Hardware Support for Embedded Operating System Security

Arman Pouraghily, Tilman Wolf and Russell Tessier
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA, USA
{apouraghily, wolf, tessier}@umass.edu

***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 embedded operating system or any running application deviates 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.**

***Index Terms*—embedded system, security, attack, defense, hardware monitor, operating system, monitoring graph, FPGA prototype**

I. Introduction

Embedded processing systems are core components of the emerging Internet of Things (IoT), as well as general control systems, smart health systems, and many other application domains. These systems are typically connected to each other and cloud computing infrastructure through the Internet. The value of data on these systems, their access to physical environments, and the potential destruction that can be caused by them make these devices attractive targets for attackers. There are two aspects that are of particular importance in this security context. First, the embedded systems are connected to the Internet and thus vulnerable to remote attacks. Second, the embedded systems typically do not have the processing capacity or power budget to run software-based defense mechanisms, such as virus scanners or intrusion detection systems, which are commonly used as security solutions in network-connected workstation and server computers.

Therefore, it is critical to develop defense mechanisms for these embedded systems that are effective to defend against attacks and that are practical to implement in a resource-constrained environment. In our work, we present a hardware-based monitoring system that can track each instruction that is executed by the embedded processor and check if it matches the expected behavior of the system. To determine what behavior is correct, we analyze the operating system (OS) and application binaries to create a *monitoring graph* for each. If the system is attacked, it necessarily will execute instructions that are not part of such a monitoring graph and thus the hardware monitor can detect this deviation. Our system is able to track the dynamics of the system (e.g., context switches, operating system interrupts, etc.) to ensure that the monitor can verify the faithful execution of every single instruction on the embedded processor.

The main contribution of our work is a lightweight security mechanism that can track operating system and application execution and detect attacks at the granularity of individual processor instructions without needing to know any characteristics of such an attack. Specifically, our paper presents the following contributions:

- Design of a hardware-based monitoring system that can detect any deviation in processing behavior in the operating system or application tasks, even when caused by previously unknown attacks.
- Prototype implementation of a hardware-based monitoring system on an Altera DE4 FPGA board using the μC/OS-II operating system to show the feasibility of this approach.
- Evaluation of the prototype system shows the ability to dynamically switch contexts, handle interrupts, and detect attacks while requiring only a few hundred logic gates and memory comparable to that of the instruction code and causing a minimal processing slowdown of 6 processor cycles per context switch.

The remainder of this paper describes the design, operation, and implementation of this hardware security mechanism for operating systems of embedded processing systems.

II. Related Work

The importance of security in embedded environments, such as in IoT, has long been acknowledged in academic research [1] and by government institutions [2]. Recent attacks on Internet infrastructure exploited vulnerabilities in IoT devices to launch distributed denial-of-service attacks [3], which highlights the continued need for novel security solutions in this space.

Although general purpose operating systems contain a variety of security mechanisms, embedded OS versions are limited and monitoring for embedded operating systems is constrained. Several techniques are used to provide operating system security at run-time. Typical mechanisms include a trusted computing base and a reference monitor [4]. The

978-1-5090-4825-0/17/$31.00

TABLE I
QUALITATIVE COMPARISON TO RELATED WORK.

	malware scanner	[10]	[11]	[12]	[13]	this paper
technique	software	hardware				
overhead	high	low				
granularity	I/O	basic bl.	processor instruction			
programs	multiple	single		multiple		
OS support	yes	no			yes	
OS monitor	yes	no				yes

software mechanisms enforce a security policy and access to compute objects, respectively. Dynamic information flow security [5] can be applied to operating systems to prevent data from input channels from being used as instructions or jump targets. A data-centric approach adds security information to storage locations and registers to track security levels [6]. A more recent approach uses a neural network to evaluate use patterns for the processor program counter and cycles per instruction [7]. Anomalous operating system behavior can be observed from these parameters. The Tamper Evident Processor [8] tags data values with hashes to identify unexpected changes. These values are used to identify OS data modifications.

The idea of using a hardware-based monitoring system to detect processing deviation is certainly not new: Monitors have been used to track function and system call sequences (e.g., [9]), to verify checksums over basic blocks (e.g., [10]), and to validate execution at a per-instruction level (e.g., [11]). What is new in our work is that we present an instruction-by-instruction level hardware monitoring approach for an environment of complex, interacting software components (i.e., multiple processing tasks running on an operating system). Such a fine-grained monitoring approach has not previously been demonstrated for a full-blown operating system with multiple tasks. The work in [12], [13] considers multiple processing tasks but does not monitor the operating system itself, which is often the target of attacks. Coarser-grained approaches have considered operating systems and processing tasks, but do not track processing behavior at the level of individual instructions, which opens them up to vulnerabilities, such as described in [14], where attacks have been executed on network processors using only a few instructions of malicious code. A qualitative comparison of related work and our contribution is shown in Table I.

III. SYSTEM AND SECURITY MODEL

To provide context for our design that we describe in Section IV and evaluate in Section V, we briefly discuss the system architecture, the construction of a monitoring graph, and the security model that is the basis for our work.

A. Monitoring Graph Construction

The basic idea of hardware monitoring is to compare system behavior against a golden indicator. Here we use a fine-grained indicator called a monitoring graph. This graph is a deterministic finite automaton in which the states are the assembly instructions and edges are the possible transitions between those instructions. It can be constructed by analyzing the binary code of an application. A big challenge in graph construction is resolving indirect control flow instructions where the target address of an instruction is determined by the content of a register. To handle these instructions, source code analysis, profiling, and binary code emulation [15] can be used. The graph extraction process is discussed in detail in [16].

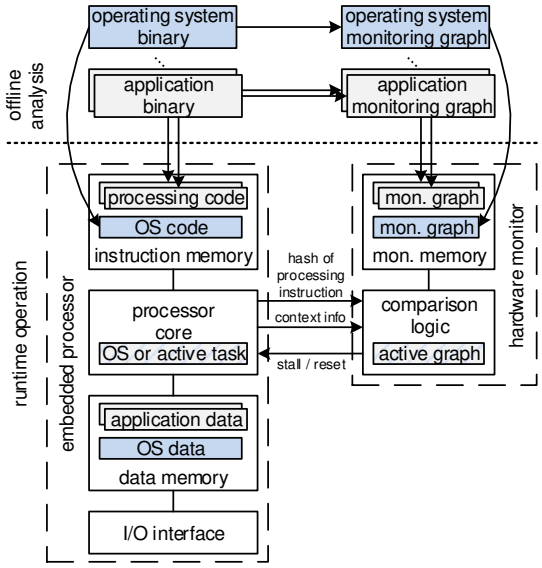

Fig. 1. System architecture of embedded hardware monitoring system that can validate correct execution of applications and operating system.

B. System Architecture

The system architecture of our hardware monitoring system is shown in Fig. 1. The processor core reports each executed instruction to the hardware monitor which compares the instruction against an entry in the monitoring graph. In the case of deviation for an expected graph traversal due to attack, a recovery procedure is initiated, as discussed in Section V.

Such a hardware monitoring approach has been described in related work [9]–[11]. We choose the per-instruction monitoring approach proposed in [11], rather than the per-basic-block monitoring presented in [10] or the system-call monitoring presented in [9], to ensure fast response to attacks. Also, we use the 4-bit hash of the processed instruction for reporting described in [17] (rather than the full instruction word) to reduce memory requirements. Finally, we use the security techniques described in [17] to prevent an attacker from tampering with the monitoring graph to avoid detection.

The challenge for this system, which is the main novelty over related work, is the need to associate the current processing context (operating system or one of multiple applications) with the correct monitoring context. As illustrated in Fig. 1, each application and the OS have their own monitoring graph (and associated monitoring state). When the processor switches between application and operating system processing

(e.g., due to context switch, interrupts, etc.), the hardware monitor needs to follow along with these dynamic changes.

C. Security Model

Having described the system architecture and the construction of a monitoring graph, we consider security properties that are tied to the security model. The security requirements of our system are:

SR1 The system should only execute code that belongs to the operating system or any of the validly installed applications.

SR2 Any attack that introduces malicious code should be detected and stopped.

The attacker capabilities that we assume are:

AC1 An attacker has access to the embedded system through any input/output channel.

AC2 An attacker can tamper with application and operating system binaries loaded into the main memory, the processing stack, and data memory.

In our work, we also assume the following limitation on attacker capabilities:

AC3 An attacker cannot tamper with the monitoring graph (e.g., by using the techniques from [17]).

In addition to these security requirements, there are performance requirements, such as low implementation cost and low performance overhead, to make the system practically useful. The hardware monitoring system described in the following section meets these requirements as we show in Section V.

IV. Monitor Design

This section describes the various aspects of our hardware monitor design in detail.

A. Operating System Task Management

The key aspect of our monitoring system is its ability to monitor both user and operating system tasks. Task switching can be initiated by the operating system (e.g., new user task is scheduled), by applications invoking system calls, or by external events (e.g., timer or external interrupt). In the first two cases, the context switch happens synchronously, meaning that the instruction after which the context switch occurs is known. Therefore, the appropriate information can be provided to the hardware monitor prior to the event by adding a small piece of code to the OS and the applications. However, in the latter case, the context switch can happen with no prior notice.

Context switch procedure on the hardware monitor includes saving the state of the monitoring graph for the code currently being executed and switching to the graph for the next processor task. Since interrupts can happen asynchronously, this whole procedure should be done seamlessly and without any coordination between the main processor and the hardware monitor. It also must be ensured that monitoring is synchronized with OS task execution following OS context switch operations such as register file save and restore.

Our OS monitoring approach has been developed for μC/OS-II[1], a widely used embedded operating system. All the OS internal functions (e.g. task scheduling), interrupt service routines, and system calls, handling software traps were continuously monitored.

B. Multi-Task Hardware Monitor System

A detailed view of our monitoring subsystem is shown in Fig. 2. The portions of the monitoring system can be split into *monitoring hardware*, which checks the per-instruction operation of the companion processor, *graph memory*, which stores state information about monitoring graphs, *sequencing logic*, which determines the next state in the graph, *processor interfaces*, which coordinates with the processor when context-related information is received, and *bookkeeping tables*, which associate monitoring graphs with specific user and OS tasks. The tables also keep track of the monitoring status for each graph. Monitoring graphs can be loaded into a secure memory from external memory via a cryptographic coprocessor. Activity in the monitoring hardware is controlled by a finite state machine. We have implemented this system and evaluated its performance on a DE4 FPGA board.

For each executed instruction, the monitoring hardware checks the instruction versus an entry in the associated monitoring graph. If an unexpected result is determined from the comparison, the instruction execution is flagged as a possible attack and the processor is either reset or interrupted. Graphs are loaded into *slots* in the graph memory. Once loaded, the starting address of the slot is associated with the graph ID (GID) of the appropriate graph in the bookkeeping tables.

Once the OS creates a new task, it sends a message to the hardware monitor with the process ID (PID) of the newly created task and its relevant graph ID. Then the hardware monitor loads the appropriate graph into its graph memory if it is not already resident and associates this PID with the received GID in the bookkeeping tables. These bookkeeping tables are consulted and updated during a context switch.

C. Context Switch Handling

In the processor, context switches can be triggered by three different events: *Interrupts, System Calls*, and the *Scheduler* resuming a user application. Next, we discuss how the hardware monitor follows the processor's context switch in each case and thus ensures the security of the system continuously.

1) Interrupts: The most frequent triggers of context switching are *interrupts*. Since interrupts happen asynchronously, the physical interrupt signal (IRQ) is presented to both the processor and the monitoring hardware. The ISR monitoring graph is always resident in monitoring graph memory. If the monitor detects an illegal instruction execution during the execution of the ISR, it resets the processor. The processor can elect to disable interrupts. In this case, the processor writes to a register in the monitor indicating that it should ignore future IRQ strobes. The monitor keeps tracking the processor until

[1] http://micrium.com/rtos/ucosii/overview/

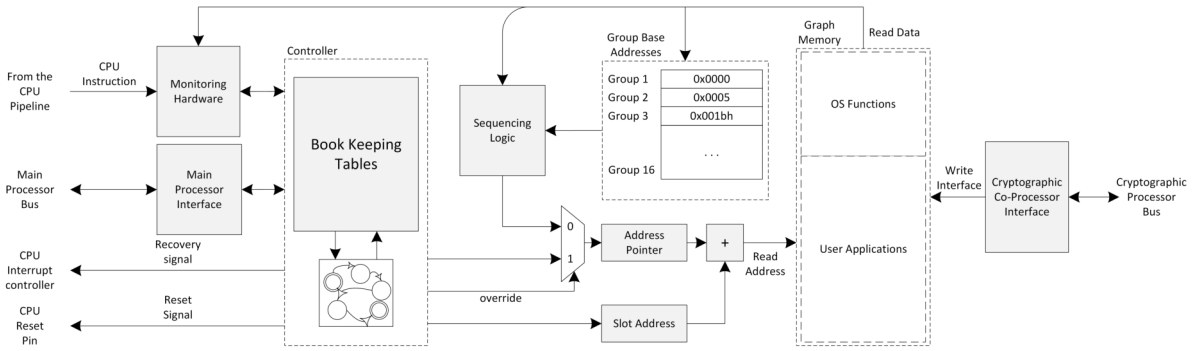

Fig. 2. Detailed view of multi-context monitoring system.

the processor writes the *disable interrupt* command into its status register.

2) System Calls: Another source of context switching is *system calls*. During these calls, the user function is suspended until the operating system returns control back to it. For monitoring, system calls are assigned a GID and control is passed to a monitoring graph for the system function. When a user task calls a system function, the GID of the function is determined from a field embedded in the user task monitoring graph for the executed instruction. Once the GID for the system call is determined, the hardware monitor saves the state of the current task's monitoring in the bookkeeping tables, finds the memory slot holding that system call's graph using its GID and starts traversing that graph. During the automatic loading of the system call's monitoring graph, the CPU is stalled by the deactivation of the *Done* signal from the monitor.

3) OS Scheduler: The most common source of context switches is the OS user task scheduler. The two most frequent invocations of the OS scheduler are from the timer ISR and an application's system call to yield the processor. When the scheduler is invoked, it chooses the next process to execute. In μC/OS-II, a priority-based scheduler is used, although round-robin or other approaches are also possible. When the next task is determined, the processor forwards the task's PID to the monitor. The monitor identifies the appropriate monitoring graph by consulting the bookkeeping tables. Once the monitoring graph information is in place, the context switch is made and monitoring is switched from the ISR or system call monitoring graph to the graph for the user task. The processor is notified that it can proceed via the *Done* output from the monitor.

D. Recovery

When an attack is detected, the monitor signals a reset to the processor. For application processes, the operating system can simply kill the process and use internal mechanisms to recover memory and restart the application. Many embedded applications can recover from such a restart. If necessary, more complex checkpointing and recovery mechanisms can be implemented. If the reset occurs during operating system processing, then the entire system needs to be restarted.

One concern with the recovery process is that a simple attack (e.g., caused by a small number of I/O operations) can cause a costly recovery operation (e.g., rebooting the system). An attacker could use this as a denial-of-service mechanism. However, the hardware monitoring system ensures that the attack does not succeed (i.e., no malicious code is executed) and no attack vectors beyond this denial of service are available to an attacker.

V. Prototype Implementation

We have developed a prototype implementation of the hardware monitoring system to show its effectiveness in providing security to embedded operating systems and their applications.

A. System Setup and Attack Scenario

Our system consists of a simple NIOS II soft processor, which is augmented by our hardware monitor. The system is described in Verilog and implemented on Terasic DE4 FPGA board utilizing an Altera Stratix IV FPGA. To install new binaries and graphs, another NIOS II core with a dedicated RSA decryption engine for secure hardware monitor loading is co-located with the main processor and the hardware monitor. The role of this co-located processor is to read encrypted binaries and graphs from an SD card, decrypt and verify them, and feed them to the main processor and the hardware monitor. Security installation of the binaries and graphs are discussed in detail in [17].

To test the ability of the hardware monitor to detect runtime attacks, we implemented a format string attack scenario [18]. Exploiting the vulnerability in `snprintf` function, we overwrite the first instruction of the interrupt service routine and replace it with a call to an arbitrary function which prints a simple message on the console. In a practical attack, this redirection of control flow can be used to execute arbitrary attack code.

B. System Operation

Under normal operation, our hardware monitor follows along with the context switches that occur in the operating system. When switching from an application process to the operating system (or the other direction), the current monitoring context is stored and the new monitoring context is loaded.

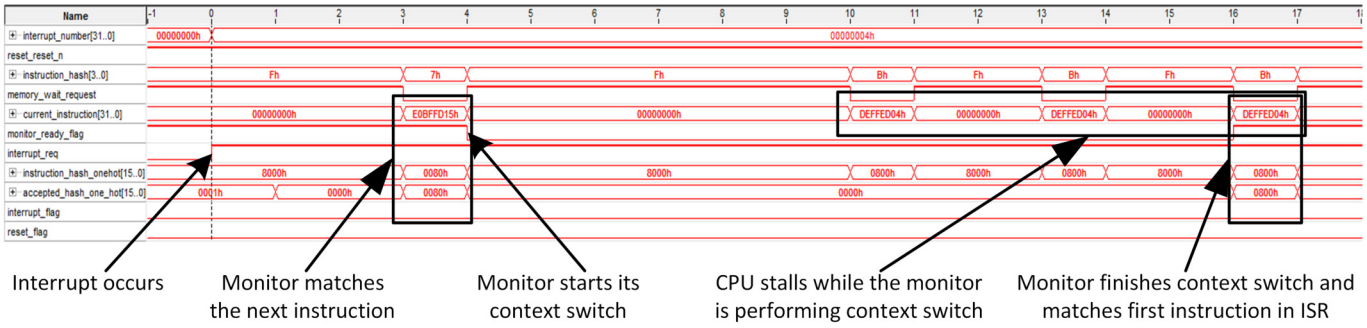

Fig. 3. Context switch interactions between processor and monitor.

Fig. 4. Attack on processor system, which is detected by hardware monitor.

Fig. 3 shows the interactions that take place during such a context switch. Our hardware monitor is blocking in the sense that the embedded processor stalls if the processor switches between contexts more quickly than the monitor (see cycles 10–16 in Fig. 3). While this stalling causes a slight overhead (see Section V-E), it ensures that no instruction is executed without being monitored.

C. Detection of Attack

When the processor is being attacked and the execution of attack code is attempted, the monitor reports an instruction execution that does not match with the monitoring graph of the application binary or operating system that is currently active. Fig. 4 shows such an attack detection. In particular, at cycle 18, there is a difference between the reported hash value (0x0008, i.e., 3 in one-hot coding) and the acceptable hash values (0x0800, i.e., 11 in one-hot coding). Thus the reset signal is asserted.

These results show that the security requirements (SR1 and SR2 in Section III-C) are met. In particular, as long as the attacker cannot modify the monitoring graphs (AC3), any change in the processing system (AC1 and AC2) that leads to any change in processing behavior can be detected by the monitoring system.

D. Monitoring Graph for Benchmarks and the OS

Using the graph extraction method described in [16], we extracted the monitoring graph for μC/OS-II and a set of benchmarks from [19]. To run the benchmarks on our NIOS based platform, minor modifications were performed. For example, our system does not have a file system. Therefore, we had to use static predefined data sets instead of reading from files. The number of instructions, the number of graph memory entries, and the total graph sizes in graph memory for each benchmark and the OS are shown in Table II.

TABLE II
MONITORING GRAPH SIZES FOR OPERATING SYSTEM AND APPLICATIONS.

	number of instructions	number of graph entries	graph size (bits)
μC/OS-II	22,913	23,625	850,500
basic_math	10,446	11,563	416,268
bitcount	6,731	7,823	281,628
qsort_small	7,113	9,055	325,980
qsort_large	7,302	9,116	328,176

E. Monitoring System Overhead

There are two types of overhead that we need to consider for our system. One overhead is the additional on-chip area that is required by the hardware monitor. This area consists of the logic necessary to implement monitoring functionality and context switching and the memory that is necessary to store monitoring graphs and contexts. We show the resources necessary for implementing the hardware monitor in Table III. The hardware monitor requires less logic than the embedded processor since its functionality is much simpler. It does require comparable memory resources because the monitoring graph needs to be stored in a format that allows fast transition

TABLE III
RESOURCE USE ON A STRATIX IV FPGA.

	Available on FPGA	Nios II w/o HW mon.	HW monitor + controller	Secure HW mon. loading
LUTs	182,400	2,997	764	2,603
FFs	182,400	3,200	922	2,936
Mem. bits	14,625,792	2,199,552	2,580,288	977,332

between states (within one processor cycle). In addition, the system requires a mechanism for securely loading monitoring graphs (to avoid tampering by an attacker). This security mechanism requires resources comparable to that of the processor and is shown in the final column of the table.

From these resource figures, the hardware monitoring system may seem *relatively* expensive to implement. However, the *absolute* resource use is very small. Also, the cost of the monitor does not increase with a higher-performance processor. For example, a higher-end processor may require more logic and have significantly more data memory, but the monitoring system would require the same amount of resources. Also, the secure loading system would only be required once when using a multi-core embedded system. Thus, the overall resource consumption is practically feasible.

The other overhead is the processing delay that is introduced by stalling the processor core during a context switch (see Section V-B). The delay for an interrupt on the processor (without any monitoring in place) is 6 cycles. As Fig. 3 shows, an additional 6 cycles of stalling is introduced by the hardware monitor which is not comparable to the hundreds of cycles needed to execute the ISR itself. The effect of this additional delay depends on the frequency of interrupts in the system. It should be noted that the original NIOS based system had the highest possible clocking rate of 198MHz and adding the hardware monitor and the cryptographic processor to it did not impose any slow down in terms of maximum clocking frequency.

These results show that our hardware monitoring system, which can detect any attack that changes processing behavior, can be implemented with reasonable amounts of additional hardware resources and practically no performance degradation on the system.

VI. SUMMARY AND CONCLUSIONS

In summary, embedded systems are particularly vulnerable to attacks. Their limited resources do not allow the use of conventional software-based defense mechanisms. In this paper, we presented a hardware-based security mechanism that can ensure correct execution of applications *and* operating system code at the granularity of individual instructions. Our prototype system shows that the proposed mechanism is feasible for these highly dynamic environments and effective in detecting any attack, even those that were previously unknown. We believe that this work presents an important step towards secure embedded processing systems for a broad range of applications domains.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1617458.

REFERENCES

[1] C. M. Medaglia and A. Serbanati, *An Overview of Privacy and Security Issues in the Internet of Things*. New York, NY: Springer New York, 2010, pp. 389–395.

[2] *Internet of Things: Privacy & Security in a Connected World*, Federal Trade Commission, Jan. 2015.

[3] D. E. Sanger and N. Perlroth, "A new era of internet attacks powered by everyday devices," *The New York Times*, Oct. 2016.

[4] T. Jaeger, Ed., *Operating System Security*. Morgan and Claypool, 2008.

[5] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, Boston, MA, Oct. 2004, pp. 85–96.

[6] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "RIFLE: An architectural framework for user-centric information-flow security," in *Proc. of 37th International Symposium on Microarchitecture*, Portland, OR, Dec. 2004, pp. 243–254.

[7] X. Zhai, K. Appiah, S. Ehsan, G. Howells, H. Hu, D. Gu, and K. D. McDonald-Maier, "Method for detecting abnormal program behavior on embedded devices," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 8, pp. 1692–1704, Aug. 2015.

[8] A. Waksman and S. Sethumadhavan, "Tamper evident microprocessors," in *Proc. of IEEE Symposium on Security and Privacy*, Oakland, CA, May 2010, pp. 173–188.

[9] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proc. of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001, pp. 144–155.

[10] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Secure embedded processing through hardware-assisted run-time monitoring," in *Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)*, Munich, Germany, Mar. 2005, pp. 178–183.

[11] S. Mao and T. Wolf, "Hardware support for secure processing in embedded systems," in *Proc. of 44th Design Automation Conference (DAC)*, San Diego, CA, Jun. 2007, pp. 483–488.

[12] K. Hu, H. Chandrikakutty, Z. Goodman, R. Tessier, and T. Wolf, "Dynamic hardware monitors for network processor protection," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 860–872, Mar. 2016.

[13] T. Thomas, A. Pouraghily, K. Hu, R. Tessier, and T. Wolf, "Multi-task support for security-enabled embedded processors," in *Proc. of 26th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Toronto, ON, Jul. 2015, pp. 136–143.

[14] D. Chasaki and T. Wolf, "Attacks and defenses in the data plane of networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 6, pp. 798–810, Nov. 2012.

[15] H. Theiling, "Extracting safe and precise control flow from binaries," in *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*. IEEE, 2000, pp. 23–30.

[16] H. Chandrikakutty, D. Unnikrishnan, R. Tessier, and T. Wolf, "High-performance hardware monitors to protect network processors from data plane attacks," in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*. IEEE, 2013, pp. 1–6.

[17] K. Hu, T. Wolf, T. Teixeira, and R. Tessier, "System-level security for network processors with hardware monitors," in *Proc. of 51st Design Automation Conference (DAC)*, San Francisco, CA, Jun. 2014, pp. 211:1–211:6.

[18] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier, "FormatGuard: Automatic protection from printf format string vulnerabilities," in *USENIX Security Symposium*, Washington, DC, Aug. 2001.

[19] 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 *Proc. of IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, Dec. 2001.