Security Breaches as PMU Deviation: Detecting and Identifying Security Attacks Using Performance Counters*

Liwei Yuan, Weichao Xing, Haibo Chen, Binyu Zang
Parallel Processing Institute
Fudan University
{yuanliwei, wcxing, hbchen, byzang}@fudan.edu.cn

ABSTRACT

This paper considers and validates the applicability of leveraging pervasively-available performance counters for detecting and reasoning about security breaches. Our key observation is that many security breaches, which typically cause abnormal control flow, usually incur precisely identifiable deviation in performance samples captured by processors. Based on this observation, we implement a prototype system called Eunomia, which is the first non-intrusive system that can detect emerging attacks based on return-oriented programming without any changes to applications (either source or binary code) or special-purpose hardware. Our security evaluation shows that Eunomia can detect some realistic attacks including *code-injection attacks*, *return-to-libc attacks* and *return-oriented programming attacks* on unmodified binaries with relatively low overhead.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*unauthorized access*

General Terms

Security

Keywords

Performance Counters, Return-oriented Attacks, Return-to-libc Attacks, PMU Deviation

1. INTRODUCTION

Security breaches are a major threat to the dependability of computer systems and can cause not only economic effect but also social impact. However, many previous mitigation approaches are either heavyweight in requiring non-trivial modifications to processor architectures [7], or intrusive in requiring source-code or binary modifications to applications [15]. Further, some emerging attacking approaches such as return-oriented programming [11] are usually difficult to be detected using existing techniques.

In this paper, we propose a non-intrusive approach to detecting a series of security breaches, which requires no changes to existing hardware, source code or binaries. The approach we propose, namely Eunomia, leverages the pervasively available performance monitoring units (PMUs) in commercial processors, to monitor performance deviation of a running application to detect possible attacks.

The key observation is that security breaches usually cause abnormal control flow that can be precisely captured by PMUs in processors. For example, attacks involving code-injection usually need to transfer control flow to injected code, while return-to-libc attacks and return-oriented programming need to break the control flow integrity of a program.

To validate our observation, we design and implement a prototype called Eunomia, to detect a series of attacks, including both code-injection and return-based[1] attacks. To detect such attacks, Eunomia continuously monitors performance samples using specific events during the program execution and correlates the events with the program execution context (such as instruction addresses and callers of library functions). Any deviation in performance samples can be used as signs of possible attacks. Upon the detection of an attack, the performance samples information such as *branch traces* can be used to locate the exploited security vulnerability and determine how the vulnerability is exploited.

To confirm the effectiveness of Eunomia, we have conducted some preliminary security tests using real-world vulnerabilities. Our evaluation results indicate that Eunomia can precisely detect the attacks at the first time they happen. Performance evaluation results show that Eunomia incurs small performance overhead for real-world applications.

2. SECURITY BREACHES AS PMU DEVIATION

2.1 Candidate PMU Features

There are a number of standard PMU features such as recording the occurrences of branch miss predictions, instruction TLB (iTLB) misses and L2 cache misses. Further, recent processors also provide several advanced features. The followings list two features from recent Intel processors:

*This work was funded by China National Natural Science Foundation under grant numbered 90818015 and 61003002, and a grant from the Science and Technology Commission of Shanghai Municipality numbered 10511500100.

APSys'11, July 11–12, 2011, Shanghai, China.

[1]We consider return-to-libc attack and return-oriented programming attack as return-based attacks

Precise Event Based Sampling (PEBS): In PEBS, performance samples are directly recorded into a predefined memory region when the occurrences of the event exceed a threshold, instead of generating an interrupt immediately. An interrupt is generated when predefined memory region is full. This mechanism enhances performance monitoring performance by batching each sample into a predefined buffer and processing them together. With the *atomic-freeze feature*, the reported IP in predefined memory is precisely the instruction immediately following the instruction causing the event.

Branch Trace Store: BTS provides a mechanism to capture control transfer events including all types of jump, call, return, interrupt and exception. The recorded information includes the source address, target address and properties of the control transfer. Thus, it provides the ability to trace the control flow of program execution. In Intel Core and Core i7, the branch trace information is generated and delivered to the system bus, and then directly recorded into a predefined memory region, similar to PEBS.

2.2 PMU Deviation in Common Attacks

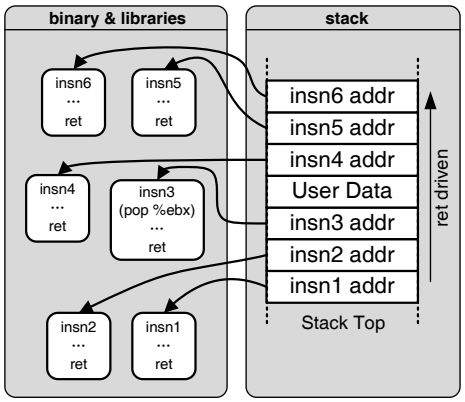

Figure 1: An example illustrating return-oriented programming.

Currently, there are two typical types of attacks according to the means to gain control of a program: code-injection attack, which directly injects external shellcode to a program; and return-based attack, which reuses existing binary code as shellcode. Code-injection attacks usually inject external shellcode to a program by exploiting some vulnerabilities, such as buffer overflow, format string and dangling pointer, and then overwriting a return address or a function pointer to transfer control to the injected code.

For return-based attacks, we further divide them into two types: return-to-libc attacks and return-oriented programming attacks. Return-to-libc attacks directly exploit security-sensitive library routines such as "execve" with attacker-supplied arguments (e.g., a root shell) to gain control. By contrast, return-oriented programming reuses binary code fragments in existing binary to form a shellcode. The essence of return-oriented programming is illustrated in Figure 1, which leverages control of the call stack to indirectly execute cherry-picked machine instructions or groups of machine instructions immediately prior to the "ret" in subroutines within existing binary. The composed code may act as a typical shellcode that binds a tcp to open a remote root shell upon connection.

According to our analysis, these attacks above, no matter how they exploit vulnerabilities (e.g., format string or buffer overflow), usually behave different in performance samples. Hence, unlike some prior attack detection schemes that focus on a particular security vulnerability, our method detects attacks not according to which vulnerability they exploit, but how they behave abnormally in performance samples. We find that both the two types of attacks result in deviation in control flow, thus incur some precisely identifiable performance samples. Table 1 summarizes the performance deviation in such attacks and how the attacks can be monitored with PMU events. As the table shown, there are some PMU events to detect each type of these attacks. For code-injection attacks, the injected code is executed on data sections (e.g., stack or heap), thus results in deviation for program counters in *branch_miss_predict*, *itlb_misses* and *branch traces*. For those reusing library routines or binary code, they will result in abnormal control flow in attacking runs that violate the control flow integrity. This will be reflected by abnormal performance samples using the BTS mechanism in x86 architecture.

2.3 Detecting Attacks Using Performance Counters

Detecting Code-injection Attacks: For attacks that inject shellcode and cause control transfer to the injected code in data sections (e.g., stack or heap), there will be abnormal performance samples such as *itlb_misses* in data sections. This provides a strong evidence of possible attacks.

Using this method to defend against code-injection attacks has similar protection ability with non-execution bit protection, but it is more flexible. Eunomia can do application-specific pattern analysis, to identify the patterns of how an application executes code in data sections and filter them out.

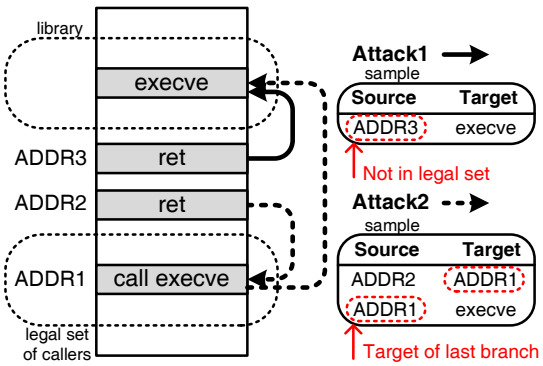

Figure 2: The algorithm to detect return-to-libc attacks.

Detecting Return-to-libc Attacks: For attacks that directly reuse sensitive library routines such as "execve" with attacker-supplied arguments, we need to check the callers of this library routine. Before running a program, Eunomia preprocess application binaries, analyzes the callers of library routines and records callers for each library routine as its legal callers, as shown in Figure 2. As library routines are called from GOT entry[2] whose entry addresses are statically determined, Eunomia can easily determine the legal callers of each library routine. We only need to record legal callers in applications.

During the check, Eunomia uses the branch trace store (BTS) mechanism to record each control transfer and checks the source and target of each transfer in the sample. If the branch target is a library routine, Eunomia checks if the source address is in the legal

[2]GOT entry is short for global offset table, which works together with program linkage table to identify loaded library routine address in dynamic linker.

Attack Type	Description	PMU Events
Code-injection	Inject code and take control transfer to injected code	Branch Tracing Event(BTS) Branch Miss Predict Event Instruction TLB Misses Event
Return-to-libc	Use library calls instead of inject code (e.g., invoke "execve" with "bin/bash")	Branch Tracing Event(BTS)
Return-oriented programming	Use instructions before "ret" in existing library and binary code to form shellcode	Branch Tracing Event(BTS)

Table 1: Deviation in performance characteristics of common attacks.

set of callers and reports an alarm if not. However, attacks can use existing "call" in binary to indirectly invoke a library routine. For such attacks, Eunomia checks one more step further. If the target address in previous sample and the source address in this sample are the same, Eunomia reports an alarm, as illustrated in Figure 2. This method may cause false positives if there are consecutive calls in normal execution, since it behaves like indirect caller of library routine in samples. To filter out this kind of false positive, Eunomia records these consecutive calls when doing binary preprocessing and avoids reporting an alarm in this situation.

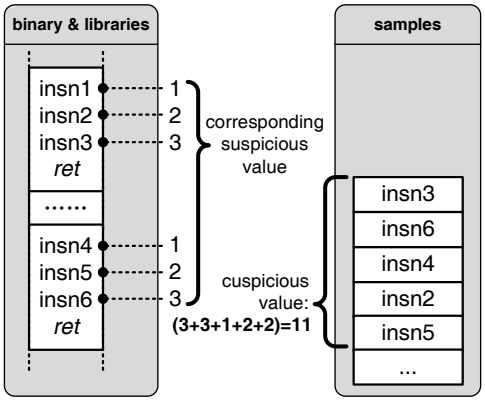

Figure 3: The algorithm to detect return-oriented programming attacks, which uses suspicious value of insns before "ret" and checks the accumulated value against the predefined threshold.

Detecting Return-oriented Programming Attacks: To detect return-oriented programming attacks, Eunomia needs to check the control flow integrity of a program execution. Ideally, Eunomia should record all legal addresses of a "ret" instruction can return to. However, due to the "unintended instruction sequences" in x86, all opcode has the byte "0xc3" could be viewed as the instruction "ret", even if the byte "0xc3" is in a jump instruction (e.g. "jmp 0x3aae9" has opcode "e9 c3 f8 ff ff"). It is usually difficult and time-consuming to build and check with the legal set of control flow graph.

Because most of such attacks need to reuse several instructions before instruction "ret" to form shellcode [14], the target of branches in return-oriented shellcode typically follows a "ret" instruction within usually one to three instructions [3].

Hence, Eunomia exploits the branch tracing ability provided by BTS to detect such attacks. Before running a program, we preprocess the binary, analyze all instructions that appear before "ret"[4]. We assign a suspicious value to each instruction according to their distance to "ret" and store the IP, opcode and the suspicious value into a hashtable, as shown in Figure 3. For dynamically linked libraries, Eunomia also processes them when the dynamic linker loads libraries and updates the hashtable accordingly. During program execution, Eunomia checks the address of each branch target in performance samples to see whether it is in the hashtable. If a consecutive number of branch targets fall into the hashtable and the accumulated suspicious value exceeds a predefined threshold, a return-oriented programming attack has likely occurred. To our knowledge, Eunomia is the first system that can detect return-oriented programming attack without modification to programs [12], binary instrumentation [8] or special-purpose hardware.

Currently, Eunomia does not consider return-oriented programming that uses an update-load-branch sequence [4] such as *pop %eax; jmp ∗%eax*. As the BTS mechanism supports collecting all branch traces, Eunomia should be able to extend Eunomia to detect such a scheme. Eunomia could be enhanced to collect all instructions with *update-load-branch* effect and use the collected branch traces to identify abnormal control transfers, which will be our future work.

3. IMPLEMENTATION OF Eunomia

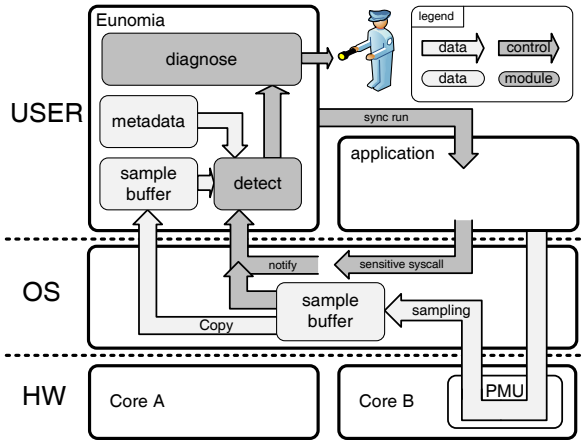

Figure 4: Architecture of Eunomia.

We have implemented a preliminary version of Eunomia based on *perf_events* on Linux kernel version 2.6.34. Currently, Eunomia supports Intel Core Duo and Core i7 processors and focuses on user-level attacks only.

The overall architecture of Eunomia is shown in Figure 4. Eunomia is composed of two parts: the kernel extension that provides interfaces for user tools to set up events and read performance samples; and a user-level tool that uses the performance

[3]Those reusing more than 3 instructions before "ret" are hard or impossible to construct in real-world [14].

[4]This "ret" refers to all "0xc3" including "ret" in "unintended instruction sequence".

samples to detect possible security attacks. The user-level tool is in the form of a monitoring process, which checks the performance samples generated by the application processes. To synchronize the monitoring process with the application processes, the monitoring process is executed as the parent process of the application processes. Eunomia uses ptrace-based techniques to synchronize monitoring process and application processes. To start an application, the monitoring process forks a child process. Before the child uses *exec* to run the application, the child process calls *ptrace* with the *PTRACE_TRACEME* flag to cause the child process to be suspended when *exec* is executed. The monitoring process sets up performance events and sample buffers for the child process and then lets the child process run. Afterwards, these two processes run simultaneously. To reduce interference between the monitoring process and the application processes, Eunomia supports the binding of each process into different cores on multicore hardware.

During normal execution, the event samples of the application processes are written to a performance sample buffer monitored by the monitoring process. As the per-sample check is very costly, Eunomia chooses to check samples in a batched mode, which notably reduces performance overhead. Fortunately, in Intel processor families, they widely use predefined buffer to record events (e.g., PEBS and BTS), this mechanism not only amortizes the cost of exception handling, but also enhances the accuracy of sampled information and fits naturally into our batch-mode checking. For events without predefined buffers (e.g., *itlb_misses* and *branch_miss_predict*), Eunomia manually records the performance samples into a buffer and signals the monitoring process when the buffer is full. Eunomia controls the frequency of security checks by setting the length of the sample buffer.

The monitoring process is executed in an event-driven way and will take action in two cases: a signal is received (e.g., the event buffer is full); or the application processes are trying to invoke sensitive system calls (e.g., execve, which is usually called by shellcode). In both cases, the monitoring process will suspend the application processes, do the security check and then resume the application processes after the check. This prevents the application processes from running out-of-sync, which may cause the sample buffer being overwritten with new data, or harmful effects to the system being made by attacks. According to our experience, the checking time in the monitoring process is very short with low CPU utilization. Thus, the overhead due to the suspend time of application processes is very small.

Eunomia also supports the post-attack analysis by using the performance samples of BTS, which provides useful information for detailed post-attack diagnosis. Thus, Eunomia is capable of finding the trace from normal function to the injected code. To use performance samples for post-attack diagnosis, Eunomia can dump a portion of performance samples containing abnormal control flow.

4. PRELIMINARY RESULTS

All evaluations were performed on an Intel Core i7 processor with 4 cores. The operating system is a Redhat Enterprise Linux with kernel version 2.6.34.

4.1 Security Analysis

We use Samba Server version 3.0.21 with a heap overflow vulnerability (CVE-2007-2446) and Squid-2.5.STABLE1 with a stack overflow vulnerability (CVE-2004-0541) to illustrate the detectability of Eunomia. We attack them by three means, namely code-injection attack, return-to-libc attack and return-oriented programming attack. However, due to the vulnerability type, we can not form a return stack in Samba Server, so we failed to exploit Samba Server with return-oriented programming attack. To detect these attacks, we set the threshold of performance events to *one* [5] to minimize false negatives. The threshold for detecting return-oriented programming attacks is set as *15*, which is enough to detect most attacks. As expected, all attacks are detected by Eunomia in the evaluation and Eunomia detects these attacks at the first time an abnormal performance sample is generated.

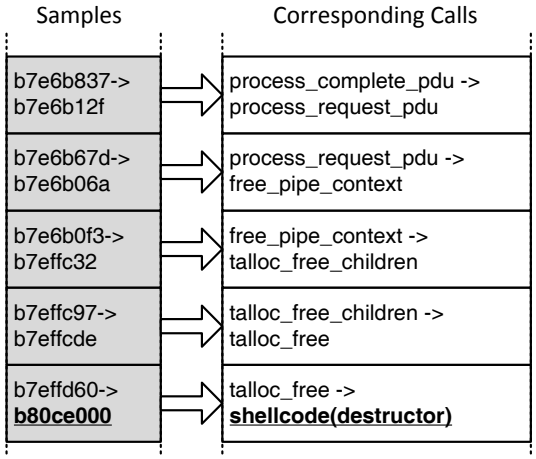

Figure 5: The results of post-attack diagnosis of code-injection attack for Samba Server.

Post-Attack Diagnosis: We also use Samba Server to illustrate the post-attack diagnosis ability of Eunomia using the BTS mechanism. As shown in Figure 5, when a code-injection attack is detected, Eunomia dumps the performance samples with abnormal control flow. From the back traces of the attack, we can easily find that the shellcode is reached by calling "destructor" function pointer in "talloc_free". Virtually, Eunomia can back trace as far as the dumped samples can reach. Here, we only provide five function records in the back trace, which is usually enough for understanding the attack.

4.2 Performance Results

We evaluate several applications in three categories: widely-used server applications, including Apache Web Server, Postgres Database Server and Exim Mail Server [1]; emerging applications including Memcached [10] and Metis [13]; and daily used tools in Linux system, including OpenSSH, a server application for secure remote shell and Tar, which is a compressing tool. For Apache Web Server and OpenSSH, we evaluate them by both latency and throughput. The latency for OpenSSH is the connection time, which is measured by calculating the average connection time of 10,000 connections. The throughput for OpenSSH is measured by transferring 1 Gbyte file using *scp*. For Apache Web Server, we use apache benchmark (*ab*) by issuing 500,000 requests with concurrency level of 100.

Performance for Detecting Code-injection Attacks: There are three events that can be used to detecting code-injection attacks, which are *itlb_misses*, *branch_miss_predict* and BTS mechanism. According to our evaluation, the event counts of *branch_miss_predict* could be more than 10 times larger than the counts of *itlb_misses* and control transfer counts captured by BTS mechanism could be 80 times larger than the counts of *itlb_misses* in the same run. Using benchmarks mentioned above to evaluate these three types of events, we found relative performance

[5]The threshold for BTS is one in nature.

overhead is 4.72%, 1% and 0.1% on average for using BTS, *branch_miss_predict* and *itlb_misses* accordingly. Thus, for detecting code-injection attacks, *itlb_misses* event is most appropriate.

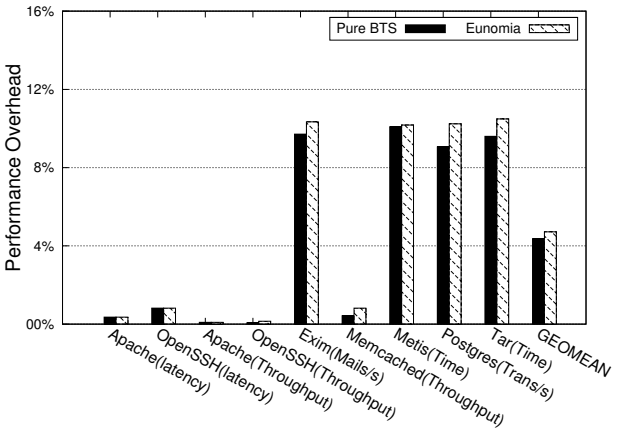

Figure 6: Performance overhead of Eunomia to detect code-injection and return-based attacks with BTS.

Performance for Attack Detection with BTS: Figure 6 shows the relative performance overhead for these applications, from the figure, we can see that Eunomia incurs acceptable performance overhead, with only 4.72% on average, ranging from 0.09% to 10.49%, which means Eunomia can be applied to real-world applications on off-the-shell systems in daily use.

We also measured the inherent overhead of the BTS mechanism alone and found that the performance overhead is almost the same as using BTS together with Eunomia, as shown in Figure 6. This indicates that the detection logic of Eunomia causes very little performance overhead itself. The major reason for overhead in BTS is due to the large number of performance samples, including *all types of jump*, *exceptions*, *calls* and *returns*. Among them, we actually only need samples for call and returns. This demands a hardware supported samples selection mechanism.

5. RELATED WORK

Performance counters have been used extensively for performance profiling and online optimization. Being aware of the importance of performance counters, previous researchers have proposed a variety of architectural techniques in order to provide low-overhead, non-intrusive and accurate performance monitoring [9, 2]. Avritzer et al. [3] performed a set of tests to measure CPU, memory and I/O usages between normal and attacking runs and concluded that the *accumulated resource usages* tend to be different. However, they failed to show how to leverage the difference for precise and in-place detection of attacks.

As Eunomia, previous researchers have also leveraged existing hardware support for security. For example, SHIFT [5] exploits existing hardware support for control speculation to implement an efficient and flexible taint tracking system. BOSH [6] uses the flow-sensitive tags in taint tracking to implement an efficient binary obfuscation system. Compared to Eunomia, these systems require instrumenting the software using compilers, thus cannot work on unmodified and deployed binaries.

6. CONCLUSION AND FUTURE WORK

This paper observed that typical security exploits usually result in precisely identifiable deviation in performance samples. Based on the observation, we designed and implemented Eunomia, which leveraged performance counters for the purpose of detecting a wide variety of security attacks. Our evaluation showed that Eunomia could effectively detect attacks on real-world security vulnerabilities with low overhead. Our future work includes more analysis on false positives and false negatives in Eunomia, enhancing existing PMUs to detect more security vulnerabilities as well as software bugs, and cooperating with compiler transformations or instrumentation to increase the precision of Eunomia.

7. REFERENCES

[1] Exim. http://www.exim.org/.

[2] AMD. Instruction-based sampling: A new performance analysis technique. developer.amd.com/assets/amd_ibs_paper_en.pdf.

[3] A. Avritzer, R. Tanikella, K. James, R. G. Cole, and E. Weyuker. Monitoring for security intrusion using performance signatures. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, pages 93–104, 2010.

[4] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proc. CCS*, 2010.

[5] H. Chen, X. Wu, L. Yuan, B. Zang, P. Yew, and F. Chong. From Speculation to Security: Practical and Efficient Information Flow Tracking Using Speculative Hardware. In *Proc. ISCA*, pages 401–412, 2008.

[6] H. Chen, L. Yuan, X. Wu, B. Zang, B. Huang, and P. Yew. Control flow obfuscation with information flow tracking. In *Proc. MICRO*, pages 391–400, 2009.

[7] J. Crandall and F. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proc. MICRO*, 2004.

[8] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proc. ACM workshop on Scalable trusted computing*, pages 49–54, 2009.

[9] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proc. MICRO*, pages 292–302, 1997.

[10] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004.

[11] R. Hund, T. Holz, and F. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proc. Usenix Security*, pages 383–98, 2009.

[12] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with Return-Less kernels. In *Proc. Eurosys*, pages 195–208, 2010.

[13] Y. Mao, R. Morris, and F. Kaashoek. Optimizing MapReduce for Multicore Architectures. *MIT-CSAIL-TR-2010-020*, 2010.

[14] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. http://www-cse.ucsd.edu/ hovav/dist/rop.pdf, 2010.

[15] W. Xu, S. Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proc. Usenix Security*, 2006.