Extrax: Security Extension to Extract Cache Resident Information for Snoop-based External Monitors

Jinyong Lee , Yongje Lee , Hyungon Moon , Ingoo Heo , and Yunheung Paek
Department of Electrical and Computer Engineering, Seoul National University, Seoul, Korea
Email: {jylee, yjlee, hgmoon, igheo, ypaek}@sor.snu.ac.kr

***Abstract*—Advent of rootkits has urged researchers to conduct much research on defending the integrity of OS kernels. Even though recently proposed snoop-based monitors have shown to provide higher performance and security level compared to conventional hypervisor-based monitors, we discovered that the use of write-back caches in a system would seriously undermine the effectiveness of snoop-based monitors. To address the problem, we propose a special hardware unit called *Extrax* which makes use of existing hardware logic, *core debugging interface*, to extract necessary information for security monitoring. Being implemented to refine the debug information for security purposes, Extrax assists snoop-based monitors to detect attacks that exploit write-back caches. Experimental results show that our system can detect more advanced attacks, which the state-of-the-art snoop-based hardware monitors cannot capture, with moderate area overhead and power consumption.**

I. Introduction and Previous Work

As electronic devices such as PCs and smartphones become essential parts of our everyday life, the potential privacy and security risks due to numerous *malwares* on the devices are rapidly growing. As a means to protect such devices from these attacks, current OSes support a variety of anti-malware solutions. These solutions usually depend on the services from the underlying OS kernel, implying that they would only work as designed when the integrity of the kernel is ensured. However, the kernel integrity has been seriously threatened since the advent of *kernel level rootkits* that manipulate the kernel so as to achieve certain goals (i.e., concealing their existence or providing backdoor accesses). Because the kernel operates at the highest privilege level in the system, the compromised kernel may nullify the effectiveness of any anti-malware measures that have their root of trust on the kernel.

The threat of rootkits have urged researchers to conduct much study to seek a more secure computing base that can safely *monitor* the system and ensure the kernel integrity even in the presence of rootkits. Two mainstream of the research directions are *hypervisor-based* [1]–[3] and *hardware-based* approaches [4]–[8]. In general, the former approaches have popularity in the security community as they do not necessitate underlying hardware modification while providing a higher privileged, thus safer, software layer for monitoring than the kernel does. However, the latest attacks [9] and reported vulnerabilities [10] pointed toward the probability that the code and data of hypervisors can also be compromised at runtime. Although the known vulnerabilities have been fixed shortly, the growing complexity of hypervisors implicates that there would be more vulnerabilities revealed in the near future.

The hardware-based approaches utilize an isolated hardware module physically independent of the monitored host system [5]–[8]. In particular, prominent monitoring schemes are recently proposed in [6]–[8]. At the center of these approaches, there is a hardware monitor, which we hereafter call the *snoop-based monitor*, whose role is to detect malicious attempts to alter the kernel by snooping every data traffic between the host CPU and main memory. Being located at the outside of the host as a dedicated hardware unit, the monitor is not only immune to rootkits attacks on the host, but also able to constantly observe the memory access behaviors of rootkits revealed on the system bus without affecting the host performance.

Although snoop-based monitors have been working well in their environments and assumptions, we have recently discovered a potential vulnerability which future attackers might exploit. It comes from the fact that most computer systems employ write-back caches. Being located in between the host CPU and main memory, caches hold copies of data or instructions recently accessed by CPU, thereby boosting the overall system performance to a large extent. However, for the perspective of snoop-based monitors, the existence of caches can be disadvantageous because they shall reduce the number of events that the monitors can watch. For example, if a rootkit tries to compromise the kernel by modifying sensitive data, and the very data hits in the cache, then the write traffic would not appear on the system bus, rendering the monitor oblivious of the write event.

Even though some previous works discussed the possibility that this problem may seriously undermine the effectiveness of their approaches [6]–[8], none of them has properly addressed this *cache-induced hiding* (CIH) effect problem. In [7], they tried to avert the problem by restricting the usage of their monitors to the systems with write-through caches. In [8], they merely mentioned a simple scheme of using periodic cache flush. Unfortunately, they did not provide any empirical data about how much loss their scheme may suffer on performance, detection rate or power consumption. However, as we will see later, our study evinces that frequent cache flush might increase the host performance overhead to a large extent.

In this paper, we propose a hardware-assisted low-overhead solution which thwarts the CIH effect by enabling the external monitors to directly access the *cache resident information* (CRI) which includes all the internal data residing within the cache without being exposed on the system bus. To implement this solution, we utilized the existing hardware logic, called the *core debug interface* (CDI), which can be found in several processors available today such as ARM Cortex series and Xilinx MicroBlaze [11], [12]. CDI has been conventionally used to supply the information relevant to the CPU internal state for the *on-chip debug* (OCD) unit [11], [12]. If CDI is plugged into a security monitor, the bountiful information provided by CDI, which contains memory access events issued by CPU, would certainly help monitor perform its desired task without the CIH effect.

This task, however, involves several complications in implementation majorly because the initial set of signals from CDI cannot be simply fed into the security monitor as they are in their present form. Some signals originally generated for

978-3-9815370-4-8/DATE15/©2015 EDAA

Signal	Description
ETMICTL [20:0]	ETM instruction control bus
ETMIA [31:1]	ETM instruction address
ETMDCTL [10:0]	ETM data control bus
ETMDA [31:0]	ETM data address
ETMDD [63:0]	ETM data write data value
ETMCID [31:0]	Current processor Context ID

TABLE I. DESCRIPTION OF CDI SIGNALS FOR ETM

debugging must be translated into another form that is required for security monitoring. Therefore, we have developed an extra hardware unit, called the *Extrax*, that being located between CDI and security monitors, carefully examine and properly refine or transform each individual signal from the interface before delivering it to the monitor.

To validate our design and further explore the implication of this additional circuits to the overall system, we have implemented a full snoop-based monitoring system in which the host system has been augmented with Extrax. With the system prototyped on a FPGA platform, we evaluated and compared the performance, power and area of our full system against the baseline system in which Extrax is not deployed. Experiment results exhibit that our monitor, with modest area and power overhead but with the host performance being almost unaffected, successfully detects rootkit attacks regardless of the type of caches while the baseline monitor often fails.

The rest of the paper is organized as follows. We first present the background information and a motivational example in Section II. Then in Section III, the assumption and threat model are presented. In Section IV, the baseline system is presented to show the overall operation of hardware-based monitoring. Section V describes the details of the proposed security extension, Extrax. After Section VI shows our experimental results, we conclude this paper in Section VII.

II. BACKGROUND

A. Core Debug Interface

The on-chip debug (OCD) unit is a debug/trace module that has been widely adopted to the commodity processors. Provided by OCD, a rich set of information allows users, on their desktop environment, to follow path that the target CPU takes as a result of code execution and monitor values in various registers and memories. CDI is an interface placed on the CPU side, and provides OCD with the CPU's internal status information. In this paper, we specifically focus on the interface that provides instruction address, current context ID (or process ID), and data address/value of memory access instructions in real-time without affecting the performance of the target processor. A representative example of OCD that supports such a tracing capability is the *embedded trace macrocell* (ETM) module of ARM [13], and the signals to ETM provided by ARM through CDI is described on Table I.

B. Cache Resident Attacks

We define *cache resident attacks* as malicious attacks that, intentionally or unintentionally take advantage of the CIH effect; the existence of write-back caches can unintentionally blindfold snoop-based monitors by impeding memory write events from appearing on the system bus, or attackers can intentionally hide the evidence of attacks by overwriting the malicious data residing in caches with benign one, thereby prohibiting the monitors to detect the symptom of attacks. To better explain, we chose the *loadable kernel module (LKM)*

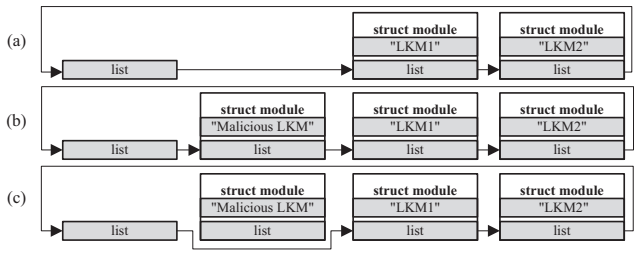

Fig. 1. Cache resident LKM hiding attack

hiding technique as a representative cache resident attack example since many rootkits in the wild employ the technique to hide themselves. LKMs are initially designed to support extension of the kernel code at runtime without recompiling the entire kernel. However, they are often used by attackers to conceal malicious processes, files or even themselves from detection mechanisms. Adversaries achieve their goal of hiding LKMs by directly modifying the kernel data structures that maintain the list of loaded LKMs.

Figure 1 shows how the LKM hiding technique is affected by write-back caches. In (a), there are several LKMs, each of which is represented by the **struct module**. The kernel handles the LKMs by maintaining the **modules** list, which is a linked list of **struct module**. Upon the module load request, in this case a request from the malicious LKM depicted in (b), the kernel adds the corresponding **struct module** to the **list**, which is the head of the **modules** list. In a system with write-back cache, the **list** will be cached after this step, and subsequent accesses to the data structure will also hit in the cache. Thus, even if the malicious LKM removes itself from the **modules** list by directly manipulating the pointers of the **modules** list as depicted in (c), this event might not be placed on the system bus. Consequently, recently proposed snoop-based monitors might no longer guarantee the integrity of the kernel since they detect attacks by snooping the system bus. Hence, a novel way to nullify CIH effect should be devised.

III. ASSUMPTIONS AND THREAT MODELS

We use the assumption taken by previous snoop-based monitors, especially by KI-Mon [7]. Therefore, we assume that adversaries have already gained administrators' privilege on the host system and thus are able to install rootkits to hide themselves or leave backdoors to the host system; for instance, the attackers can install LKMs or place hooks on critical system calls. However, we rule out *physical attacks* by an insider who has direct access to the host system and *direct kernel structure manipulation attacks* proposed in [14].

In addition, we also assume that the host system uses write-back caches, and provides CDI, that can be connected to OCD. Side-channel attacks that exploit the information from OCD/CDI are not considered in this work.

IV. THE BASELINE SYSTEM

A. The Overall System Design

Figure 2 shows a high level view of the baseline system. Since our monitoring system employs the snoop-based monitoring scheme, it is similar to the prototype of KI-Mon [7]. To ensure the integrity of the kernel, the monitor side core dynamically configures the hardware ASIC units, especially the *snooper*, based on a security policy.

We designed our baseline monitor to support various policies on detecting attacks on *immutable regions* and *mutable objects*

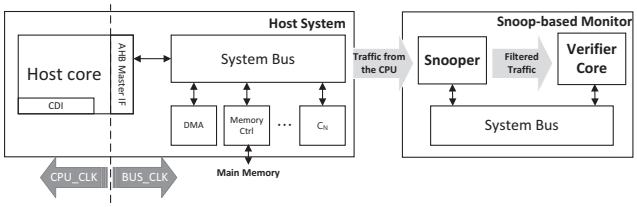

Fig. 2. The overall baseline system design

that have *invariant value sets*, as KI-Mon proposed in [7]. Immutable regions contain data that should not be modified after the boot process is complete, such as the *system call table* (SCT) and *interrupt descriptor table* (IDT). Kernel mutable object with a invariant value set is data object in which data can be updated by the kernel at run-time, but the updated value is chosen among the set of possible values that can be profiled prior to run-time; for instance, many function pointers within kernel objects are known that each function pointer points to one of its possible candidate landing sites [7].

According to the security policy of the host CPU, the snooper is configured with appropriate address ranges of the kernel data objects or regions to be monitored on main memory. Then the snooper constantly acquires write events placed on the system bus, filters out every benign event that is not relevant to the monitored regions, and transfers only the ones that violate the current security policy to the *verifier core* for further investigation.

While attacks on immutable regions can be easily detected by simply snooping the bus for write events on the region, catching evidence of malicious modifications on kernel mutable objects with invariant value sets is not straightforward since mere write events cannot be regarded as a symptom of attacks. Therefore, we employ techniques similar to the ones proposed in [7] such as the *whitelisting-based verification* and *callback-based semantic verification* that basically verifies the written values as well. The detailed explanation of these techniques are omitted in this paper since our work is focused on overcoming the CIH effect rather than suggesting new detection schemes. Therefore, readers interested in these techniques are kindly referred to [7].

The key difference between our baseline system and the prototype of [7] is that ours uses write-back caches. Therefore, snooping only the system bus may cause detection failure because of the aforementioned CIH effect.

B. Periodic Cache Flush for Cache Resident Attacks

As mentioned in [8], periodically flushing caches might help reveal more CRI on the bus when write-back caches are deployed in the system. To show the effectiveness of the scheme, we applied it to the baseline system. When implementing the scheme, the flush period should be decided with great care because a reckless choice of the period may induce either non-negligible overhead or detection failure.

For attacks that aim at immutable regions [6], the cache flush period, **p**, can be selected arbitrarily because any write attempt on the regions is deemed malicious [6], and the cacheline where the written data is located will eventually be evicted to main memory regardless of the period **p**. However, the decision of the period **p** for attacks that target kernel mutable objects is far more difficult since attackers can usually figure out a way to avoid detection by slightly modifying the original attacks.

To better explain, consider the case shown in Figure 1. Assume that the state of the **modules** list changes from the state (a) to (b) on time **s**, and from the state (b) to (c) on time **e**. In principle, the baseline monitor concludes that an LKM is malicious when the LKM is removed from the **modules** list (state change from (b) to (c)) while the corresponding memory region for the LKM remains in memory [7]. Thus, the period of cache flush **p** should be shorter than the interval **d=e-s** so that every event on **modules** list can be revealed on the system bus before the adversary achieve her own goal. Since **d** in a primitive LKM hiding technique is sufficiently large, **p** of our baseline with periodic cache flush could be long as well, so as to detect the attack with acceptable performance overhead.

However, by slightly changing the original LKM hiding technique, it is possible for attackers to reduce **d** substantially. The devised technique requires two LKMs, one of which is the malicious LKM and the other is an LKM that merely hides the first one. We call the latter the *hider LKM* whose role is to insert a callback function to the kernel timer, and set the timer with a period **q**. Then the callback function is periodically invoked to check whether the malicious LKM, which ultimately achieves the attackers goal, is inserted or not, and hide the newly added one upon detection. To insert and hide a malicious LKM, attackers first insert the hider LKM with an arbitrary period **q**, and insert the malicious one sometime later. Since the hider LKM does not hide itself and is thus added and removed legitimately, the monitor has no way of distinguishing the hider LKM from other normal LKMs.

Thus, to defend against such cache resident attacks, the flushing period **d** should be adjusted to a very small value. Our preliminary study showed that, in order to attain 100% detection rate, the period **d** need to be reduced to 30us, resulting in increased performance overhead of up to 84%. From this result, we claim that the detection with periodic cache flush might not only induce huge performace overhead, but would also cause failing in detection if attackers know the existence of periodic cache flush and modify their attacks to reduce **d**.

V. EXTRAX DESIGN

As long as CRI is accurately sent via CDI to snoop-based monitors, many security threats due to cache resident attacks would be resolved without modifying the host internal architecture. Unfortunately, realizing precise and efficient deliverance of these signals in an actual system comes at a cost with some implementation challenges, as briefly stated in Section I. Below is summarized two of those that must be resolved in order to efficaciously transfer the host internal information to snoop-based monitors through the existing CDI.

1. CDI is originally designed to send a virtual address (VA) to the OCD unit for each memory access while the monitor demands physical addresses (PA) so as not to be disrupted by the certain type of attacks which will be discussed shortly. Therefore, the original VAs from CDI cannot be directly used for a snoop-based monitor.

2. The number of memory events coming from CDI is far larger than that of those appearing on the system bus because a majority of write events originating from CPU are hidden by caches on the way to the bus. This excessive number of events sent from CDI could be burdensome to the monitoring system in terms of performance.

These key issues are tackled respectively by two new hardware units, the *address translation unit* (ATU) and the *early stage filter* (ESF), both of which constitute our Extrax.

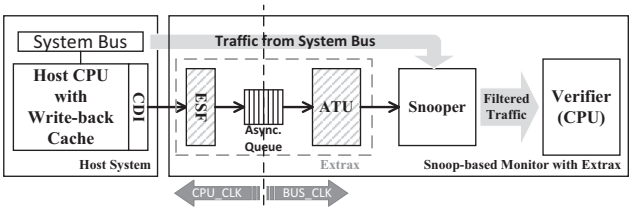

Fig. 3. The augmented baseline system with Extrax

Figure 3 depicts a block diagram of our proposed system where the baseline is extended with Extrax. It is noteworthy here that even though the information conveyed through ATU can cover all memory events, the original path from the system bus remains the same. The purpose of the path is mainly to detect attacks from other *bus masters* such as DMA. In this section, we will focus our discussion on these hardware units.

A. Address Translation Unit

Upon receiving VAs from CDI, a snoop-based monitor should decide whether the current memory access targets the regions it monitors. It seems that such a decision can be made based on VAs, but there are cases in which the monitor necessities PAs. For instance, consider the case depicted in Figure 4 where the monitor is protecting a kernel data structure contained in the critical page frame. Since this data structure is critical to the integrity of the system and is thus managed by the kernel through the *kernel page table*, no arbitrary mapping should be made to the data structure. However, in the presence of kernel-level rootkits, it would be possible for attackers to simply insert an LKM that generates another page table mapping for the data structure, denoted as the *malicious mapping* in the figure. Thus in this situation, if the monitor used the original VA to protect this type of kernel structures, the attackers could indirectly modify the kernel data with the newly mapped VA, hence successfully escaping the vigilance of the monitor.

To deal with the cases where PAs must be supplied for security monitors, we have installed ATU that translates VAs from CDI into physical ones. The overall architecture of ATU is displayed in Figure 5. ATU is configured by the monitor through the advanced high-performance bus (AHB) slave interface. On a translation lookaside buffer (TLB) miss, a *page table walk* is initiated through the AHB master interface, which is connected to the host system bus. The input to ATU includes a VA, the *context ID* and the base address of the Linux page global directory (PGD). The former two inputs are provided by CDI, while the latter one is configured through the AHB master interface immediately after the current context ID is updated. TLB is a fully associative table in which the number of entries can be configured from 16 to 32. It has a random replacement policy.

B. Early Stage Filter

With the help of CDI, our monitor is now able to snoop every write event generated by the host CPU without suffering

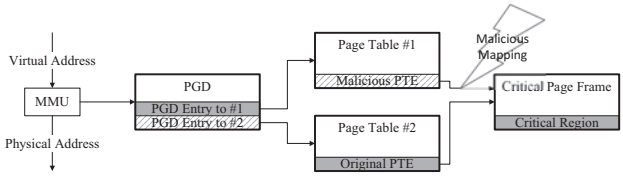

Fig. 4. Multiple virtual addresses mapping example

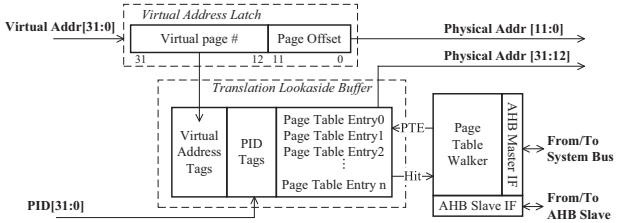

Fig. 5. The structure of the ATU

from the CIH effect. This abundant information continuously streaming into the monitor will certainly enhance the chance to detect cache resident attacks. However, it may also create an excessively large volume of information flow that will inevitably impose heavy burdens on the monitor.

Though, if we remind that the monitors usually need to watch only a subset of the memory events of the host depending on security policies, it would be wasteful if ATU exhaustively translates all incoming addresses for the monitor. The problem can be alleviated if we can filter out benign events based on a security policy. As an example, for one of the policies considered in our experiments, the monitor is interested in write attempts to the kernel data. Therefore, any read memory events can be safely discarded before reaching either the monitor or even ATU. The filter operations in this case is in fact rather straightforward since memory access types are easily discernable right after the events occur. However, we often need to apply more aggressive filtering to the events. As briefly mentioned in Section IV, the kernel data of interest occupy relatively small amount of memory compared to the whole range of main memory. Therefore, the monitor just needs to watch the access events on this limited region. Unfortunately, it is not always straightforward to decide whether or not an event just emitting from CDI falls into this region, since its address remains virtual before reaching ATU.

Nevertheless, there is still a way to provide a solution to this decision problem. For this, consider Figure 5 where we see that an address translation step does not require the *page offset* field; in fact, this field is identical for both VAs and PAs. Inspired by this fact, we implemented ESF which, being placed between CDI and ATU, removes unnecessary memory events based on page offset before the events reach ATU, thereby reducing the number of events delivered to ATU.

Figure 6 represents the internal block diagram of ESF, which rearranges the signals from CDI and filters out as many events as possible. The implementation of output rearrangement is somewhat simple in that it merely reorganizes and extracts signals that are needed for the monitors to detect attacks. Among the signals introduced in Section II, the addresses/values of memory write instructions and context IDs are selected and rearranged for monitors.

In the current implementation, ESF has eight 12-bit *address range register* pairs and comparators. The address range register pairs of ESF contain the page offset field of start/end addresses that need to be monitored. The register values can be configured by the verifier core at any time. Therefore, whenever the critical kernel regions of interest are updated, the monitor configures the snooper and ESF simultaneously.

After the configuration, ESF compare the page offset field of a VA that comes from CDI with the values of address range register pairs. If the address is included in any of the monitoring regions, ESF enables the filter output register so

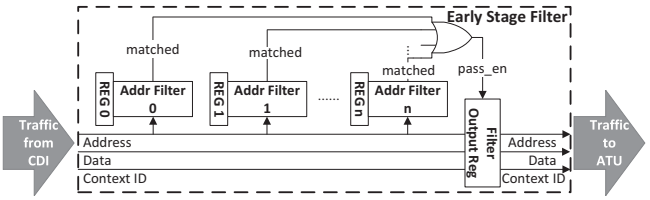

Fig. 6. The overall structure of the early stage filter

that the event can flow to ATU. Otherwise, ESF blocks the address, thus obviating unnecessary operations in ATU.

As mentioned before, current ESF implementation has 8 pairs of address range register, limiting the number of concurrent monitoring regions. Note that the number of registers can be adjusted for the environment of deployment. Alternatively, we can loosely set the address range register pairs, so that each pair contains more than one monitored region. Although it would produce unnecessary events for ATU to handle, ESF still do not miss any access to the monitored regions.

VI. Experimental Results

A. Prototype System

We have implemented our baseline system as close as possible to the design proposed in [7], as an FPGA prototype where the host CPU is the SPARC V8 processor, a 32-bit synthesizable core [15] which uses a single-issue, in-order, 7-stage pipeline. It has separate 16kB L1 caches for instruction and data. In addition, the host CPU also has a 256kB L2 cache which employs the write-back policy. The host system bus compliant with the AMBA2 AHB/APB protocol is used to interconnect all modules in the system, and Linux 2.6.21.1 is used as the host OS. The snoop-based monitor system is also implemented with the same processor and the system bus. The snooper has eight sets of address range registers which can be configured according to the security policy of the monitor.

To evaluate our approach, we augmented the baseline system with Extrax. Although our host processor, open-source synthesizable core [15], provides their own CDI specification, the information comes out of CDI is quite restricted compared to that of commercial product, such as ARM. Therefore, we slightly extended it to support the CDI signals equivalent to those of ARM architecture (see Table I). Thus, both ESF and ATU are implemented to be compatible with ARM CDI specification [13]). Since CDI is connected to both Extrax and OCD, we designed Extrax to disable signals to OCD when snoop-based monitoring is turned on. ESF is configured to have 8 address range register pairs. Our ATU, compliant with SPARC V8 Reference MMU [16], has been configured to have 16 TLB entries and 16 input queue entries.

Based on the parameters for the prototype as described above, we synthesized our system onto a prototyping board with a Xilinx SC5VLX330 FPGA. Table II provides the area of the baseline system and Extrax in terms of lookup tables for logic (LUTs), block RAMs (BRAMs) and DSP slices (DSP48E). It shows that Extrax incurs 12.09% overhead for LUTs as compared to the baseline hardware. Even though the area overhead of our Extrax seems non-negligible, it is noteworthy here that our baseline system, the open-source synthesizable core based on SPARC V8 architecture [15], has indeed very small size. Therefore, we claim that the area overhead of Extrax might be quite acceptable if deployed on the system with commercial CPU core such as Cortex-A9.

Category	Component	LUTs	BRAMs	DSP48E
Baseline System	SPARC V8 Core with L1/L2 Cache (Host System)	6856	86	4
	SPARC V8 Core with L1 Cache Only (External Monitor)	5878	15	4
	Bus components (AHB Buses + AHB/APB bridges)	908	0	0
	Memory Controller	57	0	0
	Snooper	3318	0	1
	Peripherals (TIMER, UART, and etc.)	2480	4	2
	Total Baseline System	**19497**	**105**	**11**
Extrax	Early Stage Filter (ESF)	502	0	0
	Address Translator Unit (ATU) including queue	1855	0	0
	Total Extrax	**2357**	**0**	**0**
	% Extrax over Baseline System	**12.09%**	**0.00%**	**0.00%**

TABLE II. Synthesis result of the prototype system

B. Security Evaluation

To evaluate the security monitoring capability of our approach, we chose several well-known attack techniques that are employed in real-world rootkits [8] and implemented four rootkits that target either immutable regions or kernel mutable objects. Table III lists the rootkits, of which the specific target can be deduced by their names. The first two target immutable regions while the others, the LKM and *virtual file system* (VFS) hooking attacks target mutable objects that have invariant value sets. We also implemented monitoring software that runs on our verifier core to configure the peripheral units for monitoring, such as the snooper, ESF and ATU. The current address range registers of ESF is configured to capture memory accesses only on kernel mutable objects and other related data structures such as page tables for the objects. Memory events on immutable regions can be safely filtered out by ESF since, as mentioned before, snooping the system bus would be enough to catch attacks on immutable regions.

To demonstrate the effectiveness of Extrax, we injected the four rootkits into three system versions: (*BaseWT*) the baseline system with write-through caches as in [7], (*BaseWB*) the baseline system with write-back caches and (*Ours-Extrax*) our proposed system with Extrax. As seen in Table III, the monitor in BaseWT could immediately detect all the rootkits since the host uses a write-through cache, thus immediately sending every write event onto the system bus. The monitor in BaseWB, however, was unable to detect attacks on kernel mutable objects because of the CIH effect. Ours-Extrax, on the contrary, could detect all the attacks (whether cache resident or not), thanks to our Extrax and CDI support.

Recent attackers tend to avoid launching attacks that are easily detectable like those on immutable regions. Instead, of more importance becomes the detection of attacks on mutable regions [17]. We have just seen that a snoop-based monitor deployed on the host core with a write-back cache is easily nullified when cache resident attacks are made on mutable objects. Therefore, we claim that Extrax can play a critical role in assisting such monitors, thereby increasing the security level of the systems.

C. Performance Analysis

Since CDI does not introduce any performance impact on the host, the main factor which incurs the overhead is the traffic generated by ATU as a result of address translation. To measure the performance overhead, we chose seven applications from the *SPEC 2006 benchmark suites* [18], and implemented two

Example Name		BaseWT	BaseWB	Ours-Extrax
Immutable regions	**IDT Hooking**	Detected	Detected	Detected
	SCT Hooking	Detected	Detected	Detected
Mutable objects	**LKM Hiding**	Detected	Not detected	Detected
	VFS Hooking	Detected	Not detected	Detected

TABLE III. Rootkit detection result

Application	Baseline System	Proposed System with ESF turned-off	Proposed System with ESF turned-on
h264	10.46s	10.57s	10.46s
bzip2	788.05s	813.62s	788.05s
hmmer	39.96s	39.96s	39.96s
libquantum	10.11s	10.11s	10.11s
parser	1.41s	1.41s	1.41s
omnetpp	969.1s	975.35s	969.1s
xalan	3.43s	3.46s	3.43s

TABLE IV. PERFORMANCE OVERHEAD

versions of the host system: the baseline and the proposed full system with ESF turned off. The reason of turning ESF off is to strain the system with the excessive traffic generated by CDI in the worst-case scenario.

Table IV represents this worst-case performance overhead, which is around 3.24%. The reason for this low overhead might be explained in a way that even if there seem to be a number of memory events coming from CDI, the number of events that really need memory translation in ATU is relatively small because our ATU has TLB and most memory translation end up retrieving values from the TLB. We also conducted the same experiment, with ESF turned on, monitoring the mutable objects related to the VFS hooking and LKM hiding attacks. As seen in the table, there is virtually no overhead caused by Extrax because ESF filters out most memory events that do not access the monitored memory regions.

Since turning off ESF does not cause serious performance overhead of 3.42%, some might think that ESF is not essential. As explained before, however, its main goal is to reduce the amount of CRI delivered to ATU. Reduced number of memory events would not only help ATU decrease the number of events that need address translation (main memory access), but it would also drastically reduces the number of TLB accesses, which in turn might possibly save hugh amount of power consumed by ATU.

D. Power Consumption

To assess the power consumption of Extrax, we used Synopsys Design Compiler, Mentor Graphics ModelSim, and synthesized netlists of Snooper, ATU and ESF. Switching activity interchange format (SAIF) files were extracted from Modelsim with synthesized input test sequences that maximizes the power consumption of each component. Then the SAIF files and netlists were given as the input to Synopsys Design Compiler with a commercial 45 nm process library to estimate power consumption. The results are presented in Table V with other commodity processors as reference machines. As shown in the figure, Extrax consumes relatively small power as being compared to commodity processor cores in products ranging from low- to high-end computing devices.

VII. CONCLUSION

In this paper, we proposed to reuse the CDI feature readily available for debugging in modern CPU cores in an effort to elevate the effectiveness of existing snoop-based monitors. We first discussed several implementation complications involved in the transfer of CDI signals for snoop-based monitors located outside the host CPU. Then we suggested Extrax which is an ASIC module plugged into CDI in order to convey the host internal information from CDI to the external monitor. For precise and efficient monitoring, the module performs the tasks of address translations and filtering out benign memory write events. To validate our proposed design, we have implemented a prototype on FPGA, and evaluated the security capabilities in addition to the performance, power and area overhead. Empirical results showed that our monitor, regardless of the type of caches, successfully detect all our rootkit samples, which the previous monitoring systems often failed to catch owing to CIH effect, with modest area and power overhead increase along with virtually no host performance overhead.

Component	ARM Cortex-A9 (Dual Core, no L2)	ARM Cortex-A15 (Dual Core, 1MB L2)	SPARC V8 (Single Core, 256KB L2)	Snooper	ATU	ESF
Process	40nm	30nm	45nm	45nm	45nm	45nm
Power Consumption	0.5W (@ 800 MHz) /1.9W (@ 2GHz)	3W (@ 1.7GHz)	188mW (@200 MHz)	7.24mW (@200MHz)	4.84mW (@200MHz)	560.8uW (@200MHz)

TABLE V. POWER CONSUMPTION ANALYSIS

ACKNOWLEDGMENT

This work was partly supported by the IT R&D program of MSIP/KEIT [K10047212, Development of homomorphic encryption supporting arithmetics on ciphertexts of size less than 1kB and its applications], the Brain Korea 21 Plus Project in 2014, Software R&D Center of Samsung Electronics, and the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (No. 2014R1A2A1A10051792).

REFERENCES

[1] N. L. Petroni Jr and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 103–115.

[2] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with osck," in *ACM SIGPLAN Notices*, vol. 46, no. 3. ACM, 2011, pp. 279–290.

[3] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 380–395.

[4] Y. Bulygin and D. Samyde, "Chipset based approach to detect virtualization malware," *BlackHat Briefings USA*, 2008.

[5] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor." in *USENIX Security Symposium*, 2004, pp. 179–194.

[6] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, "Vigilare: toward snoop-based kernel integrity monitor," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 28–37.

[7] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang, "Ki-mon: a hardware-assisted event-triggered monitoring platform for mutable kernel object," in *USENIX conference on Security*. USENIX Association, 2013.

[8] Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi, "Cpu transparent protection of os kernel and hypervisor integrity with programmable dram," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM, 2013, pp. 392–403.

[9] "The blue pill, doi=http://theinvisiblethings. blogspot.com/2008/07/0wing-xen-invegas.html."

[10] "Vmware: Vulnerability statistics."

[11] W. Orme, "Debug and trace for multicore socs," Sep 2008.

[12] *MicroBlaze Processor Reference Guide*, Xilinx, Apr 2012.

[13] *Embedded Trace Macrocell Architecture Specification*, ARM, 2011.

[14] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, "Dksm: Subverting virtual machine introspection for fun and profit," in *Reliable Distributed Systems, 2010 29th IEEE Symposium on*. IEEE, 2010, pp. 82–91.

[15] *GRLIB IP Core User's Manual*, Aeroflex Gaisle, January 2012.

[16] *The SPARC Architecture Manual*, SPARC International, Inc., 1992.

[17] N. L. Petroni Jr, T. Fraser, A. Walters, and W. A. Arbaugh, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data," in *USENIX Security Symposium*, 2006.

[18] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737