Hunting Trojan Horses

Micha Moffie
Computer Architecture Research Laboratory
Northeastern University
Boston, MA 02115, USA
mmoffie@ece.neu.edu

Winnie Cheng
Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
wwcheng@mit.edu

David Kaeli
Computer Architecture Research Laboratory
Northeastern University
Boston, MA 02115, USA
kaeli@ece.neu.edu

Qin Zhao
Singapore-MIT Alliance
National University of Singapore
Singapore 117576
zhaoqin@nus.edu.sg

ABSTRACT

HTH (Hunting Trojan Horses) is a security framework developed for detecting difficult types of intrusions. HTH is intended as a *complement* to anti-virus software in that it targets unknown and zero-day Trojan Horses and Backdoors. In order to accurately identify these types of attacks HTH utilizes runtime information available during execution. The information collected includes fine-grained information flow, program execution flow and resources used.

In this paper we present *Harrier*, an Application Security Monitor at the heart of our HTH framework. Harrier is an efficient run-time monitor that dynamically collects execution-related data. Harrier is capable of collecting information across different abstraction levels including architectural, system and library APIs. To date, Harrier is 3-4 times faster than comparable information flow tracking systems.

Using the collected information, Harrier allows for accurate identification of abnormal program behavior. Preliminary results show a good detection rate with a low rate of false positives.

General Terms

Security

Keywords

Program monitoring, Run time environment, Information flow control, data labeling

ASID'06 October 21, 2006, San Jose, California, USA.

1. INTRODUCTION

Criminal computer attacks grew at an alarming rate in 2004 [17]. In their 2005 Internet security threat report [21] Symantec reported that out of the top ten spyware programs, six were bundled with other programs. The report further noted that malicious code was being used for financial gains in a number of cases and that there was a noticable rise in the occurrence of malicious code that exposes confidential information. Symantec indicated that latter type of malicious code comprised 74% of the top 50 code samples reported to them in 2005 [21].

Guarding against malicious code that can either expose confidential information, or tamper with information is challenging. These exploits may take the form of Trojan Horses or Backdoors that are installed without the user's consent. Moreover, new malicious code (e.g. zero day attacks) can go undetected by even the most up-to-date anti-virus programs. Some Trojan Horses and Backdoors conceal their presence by executing as a plug-in or as a dynamically-linked library. Many have very little immediate impact on the normal operation of a system and so are difficult to detect by the user. These characteristics enable Trojan Horses and Backdoors to go undetected for a significant period of time, providing the attacker with a large window of vulnerability on the system.

To address these difficulties in identifying Trojan Horses and Backdoors, we have developed Harrier. Harrier is the centerpiece of our security framework HTH. It is capable of collecting data flow and control flow information during the execution and enables us to defend against harmful activity.

The remainder of the paper is organized as follows: In section 1.1 we highlight the characteristics of Trojan Horses and Backdoors that motivate our design choices. In section 2 we present HTH. We then give a high-level overview of our security policy in section 3. In section 4 we delve into the design and implementation of Harrier, and evaluate its performance in section 5. We review related work in Section 6 and then summarize our work, highlighting areas for future investigation, in section 7.

1.1 Security Exploits

We begin by discussing several examples of real-world Trojan Horses and Backdoors:

1. PWSteal.Tarno.Q is a Trojan Horse that logs passwords and information typed into web forms. It propagates via email attachments and registers itself as a browser helper. The Trojan monitors a subset of web pages (such as those that contain strings such as: bank, cash, gold and more) and captures keystrokes, as well as information submitted in several predefined files. The information is periodically transmitted back to the attacker via a hard-coded url [20].

2. The Trojan.Lodeight.A code tries to install malicious code on the compromised computer and opens a Backdoor on TCP port 1084. When this Trojan is executed, it connects to one of two predefined websites, downloads a remote file and then executes it. This remote file may be any arbitrary program, including a Beagle worm [20].

3. As part of an adware program, the Trojan.Vundo presents the user with pop-up advertisements. There are two components to this Trojan: a downloader and an adware injector. By exploiting a Microsoft Internet vulnerability, the downloader component is executed on the victim. It then retrieves the adware component by connecting to a specific IP address. The adware is injected into different processes as a DLL. Besides displaying advertisements on the infected machine, it also degrades performance by decreasing the amount of virtual memory available [20].

We summarize the execution patterns exhibited by a set of known malicious examples in Table 1.

Our analysis of the Trojan Horses and Backdoors mentioned above, as well as other malicious code examples reveals several distinct characteristics and behaviors:

1. The malicious code is executed without user intervention.

2. The malicious code may be directed by a remote attacker once a connection is made.

3. Resources used by the malicious code, such as file names and network addresses, are hard-coded in the binary.

4. OS resources (processes, memory) used by the malicious code may be consumed for the purpose of degrading performance.

A key characteristic of Trojan Horses and Backdoors is that they cannot be invoked by the attacker and are autonomous – at least until a connection is made.

2. HTH FRAMEWORK

In HTH, our security policy implementation is separated from our tracking mechanism to allow for flexibility and independent optimization of each component. Figure 1 shows HTH's high-level software architecture.

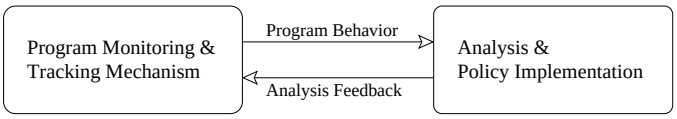

Figure 1: The HTH software architecture.

2.1 Data Sources

A key point in our framework is its ability to track *information flow* in the program and identify the exact source of the data. This capability is not only important for tracking information leakage, but also allows us to detect occurrences where the malicious code is autonomous (e.g., where the exploit uses hard-coded values).

Our policy rules take into account the source of the data. We maintain enough information about each data source in order to make fine-grained distinctions. This differientiates our approach from several solutions that offer a single "Taint" bit [16, 15, 19, 3]. We divide data sources into 5 resource types, as shown in Table 2. For some resource types, a resource specifier field is provided to include extra information on the origin of the resource. For example, for a file `/tmp/0a3f2d.txt` that belongs to resource type `FILE`, the specifier field includes the resource type of its filename. (e.g., BINARY, if the filename is a hard-coded value).

Resource Type	*Description*
USER_INPUT	data is retrieved via user interaction
FILE	data is read from a file
SOCKET	data is retrieved from a socket interface
BINARY	data is part of the program binary image
HARDWARE	data originated from hardware (e.g. cpuid)

Table 2: Resource types in HTH.

2.2 Event Monitoring

There are numerous events that need to be monitored to accommodate the information collected for our policy. We divide the events into 3 categories:

- Architectural (ISA) events - (instructions executed),
- OS (API) events - system calls, and
- Library (API) events - library routines.

Events such as OS and library calls allow us to collect information related to the program semantics and program information flow. Architectural events allow us to collect information related to program information flow. Carefully categorizing these events emphasizes the need to accommodate different levels of abstraction in our system.

2.3 Design choices

In this initial implementation, our goal was to keep the monitor as lean, and generic as possible. Analyzing source code would limit our analysis to one particular language and would require a specialized front-end for each language. There is also the issue that the source code may not be available. Therefore, we have designed Harrier to work with program binaries. This choice does tie us to a specific architecture and OS (an executable format may bound us to a specific OS).

Exploit Name	No user intervention	Remotely directed	Hard-coded Resources	Degrading performance
PWSteal.Tarno.Q	√		√	
Trojan.Lodeight.A	√	√	√	
Trojan.Vundo	√		√	√
W32.Mytob.J@mm	√	√	√	
Windows-update.com	√		√	
W32/MyDoom.B	√	√	√	
Phatbot	√	√	√	
Sendmail Trojan	√		√	
TCP Wrappers Trojan	√		√	

Table 1: Execution patterns exhibited by malicious code.

In order to maintain runtime information (user input, etc.) and enable us to perform detailed and accurate behavior analysis, we implement Harrier as a runtime monitoring system. Future research will look into developing hybrid approaches in which static analysis may be used to accelerate the runtime monitor.

Although we could potentially track all events at every abstraction level (architectural, OS, and library), it is desirable to reduce the number of events monitored to a generic and preferably small set (e.g., tracking all library APIs may very well be intractable). Since Harrier analyzes the program binary and does not rely on source code, we are only able to monitor calls that are made to shared objects. We assume that debug information is unavailable in the binary, and thus, restrict our monitoring to shared objects with a defined API. The main reason for tracking a subset of library calls is to overcome the semantic gap introduced by working at either an architectural or OS level.

Harrier is a run time security monitor that analyzes program binaries. Harrier tracks architectural, OS and selected library events. We follow this approach in order to increase both flexibility and usability. Moreover, we expect this choice to provide the best accuracy and the smallest number of false positives.

3. SECURITY POLICY

Based on the unique behavior of Trojan Horses and Backdoors described in section 1.1, we have developed a security policy composed of a set of rules designed to oversee different types of program behavior. We classify rules as one of three types:

- *Execution flow* - Rules that monitor the invocation and execution of new processes. Our target is to detect malicious code being executed.
- *Resource abuse* - Includes allocating and using different resources from the operating system (e.g., processes and memory) with the purpose of draining OS resources and reducing performance. In our policy we monitor the number of new processes created, as well as the rate of creation of new processes.
- *Information flow* - A set of rules that can enforce the flow of information between different sources and targets, classified by the different resource types as discussed in section 2. For example, a rule that notifies the user when information is flowing from a file to a socket (where both, either or none are hardcoded)

4. HARRIER DESIGN AND IMPLEMENTATION

Harrier tracks a program's execution and generates events that are further analyzed. The following sections describe how Harrier monitors and maintains this fine-grained information.

4.1 System calls

Harrier tracks multiple system calls. Monitoring the *clone* and *execve* system calls enables the enforcement of resource abuse and execution flow rules. In the case of an *execve* system call, Harrier will modify the executable target such that a new instance of Harrier is executed and can monitor the new executable.

System calls such as *open*, *close*, *read* and *write* are tracked to provide detailed information about resource types used. Tracking system calls is essential to understanding program behavior, as well as for identify entry and exit points in the data flow.

4.2 Library calls

Occasionally, tracking system calls is not enough to understand the underlying program behavior. We refer to this as a *semantic gap*. Such a semantic gap exists between the socket library API and its system calls. Monitoring a minimal set of library API functions allows us to track the information flow in order to understand the underlying program behavior.

For example, resolving strawberry.mit.edu to its IP address (18.71.0.151) may involve several system calls accessing a local host file, a domain name server, or other methods. Harrier tracks the *gethostbyaddr* and *gethostbyname* routines implemented via a library API to overcome the semantic gap.

4.3 Data flow

Harrier tags each register and memory location with one or more data sources. Whenever an assignment instruction is executed, the tag of the newly assigned memory or register

is updated. The new tag will hold a composite of the data sources of the source operands.

Another element of data flow tracking includes locating hard-coded data. This data is embedded in the binary which can be the executable itself or libraries that are dynamically loaded. Harrier monitors binary load events and tags the data with the BINARY resource type.

The use of the *stdin* file descriptor with *read* system calls identifies trivial user input.

Other channels of user input to the program include command line arguments, as well as environment and auxiliary variables (e.g., argc, argv, env, aux). All of this information is stored on the initial stack as the program begins execution. Harrier will tag the initial stack with the USER_INPUT data source. Data originating from hardware (e.g., when the `cpuid` is executed) is assigned the HARDWARE tag.

4.4 Implementation

Harrier can be viewed as a layer that virtualizes an application. It is implemented on top of a dynamic instrumentation framework. We have developed two versions of Harrier to date. The first version was based on PIN [12]. This initial implementation tracks Linux binaries, and served as a proof-of-concept. Our current version is based on dynamoRIO [2] and optimizations used in TaintTrace [3]. While our current implementation only supports Linux, the underlying instrumentation tool can also work with Windows executables.

Since dynamic instrumentation is done 'on top of' the operating system and 'below' the application, it allows us to access events across the different abstraction levels, track data flow, and easily monitor events such as system and library calls. Figure 2 shows the different types events Harrier is able to collect.

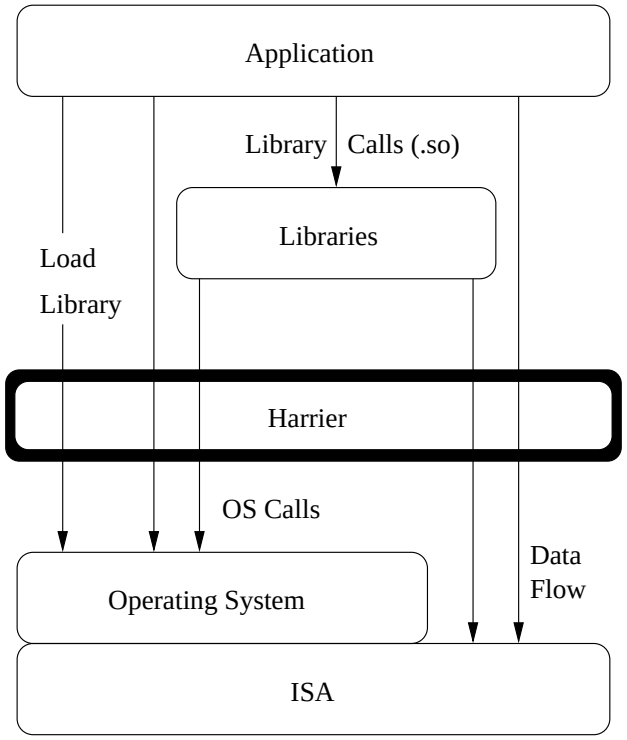

Figure 2: Events collected from different abstraction levels

The application is instrumented at different granularities:

1. Instruction – where we instrument before or after an instruction,
2. Routine – where we instrument on function call or return,
3. (image) Section – where we identify the different sections in a binary, and
4. Image - where we instrument when a binary is loaded or unloaded.

In table 3 we summarize for each policy rule type, the instrumentation granularity used and the information gathered across the different abstraction levels.

Our current implementation provides data flow analysis, as well as selected system and library calls tracking.

5. EVALUATION

The performance of HTH is dependent primarily on the performance of Harrier. In this section, we evaluate the performance of Harrier using the SPEC2000 INT benchmarks. We use the current version of Harrier, which is based on TaintTrace. We ran the benchmarks on a Linux 2.4.20, Intel Xeon (Pentium 4) 3.06GHz CPU.

TaintTrace [3] is an efficient information flow tracing tool that provides instruction-level data tracking. It is built on top of DynamoRIO [2], a dynamic binary instrumentation tool. TaintTrace provides 3 types of data information propagation on x86 instructions: Copy, Arithmetic and Control. TaintTrace uses a shadow memory that allows a fast one-to-one mapping between program memory and data-flow tracking memory. In addition, a number of techniques were used to minimize the instrumentation overhead, such as aggressive use of dead registers to avoid register spilling. TaintTrace is the fastest fine-grain monitoring tool we are aware of. Harrier is an improvement over TaintTrace to provide more well-defined data flow information for effective identification of security threats. In particular, it supports the resource types and identification fields discussed earlier in the paper.

In figure 3 we present the overhead introduced by Harrier (based on TaintTrace). For each of the benchmarks, we show the relative execution time of Harrier monitoring the benchmark (with the raw execution time of the benchmark as 100%).

Harrier introduces an average of 7X slowdown. These results are significantly better than those reported in previous work. For MemCheck [19] a median of 26.5X slowdown (for 9 SpecINT and 4 SpecFP benchmarks) was reported. Results reported for TaintCheck [15] showed around 24X slowdown for the bzip2 benchmark (the only comparable result reported).

6. RELATED WORK

A lot of work has been done in the areas of intrusion detection systems and static and dynamic information flow systems. Due to the limited space we present only closely related work.

Policy rule type	Instrumentation granularity	Information gathered
Architectural events		
Information Flow	Instruction	Data Flow (reg/mem, mem/mem, reg/reg)
Information Flow	Instruction	Hardware Information (CPUID)
OS (API) events		
Execution Flow	Instruction	System Calls (execve)
Resource Abuse	Instruction	System Calls (clone)
Information Flow	Instruction	System Calls (IO read/write)
Information Flow	Section	Binary load
Information Flow	Image	Binary load
Information Flow	Instruction	Initial stack location
Library (API) events		
Information Flow	Routine	Data Flow (Closing the semantic gap)

Table 3: Information gathered at different instrumentation granularities.

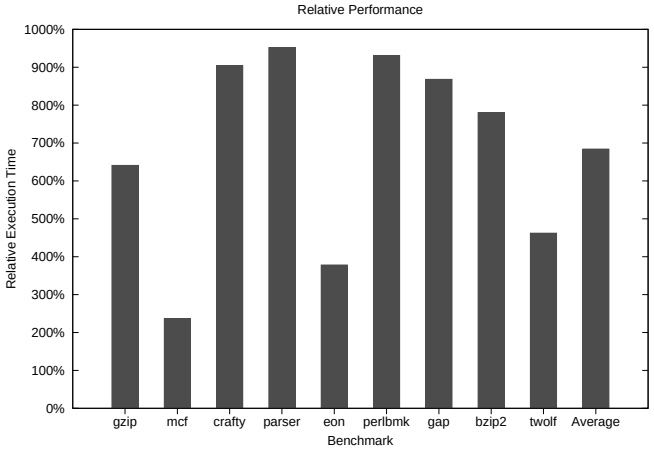

Figure 3: Relative performance of Harrier version 2.0(based on TaintTrace).

6.1 Intrusion Detection Systems

HTH builds on top of much work previously done. In particular system call monitoring was successfully used to detect malicious code in many Intrusion Detection Systems [8, 24, 11, 10, 7, 6, 18, 5]. Because system calls provide insight into program behavior and system call tracing can be done very efficiently, system call monitoring is a key aspect in many Intrusion Detection and Anomaly detection systems (as well as in HTH). A discussion of many host-based anomaly detection systems can be found in Gao et al. [6].

A specialized type of data flow, control flow, can be used to thwart attacks that change the control flow dynamically. Kiriansky et al. [9] introduced a control flow monitoring system called *Program Shepherding*.

Static analysis can also be used for intrusion detection. Wagner et al. [23] show how static analysis can be used to thwart attacks that change the program behavior from the original behavior intended by the program author. Unfortunately, this analysis assumes that the program was written with benign intent. This isn't the case for Trojan Horses or Backdoors.

6.2 Information Flow Systems

Information flow security systems have focused on language-based and static analysis mechanisms [4, 13]. These systems only allow the programmer to specify the policy. This means that the user places his trust in the code developer and is not able to enforce his own security policy.

In contrast, systems such as RIFLE [22] (an architectural framework) and Perl's interpreter [16] track the data flow dynamically and equip the user (in contrast to the programmer) with a practical way of enforcing any information flow policy.

Data Flow tracking systems based on binary rewriting or instrumentation are closely related to Harrier. Valgrind [14] has been used to rewrite the binary during runtime to dynamically check for overwrite attacks [15]. It can also detect undefined value errors at the bit level [19]. Beres et al. [1] describe an information flow monitoring approach based on dynamic label binding and code rewriting. The MIT TaintTrace project [3] uses binary rewriting to track user input (tainted data) very efficiently. TaintTrace introduces an average slow down of 5.5 times compared to native execution. Our work in HTH builds on top of TaintTrace's efficient data flow tracking.

Run time information systems are becoming more prevalent in secure systems. Many runtime systems specialize in tracking the source of data (e.g., user input), and develop security policies for common exploits. In HTH we consider different sources of data and dynamically track all of them in order to support our policy. Moreover, our current HTH implementation allows us to track data flow faster than comparable solutions.

7. SUMMARY AND FUTURE WORK

In this paper we presented HTH a security framework capable of detecting zero-day attacks. Within HTH, Harrier (which is based on TaintTrace), introduces an average overhead which is significantly less than any other current comparable system.

Harrier is able to collect vast amounts of information and allows for accurate identification of abnormal program behavior. Preliminary results are promising. A good detection rate coupled with a low false positive rate push us to continue developing HTH.

Future work will expand and improve upon HTH. First, we will explore architectural support such as RIFLE to speed up Harrier. Second we aim to monitor process communications as well as save meta-information regarding each

process behavior between executions. Lastly, we intend to present a complete security policy within HTH and show a complete evaluation of its effectiveness including the detection rate and the false positive rate.

8. ACKNOWLEDGMENTS

This work has been supported by the National Science Foundation under Grant No. 0310891.

9. REFERENCES

[1] Y. Beres and C. I. Dalton. Dynamic label binding at run-time. In *NSPW '03: Proceedings of the 2003 workshop on New security paradigms*, pages 39–46, New York, NY, USA, 2003. ACM Press.

[2] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[3] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *Proc. 11th IEEE International Symposium on Computers and Communications*. IEEE, Jun. 2006.

[4] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[5] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*, pages 38–48, 1998.

[6] D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *Proceedings of the 13th USENIX Security Symposium*, pages 103–118, San Diego, CA, USA, Aug. 9-13 2004.

[7] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, 1996.

[8] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[9] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Security '02: Proceeding of the 11th USENIX Security Symposium*, San Francisco, August 2002.

[10] C. Ko, T. Fraser, L. Badger, and D. Kilpatrick. Detecting and countering system intrusions using software wrappers. In *Proceedings of the USENIX Security Conference*, pages 145–156, Jan. 2000.

[11] A. P. Kosoresow and S. A. Hofmeyr. Intrusion detection via system call traces. *IEEE Softw.*, 14(5):35–42, 1997.

[12] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instruentation. In *Programming Language Design and Implementation (PLDI)*, Jun. 2005. Chicago, IL.

[13] A. C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM Press.

[14] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Electronic Notes in Theoretical Computer Science*, volume 89, pages 1–23. Elsevier, 2003.

[15] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *The 12th Annual Network and Distributed System Security Symposium*, Feb. 3-4, San Diego, CA, USA, 2005.

[16] perldoc.perl.org. Perl 5.8.7 documentation, perlsec - perl security. http://perldoc.perl.org/perlsec.html.

[17] Bruce Schneier. Attack trends 2004 and 2005. In *ACM Queue vol. 3, no. 5*. ACM, Jun. 2005. http://acmqueue.com/.

[18] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference*, page 209, Washington, DC, USA, 2002. IEEE Computer Society.

[19] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX 2005 Annual Technical Conference*, pages 17–30, Apr. 10-15, Anaheim, CA, USA, 2005.

[20] Symantec. Symantec security response. http://securityresponse.symantec.com/avcenter/, 2004-2005.

[21] Symantec. Symantec internet security threat report, trends for january 05 - june 05, 2005.

[22] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO 37: Proceedings of the 37th annual International Symposium on Microarchitecture*, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.

[23] D. Wagner and D. Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 156, Washington, DC, USA, 2001. IEEE Computer Society.

[24] C. Warrender, S. Forrest, and B. A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.