Hardware-based On-line Intrusion Detection via System Call Routine Fingerprinting

Liwei Zhou and Yiorgos Makris
Electrical Engineering Department, The University of Texas at Dallas, Richardson, TX 75080, USA
E-mail: {lxz100320, gxm112130}@utdallas.edu

***Abstract*—We introduce a hardware-based methodology for performing on-line intrusion detection in microprocessors. The proposed method extracts fingerprints from the basic blocks of the routine executed in response to a system call and examines their validity using a Bloom filter. Implementation in hardware renders spoofing attacks, to which operating system or hypervisor-level intrusion detection methods are vulnerable, ineffective. The proposed method is evaluated using kernel rootkits which covertly modify the system call service routines of a Linux operating system running on a 32-bit x86 architecture, implemented in the Simics simulation environment, while hardware overhead is evaluated using a predictive 45nm PDK.**

I. Introduction

As reliance of our daily activities on technology increases, so does the amount of sensitive information that is stored, exchanged, and processed in electronic form. As a result, the threat of malicious software, which seeks to steal such sensitive information or disrupt legitimate operation of a computing system, is more relevant than ever. Accordingly, development of reliable intrusion detection methods has become paramount.

Intrusion detection refers to the process of logging or monitoring operating system (OS) activities, in order to detect malicious events. State-of-the-art solutions typically rely on extracted signatures (i.e. fingerprints), or behavioral models to distinguish malware from legitimate applications and, thereby, detect intrusion [1]–[6]. Numerous methods have been developed at the OS level, leveraging its rich semantic information and flexibility [1]–[3]. OS-level solutions, however, can be subject to software attacks staged at the same level. Kernel rootkits, for example, may be used to hijack OS control flow so that it can spoof their logging/monitoring system and eliminate traces associated with malicious actions.

In order to overcome this limitation, hypervisor-level detection methods have also been proposed [4]–[6]. A hypervisor provides virtualization, allowing multiple guest-OSs to run concurrently on a single physical machine, without intruding each other's context. A management core, isolated from the guest-OSs, may naturally provide ground for more secure detection methods. Thus, data collected by an intrusion detector at the hypervisor level is generally immune to OS-level software attacks. Nevertheless, as shown through recent work [7], the hypervisor itself can be the target, as several vulnerabilities and intrusion methods have been identified. Therefore, similar attacks compromising integrity of the logged data to conceal malicious events can be staged at the hypervisor level.

In contrast, in this work we explore the possibility of relying exclusively on data collected directly through the hardware, without the intervention of a hypervisor or an OS, whereby the logged information may be compromised. In essence, we seek to leverage the fact that it is not possible to hide the actions of any executed software, even software that seeks to hide itself, from the hardware. Accordingly, traces obtained from hardware are expected to be immune to software-based tampering. The same principle governs recent malware detection/workload forensics approaches introduced in [8]–[10].

Since the majority of kernel rootkits tamper with the control flow of kernel services, using system call-related information is a popular approach for developing OS-level and hypervisor-level intrusion detection methods. Consistent with this approach, in this work we develop a hardware-based method which fingerprints system call service routines in order to assess their validity. Specifically, a Multiple Input Signature Register (MISR) is used to generate compact fingerprints for each basic block of a system call service routine, while a Bloom filter is employed for assessing whether a fingerprint is valid. This idea is demonstrated through execution of both legitimate benchmarks and kernel rootkits on an x86 architecture running Linux OS, implemented in Simics.

The remainder of this paper is structured as follows. In Section II, we discuss related intrusion detection work. The threat model considered in our approach is introduced in Section III. The proposed method is presented in Section IV and the hardware implementation of the required components is described in Section V. Section VI evaluates experimentally the effectiveness of our method and estimates its hardware overhead. Limitations and possible extensions are discussed in Section VII, while conclusions are drawn in Section VIII.

II. Related Work

Based on the level at which they are implemented, intrusion detection methods found in the literature can be broadly categorized into software-based approaches, including OS-level approaches as well as hypervisor-level approaches, and hardware-based approaches. Most of the software-based approaches leverage the effectiveness of using system call-related information to identify malicious behavior while hardware-based approaches exploit specific architecture-level information to distinguish between benign and malicious events.

A. *OS-level approaches*

OS-level approaches generally benefit from semantically-rich information, such as process ID, system call sequence, etc., which is available at this level. Program behavior can, therefore, be modeled through machine learning algorithms, based on multiple features extracted from such information, in order to identify malicious programs [1]. Alternatively, Control Flow Integrity (CFI) checking has been proposed as a promising defense against control flow hijacking attacks of OS kernel services. However, the incurred implementation and/or runtime overhead is relatively high [2], [3].

B. Hypervisor-level approaches

Hypervisor-level approaches benefit from the inherently higher security offered by virtualization and isolation, as mentioned in Section I. On the other hand, these approaches suffer from the semantic gap problem. To address this, Antfarm [11] uses the CR3 register available in the x86 architecture to identify process creation, switching and termination. Once the semantic gap is bridged, similar methods to the ones developed at the OS-level may be applied. For example, the system call sequence can be profiled through performance counters or extracted from the instruction flow and specific registers (instead of a software tracing tool, such as `strace`), in order to perform behavior-based modeling and analysis [4]–[6].

C. Hardware-based approaches

Hardware-based approaches are generally immune to software attacks, thereby enabling a new direction toward thwarting malicious software. Existing methods include statistical processing of performance counter contents or memory operation information directly in hardware to identify outlier behavior [8], [9]. A similar approach which exploits Translation Lookaside Buffer (TLB) profiling has also been proposed in the context of workload execution forensics [10].

III. Threat Model

In this section, we define the threat model that we focus on in this paper. Our choice is motivated by the fact that most OS/hypervisor-level intrusion detection methods rely heavily on system call-related information, yet their logging/monitoring mechanisms rarely inspect the actual system call execution flow. As a result, these methods can be vulnerable to software attacks, known as *system call hijacking*.

System call hijacking enables an attacker to control the execution flow of one or several system calls; thereby, malicious code can be introduced and executed without the knowledge of the legitimate system user. In Linux OS, for example, this can be achieved through a kernel rootkit exploiting the Loadable Kernel Module (LKM), which contains user defined code and is intended to extend or customize the functionality of the original kernel. Compounding the problem, when the extended or customized functionality is no longer required, the kernel module can be unloaded, restoring the kernel to its original state and, thereby, leaving no trace.

Our work focuses on *three types* of system call hijacking, which are depicted in Figure 1:

1) **System Call Table Redirection:** The simplest attack using the LKM is a redirection of the system call table. When a system call is invoked, instead of querying the original system call table to access the corresponding service routine, the OS is redirected to a different table whose content is controlled by the attacker and whose existence is unknown to the OS and the legitimate users.
2) **System Call Table Modification:** Another attack option is to modify the value of certain entries in the original system call table rather than redirecting to a different table, as the latter may leave more of a trace and can be

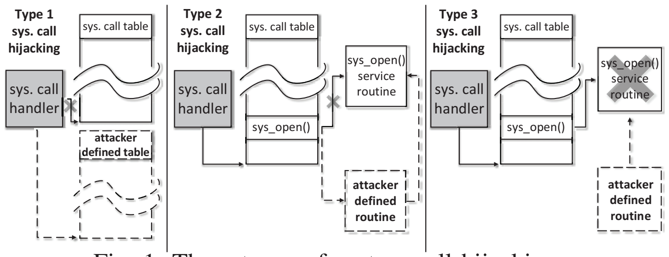

Fig. 1: Three types of system call hijacking

easily detected. In this way, the attacker can redirect the service routine of certain system calls to his/her own malicious code snippet. After the malicious code finishes, control is returned to the original service routine so that the OS remains oblivious to the attack.
3) **Service Routine Modification:** A more complicated option is to directly modify the service routines of one or more system call. In this case, the system call table as well as its entries remain unmodified, yet the actual service routines are contaminated with additional/different instructions. A smart attack may still operate covertly and hide its presence from the OS by incorporating the general service provided by the original routine.

Armed with the capability of system call hijacking, an attacker can easily spoof OS-level and hypervisor-level intrusion detection methods. Indeed, upon invocation of a system call, these methods typically log and validate the system call ID or series of IDs, yet have no mechanism of attesting that the legitimate service routine is executed.

IV. Proposed Method

In contrast to OS-level and hypervisor-level approaches, the primary objective of the method proposed herein is to ensure integrity of an executed system call service routine in a way that is immune to tampering by software. To this end, we introduce a hardware-based solution, wherein the information used for fingerprinting the executed system call service routine is generated and evaluated directly in hardware. In this way, there exists no physical pathway for the OS, hypervisor, or any application running on the system to interfere with the logged data and the validation mechanism.

The proposed hardware-based intrusion detection system, as Figure 2 shows, consists of two main components: a *data logging component* and a *validation component*. The data logging component collects three critical pieces of information related to the integrity of system call execution, namely the base address of the system call table, the contents of the system call table, and the actual system call service routines. Using this information, the validation component seeks to detect the three types of system call hijacking attacks included in our threat model. The base address of the system call table, as well as its contents, are retrieved through the Basic Input/Output System (BIOS), when the OS kernel is booted. Therefore, we also store this information directly in the hardware and contrast it against the real-time values every time a system call invocation occurs, so that any unexpected modification can be detected and an alarm can be raised to suspend execution.

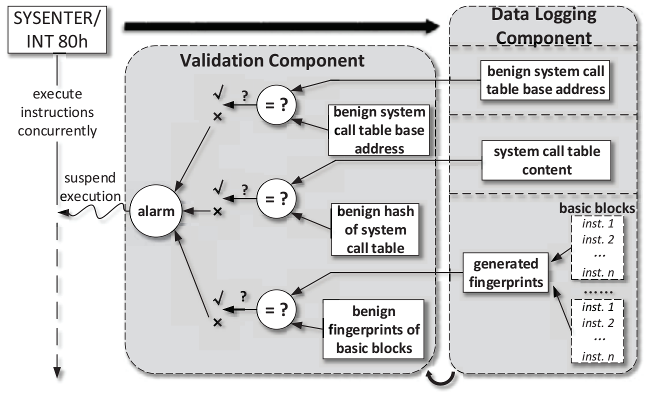
SYSENTER/ INT 80h
execute instructions concurrently
suspend execution
Validation Component
Data Logging Component
alarm
= ?
benign system call table base address
benign hash of system call table
benign fingerprints of basic blocks
system call table content
generated fingerprints
basic blocks
inst. 1
inst. 2
inst. n

Fig. 2: High-level design of proposed method

Attesting validity of the actual system call routine, however, is slightly more involved. Specifically, our method employs custom hardware to generate a fingerprint for each basic block of the executed routine and to compare it against a set of known acceptable fingerprints for this routine. The choice of a basic block, as opposed to the entire system call service routine, as the minimum entity to be fingerprinted is driven by practicality. A basic block is a snippet of atomic code executed between two control flow transfers. Therefore, the actual instructions executed are fixed, hence the golden fingerprint of a basic block can be statically computed. In contrast, the instructions executed by the entire system call service routine depend on the arguments with which it is invoked at run-time; hence there is a multitude of golden fingerprints which are not only harder to exhaustively identify but may also leave more room for malicious modifications to go undetected.

V. Hardware Implementation

A. Data Logging Component

To extract system call related information directly from the hardware without assistance from upper-level resources, such as `system.map`, we exploit several x86 hardware conventions, as described below.

1) System call table address: In a 32-bit x86 architecture, two methods can be used to invoke a system call. The first one uses the conventional software interrupt, through `INT 0x80`. In particular, this instruction consults the Interrupt Descriptor Table (IDT) to identify the entry point to the system call table, which is then indexed with a system call code to execute corresponding service routine. As shown in the first basic block of the `INT 0x80` handler in Figure 3a, the exact base address of the system call table, in this case `0xc16380a0`, can be obtained through decoding the last instruction. The second method for invoking a system call uses `SYSENTER` instructions. These instructions were introduced by Intel in order to enable fast entry to the kernel and avoid the overhead incurred by software interrupts. Similarly, the same base address is referred to in the first basic block of the `SYSENTER` handler, as shown in Figure 3b.

If an attacker launches a Type-1 attack to redirect access to the system call table, the base address of the attacker-defined table must appear as a target address in the first basic block of the system call handling routine. Therefore, comparing the

```
int 80h
lea esi,0[esi]
...
jae 0xc162d80a
call dword ptr 0xc16380a0[eax*4] /*system call table is referred to here*/
```

(a)

```
sysenter
mov esp,dword ptr 0xffffde84[esp]
...
jae 0xc162d80a
call dword ptr 0xc16380a0[eax*4] /*system call table is referred to here*/
```

(b)

Fig. 3: INT 80H handler vs. SYSENTER handler

actual address to the legitimate one at run-time directly in hardware helps in detecting such attacks.

2) System call table content: In addition to the base address of the system call table, its content is also known. In this work we store a golden fingerprint, which is generated using the mechanism introduced in the following section, for the entire system call table. When a system call is invoked, the fingerprint of the employed table is computed and compared to the golden one in hardware, in order to detect Type-2 attacks.

3) System call service routine: In order to detect Type-3 attacks, we need the ability to analyze instruction-level behavior of system calls. However, logging the entire instruction flow would introduce unacceptable hardware overhead and is, generally, unnecessary. Instead, we employ a MISR to compress the instruction flow and generate simple fingerprints.

A MISR is a variant of a Linear Feedback Shift Register (LFSR). A standard LFSR is a shift register whose output is a linear function of its previous state, where the feedback input bit is generated by the XOR/XNOR function of a subset of the register bits. A MISR has the same structure, but additional input bits are fed through an XOR/XNOR gate to every flip-flop of the shift register in each cycle, as shown in Figure 4. As a result, the next state of the MISR depends on both the current state and the input bits.

Using a MISR for our purpose has three advantages: (1) the hardware structure of a MISR is relatively simple, involving only D-Flip-Flops (DFF) and XOR gates; thus, it incurs low design overhead. The number of DFFs and XORs is decided by the characteristic polynomial of the underlying LFSR. In our case, since we seek to compact instructions and the maximum length of a single instruction in x86 is 15 bytes, the degree of the characteristic polynomial has to be at least 120[1]. Specifically, we chose to implement a MISR using $x^{120} + x^{119} + 1$ as the characteristic polynomial. (2) A MISR is scalable and can process multiple inputs simultaneously, independent of the input length; this allows us to efficiently process entire instructions in order to generate fingerprints for basic blocks. (3) The MISR has a relatively low aliasing probability. Aliasing occurs when identical signatures are generated for different input sequences and can undermine our ability to detect invalid instruction sequences. An approximation of the aliasing probability of a MISR is 2^{-n}, where n is the degree of its characteristic polynomial. In our case, since n is 120, the aliasing probability is as low as 2^{-120}, which is negligible.

[1]Intel 64 and IA-32 Architectures Software Developer's Manual

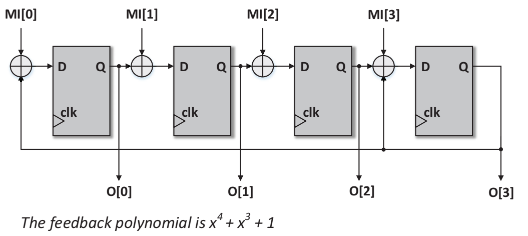

Fig. 4: An example of a 4-bit MISR

In our work, execution flow of a system call service routine is described by multiple fingerprints, one for each of its basic blocks. The generated fingerprints are then fed into the on-line validation component, which we will describe next, to detect intrusion. To associate the generated fingerprints with the corresponding system call, we use the system call code as an identifier. In x86, the system call code is stored in the `EAX` register when an `INT 0x80` or `SYSENTER` instruction is executed, hence our hardware obtains it directly from there.

B. Validation Component

The role of the validation component is to examine the validity of the fingerprints generated by the logging component for the basic blocks of a system call service routine. In particular, the golden fingerprints for each system call are identified through static code analysis and programmed in the validation component, which then checks membership of an on-line generated fingerprint in the appropriate golden set. A failed membership test implies that the system call service routine is invalid. Considering the potential design overhead of the validation component, explicitly storing in hardware all golden fingerprints for each system calls and comparing them in parallel against an on-line generated fingerprint would be prohibitively expensive. Therefore, instead of using a lookup table or a hash table for this purpose, we employ a Bloom filter for compactly storing the golden fingerprints and rapidly performing membership tests.

A Bloom filter is a space-efficient probabilistic data structure, used to test whether an element is a member of a set [12]. As shown in Figure 5a, a typical Bloom filter consists of an m-bit array and implements k different hash functions h_i, $i \in [1, k]$, each of which maps an input element E to one of the positions in the array through a uniform random distribution. An empty Bloom filter is an array with all 0s. When adding an element, all k position bits mapped by the hash functions for this element are set to 1. As a result, an element is a member of the set if and only if $\forall i \in [1, k], h_i(E) = 1$.

A Bloom filter never misidentifies a valid member of the set as a non-member. However, it is possible that due to collision in the outputs of the hash functions, a non-member may be accepted as a member of the set, which in our case would imply that an invalid fingerprint may evade detection. Fortunately, appropriate choice of m and k can bound the probability of this evasion detection occurrence [13]. Furthermore, our design leverages a partitioned and fully-pipelined Bloom filter architecture. Partitioning, which splits the m-bit array into k sections that can be separately queried by the k hash functions, may further reduce detection evasion, since

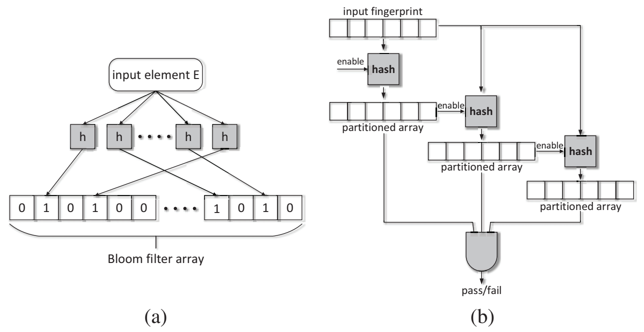

Fig. 5: (a) Typical Bloom filter, (b) Validation component

the outputs of the k hash functions themselves cannot suffer from collision [13]. Pipelining, on the other hand, allows us to disable the computation of the next hash function when the output of the previous hash function is 0, which indicates that the input element is certainly not a member of the set, thereby reducing power consumption.

In a Bloom filter, for a given array size of m bits and set cardinality n, the optimal number of hash functions k_{opt}, which minimizes detection evasion probability p, is $k_{opt} = \frac{m}{n} ln2$. For the optimal value k_{opt}, m becomes a linear function of n, which can be calculated as $m = \left\lceil -\frac{nlnp}{(ln2)^2} \right\rceil$. In this work, the maximum n (i.e., basic blocks in a system call routine) does not exceed 3500, while p is set to 3%. As a result, the chosen values for m and k are 24576 bits (3 KB) and 6, respectively.

Our Bloom filter implementation uses the hardware-friendly hash function design suggested in [14]. Specifically, given a b-bit input element E, its hash is calculated as

$$h(E) = (S_1 \cdot E_1) \oplus (S_2 \cdot E_2) \oplus \cdots \oplus (S_b \cdot E_b)$$

where S_i is a random seed coefficient $\in [1, log_2 m)$ and the operation $\cdot$ is,

$$S_i \cdot E_i = \begin{cases} S_i, & if\ E_i = 1 \\ 0, & \text{otherwise} \end{cases}$$

Further optimization is applied to the construction of hash functions. A classic Bloom filter requires k independent instances of hash functions, which introduces significant overhead when k grows. However, as described in [15], two independent hash functions h_1 and h_2, referred to as primary hash functions, are sufficient, while the remaining $k-2$ hash functions can be obtained through $h'(x) = h_1(x) + j \cdot h_2(x)$, where j is an arbitrary integer. Thus, the design overhead can be approximately bound by the cost of two hash functions, regardless of the value of k. Detailed implementation of the validation component is shown in Figure 5b. Finally, the overal architecture of the proposed method, which interfaces the logging and validation components with the microprocessor pipeline, is illustrated in Figure 6.

VI. Experimental Results

We now proceed to assess the effectiveness of our method in detecting intrusion through system call hijacking attacks and to evaluate the hardware overhead of our logging and validation components. Our experiments were performed in Simics, wherein we simulated a single-core 32-bit x86 machine running at 2GHz with 4GB of RAM, on which an Ubuntu server that embeds a Linux 2.6 kernel is loaded.

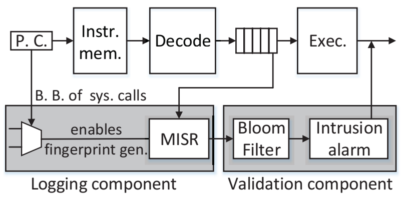

Fig. 6: Architecture of proposed method

A. *Effectiveness*

To evaluate effectiveness of our method in intrusion detection, we first perform static analysis to collect the golden fingerprints of the basic blocks for each system call service routine, and to program the corresponding Bloom filters. Next, we use MiBench [16], a free commercially-representative benchmark suite, to collect fingerprints for legitimate workload. To collect fingerprints for contaminated workload, we use five rootkits which exploit LKM to launch system call hijacking attacks of the three types introduced in Section III and which are implemented based on the Linux rootkit template maKit[2]. In order to hook our logging component onto the program counter and monitor the instruction flow, we exploit the *haps* feature of Simics. We note that, for reasons explained in Section VII, in our experiments we only fingerprint basic blocks entered through direct jump instructions.

Table I summarizes the number of times each system call was invoked by the legitimate workload, as well as the corresponding number of distinct fingerprints (only the most frequently invoked system calls are shown). All of these fingerprints were processed by the corresponding Bloom filter and passed the validation process, i.e. no false alarms occurred.

Table II summarizes the five kernel rootkits which we used to launch system call hijacking attacks. Following the definitions in Section III, the first one is Type 1, the next three are Type 2 and the last one is a Type 3 threat. All five were successfully detected by our method, as they invoked system calls whose service routine execution generated fingerprints that were rejected by the corresponding Bloom filter.

B. *Design overhead*

Since the logging and validation components are implemented directly in hardware and do not interfere with the original microprocessor execution flow, the performance overhead of our system is expected to be zero. As a point of reference, similar work developed at the hypervisor level, introduced additional performance overhead of 262.3 ms [6].

To evaluate the hardware design overhead of our method, we use a predictive 45nm Process Design Kit (PDK) [17]. The logging component and the validation component are separately synthesized in Synopsis Design Vision. Since the library does not contain a memory cell, we report the memory cost of the bit array and the seed coefficients of the Bloom filter separately, in terms of bytes required.

Table III summarizes the incurred overhead for each of the two components. Compared with a 45nm Intel Processor[3], whose area is 107 mm^2 and average power consumption is

[2]https://github.com/maK-/Syscall-table-hijack-LKM
[3]http://ark.intel.com/products/35605

TABLE I: Statistics for legitimate workload

system call	invocations	fingerprints
sys_open	1103	2708
sys_read	4480	3325
sys_write	5504	1883
sys_close	181	1470
sys_rt_sigprocmask	5640	146
sys_rt_sigaction	1029	392
sys_mmap2	218	1170
sys_ioctl	318	231
sys_brk	125	179

65 W, the total overhead of this hardware implementation is negligible. Furthermore, while other hardware-based methods performing similar analysis introduce additional 2% area overhead in general [18], [19], our implementation consumes hundreds of times less in area. Regarding the seed coefficient lookup table, two tables are required for the two independent primary hash functions, in which 120 different random 12-bit values are programmed. Therefore, the memory cost of the seed coefficient lookup table is 360 B in total. Considering that the total system call number in Linux 2.6 is 336, the memory cost of the bit array for the Bloom filters is 0.98 MB.

VII. Limitations of hardware-based approaches

A. *Indirect jump instructions*

In modern microprocessors, control flow transfers may involve either direct or indirect jump instructions. In the former, the starting point of the next basic block is explicitly known, but in the latter it is computed at run-time and may redirect execution to an instruction that is not at the beginning of a statically identified basic block. Hence, indirect jumps limit our ability to retrieve all possible basic blocks within an execution flow. Therefore, in order to avoid false alarms, basic blocks reached via an indirect jump are not fingerprinted. However, similar to other hardware-based methods which rely on behavior modeling [6], [8]–[10], if a non-zero false-positive rate is acceptable such fingerprinting can be enabled to opportunistically detect indirect jump-based attacks. Evidently, such attacks constitute a very challenging problem for hardware and software-based intrusion detection methods alike. Software solutions developed are vulnerable to tampering, incur relatively high implementation overhead which makes their deployment impractical [2], [3], and have limited effectiveness, as implied in [20]. In short, handling indirect jump-based attacks in intrusion detection remains an open question, worthy of future exploration, especially for hardware-based solutions [18], [19].

B. *OS updates*

The innate immunity of hardware-based intrusion detection methods against software-based tampering comes at the cost of sacrificing flexibility. Similar to other hardware-based methods [8]–[10], whenever the OS kernel is updated, our approach may also require an updating of its underlying configuration (i.e. the golden reference set of system call routine fingerprints). In order to maintain immunity, this information should not be modifiable through software or the OS, thus, such updating is not at all straightforward. Solutions to this limitation would likely require the use of non-volatile memory

TABLE II: Kernel rootkit detection summary

rootkit	description	detected?
Type-1 attack	system call table redirected	✓
Type-2 attack	table entry sys_open() redirected	✓
Type-2 attack	table entry sys_write() redirected	✓
Type-2 attack	table entry sys_mkdir() redirected	✓
Type-3 attack	sys_write() routine modified	✓

along with a hardware interface for reprogramming it with the updated golden reference set.

C. Multi-core environment

While our proposed method is evaluated in a single-core environment, it can be easily extended to be compatible with a multi-core environment. Specifically, a global validation component containing the golden reference set will be implemented across the processor cores, while multiple local logging components will be implemented for each core. The validation process can still work in parallel with the microprocessor execution flow, without incurring additional performance overhead. Furthermore, unlike hardware-based CFI methods whose area overhead is, generally, multiplied by the number of processor cores [18], [19], our method requires only duplicated logging components, resulting in much less additional total area overhead.

D. Hardware Trojans

Lastly, effectiveness of hardware-based on-line intrusion detection methods [8]–[10], including the one proposed herein, relies on the assumption that the underlying hardware is trusted and does not contain any malicious functionality, such as hardware Trojans [21]. Indeed, a hardware Trojan can interfere with the intrusion detection mechanism and undermine or even disable it. Of course, software-based intrusion detection methods may also be susceptible to hardware Trojan interference, since it is theoretically possible to contaminate the data that such methods are based on. In general, development of malware detection solutions which are impervious to hardware Trojan attacks is an open research problem.

VIII. Conclusion

We introduced a low-cost hardware-based approach for performing on-line intrusion detection through system call routine fingerprinting. Unlike software-based methods, which rely on information obtained through the OS and are, therefore, vulnerable to software attacks, such as system call hijacking, this hardware-based method extracts the required information directly in hardware, making it impervious to such attacks. Herein, we demonstrated an incarnation of this general idea, which logs and generates fingerprints using a MISR and examines their validity by checking their membership in a pre-specified set of golden fingerprints, implemented as a Bloom filter. The proposed method was evaluated on a 32-bit x86 architecture running Linux OS, implemented in Simics. Experimental results, using the Mibench suite as legitimate software and kernel rootkits launching system call hijacking attacks as malware, reveal that the fingerprints of the contaminated routines can be fully differentiated from those of the legitimate code. The hardware cost of our method in a 45nm process is 9229.72 μm^2 in area and 21.45 mW in power, as well as approximately 1 MB of memory, which is negligible compared to the size of a modern microprocessor.

TABLE III: Design overhead summary

	area (μm^2)	**power** (mW)
logging	1810.56	5.55
validation	7419.16	15.9
total	**9229.72**	**21.45**
microprocessor	107×10^6	65×10^3
overhead	**0.008625%**	**0.033%**

References

[1] C. Kolbitsch, P. Milani, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *18th USENIX Security Symp.*, pages 351–366, 2009.

[2] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *Proc. of the 2014 IEEE Symp. on S & P*, pages 292–307, 2014.

[3] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proc. of the 27th Annual Computer Security Applications Conf.*, pages 353–362, 2011.

[4] Y. Fu and Z. Lin. Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *IEEE Symp. on S & P*, pages 586–600, 2012.

[5] J. Pfoh, C. Schneider, and C. Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *6th Intl. Conf. on Advances in Information and Computer Security*, pages 96–112, 2011.

[6] X. Wang and R. Karri. Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits. In *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, volume 35, pages 485–498, 2016.

[7] D. Perez-Botero, J. Szefer, and R. Lee. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Intl. Workshop on Security in Cloud Computing*, pages 3–10, 2013.

[8] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. On the feasibility of online malware detection with performance counters. In *40th Annual Intl. Symp. on Computer Architecture*, pages 559–570, 2013.

[9] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev. Malware-aware processors: A framework for efficient online malware detection. In *IEEE 21st Intl. Symp. on High Performance Computer Architecture*, pages 651–661, 2015.

[10] L. Zhou and Y. Makris. Hardware-based workload forensics: Process reconstruction via TLB monitoring. In *IEEE Intl. Symp. on Hardware Oriented Security and Trust*, pages 167–172, 2016.

[11] S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Annual Conf. on USENIX*, pages 1–14, 2006.

[12] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[13] P. Almeida, C. Baquero, N. Preguica, and D. Hutchison. Scalable Bloom filters. *Information Processing Letters*, 101(6):255 – 261, 2007.

[14] R. Chhabra and V. Nath. Comparative study of Bloom filter architectures. *Global Journal of Researches in Engineering Electrical and Electronics Engineering*, 12(4), 2012.

[15] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. In *Algorithms - ESA 2006*, volume 4168, pages 456–467. 2006.

[16] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE Intl. Workshop on Workload Characterization*, pages 3–14, 2001.

[17] J. Stine, I. Castellanos, M. Wood, J. Henson, and F. Love. FreePDK: An open-source variation-aware design kit. In *Proc. of the IEEE Intl. Conf. on Microelectronic Systems Education*, pages 173–174, 2007.

[18] L. Davi, M. Hanreich, D. Paul, A. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin. HAFIX: Hardware-assisted flow integrity extension. In *Proc. of the 52nd Annual Design Automation Conf.*, pages 1–6, 2015.

[19] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis. Hcfi: Hardware-enforced control-flow integrity. In *Proc. of the Sixth ACM Conf. on Data and Application Security and Privacy*, pages 38–49, 2016.

[20] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proc. of the 24th USENIX Conf. on Security Symp.*, pages 161–176, 2015.

[21] Y. Jin, M. Maniatakos, and Y. Makris. Exposing vulnerabilities of untrusted computing platforms. In *Proc. of the IEEE Intl. Conf. on Computer Design*, pages 131–134, 2012.