Multi-Task Support for Security-Enabled Embedded Processors

Tedy Thomas, Arman Pouraghily, Kekai Hu, Russell Tessier, and Tilman Wolf
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA, USA

***Abstract*—Embedded systems require low overhead security approaches to ensure that they are protected from attacks. In this paper, we propose a hardware-based approach to secure the operation of an embedded processor instruction-by-instruction, where deviations from expected program behavior are detected within the execution of an instruction. These security-enabled embedded processors provide effective defenses against common attacks, such as stack smashing. Previous work in this area has focused on monitoring a single task on a CPU while here we present a novel hardware monitoring system that can monitor multiple active tasks in an operating-system-based platform. The hardware monitor is able to track context switches that occur in the operating system and ensure that monitoring is performed continuously, thus ensuring system security. We present the design of our system and results obtained from a prototype implementation of the system on an Altera DE4 FPGA board. We demonstrate in hardware that applications can be monitored at the instruction level without execution slowdown and stack smashing attacks can be defeated using our system.**

I. Introduction

Embedded processing systems are widely used and are key technology for control systems, the Internet of Things, personal health monitoring, home automation, and many other application domains. Due to their wide use and the importance of their tasks, embedded systems need to be protected from hacking attacks. With an increasing number of embedded systems being connected to networks, one typical attack vector against embedded systems is through the global Internet.

Many embedded systems are based on general-purpose processing systems that are vulnerable to the same type of attacks as conventional desktop and server computers, albeit for a different set of applications. The National Vulnerability Database (NVD) [1] shows that around 10% of vulnerabilities (6,518 out of 66,399) in systems are related to overflows that can be exploited via a network. Many of these overflows then enable an attacker to execute malicious code. Thus, our work focuses on protecting embedded systems from this important type of attack using a security-enhanced processor.

While desktop and server computers have the processing power to run malware detection software (e.g., virus scanner, intrusion detection system, etc.), embedded systems are typically not able to do so due to resource constraints (e.g., limited power budget, limited processing capacity, etc.). Instead, hardware-based protection mechanisms have been developed, in particular "hardware monitors," which track the operation of the processing system and aim to detect and suppress malicious activity.

A variety of different hardware-based solutions have been proposed to protect embedded processing systems. In general, there have been two shortcomings in existing work:

- Monitoring on systems with complex workloads is based on coarse indicators (e.g., function call sequence [2]). This approach leaves the system vulnerable to attacks that happen between indicators (e.g., within a function call).
- Fine-grained monitoring systems do not support multi-task workloads on operating systems. This constraint limits the applicability of this single-task monitoring to specialized domains (e.g., embedded control systems, network processors, etc.).

To make hardware monitors an effective protection mechanism for attacks on embedded systems in any application domain, it is critical to develop fine-grained monitoring on multi-task embedded systems. In our work, we present the design of a hardware monitoring system that coordinates with the task switching dynamics of an operating system to verify every instruction executed by applications.

The specific contributions of our work are:

- Design of a Multi-Task Hardware Monitor System (MTHM) that supports multi-tasking contexts and that operates in sync with an embedded operating system (OS).
- Prototype implementation of a hardware monitoring system on an FPGA-based DE4 board.
- Evaluation of the prototype and a demonstration of system protection from a stack smashing attack.

This security enhancement for embedded processors allows for the simultaneous use of application-specific monitoring information for multiple applications. The remainder of this paper describes the design, operation, and implementation of our hardware monitoring system in more detail. In Section II we discuss other security approaches for embedded processors, including monitoring. Section III provides the security model and operation of our system while Section IV describes the hardware details and protocols involved in task switching using monitoring. Experimental results are presented in Section V. Section VI concludes the paper.

II. Related Work

Protection mechanisms for processing systems against code injection attacks are manifold. Network devices, such as firewalls [3] and intrusion-detection systems [4], can block malicious network traffic if packet payloads are not encrypted and if detection rules (e.g., Snort [5]) are updated quickly enough. Programming language extensions can generate code that is not vulnerable [6] if source code is available and can be transformed appropriately. Stack protection mechanisms in program code or in the operating system can defend against some attacks [7]. Memory protection mechanisms that separate instruction and data memory (e.g., Harvard architecture or No-eXecute (NX) bit) can avoid some attacks, but are still vulnerable [8]. A survey of these various techniques can be found in [9].

The prevention of stack smashing attacks has been the focus of significant work, although most approaches require significant processor modifications and run-time execution slowdown. Dynamic instruction flow tracking (DIFT) [10] tags each incoming data value or its derivative with a one-bit tag to indicate that it should not influence program control flow. The approach can require an execution slowdown due to tag checking. CHERI [11] establishes a base and bounds for pointers, preventing illegal accesses to memory which can lead to buffer overflow attacks. This approach also involves data tagging and the use of a special-purpose capability processor and registers to dynamically assess tags. The Hardbound approach [12] includes hardware to check the address bounds of every pointer access to memory. A flexible software-only approach [13] introduces a compiler pass for each application to insert bounds checking operations in the code. Although flexible for a range of applications, an increase in code size and application slowdown make the approach limited for embedded applications.

One very effective protection mechanism is the use of hardware and software monitoring to track different aspects of program behavior. The granularity of such monitors ranges from a call sequence (e.g., [2]) to checksums over basic blocks (e.g., [14]) to per-instruction verification (e.g., [15]). Coarse monitoring granularity may not be able to detect attacks that require only a few instructions to execute (such as demonstrated for a denial-of-service attack in network processors [16]). Thus, our work focuses on monitors that perform per-instruction monitoring and can detect attacks immediately when program behavior changes. Due to the need for tight coupling, such monitors are implemented in hardware and co-located with the processor core.

Existing hardware monitors have been used to monitor processors with single-task workloads (e.g., [15]) or with a small number of tasks that are managed through a control processor (e.g., [17]). However, an increasing number of embedded systems use operating systems, where multiple tasks actively share the processor core and tasks are dynamically added and removed. Our work focuses on providing security through instruction-level monitoring in such a highly dynamic workload controlled by an operating system. All instructions (both data and control) are monitored by our approach.

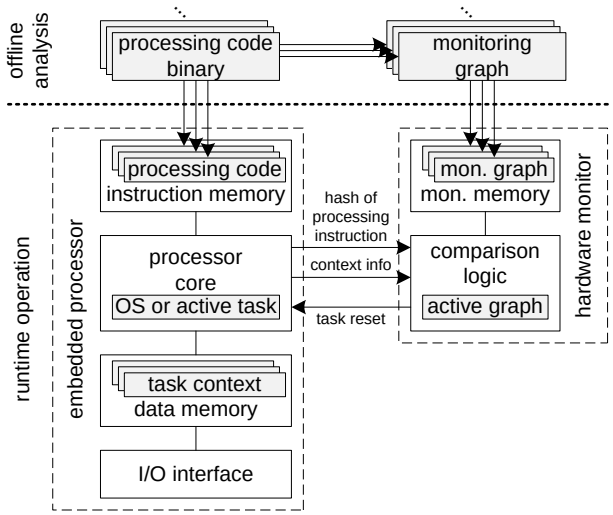

Fig. 1. System architecture of Multi-Task Hardware Monitor System.

III. System and Security Model

To provide the necessary context for the Multi-Task Hardware Monitor System design presented in Section IV, we briefly discuss the operation of MTHM and the security model for our work.

A. Secure Processing with Hardware Monitors

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). As discussed in related work, there are a number of different approaches to monitoring based on what information is communicated from the processor to the monitor and what information is used to determine if that behavior is "normal."

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 based on the analysis of the processing binary (similar to [15]). Each instruction is represented by a 4-bit 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 [18]).

The system architecture of our Multi-Task Hardware Monitor System, which supports multiple tasks, is illustrated in Figure 1. The figure shows that application binaries are analyzed offline. During runtime, the comparison logic in MTHM matches the monitoring graph to the currently active task on the processor. To do the operation, the OS-to-Monitor Interface (OMI) communicates the necessary context information

between the processor and the monitor. When the processor execution does not match the expected behavior reflected in the monitoring graph of the current task, a reset signal is sent from the monitor to the processor to terminate the current task. (More complex recovery and roll-back mechanisms could be implemented, but are not discussed here.)

It is important to note that the hardware monitoring system is isolated from the processor and thus cannot be tampered with remotely by the attacker (e.g., to change the monitoring graph to match an attack). Related work discusses how to achieve such isolation while still enabling dynamic installation of hardware monitoring graphs through the use of cryptographic mechanisms [19].

B. Security Model

To justify how our proposed system provides a secure processing environment, we briefly discuss the security model that is the basis for our work.

1) Security Requirements: We require that our system meets the following security requirements:

SC1 The system should only allow execution of code as programmed in the executable binaries of each task.

SC2 Secure processing should be provided for multiple, dynamically changing tasks.

SC3 Malicious code execution in one task should not affect other tasks.

In addition to security, there are also practical performance requirements. As we show in our results, the hardware monitor does not reduce the performance of the embedded processor in any way. The only overhead is a few instructions (five for our experimentation) in the operating system code when switching tasks, which leads to a negligible reduction in processing speed.

2) Attacker Capabilities: We make the following assumptions about the capabilities of an attacker that tries to change the operation of the embedded system and/or tries to execute malicious code on the embedded system:

AC1 An attacker can provide any input through input/output interfaces of the embedded system.

AC2 An attacker can start and stop any task from an installed binary in the embedded system (within the limitations of a maximum number of active tasks).

AC3 An attacker can tamper with any of the binaries.

In order to provide a practical solution for secure processing in an embedded system, we also require some reasonable constraints on attacker capabilities:

AC4 An attacker cannot tamper with the operating system itself.

AC5 An attacker cannot tamper with the hardware monitoring system (e.g., modifying monitoring graphs for installed executables).

As discussed above, we do not discuss the secure installation of monitoring graphs, which has been addressed in related work [19] in more detail.

IV. Monitor Design

A. Task Management in the Operating System

A key aspect of our monitoring system is its ability to fit seamlessly within the context switch operations of a typical operating system. As noted in Section V, the time required to switch monitoring graphs for different tasks is significantly less than the typical time required for other activities in a context switch. In our implementation, graph switching is synchronized with other OS actions (e.g., register file save and restore) that occur during a context switch so that user tasks are protected at all times. Typical context switch activities for embedded operating systems, such as μC/OS-II[1] used for this work, include:

1) A timer or other OS event generates an interrupt triggering a context switch.
2) The OS scheduler determines the next process for execution. Our implementation uses a priority based scheme, although round-robin or other schedulers would also be appropriate.
3) The OS provides the process ID (PID) of the next process to the monitoring system, triggering a monitoring graph switch in the monitor. This switch includes monitor state saving for the process currently being monitored, and a restoration of monitoring state for the next process.
4) Concurrently, the OS saves process state (registers, program counter, etc.) for the current task to main memory.
5) The OS retrieves process state for the next process from main memory and restores it to processor registers.
6) The OS checks the status of the monitoring system to confirm that the monitor for the next process is ready for use.
7) The OS sends a trigger to the monitoring system to start monitoring for the newly-loaded process.

After the context switch is completed, the processor sends every instruction executed for the process to the monitoring system. In the next section, we provide a detailed view of the monitoring system and how it interacts with the processor for steps 3, 5, and 6 above.

B. Multi-Task Hardware Monitor System

A detailed view of our monitoring subsystem is shown in Figure 2. The portions of the monitoring system can be split into *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 state information about monitoring for each process, *controller*, and *processor interface*. A detailed example using similar monitoring hardware and graph memory can be found in [18].

The monitoring hardware checks each processor instruction using information from the monitoring graphs stored in graph memory. In the figure, graphs for four separate applications are stored in *slots* in the graph memory. Each graph includes

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

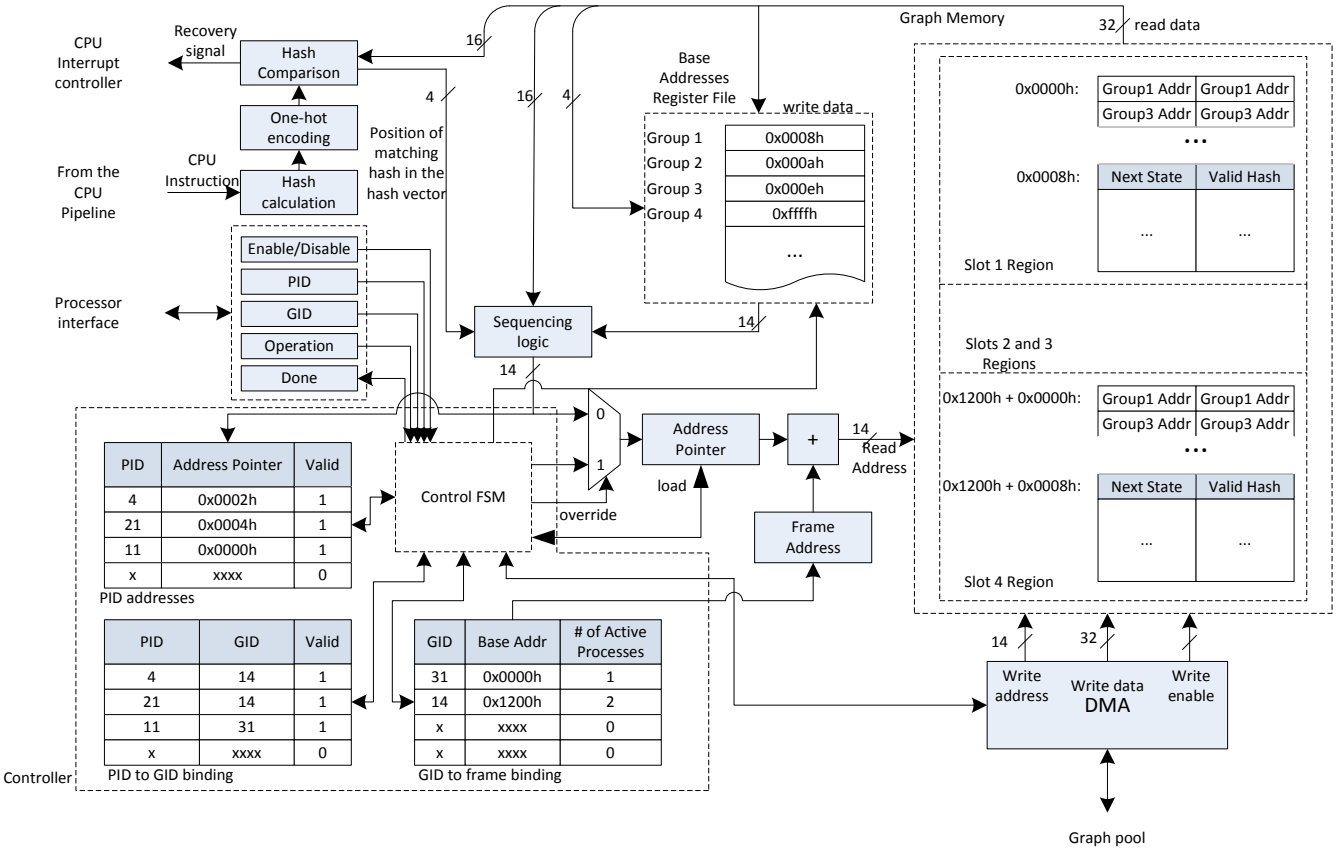

Fig. 2. Detailed view of multi-context monitoring system

one row per instruction, effectively representing expected program control flow as a state machine [18]. 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 [18]. 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 a 4-bit 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 an application 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 fanin count [18]. 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 application. The implemented monitor requires only one memory lookup per instruction.

Effectively, the monitoring information for each process 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*. If a context switch is requested, these values must be updated to use values for the requested next process. The procedure required for a context switch inside the monitoring system is described next.

C. OS-to-Monitor Interface for Context Switch

In case of a context switch, control information is exchanged between the processor and the monitoring system. The exchange of monitoring information (Step 3 in Section IV-A) starts when the processor writes the *PID* of the next process into the *PID* register in the *processor interface* of the monitoring system and sets a bit in the *Operation* register. The *control FSM* then performs the following actions:

1) The *address pointer* for the currently executing process is saved in the *PID addresses* storage so that it can be restored for the next invocation of the process.
2) The graph ID (*GID*) associated with the next process is located in the *PID to GID binding* storage using the *PID* written to the processor interface.
3) If the graph ID of the next process differs from the ID of the previous one, the *base address registers* are loaded with values for the graph of the next process. These values are loaded from the graph memory (e.g., Group1 Addr, etc).
4) The *GID* is used to determine the *frame address* for the start of the appropriate monitoring graph in graph memory for the process. This information is stored in the *GID to frame binding* storage.
5) The *address pointer* value for the next process is restored from the *PID addresses* storage.
6) The *Done* bit is set in the *processor interface* indicating that the monitoring system is now ready to monitor the next process. This bit can be read by the processor.
7) Once all other context switch activity for the next process has concluded (e.g., processor registers are loaded), the processor sets an *Enable* bit in the *Operation* register of the processor interface, restarting monitoring. The processor waits until this bit set is successfully made, ensuring synchronization. Instructions of the newly-loaded process are then monitored.

In Section V we show that these steps can be performed in 17 clock cycles for our prototype system.

D. OS-to-Monitor Interface for Process Creation

When a new task is being created by the OS, it is assigned a unique *PID* and *GID* by the operating system. Since many processes of the same application may exist, the *GID* may not be unique. The following steps are used to initialize the security monitor for the new process.

1) The two identifiers (GID and PID) are passed to the monitor via the *processor interface.* The monitor first searches for an empty slot in the *PID addresses* storage and *PID to GID binding* storage to insert the new bindings.
2) While making these associations, the *GID to frame binding* storage is searched to determine if the appropriate graph is already loaded. If it is available, the next step is skipped.
3) If the *GID* is not found in the *GID to frame binding* storage, the *GID* is inserted into the table and the new graph is loaded into graph memory using the DMA interface. If the graph memory is full, a graph to remove is determined using a least recently used approach. Following graph loading, base addresses are updated.
4) The *Done* bit is set in the *processor interface* indicating that the monitoring system is now ready to monitor the next process. This bit can be read by the processor.

During system startup, monitoring graphs are loaded from an external memory graph pool for the new processes that will be executed by the processor. Concurrently, the processor performs a series of process creation operations including initialization of the process stack and control block (registers, etc.). In Section V, it is noted that while process creation can require hundreds of cycles for the processor, if the appropriate monitoring graph is already in the monitoring system, monitoring information update for process creation requires less than 20 cycles for the monitoring system.

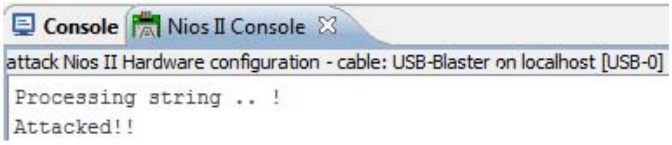

Fig. 3. Console display during stack smashing

V. Prototype Implementation

A. System Setup

To verify the functionality of our monitoring system, we implemented an embedded NIOS II processor plus monitoring system using a Stratix IV GX230 FPGA located on an Altera DE4 board. A single-core NIOS executing a μC/OS-II operating system was used for testing. Monitoring logic and memory were implemented in on-chip resources. Monitoring graphs were generated by passing code through a standard MIPS_GCC compiler flow to generate assembly-level instructions [18]. The output of the compiler allows for the identification of branch instructions and their target addresses. This information was used to generate monitoring graphs for four MiBench[2] applications (*bitcount*, *qsort*, *basicmath*, and *stringsearch*) and malicious stack-smashing attack code. Our examination of all MiBench benchmarks determined that the target for all dynamic branches could be determined at compile time.

The attack code we use for our system is a simple C function which accepts a character string from an I/O port and copies it to a buffer located on the processor stack [20].

```
void process_input(char *stringpassed) {
    char name[90];
    strcpy(name,stringpassed);
    printf("Processing string .. !\n");
    return;
}
```

In this poorly designed code, no check is made to determine if the string **stringpassed** is longer than the target buffer, so the return address of the function can be overwritten with an address which points into the user-provided input string. Instead of characters, this "string" can contain processor instructions which repetitively print out "Attacked!!" on a

[2]http://wwweb.eecs.umich.edu/mibench/

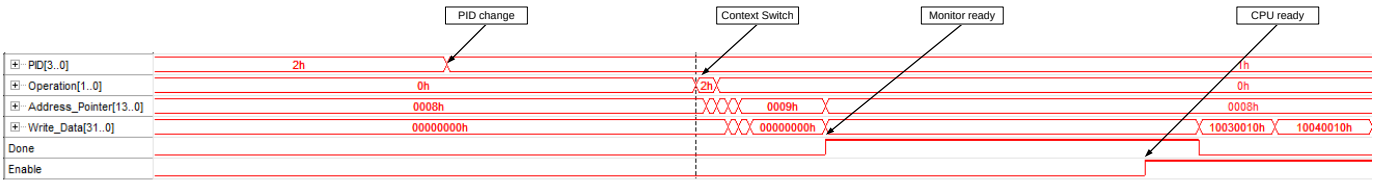

Fig. 4. SignalTap waveforms showing the trigger for monitor context switch (Operation = 0x2), monitor switch finished (Done), and monitor restart monitoring when CPU ready (Enable)

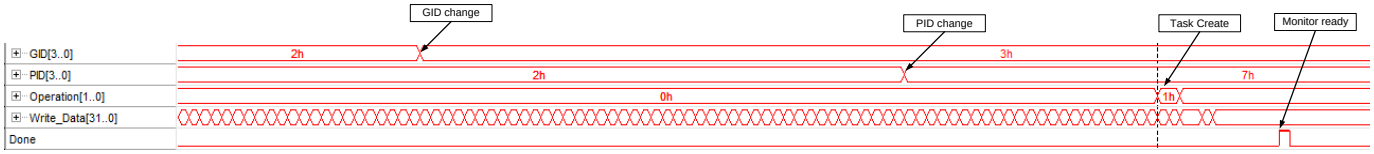

Fig. 5. SignalTap waveforms showing the operations for monitor process creation (Operation = 0x1) and monitor update finished (Done)

terminal in a loop, although much more malicious behavior could be imagined. A monitor for the code is able to detect the unplanned control flow jump and kill the process before the attack can perform this activity. The hash values stored in the monitoring graph for the application will not match the values for the malicious instructions as they are executed during the attack. As shown in Figure 3, we have confirmed that this attack will lead to unexpected results (an attack message) if monitoring is not used.

B. Monitor Context Management

We have verified our ability to perform numerous context switches between multiple processes of the four monitored MiBench benchmarks both via simulation and in emulation hardware. This switch includes both standard process state used by the processor (e.g., register information, stack) and monitoring information using the mechanism outlined in Section IV-C. Altera SignalTap, a hardware debugger, was used to generate the waveforms described in this section.

The waveforms in Figure 4 show the synchronization between the processor and the monitor as a result of the context switch. First, the processor notifies the monitoring system of the switch by writing the *PID* of the next process into the processor interface. The monitor switch is started by the processor writing into the *Operation* register of the interface. The value of the *address pointer* for the old process is stored and the value for the new process is restored to/from *PID* address storage immediately after this trigger. The base address registers are then configured using the *write_data* port shown in Figure 2. After the control FSM performs the monitor update, the *Done* signal is set in the processor interface indicating the monitor context switch is finished. Finally, after the processor finishes other context switch operations, it sets the *Enable* signal in the processor interface to restart monitoring. The processor waits a cycle until this write is complete. Monitoring for the new process starts with the first instruction received from the process.

Application	Instructions	Graph entries
qsort	96	111
bitcount	60	74
basicmath	107	132
stringmatch	77	97

TABLE I
APPLICATION INSTRUCTION COUNT AND MONITORING GRAPH SIZE

Experiments in simulation and in the lab on FPGA hardware showed that the processor is able to process data for the MiBench benchmarks equally fast both with and without monitoring (e.g., no slowdown for monitoring). Context switch time is extended by 5 cycles versus no monitoring to allow for monitor context switches. This overhead accounts for the data exchanges between the processor and monitoring system for synchronization. Overall, we found that the number of cycles needed to perform a monitor context switch is 17 versus the 34 cycles needed for the processor to save and restore registers (note that monitor and processor context switch operations occur in parallel).

The amount of time needed to create a new process in the OS is about 600 clock cycles versus 17 to create process information in the monitor (Figure 5). If a monitoring graph is loaded from main memory, the cycle count required for the monitor increases to include reading the number of rows in the monitoring graph for the new process into graph memory (about 104 on average for our applications, as seen in Table I). The number of graph entries (rows) for each application is somewhat larger than the application instruction count due to the DFA representation of the graphs [18].

C. Attack Detection and Protection

We have verified in both simulation and in hardware that our monitoring system is able to detect the stack smashing attack described in Section V-A and notify the processor so that the malicious process can be terminated. SignalTap waveforms derived from observing hardware operation in system are

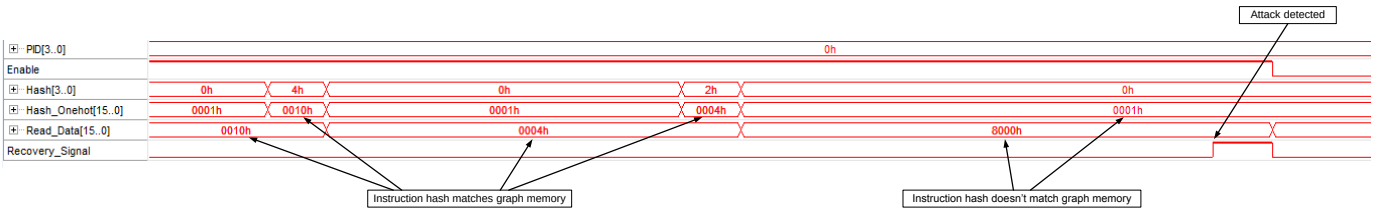

Fig. 6. SignalTap waveforms showing the successful identification of a stack smashing attack. Instruction hash values are checked against expected values stored in the monitor graph. When a mismatch occurs, a recovery signal is triggered indicating the process should be terminated.

Operation	*Number of cycles*
Interrupt latency	1
Save CPU registers	25
Interrupt handler	129
Interrupt service routine	30
Task delete	126
Total	311

TABLE II
DETAILED ACCOUNTING OF PROCESS DELETE TIME AFTER A MONITOR DETECTS AN ATTACK.

	Available on FPGA	Nios II with no HW monitor	HW monitor and controller
LUTs	182,400	1,341	406
FFs	182,400	1,166	522
Mem. bits	14,625,792	2,108,416	524,512
Pwr (mW)	-	105.97	41.83

TABLE III
RESOURCE USE AND POWER CONSUMPTION ON A STRATIX IV FPGA

shown in Figure 6. As described in Section IV-B, an attack is detected when the hash of the CPU instruction does not match the expected value stored in the monitoring graph for the application. In our system, the implemented hash function counts the number of ones in the instruction to form a four-bit hash value. The figure shows the four-bit hash value, a one-hot version of the hash value, and the retrieved, expected hash value for the instruction from the monitoring graph (*read_data[15:0]*). In the waveforms, it can be seen that the correct hash value is matched twice, but the third hash value is incorrect, indicating a branch to an unexpected section of code. As a result of this detection, a recovery signal is generated and used to trigger an interrupt, notifying the processor that the process should be terminated.

From Table II, the interrupt latency in this termination activity is only 1 cycle, limiting the number of executed attack code instructions to two. Once the interrupt occurs, the operating system takes control and saves the CPU registers. Subsequently, the interrupt handler is called to determine what caused the interrupt to occur. The interrupt handler also disables certain OS features like context switching. An interrupt service routine then determines the attack task by reading the *PID* from the processor interface in the monitor. The delete operation for this task in the OS takes 126 cycles. Overall, 311 cycles are required to recover from an attack and to continue normal operation.

D. Monitoring System Resources

To provide some context regarding the amount of overhead required by the monitoring system relative to the processor, hardware results of the system reported by the Altera Quartus II tool are shown in Table III. The lookup table (LUT), flip flop (FF), and memory resources required for the monitor are appropriate compared to the processor core. Dynamic power values are also shown in the table. These power numbers were generated using Altera PowerPlay with standard node toggle rate settings.

E. Discussion of Security Properties

We argue that the system we have designed and prototyped achieves the security requirements we put forth in Section III-B.

The key observation is that our hardware monitor can detect when a specific task executes code that is different from the binary. In such a case, the hash value that is reported from the processor core to the monitor does not match. (There is a chance that the attacker is lucky and the hash matches by coincidence or the attacker is clever and aims to construct code that matches. This action, however, is very difficult to achieve in practice and can be defeated by hiding the hash function [19].) If the monitor detects deviation from the binary, then the processor is signaled to stop execution of the attacked task. Thus, SC1 (no execution of attack code) is achieved.

Our system supports multiple tasks that are switched dynamically by the operating system. The hardware monitor follows along in sync and associates the current task on the processor core with the correct monitoring graph. Thus, we achieve SC2 (secure processing for multiple tasks).

Finally, when an attack occurs, the hardware monitor informs the operating system about the attack and the targeted tasks are stopped using a conventional task termination mechanism (similar to the `kill` command). This mechanism is specifically designed to not affect other tasks. Thus, SC3 (isolation of attacked task) is achieved.

We rely on the limitations of attacker capabilities, such as AC4 and AC5 (no tampering of operating system or hardware monitor), to ensure that an attacker cannot circumvent the security mechanisms we have put in place.

VI. Conclusions and Future Work

In this paper, we have presented security hardware for embedded processors that execute multiple processes under the control of an operating system. Our monitoring approach allows the operation of each process to be tracked at the instruction execution level. If a deviation from the expected instruction execution sequence is detected, the monitor can quickly identify it and notify the processor to initiate process termination. A significant contribution of the work is the inclusion of multi-context support in the monitoring system. Monitoring state for each process can be quickly saved during a process context switch and previously-stored state can be reloaded. We document the specific steps needed to ensure synchronization between the processor and monitor to ensure that each process is always protected during execution. Using prototyping, we show that our system is effective for multiple processes managed by an embedded OS. A stack smashing attack is identified and suppressed. The monitoring system does not impact application execution time.

In the future, we plan to explore expanding the monitoring system to support multiple processor cores [21]. This extension will require monitor sharing and enhanced graph loading.

Acknowledgments

The authors wish to thank Altera Corporation for the donation of the Quartus II software and DE4 board. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1115999.

References

[1] *National Vulnerability Database*, National Institute of Standards and Technology, http://nvd.nist.gov.

[2] 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.

[3] J. C. Mogul, "Simple and flexible datagram access controls for UNIX-based gateways," in *USENIX Conference Proceedings*, Baltimore, MD, Jun. 1989, pp. 203–221.

[4] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a content-scanning module for an Internet firewall," in *Proc. of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, Apr. 2003, pp. 31–38.

[5] M. Roesch, "Snort - lightweight intrusion detection for networks," in *Proc. of the 13th USENIX Conference on System Administration (LISA)*, Seattle, WA, Nov. 1999, pp. 229–238.

[6] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *Proc. of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC)*, Monterey, CA, Jun. 2002, pp. 275–288.

[7] T.-C. Chiueh and F.-H. Hsu, "Rad: a compile-time solution to buffer overflow attacks," in *Proc. of 21st International Conference on Distributed Computing Systems (ICDSC)*, Apr. 2001, pp. 409–417.

[8] A. Francillon and C. Castelluccia, "Code injection attacks on Harvard-architecture devices," in *Proc. of the 15th ACM Conference on Computer and Communications Security (CSS)*, Alexandria, VA, Oct. 2008, pp. 15–26.

[9] Y. Younan, W. Joosen, and F. Piessens, "Runtime countermeasures for code injection attacks against C and C++ programs," *ACM Computing Surveys*, vol. 44, no. 3, pp. 17:1–17:28, Jun. 2012.

[10] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Dec. 2004, pp. 85–96.

[11] J. Woodruff, R. Watson, D. Chisnall, S. Moore, J. Anderson, B. Davis, B. Laurie, P. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proc. of the International Symposium on Computer Architecture*, Jun. 2014, pp. 457–468.

[12] J. Devietti, C. Blundell, M. Martin, and S. Zdancewic, "Hardbound: Architectural support for spatial safety of the c programming language," in *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Seattle, WA, Mar. 2008, pp. 103–114.

[13] S. Nagarakatte, J. Zhao, M. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *Proc. of the International Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun. 2009, pp. 245–258.

[14] 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)*, Munich, Germany, Mar. 2005, pp. 178–183.

[15] S. Mao and T. Wolf, "Hardware support for secure processing in embedded systems," *IEEE Transactions on Computers*, vol. 59, no. 6, pp. 847–854, Jun. 2010.

[16] 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.

[17] K. Hu, H. Chandrikakutty, R. Tessier, and T. Wolf, "Scalable hardware monitors to protect network processors from data plane attacks," in *Proc. of First IEEE Conference on Communications and Network Security (CNS)*, Washington, DC, Oct. 2013, pp. 314–322.

[18] H. Chandrikakutty, D. Unnikrishnan, R. Tessier, and T. Wolf, "High-performance hardware monitors to protect network processors from data plane attacks," in *Proc. of 50th Design Automation Conference (DAC)*, Austin, TX, Jun. 2013, pp. 80:1–80:6.

[19] 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.

[20] A. B. Sikiligiri, "Buffer overflow attack and prevention for embedded systems," Master's thesis, Department of Electrical and Computer Engineering, University of Cincinnati, 2011.

[21] T. Thomas, "Hardware monitors for secure processing in embedded operating systems," Master's thesis, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, 2015.