].

Our technique builds on the idea of automatic unpacking. However, we do not perform unpacking (hence we do not face the problem of memory dumping and executable reconstruction [12]) but refine the process for finding dynamic code. We extend it to work with multiple levels of execution and also log the memory reads. We can then define behavior patterns such as code decryption, integrity checking and code scrambling and detect these patterns efficiently.

Since we do not have the same output as the related tools, we can not accurately compare our performance. However, based on our experiment we can expect the output of our prototype to be more detailed but less robust. With the use of dynamic binary instrumentation, we were able to quickly develop a lightweight prototype to confront the theoretical framework with actual malware samples, at the price of stability.

1.2 Memory Layering

We consider the execution of a program to be an arbitrarily large sequence of instructions i_1, ..., i_x, ..., i_{max}, ... If we know the effects of each instruction on memory, we can precisely know the state of the program at each step x.

We are going to associate each memory address m at step x (i.e. after the execution of i_x) with an execution level $Exec(m, x)$, a read level $Read(m, x)$ and a write level $Write(m, x)$. Initially, for all m, we have $Exec(m, 0) = Read(m, 0) = Write(m, 0) = 0$. These levels are then updated after the execution of each instruction, depending on its effect on memory.

Suppose we are at step $x + 1$, we want to update $Exec(\_, x + 1)$ (resp. $Read$, $Write$) given i_{x+1} and $Exec(\_, x)$ (resp. $Read$, $Write$). We first apply the execution rule:

- *Execution Rule:* if i_{x+1} is at address $m_{i_{x+1}}$, then for all m:

$$Exec(m, x{+}1) = \begin{cases} Write(m, x) + 1 & \text{if } m = m_{i_{x+1}} \\ Exec(m, x) & \text{otherwise} \end{cases}$$

  Code written at some level k has an execution level of $k + 1$.

Then, we apply the rules below depending on i_{x+1}:

- *Memory Read Rule:* if i_{x+1} reads the memory address m' (such as `mov eax, [m']`), then for all m:

$$Read(m, x{+}1) = \begin{cases} Exec(m_{i_{x+1}}, x + 1) & \text{if } m = m' \\ Read(m, x) & \text{otherwise} \end{cases}$$

  A memory address read by an instruction at some level k has a read level of k.

- *Memory Write Rule:* if i_{x+1} writes to the memory address m' (such as `mov [m'], eax`),

$$Write(m, x{+}1) = \begin{cases} Exec(m_{i_{x+1}}, x + 1) & \text{if } m = m' \\ Writ(m, x) & \text{otherwise} \end{cases}$$

  A memory address written by an instruction at some level k has a write level of k.

Note that the execution rule is always applied *after* the execution of the instruction, no matter how the control was transferred to this instruction (direct, indirect, fall through and asynchronous). In some cases the three rules can be applied, for instance when an instruction both reads from and writes to memory.

1.3 Building the Code Waves

As we have just seen, during a computation each memory address can have different levels. A code wave, sometimes referred to in the literature as a code layer [18], can be seen as the sets of memory addresses that were at the same level at some point during the execution of the program.

Therefore, we define $\mathbf{R}_k$ (resp $\mathbf{W}_k$, $\mathbf{X}_k$) the set of every memory addresses that had read (resp. write, execution) level k during the execution:

$\mathbf{R}_k = \{m \mid \exists x \, s.t. \, Read(m,x) = k\}$
$\mathbf{W}_k = \{m \mid \exists x \, s.t. \, Write(m,x) = k\}$
$\mathbf{X}_k = \{m \mid \exists x \, s.t. \, Exec(m,x) = k\}$

We can now define the *code wave k* as the tuple $(\mathbf{R}_k, \mathbf{W}_k, \mathbf{X}_k)$. The existential operator is used in the definition for brevity, it does not imply that code waves can not be computed efficiently. It is indeed simple to build the sets $\mathbf{R}_k$, $\mathbf{W}_k$ and $\mathbf{X}_k$ incrementally by inserting the memory addresses affected by each instruction i_x in the right set. Building the sets $\mathbf{R}_k$, $\mathbf{W}_k$ and $\mathbf{X}_k$ can be done with a complexity $O(max.log(max))$ where max is the number of instructions executed by the program. This can be done in real-time or offline, given an instruction-level run trace.

1.4 Behavior Patterns

Once the code waves have been reconstructed, it is possible to exhibit specific protection patterns commonly used for code armoring.

For instance, a self-modifying program executes some code at level k' which was written at level $0 < k < k'$. We construct the set *Self*(k,k') of locations modified at level k and then executed at level k', as follows:

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

Then a self-modifying program is a program such that $\cup_{k<k'} \textit{Self}(k,k')$ is not empty.

We now present some usual behavior patterns that we use in TraceSurfer:

- *Blind Self-Modification*: Wave k performs a blind self modification on wave k' if instructions in k' have been written but not read by k:

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

- *Decryption*: Wave k decrypts wave k' if instructions in k' have been both read and written by k:

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

- *Integrity Checking*: Wave k checks the integrity of wave k' if it reads instructions in wave k' that were not written by waves between k and k':

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

- *Code Scrambling*: Wave k is scrambled by wave k' if instructions in k have been written by k' for $k < k'$:

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

Of course, we can define other behavior patterns. In all cases, a trace satisfies a behavior pattern A, $A \in \{\textit{Blind}, \textit{Decrypt}, \textit{Check}, \textit{Scrambled}\}$, if $\cup_{k,k'} A(k,k')$ is not empty.

The algorithm we use to detect the code protection patterns works in time $O(n^2.m)$ where n is the number of waves and m is the size of the waves. In most cases, the number of waves is relatively low and can be considered constant. Therefore, the average complexity of the code protection detection is in $O(m)$.

1.5 Other Analyses Performed

In addition to the code protection patterns based on code waves (algorithmic protection), we can also be interested in more technical anti-reversing techniques such as anti-debugging and anti-virtualization techniques. Since we use VMware to run the samples, we were particularly interested in preventing the detection of VMware by the samples.

TraceSurfer detects common techniques for virtualization detection such as RedPill [24] and its variants using the SLDT and SGDT instructions as well as the VMware Channel technique used in ScoopyNG [19] for instance. Using dynamic binary instrumentation, it is fairly straightforward to detect the specific instructions used in these tests (we can therefore raise alerts when such techniques are detected), but we can also change the output of these instructions so that the attempt to detect the virtual machine monitor fails. As a result, we can effectively hide the presence of the VMM and thus enhance the transparency of virtualization. However, the transparency is still far from being perfect and there are many other ways to detect the presence of virtualization [11], some of them being impossible to counter effectively [2, 13].

To sum things up, we can currently detect and counter the following virtualization detection techniques:

- RedPill and its variants (SIDT, SLDT, SGDT)
- STR
- the VMware channel

Though we did not implement it yet, we consider adding support for the detection of:

- other virtualization detection techniques (for VirtualPC, VirtualBox, Xen and so on)
- anti-debugging techniques
- anti-sandbox techniques. Some packers include countermeasures for online malware analysis services, such as Anubis [3], Joebox [7], CWSandbox [25], etc. These techniques are particularly suspicious and would be worth detecting.

1.6 Implementation

TraceSurfer is currently implemented using dynamic binary instrumentation. It is made of 620 lines of C++ written as a plugin for Pin (a pintool in Pin jargon). Dynamic binary instrumentation has many advantages, such as full control over the binary and the ease with which one can write new analysis tools. But it also has several drawbacks such as imperfect transparency [6] and a potentially massive slowdown of the target binary compared to native execution.

It is also possible to compute the same output as TraceSurfer by using a full system emulator such as QEMU [4]. The slowdown is presumably worse than with DBI but the transparency problem is mostly solved.

Additionally to the transparency problem, our current implementation has the following limitations:

- no support for 64 bit binaries
- no support for kernel mode instrumentation
- limited support for multithreading

2 Experimental Protocol

To assess the effectiveness of our tool, we tested it on several thousands of potentially malicious binaries. The aim of this experience was to detect which samples are protected, the kind of protection used (packing, anti-virtualization, integrity checking...), and the scalability to high volumes of binaries.

2.1 Samples Selection

We deployed a Nepenthes powered honeypot [1] which collected thousands of potentially malicious files. Since TraceSurfer only works with Win32 executable binaries, we had to eliminate the samples reported by the `file` command as not DOS/Win32 executable. This selection pass returned 59,554 binaries out of the initial 62,498 (see Subsection 3.1). These samples were then ready to be sent on the cluster for execution.

2.2 Execution on the cluster

We use a 12 nodes cluster running under Ubuntu Linux 8.04 (with two Quad Core Xeon L5420 and 16 Gb of RAM on each node) dedicated to this type of experiment. It was configured with the following requirements:

- Isolation: we must be able to run malware on the cluster without compromising it. Additionally, malware must not spread on the network.
- Scalability: we designed the architecture for X nodes and Y virtual machines per node. It should be easy to add or remove nodes and virtual machines.
- Automation: there should be no need for human intervention during the analysis.

The isolation requirement is fulfilled by the virtualization layer, the locked snapshots (so that each virtual machine boots to a clean state) and the network isolation of the cluster. We have deployed a virtual machine with Windows XP Professional x64 Service Pack 2, Pin and an SSH server. This machine runs under VMware Server 2.0 with a virtual Ethernet card in HostOnly mode to allow ssh connections between the host and the guest systems. VMware was selected among other virtualization systems because the server version is free of charge and for the VMware VIX API that allowed us to interact with the guest machine from the host. For instance, we use this API to power on a virtual machine for each binary and to power it off once the analysis is finished.

The virtual machine was deployed twice on each of the 12 nodes. Then, we use a program to launch the analysis of the samples and distribute them between our 24 virtual machines. The functioning of the program is quite simple: for each sample, a virtual machine is started and the file is uploaded with an sftp connection. Then, we launch TraceSurfer on the sample with an ssh connection and when it returns or times out, the report file is retrieved with sftp. Finally, the virtual machine is stopped, discarding all the changes on the guest operating system. Synchronization between the cluster nodes is performed by a simple lockfile mechanism to prevent multiple analyses of the same binary. This program is asynchronously launched once for each virtual machine on the nodes (2 times in our case).

To fulfill the automation requirement, we had to implement timeout mechanisms, mostly due to samples going resident in memory or waiting for user input. There is an *internal* timeout mechanism implemented

Nb of files from the honeypot	62,498	
Nb of executable files	59,554	
Detection by Kaspersky AV	58,089	97.54%
Detection by ESET NOD32	57,685	96.86%

Tab. 1: Honeypot Samples

in TraceSurfer and an *external* timeout mechanism on each node to shut down TraceSurfer and the virtual machine in case anything goes wrong.

3 Results

3.1 A Bird's-Eye View of the Samples Set

Table 1 gives a high-level view of our input set.

We ran two commercial anti-virus scanners on the 59,554 executable files. The detection rates of 97.54% and 96.86% confirm the intuition that the executables are mostly malware.

We were able to analyse correctly[1] 48,404 (81.28%) of these binaries with TraceSurfer in approximately 34 hours and 10 minutes, with a timeout of 90 seconds for each sample. Out of these binaries, 13,409 (27.70%) were stopped because of the timeout mechanism.

TraceSurfer failed to analyse 18.72% of the samples for different possible reasons:

- Pin fails to instrument some aggressively protected binaries
- some binaries are unsupported by the current version of TraceSurfer (rootkits, 64 bit binaries)
- some PE files are broken, or crash even with no instrumentation
- some binaries may be written for a specific version of Windows and will not run on our virtual machines

See Subsection 3.2 for a more detailed analysis of the output of TraceSurfer.

Finally, we ran pefile [8] with approximately 2,600 signatures on our samples set. The notable result is that no packer is detected in the vast majority of the samples (97.08%).

[1] We consider the analysis correct if TraceSurfer gives at least a partial output on a given binary. Therefore "correct" does not imply "complete". It is correct in the sense that the number of waves and features reported by TraceSurfer is conservative, but in some cases features can not appear in the report, for instance when a crash occurs before a partial report has been printed.

Nb of Waves	Nb of Binaries	% of Analysed Files
1 wave	318	0.66%
2 waves	4,184	8.64%
3 waves	516	1.07%
4 waves	589	1.22%
5 waves	42,455	87.71%
6 waves	86	0.18%
7 waves	41	0.08%
8 waves	92	0.19%
9 waves	10	0.02%
10 waves	38	0.08%
11 waves	32	0.07%
12 waves	40	0.08%
14 waves	2	0.00%
15 waves	1	0.00%

Tab. 2: Code Waves Analysis in Honeypot Samples by TraceSurfer

3.2 Detailed Analysis

Table 2 shows the number of code waves found by TraceSurfer. The proportions in this table are based on the 48,404 files analysed correctly by TraceSurfer.

We can note that:

- 99.34% of the analysed samples use dynamically generated code (i.e. 2 waves or more). They are not necessarily all packed with a conventional packer, but they almost all use at least some form of self-modification.
- the number of waves is relatively low, since the maximum is 15.
- 87.71% of the analysed samples use 5 waves. This is quite a surprise, the original assumption was to find a peak at 2 waves since most simple packers work with only 2 waves. The cause of this peak might be that a 5-waves packer is over-represented in our samples set, maybe due to the particular configuration of the honeypot.

Table 3a shows the different behavior patterns detected, as defined in Subsection 1.4.

We can note that:

- most binaries use both types of dynamic code: decryption (91.00%) and blind self-modifying code (90.50%)
- the number of binaries that use integrity checking is surprisingly high (88.14%). The reason is probably that the over-represented 5-waves packer in our samples set also uses integrity checking.

Table 3b shows the different anti-virtualization techniques used. Very few samples use such techniques (0.15%), and most of them only use the SIDT (also known as RedPill) detection method.

3.3 Normal Files Analysis

We now run TraceSurfer on a small set of normal programs (i.e. supposedly not malicious). We selected the 467 unique .exe files on a clean install of Windows XP. We were able to analyse 388 of these files (83.08%) in the same conditions as the honeypot samples in approximately 30 minutes.

Table 4a shows that dynamic code (i.e. more than 1 code wave) was found in 66 of these 388 files (17.01%). The most probable explanation for this is that they are .NET programs and TraceSurfer detects the effect of the .NET Just-In-Time compiler.

As expected, no anti-virtualization technique was detected.

4 Limitations and Further Research

Our approach suffers from some limitations:

- the *timeout* mechanism is a typical problem for dynamic analysis tools. A potential solution would be to use multiple-path exploration [21], but this is a hard problem.
- the *transparency* of the underlying tool: our model works on instruction-level traces and is thus very generic. However, we chose DBI to extract the run trace, and it requires heavy modifications of the target binary. These modifications can be detected, and as a consequence the analysis might not be transparent. The solution would be to extract the run traces with a different mechanism, such as a CPU emulator.
- *kernel-mediated writes*: since we only instrument userland code, memory areas written by the kernel will not be visible in the run trace. Therefore, we might miss some dynamic code generated with system calls. A workaround would be to monitor system calls that write in the process address space or to use full system emulation.
- *heuristic* labelling: the informal naming of behavior patterns can in some cases not correspond to the intuition. For instance the "code decryption" tag can be attributed to a memory address that has been read, written and then executed but with no actual data flow between the read and the write. The solution would be to actively follow data flow dependencies.

Conclusion

We defined a theoretical framework for the analysis of programs with dynamically generated code, and we defined a taxonomy of self-modifying behaviors based on this framework. We also developed a prototype implementation for run time analysis of potentially malicious files that can automatically detect the use of suspicious behaviours. Finally, we distributed this implementation on a cluster that can currently analyse about 1,400 binaries per hour. In future versions, we consider adding detections for more suspicious behaviours, such as anti-debugging and anti-sandboxing techniques.

Another future research area would be the use of TraceSurfer to automatically generate accurate behavioural signatures based on memory access patterns and to implement malware detection mechanisms based on these signatures.

Acknowledgements

We would like to thank Vincent Mussot for his implementation of virtualization counter-countermeasures in TraceSurfer and his work on DBI transparency.

References

[1] P. Baecher, M. Koetter, M. Dornseif, and F. Freiling. The nepenthes platform: An efficient approach to collect malware. In *In Proceedings of the 9 th International Symposium on Recent Advances in Intrusion Detection (RAID*, pages 165–184. Springer, 2006.

[2] E. Barbosa. Blue pill detection. In *SyScan*, 2007.

[3] U. Bayer, C. Kruegel, and E. Kirda. Ttanalyze: A tool for analyzing malware, 2006.

[4] F. 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.

[5] L. Boehne. Pandora's bochs: Automatic unpacking of malware. 2008.

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

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

Code Protection Used	Nb of Binaries	% of Analysed Files
Code decryption (*Decrypt*(k, k'))	44,046	91.00%
Blind self-modifying code (*Blind*(k, k'))	43,805	90.50%
Integrity checking (*Check*(k, k'))	42,665	88.14%
Code scrambling (*Scrambled*(k, k'))	601	1.24%

(a) Memory Access Patterns Detected in Honeypot Samples by TraceSurfer

Anti-Virtualization Technique used	Nb of Binaries	% of analysed files
At least one	71	0.15%
SIDT anti-virtualization technique	65	0.13%
SLDT anti-virtualization technique	0	0.00%
SGDT anti-virtualization technique	0	0.00%
STR anti-virtualization technique	0	0.00%
VMWare channel anti-virtualization technique	14	0.03%

(b) Anti-Virtualization Techniques Detected in Honeypot Samples by TraceSurfer

Tab. 3: Honeypot Samples Analysis

Nb of Waves	Nb of Binaries	% of Analysed Files
1 wave	322	82.99%
2 waves	66	17.01%

(a) Code Waves Analysis in Normal Files by TraceSurfer

Code Protection Used	Nb of Binaries	% of Analysed Files
Code decryption (*Decrypt*(k, k'))	61	15.72%
Blind self-modifying code (*Blind*(k, k'))	1	0.26%
Integrity checking (*Check*(k, k'))	0	0.00%
Code scrambling (*Scrambled*(k, k'))	0	0.00%

(b) Memory Access Patterns Detected in Normal Files by TraceSurfer

Anti-Virtualization Technique used	Nb of Binaries	% of analysed files
At least one	0	0.00%

(c) Anti-Virtualization Techniques Detected in Normal Files by TraceSurfer

Tab. 4: Normal Files Analysis

[8] E. Carrera. 4 x 5: Reverse engineering automation with python. In *Black Hat USA*, 2007.
[9] E. Carrera. Malware - behavior, tools, scripting and advanced analysis. In *HITB*, 2008.
[10] S. Cesare. Security applications for emulation. In *Ruxcon*, 2008.
[11] P. Ferrie. Attacks on virtual machine emulators. In *AVAR Conference*, 2006.
[12] P. Ferrie. Anti-unpacker tricks. 2008.
[13] E. Filiol. Formal model proposal for (malware) program stealth. In *Virus Bulletin*, 2007.
[14] T. Graf. Generic unpacking – how to handle modified or unknown pe compression engines? In *Virus Bulletin*, 2005.
[15] F. Guo, P. Ferrie, and T.-C. 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.
[16] S. Josse. Secure and advanced unpacking using computer emulation. *Journal in Computer Virology*, 3, 2007.
[17] S. Josse. Analyse et detection dynamique de code viraux dans un contexte cryptographique, 2009. Ph.D. Thesis, Ecole Polytechnique.
[18] M. G. Kang, H. Yin, and P. Poosankam. Renovo: A hidden code extractor for packed executables. In *5th ACM Workshop on Recurring Malcode*, 2007.
[19] T. Klein. Scoopyng - the vmware detection tool, 2008. `http://www.trapkit.de/research/vmm/scoopyng/index.html`.
[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, K. Hazelwood, and V. J. Reddi. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation (PLDI)*, 2005.
[21] A. Moser, C. Kruegel, and E. 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.
[22] D. Quist and Valsmith. covert debugging, circumventing software armoring techniques. In *Black Hat USA*, 2007.
[23] P. Royal. Alternative medicine: The malware analyst's bluepill. In *Black Hat USA*, 2008.
[24] J. Rutkowska. Red pill... or how to detect vmm using (almost) one cpu instruction, 2004. `http://www.invisiblethings.org/papers/redpill.html`.
[25] 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.