CFIMon: Detecting Violation of Control Flow Integrity using Performance Counters

Yubin Xia† ‡, Yutao Liu† ‡, Haibo Chen†, Binyu Zang‡

†*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*

‡*School of Computer Science, Fudan University*

Abstract

Many classic and emerging security attacks usually introduce illegal control flow to victim programs. This paper proposes an approach to detecting violation of control flow integrity based on hardware support for performance monitoring in modern processors. The key observation is that the abnormal control flow in security breaches can be precisely captured by performance monitoring units. Based on this observation, we design and implement a system called CFIMon, which is the first non-intrusive system that can detect and reason about a variety of attacks violating control flow integrity without any changes to applications (either source or binary code) or requiring special-purpose hardware. CFIMon combines static analysis and runtime training to collect legal control flow transfers, and leverages the *branch tracing store* mechanism in commodity processors to collect and analyze runtime traces on-the-fly to detect violation of control flow integrity. Security evaluation shows that CFIMon has low false positives or false negatives when detecting several realistic security attacks. Performance results show that CFIMon incurs only 6.1% performance overhead on average for a set of typical server applications.

1. Introduction

Security breaches have been a major threat to the dependability of networked systems, due to the inevitable security vulnerabilities in many software systems. Viruses exploiting such vulnerabilities have caused loss in millions of dollars [1], [2], [3], resulting in not only economic problems, but also significant social impact [4], [3].

Currently, many classes of security exploits usually involve introducing abnormal control flow transfers. For example, the code-injection attack leverages security vulnerabilities to inject malicious code to a program and then redirects control flow to the injected code to gain control. To bypass protection from processor and OS support for non-executable stack, sophisticated attackers switch to *code-reuse* attack that leverages existing code to form malicious *gadgets*. There are currently multiple classes of code-reuse attacks: 1) return-to-libc attack [5], which overwrites stack to redirect the control to library functions in libc; 2) return-oriented programming [6], which injects a forged stack containing instruction addresses in existing binary and leverages *ret*-like instructions to transfer control flow among these instructions to form malicious *gadgets*; 3) jump-oriented programming [7], which uses indirect branches instead of "ret" and a *dispatcher gadget* to transfer control flow among existing binary to form malicious *gadgets*.

There are current many countermeasures to defeat against these attacking means. Some approaches defend against code-injection attacks, including StackGuard [8], FormatGuard [9] and non-executable stacks. Some systems defend return-oriented programming by leveraging either heuristic characteristics [10] and eliminated all "ret" instructions [11], [12]. However, these approaches are usually ad-hoc to a specific protection means. For jump-oriented programming, there are currently few effective means to defend against it.

There are also several general approaches that may defeat against these attacks. For example, control flow integrity [13] statically rewrites a program and uses dynamic inlined guards to check the integrity of control flow. However, this approach may suffer from the coverage problems as static analysis along can easily either overlook legal or tolerate illegal control flow transfers. Control flow locking [14] recompiles a program to limit the number of abnormal control flow transfer, which is thus difficult to be applied to legacy applications.

To improve the efficiency and/or coverage of detection of control flow violation, researchers have also proposed architectural support to validate or enforce control flow integrity [15], [16], [17]. For example, Shi et al. [17] enhance the branch target buffer with

a bloom-filter like signature table to abnormal control flow. However, these proposals need to change existing processors, thus are not readily deployable in stock systems.

In this paper, we propose a non-intrusive approach to detecting a set of attacks that cause abnormal control flow transfers, without changes to existing hardware, source code or binaries. The approach we propose, namely CFIMon, leverages the pervasively available hardware support for performance monitoring in commercial processors, to collect the legal sets of control transfers and monitor control flow deviation of a running application to detect possible attacks. The key observation of CFIMon is that security breaches causing abnormal control flow that can be *precisely* captured by the *branch tracing* mechanisms in performance monitoring units in commodity processors.

Performance monitoring units have been standard components in almost all commercial processors. They provide non-intrusive and low-overhead ways of online performance monitoring and optimization. To improve monitoring precision [18], [19] and lower performance overhead, commercial processors have been integrated with support for precise monitoring mode, including Intel's Precise Event Based Sampling (PEBS) [20], AMD's Instruction-based Sampling (IBS) [21] and PowerPC's Instruction Marking. To analyze control flow behavior of a program, commercial processors have been integrated with support for *Branch Tracing*, including Intel's Branch Trace Store (BTS) and Itanium2's Branch Trace Buffer (BTB). These features allow collecting all branch instructions in a predefined buffer for future analysis.

We leverage hardware support for performance monitoring, which is originally designed for tuning the performance of both applications and system software, to collect legal control transfers and detect violation of control flow integrity. To detect such attacks, we use both static analysis and runtime training to collect the legal set of control flow transfers. During the training phase, CFIMon continuously monitors the performance samples using the BTS mechanism and correlates the traces with the set obtained from static analysis to generate a high-precise set of control transfers. During normal execution, CFIMon also uses the BTS mechanism to collect and analyze in-flight control transfers. Any deviation in performance samples can be used as signs of possible attacks. Upon the detection of an attack, the recorded *branch traces* can be used to locate the exploited security vulnerability and reason how the vulnerability is exploited.

We have designed and implemented a prototype based on *perf_events* [22] supports in Linux kernel 2.6.34 [1], to detect possible attacks. CFIMon currently supports x86 architecture, including Intel Core Duo, Core i5 and i7, using the PEBS and branch trace store mechanism in these processors.

To measure the effectiveness of CFIMon, we have conducted a variety of security tests using real-world vulnerabilities, including heap/stack/integer overflow, format string vulnerabilities and dangling pointers. Our evaluation results indicate that CFIMon can precisely detect the attacks at the first time it happens. We also show that it is very easy to reason about the attacks using the branch traces collected by CFIMon. Performance evaluation results show that CFIMon incurs modest performance overhead for real-world applications.

Based on our experience, we further propose several enhancements to existing performance monitoring units (PMUs) for both performance and detection ability. To further lower performance overhead, we propose adding event filtering mechanism in *Branch Trace Store* to selectively record a few branches instead of recording all branching instructions. To further enhance the detection ability of CFIMon, such as detecting non-control-data attacks [23], current PMUs can be enhanced to support collecting precise linear address of each memory operations. Finally, current PMUs can be enhanced to support simultaneously monitoring of multiple events precisely, so that multiple events could be used to simultaneously detect a variety of attacks (e.g., control and non-control data attacks).

In summary, this paper makes the following contributions:

- The key observation that abnormal control transfers in security breaches can be *precisely* captured in performance samples of the *Branch Trace Store* mechanism.
- The CFIMon system for detecting security breaches, which is the first system that leverages the hardware support for performance counters to precisely detect and analyze attacks.
- A working implementation of the above techniques on commercial processors, as well as security and performance evaluation to demonstrate the effectiveness of our approach.

The rest of the paper is organized as follows: The next section provides some background information on existing hardware support for performance monitoring. Section 3 illustrates the idea and design of CFIMon, followed by the implementation issues in section 4. After describing the experimental setup, the security analysis of CFIMon and its incurred performance overhead are

1. It should be easy to port CFIMon to other OSes such as Windows and FreeBSD, which will be our future work.

evaluated in section 5.1 and section 5.2 accordingly. Section 6 discusses some implications on hardware enhancement for further enhancing performance and detection ability in CFIMon. Finally, we review previous literature in section 7 and conclude the paper in section 8.

2. Performance Monitoring Units

There are generally two working modes of PMUs: interrupt-based mode and precision mode. In the first mode, a counter will automatically increase and generate an interrupt when it has reached a predefined threshold (i.e., event-based sampling) or predefined time has elapsed (i.e., time-based sampling). This is the basic performance counter mode, which supports most types of events, but lacks precise instruction pointer information, resulting in that the reported IP (instruction pointer) is up to tens of instructions away from the instruction causing the event, due to the out-of-order execution in modern processors. For example, according to AMD's manual, the reported IP may be up to 72 instructions away from the actual IP [21] causing the event.

To improve the precision and flexibility of PMUs, most commodity processors also support a precise mode of performance monitoring, including the Precise Event-Based Sampling (PEBS), Branch Trace Store (BTS), Last Branch Record (LBR) and Event Filtering (EF). Currently, most existing commodity processors support parts of the features mentioned above.

Precise Performance Counter: In PEBS, the samples of performance counters are written into a pre-registered memory region. When the memory region is nearly full, an interrupt is generated to trigger the handler. By batching the samples and processing them together, this mechanism improves the performance of monitoring significantly. Meanwhile, thanks to the *atomic-freeze* feature, the IP addresses recorded in traces are exactly the ones causing the event. However, only a few events are PEBS events in Intel Core and i7 processors.

Branch Trace Store: Intel's BTS mechanism provides the capability of capturing all control transfer events and saving the events in a memory-resident BTS buffer. The events include all types of jump, call, return, interrupt and exception. The recorded information includes the addresses of branch source and target. Thus, it enables the monitoring of the whole control flow of an application. Similar as PEBS, the branch trace is also recorded in a pre-registered memory region, which makes the batching processing possible.

Last Branch Record: LBR in Intel Core and Core i7, as well as Branch Trace Buffer (BTB) in Itanium2, records the most recent branches into a register stack. This mechanism records similar data as in BTS. It records the source address and target address of each branch, thus provides the ability to trace the control flow of a program as well. However, due to the small size of the register stack (e.g., Intel Core has 4 pairs, Core i7 has 16 pairs, Itanium2 has 8 PMD registers), previous samples may be overwritten by upcoming samples during monitoring.

Event Filtering: The Event Filtering mechanism provides additional constraints to record events. It is used to filter events not concerned with. For example, latency constraints can be applied in Itanium2's cache related events, which only count on high latency cache misses. Further, constraints such as "do not capture conditional branches", "do not capture near return branches" are generally available on recent processors, which support LBR/BTB such as Intel Core i7 and Itanium2. However, this mechanism is currently only available in LBR/BTB, control transfers recorded in BTS lack this type of filtering support.

Conditional Counting: To separate user-level events from kernel-level ones, PMUs also support conditional event counting: they only increment counter while the processor is running at a specific privilege level (e.g. user, kernel or both). Further, to isolate possible interferences in performance counters among multiple processes/threads, operating systems are usually enhanced by saving and restoring performance counters during context switches.

3. CFI Enforcement by CFIMon

CFIMon adopts two phases: *offline phase* and *online phase*. During the offline phase, CFIMon builds a *legal set* of target addresses for each branch instruction. During the online phase, CFIMon collects branch traces from applications and diagnoses possible attacks with *legal set*s following a number of rules. A rule can be applied to a portion or all of branch traces, and can determine the status of the branch as legal, illegal or suspicious. Further decision will be made depending on the status of the branch and context.

This section first describes the requirements of different branch types for enforcing control flow integrity, and then presents the reasons of choosing the BTS (Branch Trace Store) among all the performance counters to monitor the control flow. Finally, we describe our approaches to detecting typical attacks and use several real-world vulnerabilities to show how to detect control flow violation when the vulnerabilities are exploited, and discuss possible issues with CFIMon.

Branch Type	Branch Example	Target Instruction	Target Set	In Binary	Run-time
Direct call	callq 34df0 <abort>	1: taken	/	16.8%	14.5%
Direct jump	jnz c2ef0 <__write>	1 or 2: taken or fallthrough	/	74.3%	0.8%
Return	retq	Limited: insn. next to a call	*ret_set*	6.3%	16.3%
Indirect call	callq *%rax	Limited: 1st insn. of a function	*call_set*	2.1%	0.2%
Indirect jump	jmpq *%rdx	Unlimited: potentially any insn.	*train_set*	0.5%	68.3%

TABLE 1. Branch Classification. The distribution is from Apache and libraries it uses.

3.1. Branch Classification

The control flow integrity of an application can be maintained if we can 1) get a legal set of branch target addresses for every branch, and 2) check whether the target address of every branch is within the corresponding legal set at runtime. There are five types of branches in x86 ISA, including direct jump, direct call, indirect jump, indirect call, and return. Table 1 shows examples for each branch type.

A direct jump has only one target address if it is an unconditional jump, or two target addresses if it is a conditional jump. For example, instruction *"0x403291: jnz 0x403200"* has two possible target addresses: *0x403200* if the branch is taken, and the address of the next instruction if the branch falls through. Similarly, a direct call also has only one target address. Since the code is read-only and cannot be modified during runtime, a direct branch, either a direct jump or call, is always considered as a safe one.

However, not all branch instructions have deterministic target address set. An indirect jump, e.g., *"jmp %eax"*, may theoretically branch to any instruction in the memory space. It is not possible to gain the whole legal target address set of indirect jump just by statically scanning the binary.

Unlike indirect jump, the legal target set of an indirect call is limited. A call can only transfer control to the start of a function, which could be obtained by statically scanning the binary code of the application and the libraries it uses.

A return instruction branches to an address popped from the stack, which could only be determined during runtime. Since a function maybe be invoked through a function pointer, we cannot know exactly all of its callers. Fortunately, we finds that in most cases, a *return* follows a *call* instruction. Thus the target address of a return has to be the one next to a *call*, which could also be obtained by scanning the binary code. However, there do exist several cases of "return-without-call", which will be discussed in Section 3.3.

We analyze the distribution of branches in binary and at runtime. Table 1 presents the distribution of different types of branches of Apache and the libraries it uses. As shown in the figure, the indirect branch (including indirect call/jump and return) takes up only 8.5% in static binary, but 84.8% at runtime. However, among all the executed indirect branches, 94.7% have only one target address, 99.3% have less than or equal 2 target addresses. There are only 0.1% of all 7736 branches (9 branches) have more than 10 different target addresses. It indicates that the variation of an application's control flow is limited. Thus, only a small number of rules are needed to diagnose and analyze the branch trace at runtime.

3.2. Monitoring All Branches at Runtime

To accurately and effectively identify an attack, CFIMon needs precise information of every branch at runtime for detection. As mentioned in section 2, there are some mechanisms that can record each control transfer, e.g. LBR and BTS, thus provide users with the ability to trace back program execution flow and find how attackers transfer control flow to the malicious code. However, since LBR uses a small register stack to store the branch information, previous samples may be overwritten by upcoming samples during monitoring, and the overwritten events cannot be detected. Thus, LBR provides no opportunity to check the samples, which makes this mechanism hard to be used in detecting security attacks. On the other hand, BTS can precisely record all control transfers into a predefined buffer. An interrupt will be delivered when the buffer is nearly full. The monitor can then get the trace in a batch and do the security check. Meanwhile, since the monitor can obtain all the branch information of a running application, it can not only detect security attacks, but also identify the control flow of the execution of malicious code, thus help users locate the vulnerabilities.

3.3. Detecting CFI Violation

During the offline phase, CFIMon first scans the binary of application and dynamic libraries to get *ret_set* and *call_set*. The *ret_set* contains addresses of the instructions next to each *call*. The *call_set* contains all addresses of the first instruction of each function. CFIMon gathers branch traces from training runs to get

the legal set of branch target for each indirect jump, namely *train_sets*.

There are several cases that the calling convention may be violated, including setjmp/longjmp and Unix signal handling. In the setjmp/longjmp situation, the longjmp() will not return to its own caller, but return to the caller of setjmp() instead, which is also a legal return address. Hence, no false positive will occur. In the case of Unix signal handling, when a signal has been received, the OS will invoke the signal handler, and push a return address on the stack. When the signal handler returns, it will pop and branch to the address as if it is invoked from there. Since an application may be trapped into OS at any instruction, the address may be any location in the memory space, thus violates the rules of return. In order to eliminate such false positives, we modify the OS to notify the monitor when a signal handler is invoked. The monitor will then omit the alarm when a signal handler returns.

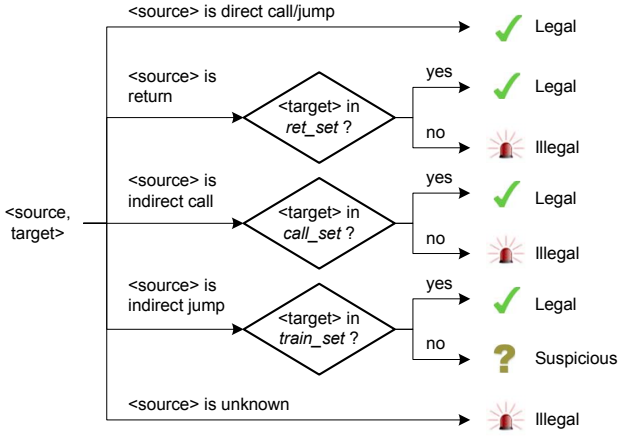

Fig. 1. Rules in CFIMon

Figure 1 shows the detail of diagnose module in CFIMon. Any branch has one of three states: legal, illegal or suspicious. For every branch sample, the diagnosis module first handles special cases such as a return from legal signal handler. Then, it switches into different cases according to the type of source address, and considers the state of branches depending on the target address.

The *train_set*s are obtained through training runs. CFIMon collects branch traces of an application with training input, and parses the trace to get the legal target addresses for each indirect jump, namely *train_set*. However, the *train_set* may not be complete since there could be corner cases which are not covered. Thus, during online checking, if a branch is not in the *train_set*, it is not considered as illegal but suspicious, which will be delivered to the diagnose module to make further decision.

Once the diagnose module discovers an illegal branch, it will take serious actions such as suspending the application and triggering an alarm immediately. For suspicious branches, the diagnose module can make a flexible decision, depend on the pattern of the branches. For example, the diagnose module can maintain a window of the states of recent n branches, and apply a rule of tolerating at most m suspicious branches in the recent n ones. The parameter m and n can be adjusted by the users to make a balance between availability and security, according to specific requirements of application. Our current prototype adopts this slide-window mechanism.

CFIMon also collects all suspicious branches at runtime. If the suspicious branches are considered as corner cases, the trace will be used as the input of online training to further improve the accuracy of *train_set*. In order to make the training more accurate, machine learning technologies can be adopted to analyze the pattern of continues branches, which is our future work.

3.4. Case Studies of Real-World Exploits

In this section, we use three real-world examples of the above mentioned attack types and show on how they could be detected with our approach.

Code-injection Attack of Samba: In this attack, we exploits a heap overflow vulnerability in the "lsa_trans_name" function to overwrite a function pointer called "destructor" in Samba's malloc header. When a memory buffer is freed, the destructor will be called, causing the control to be transferred into the injected *nop-sled* [2], which eventually executes shellcode. The shellcode will open a socket and listen to tcp connections. Upon each connection, the shellcode will provide attackers with a remote shell. The attack is detected since the branches have never appeared in the *train_set*. The monitor detected such event and triggered an alarm when the number of suspicious branches exceeds the threshold.

There may be cases where code execution on stack is legal, such as trampoline code on the stack for nested functions in GCC and signal handling code in old versions of Linux. Fortunately, recent Linux kernel has abandoned the need of execution on stack for signal handling. GCC generates trampoline code only when an nested function address is referenced and the nested function accesses variables from its outer closure, which is a very rare case. Even if an application indeed use code execution in data section, CFIMon can solve the situation without false positive since the branches will be in the *train_set*.

2. *nop-sled* is a piece of code that is semantic equivalent to nops, which is used to enlarge the chance of transferring to injected code.

Return-to-libc Attack of GPSd: GPSd is a service daemon that monitors GPSes or AIS receivers attached to a host computer through serial or USB ports. It makes all data on the location/course/velocity of the sensors available to be queried on TCP port 2947 of the host computer. We use *GPSd* of version 2.7, which has a format string vulnerability in "gpsd_report" function. Attackers can overwrite arbitrary memory addresses with arbitrary values. In our evaluation, we use this format string vulnerability to overwrite the *GOT entry* of "syslog" into "system" library routine address, and the subsequent calls to "syslog" library routine will actually invoke "system" with attacker-supplied arguments.

We evaluate CFIMon by detecting this return-to-libc attack, according to our detection scheme. When the "system" library routine appear in the branch target address of the collected traces, CFIMon marks it and the following branches as suspicious. The number of suspicious branches quickly exceeds the threshold and an alarm is triggered as expected.

Return-oriented Programming Attack of Squid: We use *Squid* with version 2.5-STABLE1, which is a widely-used proxy server. In *Squid*, the helper module for ntlm authentication has a stack overflow bug in its function "ntlm_check_auth". Attackers can supply arbitrary password of at most 300 bytes to smash the stack. After the attack, the return value stored before old %ebp is overwritten to the address of the first instruction of our return-oriented shellcode and the stack is overwritten as the return addresses of return-oriented shellcode, as shown in Figure 2. When the program gets to execute "leave", the stack pointer now points to our injected return address stack. After the "ret" instruction execution, control transfers to our shellcode finally.

We use CFIMon to detect this return-oriented programming attack. When the malicious code executes the first "ret" instruction, the monitor finds that the target of the "ret" is not an instruction next to a "call". Since the instruction is not a special case, e.g., signal handling, CFIMon indicated it as an illegal return.

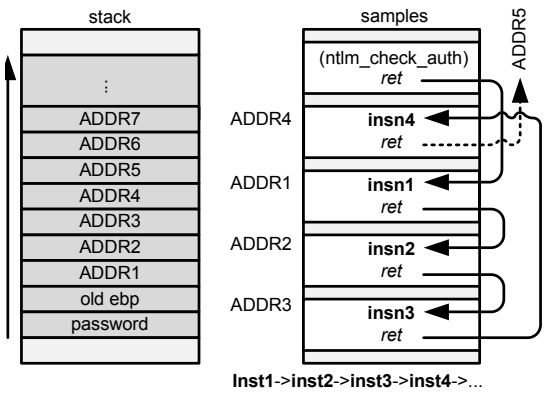

Fig. 2. An example return-oriented programming attack of Squid.

3.5. Discussions

False Alarms: As most attack detecting systems, CFIMon might have false positives or false negatives. For code-injection attacks, since the attack needs to trigger abnormal control transfer to injected code. Thus, CFIMon is able to detect code-injection attacks without false negatives. However, there may have some false positives when detecting code-injection, besides the trampoline and signal usage in stack or heap, applications can still execute code on heap or stack, including some self-modifying code, binary translators or Java virtual machines. In such cases, CFIMon can learn application-specific knowledge to filter such false positives.

For code-reuse attacks, CFIMon needs to preprocess binaries and use training to collect legal sets of branches. Consequently, if there is imprecision during the preprocessing, CFIMon might have some false positives or false negatives. However, CFIMon is able to learn from the program execution to minimize false alarms.

Informed Attackers: Attackers knowing the mechanisms in CFIMon can still hardly bypass CFIMon. First, an attacker can leverage the detection latency. Since CFIMon is triggered when the buffer is full or a sensitive system call is made, there's latency from attack starting. An attacker may carefully construct malicious code with few or even no branches. When such code is running, it delays the detection.

Second, an attacker can leverage the slide-window size. If the attack code is constructed in the form of loop of "few abnormal-branches + many normal-branches", it may use normal-branches to fill the slide-window and not trigger alert.

We argue that both attacks are hard to construct that increases the cost of attacking. Meanwhile, in the first case, one can make a tradeoff between the security level and performance by setting the buffer size. In the second case, one can balance between the security level and false-positive rate by setting the slide-window size.

For example, for attackers exploiting return-oriented programming, they now have to choose a consecutive number of unusual branch sequences not covered in the *train_set*s, which, unfortunately, are very rare in reality. Further, once the number of suspicious branch instructions increase to a threshold, CFIMon will report an alarm. The memory buffer used by BTS is also protected by CFIMon. If an attacker aims to compromise the memory buffer, there'll be abnormal branches too.

4. Implementation of CFIMon

We have implemented CFIMon based on *perf_events* on Linux kernel version 2.6.34, which is a unified kernel extension in Linux for user-level performance monitoring. Currently, CFIMon supports Intel Core Duo, Core i5 and i7 processors and focuses on user-level attacks only.

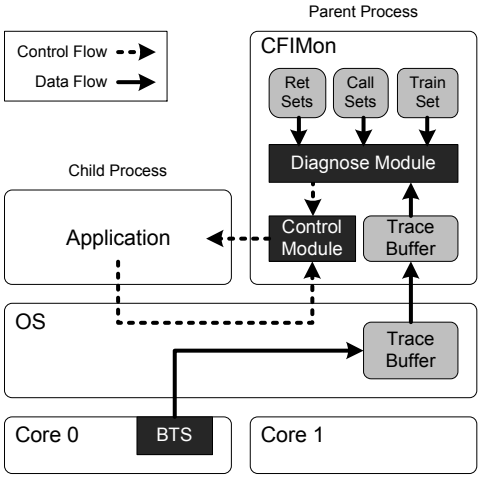

Fig. 3. Architecture of CFIMon.

Figure 3 presents the overall architecture of CFIMon. There are two components of CFIMon: a kernel extension and a user-level tool. The kernel extension is responsible to operate the performance samples, monitor signals, and provide the interfaces to user-level tool. The user-level tool has two modules: diagnose module and control module. The diagnose module uses branch traces, *call_set*, *ret_set* and *train_set*s as inputs to check the control flow integrity, and receives information from the OS to solve special cases such as signal handling. The control module is in charge of initializing the environment, launching and synchronizing with an application.

The user-level tool is executed as a monitoring process, which is the parent process of the application processes. It uses *ptrace* to synchronize with the application processes. When launching an application, the monitoring process forks a child process. The child process first calls *ptrace* with the flag *PTRACE_TRACEME* on. Thus, when making the *exec* system call, it will be suspended by the OS and its parent process gets to run. Since the addresses in *call_set* and *ret_set* are obtained from the binary file of application and dynamic libraries, they are in the form of relative address. Thus the parent process will get the memory mapping information of the child process and transfer the address to absolute address. After the monitoring process sets up the performance events and trace buffers, it resumes the child. Afterwards, the two processes can run simultaneously without any synchronization until next time the child calls *exec*, so that the monitoring process has a chance to run for security check at the critical point. The monitoring process and the application processes are binded on different cores on multicore hardware, for the purpose of further reducing interference between the two. When the application processes further forks other children process, the monitoring process can automatically monitor them as well.

When the application starts to run, the BTS counter generates trace of branch and writes the trace directly into a memory buffer. Once the buffer is nearly full, the kernel will copy the trace to user space, and the monitoring process will start to diagnose. This batching mode significantly reduces the performance overhead compared with per-sample check mode. Meanwhile, when the application processes is trying to invoke sensitive system calls (e.g. *execve* which is usually used by a shellcode), the monitoring process will suspend the application processes and resume it after the check. This prevents the application processes from running out-of-sync, which may cause harmful effects to the system being made by attacks. The suspend time of application processes is small since the diagnose process is simple and effective that only utilizes a little CPU during execution.

Once a CFI violation is detected, the monitoring process can take different actions according to different requirement of applications. It may immediately kill the application processes or email the administrator, or both. It can also store the recent branch trace for post-attack analysis. The administrator can know the process of the attacking by diagnosing the trace to malicious code and further fix the vulnerability of the application.

5. Experimental Setup

All evaluations were performed on an Intel Core i5 processor with 4 cores. Each core is with 32k L1 instruction and L1 data cache and a 256K L2 data cache. The four cores share an 6 MB L3 cache. The machine has 2BG 1066MHz main memory, a 500GB sataII disk of 7200 rpm, and a 100Mbps NIC. The operating system is a Debian-6 with kernel version 2.6.34.

5.1. Security Analysis

We use several real-world applications as well as two demo programs with the dangling pointer and integer overflow vulnerabilities to evaluate the detection ability of CFIMon, which is shown in Table 2. For these applications with different vulnerabilities, we used

Application	Reference	Description	Vulnerability	Attack Means		
				Injected	Ret-to-libc	Ret-oriented
Samba-3.0.21	CVE-2007-2446	file and print server	heap overflow	√	√	×
Squid-2.5.STABLE1	CVE-2004-0541	cache proxy	stack overflow	√	√	√
GPSd-2.7	CVE-2004-1388	gps device agent	format string vul.	√	√	×
Wu-ftpd-2.6.0	CVE-2000-0573	ftp server	format string vul.	√	√	×
Wu-ftpd-2.4.2	CVE-1999-0368	ftp server	stack overflow	√	√	√
Bug1	Demo	bug test program	dangling pointer	√	√	√
Bug2	Demo	bug test program	integer overflow	√	√	√

TABLE 2. Security vulnerabilities for evaluation, which are exploited using three means of attacks: code-injection (Injected), return-to-libc (Ret-to-libc) and return-oriented programming (Ret-oriented).

three types of attacks to exploit them, namely code-injection attacks (Injected), return-to-libc attack (Ret-to-libc) and return-oriented programming (Ret-oriented). For Samba, GPSd and Wu-ftpd-2.6.0, as we cannot overflow the stack to construct a return stack with instruction addresses, we failed to exploit the vulnerabilities in the three applications using return-oriented programming. For all these attacks, we set the window size as 20, and tolerant at most 3 suspicious branches within the window. **Evaluation for Code-Injection Attacks:** To effectively and reliably attack these applications using code-injection and finally transfer control to injected code, we use the metasploit framework [24] to generate *nop-sled* before the injected code. We attack each application with injected code five times to test the false negatives. As expected, all attacks are detected by CFIMon in the evaluation and CFIMon detects these attacks at the first time an abnormal performance sample is generated. During this evaluation, we simply report a security alarm upon the detection of attacks.

Evaluation for Return-to-libc Attacks: All vulnerabilities that can be attacked with code-injection can also be attacked with the return-to-libc attack. Similar to our evaluation on code-injection attacks, CFIMon successfully detects all these attacks without experiencing false negatives.

Evaluation for Return-oriented Programming Attacks: Similar to other evaluation, CFIMon successfully detects all these attacks without experiencing false negatives. Return-oriented programming attacks have the following features: it uses return to organize logic and heavily use "unintended instruction sequence" to form code gadgets. It violates the rules of CFIMon which enforces that the target address of a return instruction must be the one next to a call. Even if the start address of the first gadget happens to be the legal target, it is hard to make all the gadget legal. Such attacks are hard to be applied on applications using heap overflow or format string vulnerability, because we cannot modify the stack top pointer(e.g. %esp) to our supplied return addresses stack.

Evaluation for Jump-oriented Programming Attacks: The jump-oriented programming attack is similar with return-oriented programming except it uses jump to organize the malicious code gadgets. We didn't make a jump-oriented programming attack on real application. However, we argue that since this kind of attack relies heavily on "unintended instruction sequence", it is likely to issue an invalid jump instruction, which will be captured by the CFIMon. Even if it uses all legal jump, the branches will be marked as suspicious for their rareness, and an alarm will be reported accordingly.

Evaluation for False Positives: We run several typical server daemons (e.g., squid, sshd) using CFIMon in our daily use, to evaluate the false positives in CFIMon. We check the log every day to see if there are any false alarms in daily use. With several days of monitoring, we experience no false positive in our daily use. Thus, CFIMon could be practically used for real-world applications in off-the-shell systems with few false positives.

Samples	Corresponding Calls
b7e6b837->b7e6b12f	process_complete_pdu->process_request_pdu
b7e6b67d->b7e6b06a	process_request_pdu->free_pipe_context
b7e6b0f3->b7effc32	free_pipe_context->talloc_free_children
b7effc97->b7effcde	talloc_free_children->talloc_free
b7effd60->**b80ce000**	talloc_free->**shellcode(destructor)**

TABLE 3. The results of post-attack diagnosis of code-injection attack for Samba Server

Post-Attack Diagnosis: We also use Samba Server to demonstrate the post-attack diagnosis ability of CFIMon. As shown in Table 3, CFIMon dumps the performance samples with abnormal control flow when a code-injection attack is detected. By analyzing the back traces of the attack, we can easily find that when calling "destructor" function pointer in "talloc_free', the shellcode is invoked. Virtually, CFIMon can back trace as far as the dumped samples can reach. Here only five function records are presented which is enough for understanding the attack.

Application	Description	Performance Matrix	Parameters
Apache	A widely used web server	Throughput of get/put	4 clients put 64KB files and get 1MB file
		Latency of get	Latency of 4 clients getting 1MB file
Exim	A mail transfer agent	Mails per second	Send 1MB mails
Memcached	An object caching server	Throughput of values get by key	Use trace of Facebook
Wu-ftpd	A widely used FTP server	Throughput	Client gets 700MB file from the server

TABLE 4. Different types of real-world applications for benchmarking

5.2. Performance Evaluation

In this section, we quantitatively evaluate the performance of CFIMon using several real-world applications, which many attacks target at.

Benchmark Selection: To show CFIMon can be applied to a variety of applications with practical performance, we choose different types of server applications, as shown in Table 4. These applications can be divided into two categories: 1) widely-used server side applications, including Apache Web Server, wu-ftpd and Exim Mail Server [25]; 2) emerging applications including Memcached [26], which is a distributed memory object caching system widely used in many productions systems in companies such as Facebook, Google and Yahoo!.

Performance Results: We evaluate these benchmarks by different performance matrix as shown in Table 4. Figure 4 shows the relative performance overhead for these applications, from the figure, we can see that CFIMon incurs modest performance overhead, with only 6.1% on average, ranging from 2.3% to 8.4%. We also compare the overhead of CFIMon with pure BTS (trace recording only). The overhead of pure BTS is 5.2%, which takes 86% of all the overhead of CFIMon. The result means that CFIMon can be applied to some real-world server applications on off-the-shell systems in daily use.

Figure 5 shows the overhead of CFIMon when clients get/put files of different size from/to an Apache server. The performance overhead is less than 5% when the file size is larger than 2MB, but increases as the file getting smaller. This is because when the file is large, the server is I/O-bound. But for small-size file, the server consumes more CPU. The throughput of apache-put and apache-get are 42% and 15% of the original run when the file size is 1KB, respectively. We further broke down the overhead and found that in this case, 97% of the overhead came from the BTS itself, because of the frequent memory write of trace buffer.

Memory Overhead: The *ret_set*, *call_set* and *train_set* of target addresses are organized in the form of hash table. The size of the tables is quite small: in Apache, the sum size of both is only a little more than 200KB. Thus the memory overhead incurred by CFIMon is negligible.

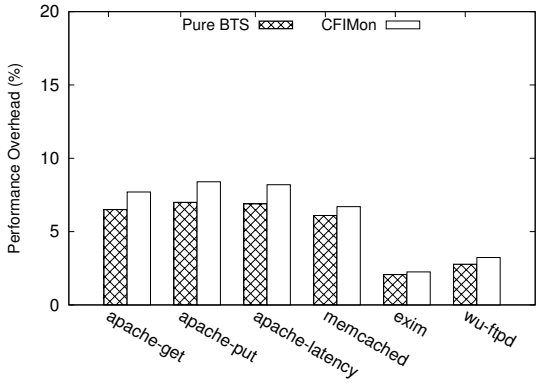

Fig. 4. Performance overhead of CFIMon

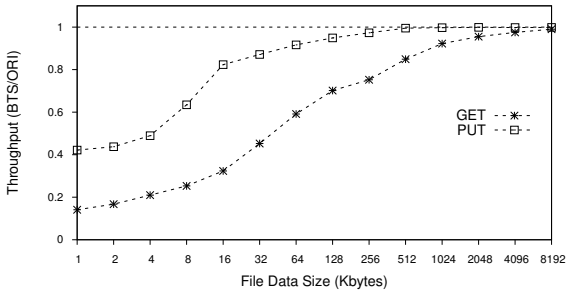

Fig. 5. Performance overhead in Apache

6. Implications on Hardware Enhancement

According to our experience, existing performance monitoring units could be enhanced in the following ways for the purpose of attack detection:

Event Filtering for BTS: Currently, none of commercially available processor supports Event Filtering for BTS. Although Intel Core i7 supports Event Filtering for LBR, its register stack is not big enough for security diagnosis. If the hardware support for more flexible filtering of events is available, the performance overhead and complexity of CFIMon could be further reduced in different usage scenarios. For example, if the BTS is with the support of selective sampling of only call or return instructions, the overhead in CFIMon would

be significantly lowered when detecting return-oriented programming.

Co-existing Multiple Counters for Security: The support for simultaneous monitoring of multiple events is poor in commodity processors. For example, when monitoring 4 events at the same time in Intel processors, the precision is lowered and the performance overhead increases significantly. Hence, we plan to investigate the hardware support to increase the concurrency level of performance monitoring, yet without sacrificing performance and precision significantly. This also enables the co-existence of CFIMon with performance tuning.

Precise Linear Address Information of Memory Stores: Although Intel Core i7 supports precise linear address information of memory operation with event *mem_inst_retired:latency_above_threshold*, it cannot be used to detect non-control-data attacks [23] due to the following reasons: it randomly tags instructions by hardware and only tagged instructions have linear address information; as the minimal latency threshold is set to 4, it is unable to report memory loads of latency less than 4 cycles; To enable the detection of non-control-data attacks by checking the data flow integrity or write integrity testing [27], it is desirable for the PMUs to provide with a precise event which can record specific memory stores with linear address information.

7. Related Work

There is already a considerable amount of work aiming at detecting or preventing security attacks and improving performance counters. However, none of them has exploited performance counters for the use of attack detection and analysis. In this section, we shortly describe related literatures in performance counters and discuss some typical systems in the security area:

7.1. Performance Counters

Performance counters have been used extensively for performance profiling [28] and online optimization [29]. 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 [19], [21]. Software developers have also provided a number of interfaces to support simple and portable uses of diverse performance counters. In this paper, by exploiting existing hardware and software support for performance monitoring, we demonstrate the novel use of performance counters to non-intrusive detection of security attacks with unmodified, deployed applications.

In a recent positional paper, Yuan et al. [30] conduct a survey on how diverse PMU features such as *itlb_misses*, *branch_miss_predict* and branch trace store could be used to detect various attacks. However, their approach are ad-hoc and there is no uniformed to detect and analyze different attacks related to violation of control flow integrity. Avritzer et al. [31] 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.

7.2. Control Flow Attacks and its Countermeasures

Code-reuse Attacks: Code-reuse attacks have emerged recently. Return-to-libc attacks [5] has been used as an effective means to exploit many security vulnerabilities. Return-oriented programming [6] and its variety "pop+jmp" attacks [11] and jump-oriented programming [7] go a step further by reusing existing binary sequences instead of function calls, and thus place much less assumptions on victim programs.

Defending Against Code-reuse Attacks: There are also a number of efforts aiming at detecting or defending against code-reuse attacks. For example, ROPdefender [32] uses a shadow stack together with binary rewriting to validate each return target. DynIMA [10] instead leverages the characteristics of return-oriented programming of using short code sequences before "ret" to detect possible attacks. Return-less kernel instead using compiler-rewriting However, these approaches are ad-hoc in defending only a special class of code-reuse attack and most of them require rewrite either source code or binaries.

Security Through Diversity: Security through running several diverse copies and comparing the results [33], [34] has been a useful technique to defend against a variant of attacks, by increasing the attacking difficulty in requiring understanding and attacking several copies simultaneously. When implementing purely in software, this usually means that the resource consumption will be increased by approximately the number of diverse copies. Hence, recent researchers exploit the architectural support and multicore hardware [35] to reduce the resource consumption and increase performance.

Security Through Randomization: Randomizing the execution environments such as instruction sets [36], [37], and address spaces [38] is an effective approach to defend code-injection attacks or memory errors. As the environments (ISA, address spaces) assumed by attackers are different from the real execution environments,

attacking code will fail to execute. While effective, it could incur significant performance degradation without the hardware support [36], or is only effective to a specific attack [38].

Control and Data Flow Integrity: Dynamically enforcing the integrity of control flow [13] or data flow [39] could defend against attacks aiming at altering the normal control and data flow. It has also been implemented in the system address space [40]. However, it requires binary rewriting of software and would incur non-trivial performance overhead [16].

Taint Tracking: Taint tracking is a general security defense technique. It works by marking data from untrusted channels as tainted, tracking the propagation of the tags during execution, and checking the tags before critical uses of data to detect attacks. There has been a considerable number of systems that extend existing hardware to support taint tracking [41], as well as software-based implementation using compiler instrumentation [42], running the code in an emulator [43], binary translator [44] and JVM [45]. Compared to its hardware counterparts, software-based taint tracking is more expressive but would result in significant performance overhead (e.g., 3.6X for LIFT [44] and 37X for TaintCheck [43]), or require instrumenting software [42].

Security on Existing Hardware: As CFIMon, previous researchers have also leveraged existing hardware support for security. For example, SHIFT [46] exploits existing hardware support for control speculation to implement an efficient and flexible taint tracking system. BOSH [47] uses the flow-sensitive tags in taint tracking to implement an efficient binary obfuscation system. Compared to CFIMon, these systems require instrumenting the software using compilers, thus cannot work on unmodified and deployed binaries.

8. Conclusion and Future Work

In this paper, we observed that many security exploits against control flow can result in precisely identifiable control flow deviation in performance samples. Based on the observation, we designed and implemented CFIMon, which leveraged the branch trace store mechanism in performance counters for the purpose of detecting a wide variety of security attacks to control flow, including classic code-injection attacks and emerging code-reuse attacks. Our evaluation using several realistic vulnerabilities showed that CFIMon can effectively detect attacks on these vulnerabilities. Performance results indicates that CFIMon has modest performance overhead for real-world applications, and we proposed our several hardware proposals for further enhancement to the detection ability and performance in existing processors.

CFIMon has made its first step in using performance monitoring units for the purpose of attack detection. In future work, we plan to extend and improve CFIMon in several directions. First, while this paper only explores the use of PMU for security attacks, the idea of CFIMon could also be similarly applied to other types of bugs such as race conditions, ordering violations and deadlocks, whose behaviors might also result in some performance anomaly. However, as the performance anomaly of some types of bugs could be insignificant compared to that of security attacks, it would also be interesting to couple with minor architectural support to *filtering out* possible false positives, as those in Event Filtering. Second, CFIMon does not detect high-level semantic attacks now, whose explosion usually requires understanding the high-level semantics of a program. In our future work, we plan to extend our system with program semantics to detect such attacks. Finally, CFIMon is designed with the aim of supporting unmodified applications. However, if being coupled with compiler transformation or instrumentation support, it could further reduce the complexity and increasing the precision of CFIMon. In the future, we plan to investigate how compilers could be used to make applications friendlier to CFIMon.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work was funded by China National Natural Science Foundation under grant numbered 61003002, a grant from the Science and Technology Commission of Shanghai Municipality numbered 10511500100, Fundamental Research Funds for the Central Universities in China and Shanghai Leading Academic Discipline Project (Project Number: B114).

References

[1] C. Zou, W. Gong, and D. Towsley, "Code red worm propagation modeling and analysis," in *Proc. CCS*, 2002, pp. 138–147.

[2] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the Slammer worm," *IEEE Security & Privacy*, vol. 1, no. 4, pp. 33–39, 2003.

[3] M. Bailey, E. Cooke, D. Watson, F. Jahanian, and J. Nazario, "The Blaster Worm: Then and Now," *IEEE Security & Privacy*, vol. 3, no. 4, pp. 26–31, 2005.

[4] PITAC Team, "Cybersecurity: A crisis of prioritization." President's Information Technology Advisory Committee (PITAC), Tech. Rep., Feb. 2005.

[5] R. Wojtczuk, "The advanced return-into-lib (c) exploits: Pax case study," *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*, 2001.

[6] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 552–561.

[7] T. Bletsch, X. Jiang, V. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 30–40.

[8] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. Usenix Security*, 1998.

[9] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier, "FormatGuard: Automatic protection from printf format string vulnerabilities," in *Proc. Usenix Security*, 2001.

[10] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks," in *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, 2009, pp. 49–54.

[11] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proc. CCS*, 2010, pp. 559–572.

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

[13] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[14] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proc. ACSAC*, 2011.

[15] T. Zhang, X. Zhuang, S. Pande, and W. Lee, "Anomalous path detection with hardware support," in *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 2005, pp. 43–54.

[16] M. Budiu, Ú. Erlingsson, and M. Abadi, "Architectural support for software-based protection," in *Proc. Workshop on Architectural and system support for improving software dependability*, 2006, p. 51.

[17] Y. Shi and G. Lee, "Augmenting branch predictor to secure program execution," in *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*. IEEE, 2007, pp. 10–19.

[18] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, R. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl, "Continuous profiling: where have all the cycles gone?" *ACM SIGOPS Operating Systems Review*, vol. 31, no. 5, p. 14, 1997.

[19] 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*, 1997, pp. 292–302.

[20] B. Sprunt, "Pentium 4 performance-monitoring features," *IEEE Micro*, vol. 22, no. 4, pp. 72–82, 2002.

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

[22] I. Molnar, "Performance counters for linux, v8." http://lwn.net/Articles/336542, 2009.

[23] S. Chen, J. Xu, E. Sezer, P. Gauriar, and R. Iyer, "Non-Control-Data Attacks Are Realistic Threats," in *Proc. USENIX Security*, 2005.

[24] Metasploit Team, "Metasploit," http://www.metasploit.com/.

[25] "Exim," http://www.exim.org/.

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

[27] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *IEEE Symposium on Security and Privacy*, 2008, pp. 263–277.

[28] G. Ammons, T. Ball, and J. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *Proc. PLDI*, 1997, pp. 85–96.

[29] R. Azimi, M. Stumm, and R. Wisniewski, "Online performance analysis by statistical sampling of microprocessor performance counters," in *Proc. Supercomputing*, 2005, pp. 101–110.

[30] L. Yuan, W. Xing, H. Chen, and B. Zang, "Security breaches as pmu deviation: Detecting and identifying security attacks using performance counters," in *2011 ACM SIGOPS Asia-pacific Workshop on Systems*, 2011.

[31] 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*, 2010, pp. 93–104.

[32] L. Davi, A. Sadeghi, and M. Winandy, "Ropdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 40–51.

[33] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity," in *Proc. USENIX Security*, 2006, pp. 105–120.

[34] A. Nguyen-Tuong, D. Evans, J. Knight, B. Cox, and J. Davidson, "Security through redundant data diversity," in *Proc. DSN*, 2008, pp. 187–196.

[35] R. Huang, D. Deng, and G. Suh, "Orthrus: efficient software integrity protection on multi-cores," in *Proc. ASPLOS*, 2010, pp. 371–384.

[36] G. Kc, A. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proc. CCS*, 2003.

[37] E. Barrantes, D. Ackley, S. Forrest, and D. Stefanović, "Randomized instruction set emulation," *ACM Transactions on Information and System Security (TISSEC)*, vol. 8, no. 1, pp. 3–40, 2005.

[38] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proc. CCS*, 2004, pp. 298–307.

[39] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proc. OSDI*, 2006.

[40] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula, "XFI: Software guards for system address spaces," in *Proc. OSDI*, 2006, p. 88.

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

[42] 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, pp. 121–136.

[43] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. NDSS*, 2005.

[44] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu, "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks," in *Proc. MICRO*, 2006, pp. 135–148.

[45] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proc. OSDI*, 2010.

[46] 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*, 2008, pp. 401–412.

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