A Hardware Monitor to Protect Linux System Calls

George Provelengios, Arman Pouraghily, Russell Tessier, and Tilman Wolf
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA, 01003 USA

***Abstract*—Internet-connected embedded systems have limited capabilities to defend themselves against remote hacking attacks. The potential effects of such attacks, however, can have a significant impact in the context of the Internet of Things, industrial control systems, smart health systems, etc. Embedded systems cannot effectively utilize existing software-based protection mechanisms due to limited processing capabilities and energy resources. We propose a novel hardware-based monitoring technique that can detect if the system calls of sophisticated embedded operating systems (e.g. Linux) deviate from the originally programmed behavior due to an attack. We present an FPGA-based prototype implementation that shows the effectiveness of such a security approach using a known Linux exploit. Our approach detects the attack with minimal overhead and without slowing processor operation.**

***Index Terms*—hardware security, hardware monitor, system call, FPGA**

I. Introduction

The Internet of Things (IoT) represents the convergence of cyber-physical systems (CPS), which control physical processes, and the Internet, which provides global interconnectivity for access to data systems. Embedded systems are at the core of any IoT solution as they provide the necessary computational power at the location where devices interact with the physical world. Due to their deployment in the environment, these embedded systems are typically constrained in their computational resources (performance and/or energy) but still connected to a network to interact with the other components of the IoT solution. This type of networked embedded system experiences a particularly challenging problem when it comes to security, specifically, protection from attacks on the embedded operating system. The network connectivity provides an attack vector to the system and the system performance or energy resources are insufficient to run conventional defense mechanisms, such as virus scanners and malware detection software, that provide protection on conventional computers. An effective defense mechanism that has been developed in related work is "hardware monitors". These monitors are logic components that are co-located with the embedded system processor core and track the execution of software. Hardware monitors require no change or addition to the software that is run on the processing system. Instead, such monitors verify that a processor executes a piece of software faithfully by comparing two pieces of information: (1) processing steps reported by the processor at runtime and (2) a model of what is considered correct execution of the software that is to be executed. Attacks that hijack the processor inherently cause the processing to deviate from the model of the original software and thus can be detected.

When discussing related work in Section II-A, we show how our work distinguishes itself from other monitoring techniques. The main novelty of our work is that our hardware monitoring system *works with Linux, a common, widely-used operating system*, whereas previous work has either looked at specific applications running directly on the processor or highly constrained, simplistic embedded operating systems. In addition, we show that our system defends against *real, practical attacks* (in our case the CVE-2013-1828 vulnerability, which has a known exploit), whereas previous work has shown defenses against attacks exploiting synthetically crafted vulnerabilities.

The main idea of our work is to focus the monitoring system on the portions of the operating system that are particularly vulnerable. Since many vulnerabilities and associated exploits occur in the context of system calls, we have designed our hardware monitor to track their processor operations at very fine granularity. By verifying operation at the level of an individual processor instruction, we can detect any deviation (i.e., attack) almost instantaneously. By limiting the monitoring to a fraction of the operating system code (i.e., system calls) and not the entire code base, we can achieve low overhead compared to other hardware monitoring approaches. This combination of sensitivity to attacks on vulnerable code and low hardware overhead (and no modification to any software) provides a promising approach to protecting embedded systems in the IoT domain or anywhere else.

The remainder of the paper is organized as follows. Section II discusses how our work relates to other efforts to protect embedded systems. The principles of our monitoring system are described in Section III. The design and implementation of our prototype system are presented in Section IV. Experimental results are shown in Section V, and the paper is summarized and concluded in Section VI.

II. Background

A. Related Work

Monitoring of correct program execution has been proposed in various forms, such as verification of control-flow integrity (CFI) [1]. These software techniques may slow down program execution and do not validate individual processor instructions. Hardware monitoring reduces the performance impact of monitoring. The seminal work by Arora *et al.* described a fine-grained hardware monitoring system that verifies correct execution at the granularity level of a basic block [2]. This work was advanced by Mao *et al.* in verifying

TABLE I
RELATED WORK ON HARDWARE MONITORING.

	Abadi et al. [1]	Arora et al. [2]	Mao et al. [3]	Pouraghily et al. [4]	this paper
verification	control flow operations	all processor instructions			
granularity	basic block		single processor instruction		
target	application / OS	monolithic application		simplistic OS	Linux OS
coverage	application / OS	entire application		entire OS	system calls
overhead	software	high hardware cost			low hardware cost

individual processor instructions and the resulting ability to stop attacks within one processor clock cycle instead of having to wait until the basic block has ended [3]. Recent work by Pouraghily *et al.* further expanded the previous work to not only monitor monolithic applications, but the underlying operating system [4].

Our work also focuses on monitoring the operating system. However, unlike related work, we aim to work with a real Linux operating system, not a light, embedded variant of a simplistic operating system. The large code size of the Linux kernel makes previous approaches to monitoring impractical due to their large overhead. In our work, we focus the monitoring effort on the portions of the code that are particularly vulnerable to attacks: system calls. Thus, we can effectively detect a good number of attacks while keeping the monitoring overhead low enough to make such a system practically useful. The progression of work on hardware monitoring and the context of our contribution is summarized in Table I.

System-call monitoring is another technique that attempts to detect intrusion. The approach tracks the system calls that are executed by an application, which is much coarser than tracking individual processor instructions. A survey on system-call monitoring [5] describes how the work has evolved over time. The main difference between this work and our approach is that we do not track patterns of system calls. Instead, we focus on ensuring that the processor instructions associated with a system call are executed faithfully. This approach ensures that attacks via system calls do not succeed. The existing approaches to system call monitoring can be used orthogonally to our work.

B. Focus on System Calls

As mentioned above, our hardware monitoring system focuses on validating the correct execution of system calls in the operating system. The current Linux kernel (version 4.13.15) contains code for 337 different system calls. Between 1999 and 2017, 1,931 vulnerabilities in the Linux kernel were reported to the Common Vulnerabilities and Exposures (CVE) database that is maintained by MITRE. Of those, 45 vulnerabilities (2.3%) directly relate to system calls. This may seem like a small percentage. However, the existence of a vulnerability is particularly problematic if an exploit exists that can let an attacker use the vulnerability in a practical manner. Of 148 publicly available exploits (listed in the Exploit Database maintained by Offensive Security) that lead to privilege escalation attacks (which gives the attacker full control over the system), 25 exploits (16.9%) are based on vulnerabilities in system calls.

A typical attack, as we describe in more detail in Section IV, uses a buffer overflow to redirect program execution to shell code or other attack code. Since the kernel operates at the highest level of privilege in the system, achieving the execution of malicious code through redirection of a system call can give an attacker the highest level of access. By protecting system calls from such attacks through verification of correct execution, which can detect buffer overflow attacks that change code execution, we can protect the system from exploits that use known and unknown vulnerabilities. This protection works for attacks that are launched through software that is executed on the system directly, as well as attacks that are launched remotely through the network.

III. MONITORING ARCHITECTURE

The main goal of our monitoring system is to prevent execution deviations from system calls to malicious code. If such a deviation is detected, execution is stopped and the processor is reset. Our security model assumes that an attacker may access the target system and tamper with processor instructions and data remotely through an I/O interface, although it is not possible to tamper with the monitoring system.

A. Basic Monitor Operation

As mentioned in Section I, hardware monitors are components that are co-located with processor cores to track the processing of software on that core. The objective is to assess the operation of the processor and determine when incorrect behavior is detected (which can be due to benign faults or malicious attacks). In our work, we use a hardware monitor that receives information about every instruction executed on the processor core and compares it to a "monitoring graph" that is generated from the processing binary. Each instruction is represented by a hash value (to reduce the size of the monitoring graph compared to the size of the binary) and state transitions correspond to possible control flow paths between instructions. We use a deterministic finite automaton (DFA) representation of the monitoring graph (as detailed in [6]).

For this work, a monitoring graph is generated during design compilation [6] for selected system calls. Each instruction in the system call is encoded as an entry in the graph that includes the valid hash value(s) of the next instruction (or instructions in the case of a branch) and the next graph state(s). A detailed view of our monitoring subsystem is shown in Figure 1. The portions of the monitoring system can be split into *monitoring*

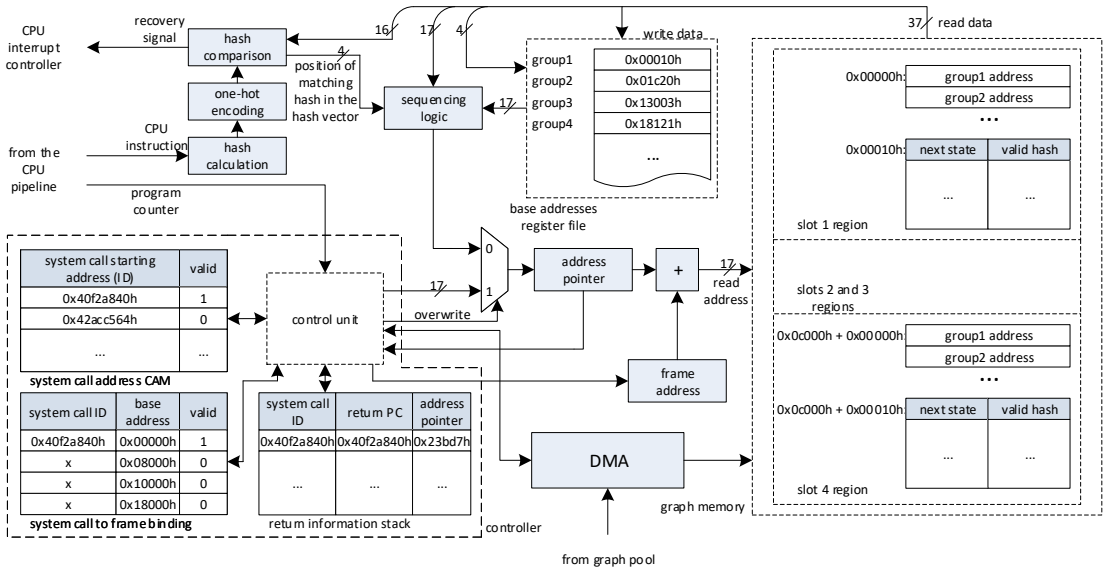

Fig. 1. System architecture for a hardware monitor that supports selective system call monitoring

hardware (three boxes in upper left corner of the figure), which checks the per-instruction operation of the companion processor, *graph memory*, which stores monitoring graphs, and *controller*. The monitoring hardware checks each processor instruction using an entry from the monitoring graphs stored in graph memory. In the figure, graphs for four separate system calls are stored in slots in the graph memory. Each graph includes one row per instruction, effectively representing expected program control flow as a state machine. A *read address* pointer indicates the entry in the graph that corresponds to the instruction that has just completed execution. During the execution of an instruction, a multi-bit (in our case 4-bit) hash value of the instruction is generated and converted to a one-hot representation. Previous work has shown a 4-bit hash value to be sufficient to limit collisions [7]. The one-hot encoding is compared against the expected next instruction hash values (*valid hash*) that are stored in the graph entry for the previously executed instruction. The use of a one-hot representation simplifies these comparison operations.

A match of an instruction hash against a stored valid hash indicates a valid instruction. If no match occurs, an illegal instruction has been executed, leading to the generation of a recovery signal which is used by the processor for process termination. Since control flow instructions (e.g. branch) may have several possible next instructions, and, consequently, several possible valid hashes, multiple one-hot valid hash bits may be set per entry. A match of any of these hashes indicates a valid instruction. Our approach can handle dynamic branch targets by profiling the code to determine all branch targets for a system call prior to graph generation. Entries for these targets are then added to the graph.

The next *read address* (memory row) in the monitoring graph is determined using next state information stored in the current entry, the matched hash value, and information stored in *base address registers* which group states based on fan-in count [6]. These values are combined via addition in the *sequencing logic* box in the figure. The resulting address is stored in the *address pointer* and subsequently added to the start address for the appropriate graph slot for the system call. The implemented monitor requires only one memory lookup per instruction. Effectively, the monitoring information for each system call at any given point in execution is defined by the contents of the address pointer, the monitoring graph for the process and the contents of the base address registers. The location of each system call monitoring graph in the graph memory is stored in the *system call to frame binding* memory. The procedure required for activating monitoring for system calls is described next.

B. Enabling and Disabling Monitoring

Since monitored system calls can be invoked from within user applications or unmonitored system calls, a mechanism to seamlessly enable and disable the hardware monitor once a system call is invoked or retired is included in our monitoring system. Monitoring is stopped after the monitored system call is finished and the user application or unmonitored system call execution is restarted. We consider four specific scenarios: (1) a monitored system call is called from an application (monitor activated), (2) a return from a monitored system call to an application or unmonitored system call (monitor deactivated). (3) an unmonitored system call is called from a monitored system call (monitor deactivated), and (4) a return from an unmonitored system call to a monitored system call (monitor activated).

1) Call to monitored system call from unmonitored code: After Linux is compiled into a loadable image, the addresses of the kernel functions and system calls are fixed. The starting address of each system call is used as a unique identifier. For each system call, there is only one entry point, which is

used to trigger the monitor. A hardware-based solution triggers monitoring upon entry into a system call by matching the system call program counter to one of a series of valid stored values (valid bit = 1) in a content addressable memory. It is shown in Figure 1 as the *system call address CAM*. As a transition to the monitored system call is made, the monitor is enabled.

For example, when the microprocessor executes an instruction, the program counter which has been extracted from the exception stage of the processor pipeline is compared against all of the valid system call starting addresses in the CAM. If it matches a stored address, the monitor is activated to start tracking microprocessor code execution using the monitoring graph generated during the compilation process for the system call. Prior to Linux execution, the CAM is loaded with the start addresses of the monitored system calls. Information in the monitor, including monitoring graphs and the system call address CAM, are loaded through a secure channel that is not accessible to application users. Any modifications to the CAM table is performed using secure techniques [6].

2) Return from monitored system call: A scalable approach is used to disable the monitor upon leaving a monitored system call since multiple exit points in the call may exist. To avoid using a large CAM to match the PC against all exit points, monitor disable information is embedded within the monitoring graph of the system call. As discussed in Section III-A, each entry in the monitoring graph contains a one-hot encoding of the valid hashes for the next instructions which succeed the current one. Normally, one or more of those bits are set to one according to the number of legitimate next instructions. However, if the instruction is the last instruction of the system call, all hash bits are set to zero indicating a system call return. This value disables monitoring.

C. Handling Nested System Calls

The mechanism described above is most effective if the call to a monitored system call is made from application code and a return to this code is made when the system call terminates. However, in many cases a monitored system call may invoke another system call that may be monitored or unmonitored. Thus, monitoring may need to be suspended for a time and then restarted upon return to the monitored system call.

1) Call to unmonitored system call from monitored system call: If unmonitored code is called from the monitored system call, the return address of the monitored code is stored on the return information stack and the monitor is deactivated. When a current, monitored system call switches to a new system call, its return address is stored on the stack. The stack consists of three different fields: system call ID which is the starting address of the monitored system call, return PC which is the next PC of the current system call which will be executed on the microprocessor after returning from the callee, and finally the current pointer to the monitoring graph of the current system call.

2) Return from unmonitored system call to monitored system call: When a return is made from the unmonitored code to monitored code, the return PC is checked against the top of the return information stack to determine if monitoring should be re-enabled. If a return is made to the monitored code, the monitor is reactivated and the return PC is popped from the stack.

TABLE II
MONITORING GRAPH SIZES FOR FOUR LINUX SYSTEM CALLS

system call	num. instr,	num. entries	graph size (bits)
getsockopt	49,252	68,422	2,531,614
execve	49,816	70,318	2,601,766
open	37,953	54,520	2,017,240
mmap	171	254	9,398

IV. PROTOTYPE IMPLEMENTATION

Our experimental system uses a 7-stage LEON3 processor, release 2017.2-b4193 [8] and an attached hardware monitor. The floating point unit was not included in the design. The hardware was synthesized and mapped to a Stratix IV FPGA on a Terasic DE4 board with 1GB of DDR2 memory. To perform monitoring, the instruction under execution and the program counter (PC) from the processor are tapped for use by the monitor. For monitoring to work effectively, it is necessary to ensure that only committed instructions are monitored, since a number of fetched instructions may be flushed or annulled from the processor pipeline. For this reason, the PC and associated instruction are tapped from the exception stage of the processor after the annul signal can be examined. As discussed in Section III-A, the instruction is subsequently converted to a hash value and compared to a stored entry in the monitoring graph. The PC is used to determine if monitoring should be enabled or disabled. If the monitor detects a deviation from expected computation, the processor is reset using a recovery signal. Detection and reset takes place as the inappropriate instruction is executed. The processor additions needed to tap the PC and instruction are negligible and our results show no loss of processor clock speed performance as a result of this action.

A. System Calls

In a secure system, all system calls should be monitored to prevent any system-call-based attack. Our monitor micro-architecture shown in Figure 1 is designed to monitor a subset of all calls as needed. For this work, we focus on the four system calls shown in Table II (more calls can be easily added). We chose these four system calls since the first contains the known vulnerability CVE-2013-1828 and others have been characterized as particularly vulnerable calls and used for kernel exploitation [9].

B. Attack Scenario

To evaluate the ability of our monitor to detect and prevent an attack, we tested our processor/monitor system with a known and published Linux attack from the Exploit Database, ID 24747 [10] and an additional attack that is derived from

```
➜  ~ ssh test@192.168.2.40
test@192.168.2.40's password:
[test@buildroot ~]$ cat /etc/passwd
root:x:0:0:root:/root:/bin/sh
test:x:1001:1001:Linux User,,,:/home/test:/bin/sh
[test@buildroot ~]$ ./priv_escalation
[test@buildroot ~]$ cat /etc/passwd
root:x:0:0:root:/root:/bin/sh
test:x:0:0:Linux root,,,:/home/test:/bin/sh
[test@buildroot ~]$ exit
logout
Connection to 192.168.2.40 closed.
➜  ~ ssh test@192.168.2.40
test@192.168.2.40's password:
[root@buildroot ~]#
```

Fig. 2. Console output showing that the attack script changes the *test* account privilege from a normal user to root

it. The latter attack exploits a vulnerability in the function *sctp_getsockopt_assoc_stats()* of the *getsockopt* system call and leads to a privilege escalation.

In the function, a call to *copy_from_user()* is used to copy the contents of a user-provided buffer into a data structure defined inside the function's local scope. Since there is no size check before calling the function, the user can provide a buffer to the system call which is bigger than the size of the local buffer. Therefore copying the buffer contents to the *sctp_getsockopt_assoc_stats()* function's local stack frame can overwrite substantial portions of the stack.

In Linux, the */etc/passwd* plain text file holds information about user accounts and their access levels. By modifying this file, one can grant any account root access. However, all users except root can only read this file and write access to this file is only granted to the root account. In our attack, instead of rewriting the stack with some random data and therefore destroying the return address, the system call is fed a buffer with meaningful data so that a user can gain root access. Specifically, the return address of the *sctp_getsockopt()* function is changed to the starting address of the *call_usermodehelper()* function which is a part of kernel and is used to prepare and run a user mode application from within the kernel. Using this function, */bin/sed*, a stream editor in Unix based operating systems, is executed to rewrite */etc/passwd* and grant root access to the user. Figure 2 shows the attack in action.

A key aspect of this attack is the writing of the attack arguments to call *usermodehelper()* that are passed to */bin/sed* and the branch to the function from the system call *sctp_getsockopt()*. When *call_usermodehelper()* is called, it receives its four operands on the stack, (two char*, one char**, and an int). Using monitoring, it is possible to detect the unexpected branch to *call_usermodehelper()* since the instructions of this function will not have entries in the monitoring graph.

V. Experimental Results

To evaluate performance, our processor and monitor architecture was mapped to the DE4's Stratix IV EP4SGX230 FPGA. A maximum system clock frequency of 110 MHz was achieved both with and without the monitor. Signals internal to the FPGA were monitored using Intel SignalTap, leading to the waveforms shown in Figure 3. The observed waveforms come from an attempted return from the system call function *sctp_getsockopt()*. Figure 3 (top) shows processor behavior during a normal return from the function starting at cycle 130. At this point, the one-hot hash encoding (0000 0000 0000 0100) of the next instruction matches one of the acceptable encoded valid hashes in the stored monitoring graph (0100 0000 1110 0100) in bit 2. The same observation can be made for the hashes of the next instruction. Thus, the instruction execution matches one of the expected execution paths determined during design compilation.

A. Attack Detection and Recovery

Figure 3 (bottom) shows the details of monitoring activities when the attack described in Section IV-B is performed. In this case, the return address of the *sctp_getsockopt()* function has been overwritten with the address of the *call_usermodehelper()* function. Since the first instruction in this function was not an acceptable return target for *sctp_getsockopt()*, the one-hot hash of this instruction will not match a valid hash value in the monitoring graph entry. Figure 3 (bottom) shows that this is the case. The one-hot hash of the instruction at cycle 130 is (0000 0001 0000 0000) while the stored valid hash value is (0100 0000 1110 0100). Since bit 8 is not set in the valid hash value, an unexpected instruction has been executed and the processor reset (recovery signal) can be asserted low. Note that the set bit in the one-hot hash of the next instruction also does not match the appropriate bit in the valid hash value. It should be noted that although the reset signal causes the processor to restart, possibly leading to a denial of service attack, this outcome is preferable to an unauthorized user gaining superuser access to the system.

Using the graph generation approach described in Section III-A, we examined the size of four representative Linux (version 3.8.0) system calls, including the *getsockopt* call described in Section IV-B. The number of instructions, the number of monitoring graph memory entries, and the total graph sizes in monitoring graph memory in bits for each system call are shown in Table II.

B. Monitoring System Overhead

For performance reasons, the monitoring graph is stored on-chip to allow for instruction-by-instruction hash value comparisons. Thus, we assess both the logic overhead and the overhead of on-chip memory. In addition, if a new system call is used, its monitoring graph may need to be securely loaded from off-chip memory using DMA to one of the graph memory slots shown in Figure 1 [4]. The resources needed to implement the microprocessor, the monitor and its associated graph transfer circuitry are shown in Table III.

The table shows that the monitor and associated control circuitry require dramatically less circuitry than the processor since it is a simple finite state machine. On-chip memory is needed so that each graph entry can be quickly obtained and compared to the currently-executing instruction. The table also includes the resources needed to securely load encrypted system call monitoring graphs from external memory. This

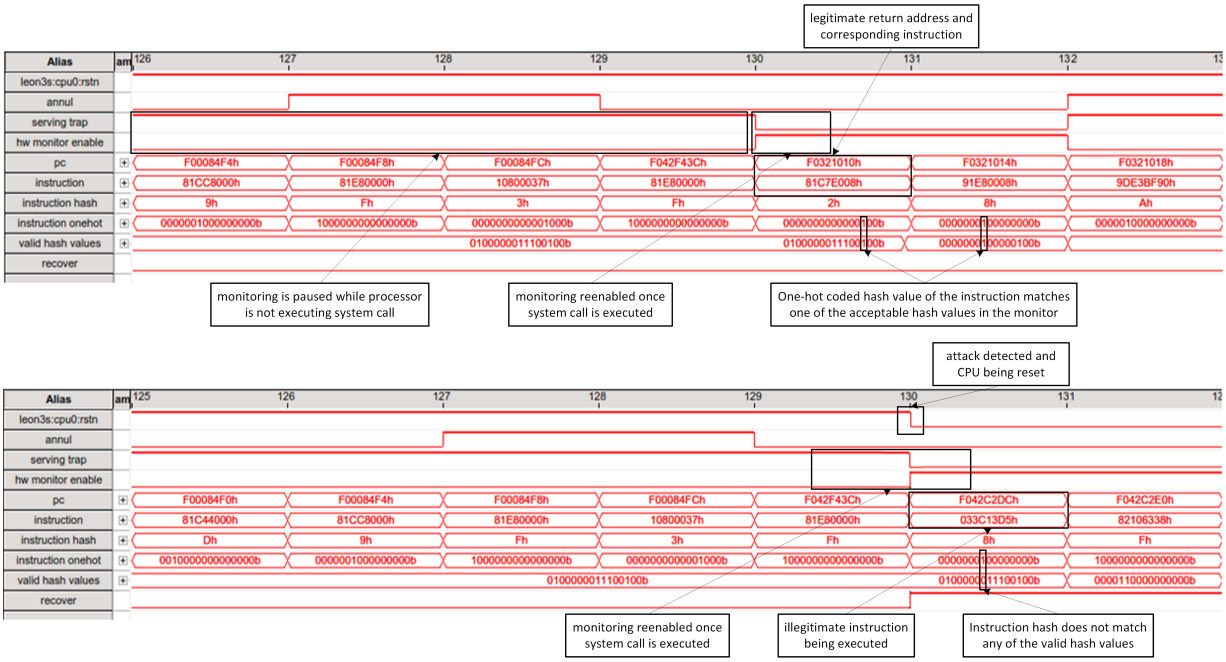

Fig. 3. Waveforms showing normal execution of the system call (top) and detection of the attack (bottom).

TABLE III
RESOURCE UTILIZATION OF THE HARDWARE MONITOR AND LEON3 PROCESSOR

resource		Available	LEON3 w/o hardware monitor	Hardware monitor and controller	CAM/stack	Secure HW mon. loader
Logic LUTs		182,400	20,070	380	6,555	2,603
Memory LUTs		91,200	170	0	0	0
Flip flops		182,400	15,053	324	11,457	2,936
Memory (bits)	off-chip	8,589,934,592	100,326,512	0	0	0
	on-chip	14,625,792	534,752	3,054,752	0	977,332

circuitry includes a decryption circuit which increases the overhead of the interface. Finally, the resources needed to implement the system call address CAM and return information stack used to identify monitoring start and stop points (described in Section III-B) for up to 337 different system calls are shown in the table.

By far, the most expensive part of monitoring is the on-chip memory needed to store the monitoring graphs. In this system, Linux instructions are stored off-chip so monitoring storage takes up the bulk of the on-chip storage. In our design, the monitoring graphs consume less than one-quarter of available on-chip memory so sufficient space is available for other circuitry. Overall, our results show that system call monitoring for advanced embedded operating systems, such as Linux, can be performed efficiently.

VI. SUMMARY AND CONCLUSION

System calls in sophisticated embedded operating systems are known to be vulnerable targets for attackers. We present a low-overhead monitoring approach that allows for selective instruction-by-instruction monitoring of system calls. Our approach has been demonstrated in hardware to successfully identify and prevent a known Linux system call attack. The overhead of the monitor is modest and does not impact the performance of the microprocessor.[1]

REFERENCES

[1] M. Abadi *et al.*, "Control-flow integrity principles, implementations, and applications," in *ACM CCS*, Nov. 2005, pp. 340–353.

[2] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Secure embedded processing through hardware-assisted run-time monitoring," in *IEEE/ACM DATE*, Mar. 2005, pp. 178–183.

[3] S. Mao and T. Wolf, "Hardware support for secure processing in embedded systems," in *IEEE/ACM DAC*, Jun. 2007, pp. 483–488.

[4] A. Pouraghily, T. Wolf, and R. Tessier, "Hardware support for embedded operating system security," in *IEEE ASAP*, Jul. 2017, pp. 61–66.

[5] S. Forrest, S. Hofmeyr, and A. Somayaji, "The evolution of system-call monitoring," in *Comp. Security Appl. Conf.*, Dec. 2008, pp. 418–430.

[6] K. Hu *et al.*, "System-level security for network processors with hardware monitors," in *IEEE/ACM DAC)*, Jun. 2014, pp. 211:1–211:6.

[7] T. Wolf *et al.*, "Securing network processors with high-performance hardware monitors," *IEEE TDSC*, vol. 12, no. 6, pp. 652–664, Nov. 2015.

[8] J. Gaisler and S. Habinc, "Grlib IP library user's manual," Cobham, Tech. Rep., Nov. 2017.

[9] C. Linn *et al.*, "Protecting against unexpected system calls," in *Usenix Security Symposium*, Aug. 2005.

[10] "Exploit database," https://www.exploit-db.com, accessed: 2017-11-18.

[1]This research was sponsored by NSF grant CNS-1617458.