Hardware Support for Secure Processing in Embedded Systems *

Shufu Mao and Tilman Wolf
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA, USA
{smao,wolf}@ecs.umass.edu

ABSTRACT

The inherent limitations of embedded systems make them particularly vulnerable to attacks. We have developed a hardware monitor that operates in parallel to the embedded processor and detects any attack that causes the embedded processor to deviate from its originally programmed behavior. We explore several different characteristics that can be used for monitoring and quantitative tradeoffs between these approaches. Our results show that our proposed hash-based monitoring pattern can detect attacks within one instruction cycle at lower memory requirements than traditional approaches that use control-flow information.

Categories and Subject Descriptors

C.3 [**Special purpose and application-based systems**]: Real-time and embedded systems.

General Terms

Design, Security.

Keywords

Embedded system security, processing monitor, hardware monitor.

1. INTRODUCTION

Embedded systems are widely deployed and used in application domains ranging from cellular phones to smart cards, sensors, network infrastructure components, and a variety of control systems. Two key characteristics make these systems particularly vulnerable to attacks. First, the embedded nature of the processing system limits the complexity of the device in terms of processing capabilities and power resources. It also exposes the device to a number of potential physical attacks. Second, as a direct result of the limited processing capabilities, embedded systems are limited in their capabilities to run software to identify and mitigate attacks. Unlike workstation computers that can afford to run virus scanners and intrusion detection software, embedded systems typically only run the target application. Thus, embedded systems are inherently more vulnerable to attacks than conventional systems.

Attacks on embedded systems can be motivated by a number of different goals. The following list illustrates this point (but is not meant as a complete enumeration of all possible scenarios): (1) Extraction of secret information (e.g., reading of cryptographic key material from a smart card); (2) Modification of stored or sensed data (e.g., tampering with utility meter readings); (3) Denial of service attack (e.g., reducing the functionality of a sensor network); and (4) Hijacking of hardware platform (e.g., reprogramming of TV set-top box). In each of the cases, the attack relies on the ability to get access to the embedded system and change its behavior (i.e., change in instruction memory) or its data (i.e., change in data memory). It is important to note that in most attack scenarios a modification of behavior is necessary even when modification of or access to data is the ultimate goal of the attack. Therefore, we focus on the security of processing in this paper.

When proposing our security mechanism for embedded systems we take into consideration the following important observations:

- Independence: A monitoring subsystem should use independent system resources that overlap as little as possible with the target of a potential attack. In particular, using a single embedded Processor for processing applications and security-related software is a bad choice. If an intruder can access the processor, then the security-related software is just as vulnerable to attacks.
- Low Overhead: Embedded systems require a lightweight security solution that considers the limitations of embedded systems in terms of adding additional logic and memory for monitoring.
- Fast Detection: A monitoring subsystem should be able to react as quickly as possible to an attack. In particular, attacks on embedded systems that simply change memory state or extract private data may require only a few instructions to cause damage. Therefore it is important to be able to detect an attack within a few instructions.

Our proposed secure monitoring system takes these design goals into account and achieves the required performance in terms of low system complexity and fast detection speed.

The main idea behind our monitoring system is to analyze the binary code of an embedded system application and derive an augmented control flow graph. During run-time, the embedded processor reports on the progress of application processing by sending a stream of information to the monitoring system. The monitoring system compares the stream to the expected behavior of the program as derived from the executable code. If the processor deviates

*This work was supported in part by NSF Grant CNS-0447873.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2007, June 4–8, 2007, San Diego, California, USA.
Copyright 2007 ACM 978-1-59593-627-1/07/0006 ...$5.00.

from the set of possible execution paths, then it is assumed that an attacker has altered the instruction store or program counter to alter the behavior of the system. Our evaluations on an embedded system benchmark show that the proposed monitoring technique can detect deviations from expected program behavior within the time of a single instructions while only requiring a small amount of additional logic and memory in the order of one tenth of the application binary size.

The remainder of this paper is organized as follows. Section 2 discusses related work. The overall system architecture and details on the monitored information stream are presented in Section 3. Section 4 presents an extensive evaluation of the proposed architecture. Section 5 summarizes and concludes this paper.

2. RELATED WORK

The term "embedded system" covers a broad range of possible system designs, processor architectures, and performance and functionality characteristics. In our work, we focus on embedded systems that can be broadly characterized as middle to lower end in the performance spectrum. Their main characteristics are: (1) Medium to low-performance embedded processor core (e.g., single RISC processor); (2) Targeted use for one or only a handful of applications; and (3) Typically used in a networked setting. Examples for practical embedded systems that fit these characteristics are: cell phones, networked sensors, smart cards (typically not networked, though), low-end network routers (e.g., home/small office gateway), networked printers, etc.

Attacks on embedded system can have a wide range of approaches. Ravi et al. describe mechanisms to achieve physical security by employing tamper resistant designs [9]. Wood et al. consider a networked scenario where systems are exposed to additional remote attacks [11]. Embedded systems are also susceptible to side-channel attacks (e.g., differential power analysis [8]). Solutions to this problem have been proposed [4], and we do not consider this aspect in our work.

In terms of developing a general, hardware-based architecture to protect embedded systems against a range of attacks, Gogniat et al. have proposed such in [6]. This work does not give details on what the proposed monitors would look like. Our work can be seen as one example of how to monitor processing to ensure secure execution of applications.

In the context of monitoring processing on embedded systems, Arora et al. have proposed a system [2] that is conceptually similar to our work. The main difference is that their finest granularity of monitoring is the basic-block level due to the use of per-block hash values, and deviations in the program execution are detected when the hash value at the end of a basic block does not match. In our work, deviations from the binary can be determined within a single (or a few) instructions. Also, Arora et al. use control flow information to track program execution. As we discuss in Section 4, our proposed hash-based monitoring performs significantly better (i.e., faster detection) than control-flow based monitoring.

Another work about control flow integrity is from Abadi et al. [1], control flow graph is also used for monitoring in their approach. The main difference between their approach and ours is that, Abadi et al. rewrite the machine code to implement the necessary checks, while binaries do not need to be modified in our work. We also believe it is important to separate the processor from the monitor by using separate system resources to reduce vulnerability. Suh et al. use the concept of "information flow" to track if data is considered authentic or spurious (i.e., potentially malicious) [10]. This system requires a much more complex design that needs to be integrated with the processor.

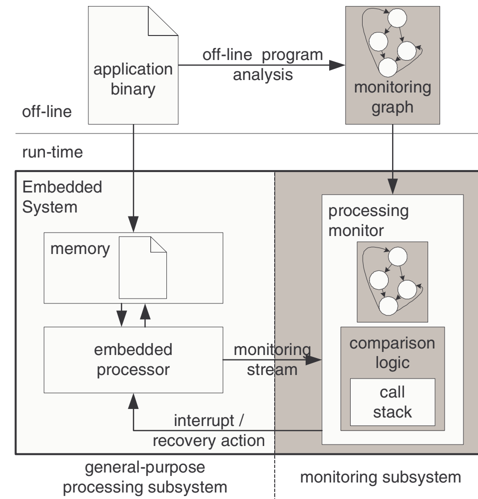

Figure 1: System Architecture for Secure Embedded Processing.

A completely different approach to ensuring secure execution of programs is to identify non-instruction memory pages with an NX (No eXecute) or XD (eXecute Disable) bit. The idea is to avoid a change of control flow to a piece of code that belongs to data memory. This mechanism is useful to avoid, for example, buffer overflow attacks. It does not consider a scenario where an attacker overwrites instruction memory.

Anomaly and intrusion detection by comparing behavior against a model is also used in other domains (e.g., mobile ad-hoc networks [5]). In our case, we have a simpler problem since our model is derived from the actual binary of the application. Thus, there is no guess-work on how accurate the model is – it is exactly the same as the application.

3. PROCESSING MONITOR

To achieve secure processing on an embedded system, we propose a monitoring system that verifies that the processor indeed performs the operations that it was intended to. In order for an attacker to abuse an embedded system, it is necessary to modify its operation in some way: either by adding a new piece of instruction code that performs malicious operations or by modifying the existing application to execute malicious code. In this work we assume that the embedded system workload is "secure" when the applications are executed correctly without any deviation from their binary code. Execution of any instruction that is not part of that binary or that is executed not in the correct order is considered an attack.

3.1 System Architecture

To detect an attack, we employ the system architecture shown in Figure 1. The system architecture consists of two major components operating in parallel, the conventional embedded processing subsystem and the security monitoring subsystem.

The conventional embedded processing subsystem consists of a general-purpose processor, memory, I/O, and any other components that are necessary to execute the embedded system application. The only addition is an extension to the processor core that continuously sends a stream of information to the monitoring subsystem. There is also a feedback component from the monitoring system to the processor. In the case an attack is detected, the monitor can halt the processor and initiate a recovery attempt.

The security monitoring subsystem implements the monitoring capability that compares the stream of information sent from the processor with the expected behavior derived from the off-line analysis of the binary. A “monitoring graph” represents the sequence of possible control flows between basic blocks. More detailed information about the processing steps within each basic blocks is also maintained. In order to be able to keep track of all permissible control flows, a call stack is necessary. If the comparison logic determines that there is a discrepancy between the stream of information from the processor and the monitoring graph, it determines that an attack occurred and initiates an interrupt to the processor.

As indicated in the figure, the monitoring graph is generated in an off-line process, where the binary of the application is simulated and analyzed. The simulation is necessary to resolve some branch targets that cannot be determined statically. It is important to note that this process indeed only requires the binary and not the source code of the application.

This system architecture reflects the design goals of a secure processing system that we have discussed in Section 1: The monitoring component uses system resources that are independent from the embedded processor (Independence). Thus, an attacker would need to attack the general-purpose processor and the monitoring system (at the same time) to avoid detection. The monitoring component operates in parallel with the embedded system processor. Since it does not replicate the data path of the processor, but solely monitors the control flow, it is considerably less complex (Low Overhead). We propose a number of different alternatives for the information stream below. In most cases, this stream contains information about individual instructions, or a block of a few instructions. This fine level of granularity allows the monitor to react quickly when processing deviates from what is expected (Fast Detection).

The complexity of the monitoring graph and the ability to detect attacks clearly depend on the choice of information that is passed between the processor and the monitoring system.

3.2 Information Stream Alternatives

There are endless choices of what characteristics to monitor with when attempting to detect attacks. We consider several alternatives of information streams that reflect various “processing patterns” that occur when executing an application. When implementing secure processing and monitoring on an actual system, only one such pattern would be used. Also, the off-line analysis and the monitoring graph representation discussed in Figure 1 need to change with different pattern. All of the information required for each pattern can be obtained from the application binary. Figure 2 illustrates examples for each of the five patterns:

- **Address Pattern:** The idea behind using the instruction address as an indicator for monitoring processing is that each instruction address is unique. Assuming instruction memory cannot be corrupted, a program must follow exactly the same sequence of instructions as it had been programmed to do. Using addresses, however, is vulnerable for the same reason. If an attacker can replace parts of the application code with a sequence of instructions that has the same basic block structure as the original, this change goes undetected. This vulnerability is due to the pattern using no information on what instructions are actually executed on the processor.

- **Opcode Pattern:** In contrast to the address pattern, the opcode pattern focuses solely on the operations that are performed on the processor. The intuition behind using this information for monitoring processing is that the sequence of operations parallels the underlying functionality of the program. An attacker would need replace instructions with malicious instructions that use the same opcodes (but possibly different operands) in the same sequence. This type of attack is likely to be more challenging than in the case of the address pattern.

- **Load/Store Pattern:** A pattern that considers the operands in instructions to monitor processing is the load/store pattern. In this pattern, only load and store instructions and their target register are considered. Instructions that are not memory accesses are ignored (shown as wildcard in the figure) and only the number of wildcards between memory accesses is stored in the pattern graph. The reason for considering the target register rather than the target address in memory is that the memory address cannot be determined statically.

- **Control Flow Pattern:** Another intuitive pattern is the control flow pattern. In this pattern, all control flow operations are stored (e.g., branches, calls, returns) including their branch targets if applicable. It allows the monitor to track any change in the program counter, but exhibits a similar vulnerability as address patterns since there is no information exchange on the actual operation of the processor. In related work, similar information is used to monitor processing [2]. In some cases the control flow information is limited to system calls. We consider control flow at the level of basic blocks.

- **Hashed Pattern:** Another pattern we propose to use for monitoring is the hashed pattern. In this case, several pieces of information (in our case instruction address and instruction word) can be compacted to a smaller hash value. This is particularly useful since opcode, operands, etc. can consume a lot of memory space. This pattern can be used with different lengths of hash functions. We us the function name hash$n$ to indicate that n-bit hash function is used.

3.3 Monitoring Graph and Comparison Logic

Given the monitoring graph that matches one of the patterns from above, the comparison logic can verify that the processing on the embedded system follows a possible path of execution.

When monitoring within a basic block, the comparison logic simply follows the sequence of patterns that is stored in the monitoring graph. For example, in the case of the opcode pattern, the monitor compares the opcodes reported by the processor to those in the current basic block of the monitoring graph. If wildcards are used (e.g., for the load/store pattern), any instructions reported by the processor can match the wildcard, except those that are part of the pattern (loads and stores in this case). The necessary logic is straightforward to implement since it comes down to a simple comparison between what the processor reports and what is stored in the monitoring graph.

When the end of a basic block is reached, control flow branches to one of up to two targets. The monitoring logic does not replicate the data path of the processor and thus cannot determine which branch is taken. Instead, the comparison logic allows for multiple parallel execution paths. That is, the monitor allows the current state of execution to be in multiple locations in the monitoring graph at the same time. As monitoring progresses, some of these states turn out to be invalid and thus are pruned from the set of concurrent states. If all states lead to invalid comparisons, then an attack is detected.

To illustrate this process, consider an opcode monitor at the end of a basic block where the current instruction is a conditional branch. In the next instruction, the processor either jumps to the branch

sample object code	monitoring graph				
	address	opcode	load/store	control flow	hash4
...	...	...	...	...	...
020004d0 str r0, [sp]	020004d0	str	str r0	*	0011
020004d4 str r0, [sp, #4]	020004d4	str	str r0	*	0001
020004d8 ldr r1, [pc, #1c4]	020004d8	ldr	ldr r1	*	0001
020004dc sub r4, r11, #2080	020004dc	sub	*	*	0110
020004e0 ldr r3, [pc, #1c0]	020004e0	ldr	ldr r3	*	1001
020004e4 sub r4, r4, #8 ; 0x8	020004e4	sub	*	*	0010
020004e8 ldr r2, [r11, -#2136]	020004e8	ldr	ldr r2	*	1010
020004ec mov r0, r4	020004ec	mov	*	*	1111
020004f0 bl 02091aa0	020004f0	bl	*	bl 02091aa0	0011
020004f4 mov r0, r4	020004f4	mov	*	*	1010
020004f8 mov r1, #0 ; 0x0	020004f8	mov	*	*	0111
020004fc bl 020905dc	020004fc	bl	*	bl 020905dc	1100
...	...	...	...	...	...

Figure 2: Examples of Monitoring Graphs for Different Information Streams.

target (e.g., an add instruction) or continues with the following instruction (e.g., a sub instruction). After validating the branch, the opcode monitors allows both following instructions to be valid states. If either an add or a sub is reported by the processor, the monitor accepts it as correct. At the same time, the path that does not match gets pruned. Depending on the code of the application, the duration for which the monitor is in an ambiguous state varies. As a result, detection of possible attacks can be drawn out until all ambiguity is removed and the monitor is certain that a reported processing sequence is invalid.

In addition to a data structure to maintain parallel state in the monitoring graph, the comparison logic also needs to maintain parallel call-stacks for each state. A call-stack is necessary since most instruction set architectures provide call and return instructions. The return instruction has an unknown target unless a stack of previously observed call instructions is maintained. Since different execution paths may traverse a different sequence calls and returns, these call stacks need to be maintained independently for each monitoring state.

4. RESULTS

We present a number of results on the performance and resource requirements of the proposed monitoring system for the five different information stream patterns. The setup to obtain results is as follows: We simulate the behavior of the monitoring system using an embedded system application workload on an ARM instruction set architecture. We use the MiBench benchmark suite [7] to generate realistic workloads. This suite encompasses over two dozen applications from six different application domains (automotive/industrial, consumer, office, network, security, and telecomm). These application domains match very well with the embedded systems complexity that our research targets and thus can be considered a representative workload. We employ the SimpleScalar simulator [3] to extract relevant monitoring information and the objdump utility for binary analysis to generate monitoring graphs. A 4-bit hash function is used as a representative of hashed pattern.

The first important results is that our implementation of the monitoring system performs application monitoring correctly for all application. That is, no false positives are reported by the monitor when executing the applications on the given benchmark inputs. To quantify the trade-offs between different monitoring patterns, we consider two performance metrics: (1) memory requirement for the monitoring graph and (2) duration of monitoring ambiguity.

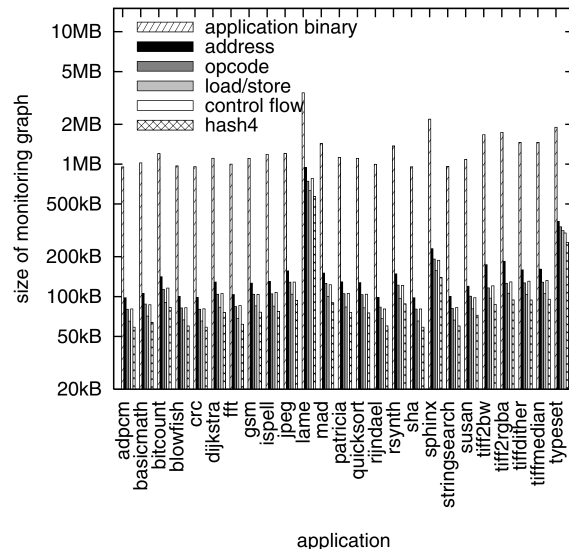

Figure 3: Size of Monitoring Graph for Different Benchmarks and Information Streams.

4.1 Monitoring Graph Size

The size of the monitoring graph that was generated from the binary of each application is shown in Figure 3. Each pattern is represented in a different shading. We assume a 32-bit address space and an efficient coding of the monitoring graph (e.g., run-length coding of sequences of wildcards, efficient coding of consecutive addresses). Each monitoring graph stores the patterns for each basic block as well as the branch pointer at the end of each basic block.

Figure 3 shows also the size of the application binary, which is 1–5MB in most cases. In comparison, the size of the monitoring graph is only around 100kB with the exception of two applications. This shows that the memory requirements for the processing monitor is only in the order of one tenth of the memory requirements of the application.

When comparing the monitoring graph sizes of different moni-

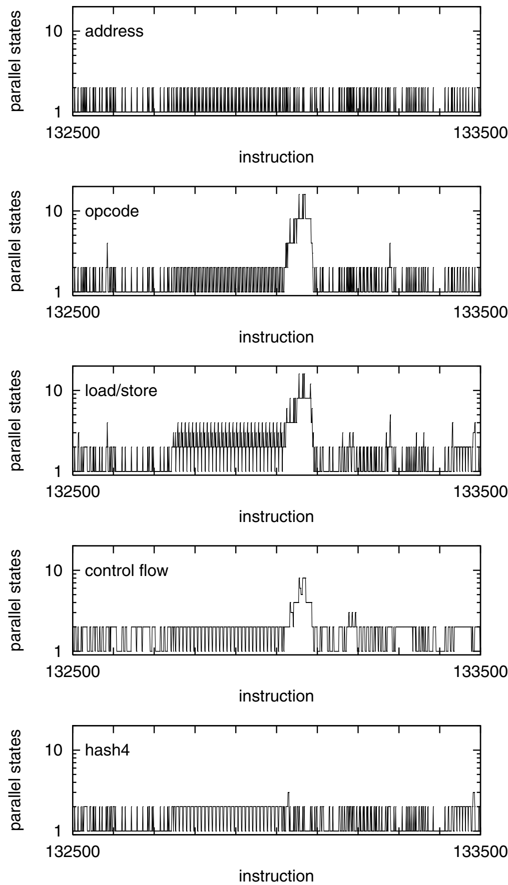

Figure 4: Snapshot of Monitoring of 1000 Instructions in *patricia* Application.

toring patterns, address is consistently the largest, while hash4 is the smallest. The difference is approximately a factor of two across all benchmarks.

4.2 Monitoring Ambiguity

To illustrate how ambiguity in the monitoring system occurs, we show a snapshot of a monitoring trace for a thousand instructions for one application in Figure 4. In most cases, applications alternate between 1 and 2 parallel states. The second parallel state is typically generated by a control flow operation where the actual path is uncertain for a few instructions. In some cases, the program causes to spawn a large number of parallel states, which can be caused by a loop or similar code that has a very regular pattern. The three patterns opcode, load/store, and control flow are particularly affected by this behavior.

The average number of parallel states in the monitoring logic is shown in Figure 5. The closer the values is to 1, the less frequently ambiguous states occur. With larger values, the chances that the monitoring system could be circumvented increases. The address, opcode, and hash4 patterns are all very close to the ideal for all benchmarks. The control flow pattern shows slightly higher averages for some applications. Large outliers occur for the load/store pattern for five applications.

To further compare control flow with our proposed hash4 monitoring pattern, Figure 6 shows a comparison of the size of the monitoring graph with the 95 percentile of ambiguous path length. The

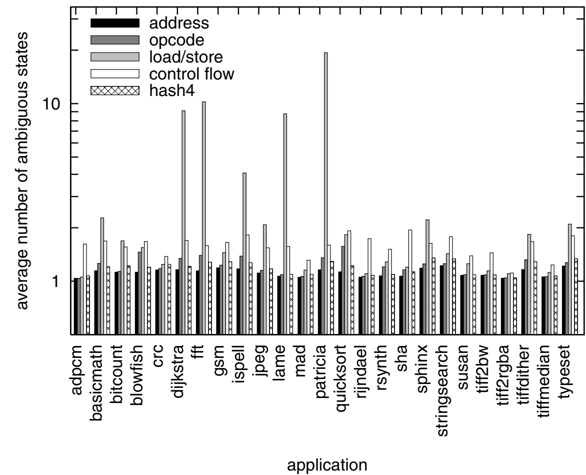

Figure 5: Average Length of Ambiguous Execution Paths for Benchmark Applications.

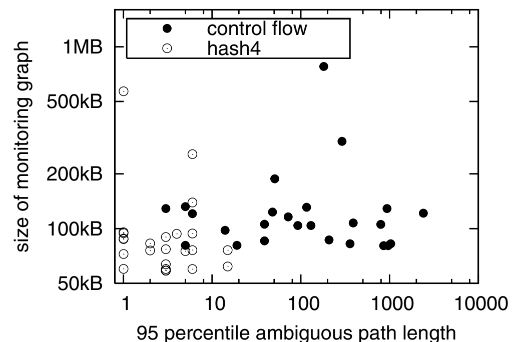

Figure 6: Monitoring Graph Length Compared to 95 Percentile Ambiguous Path Length. All 25 benchmark applications are plotted for each monitoring pattern.

shorter the ambiguous path length and the smaller the monitoring graph size, the better the overall performance. Clearly, the hash4 results cluster in the lower left corner. The control flow monitoring graphs are only slightly larger in size, but perform much worse than hash4 in terms of ambiguity. Again, this indicates that our proposed monitoring approach that uses a hash pattern to report processor information is a suitable approach to ensuring secure processing on an embedded system.

4.3 Evaluation with Random Attacks

To put the above results in context, we show the monitoring performance of our system when using "attacks" where the random bit flips are introduced into the application binary. The bit flips represent the smallest possible change an attacker could apply to a binary in order to change program behavior. Of course not all bit flips change program behavior (e.g., bit flip in unused portion of instruction word, change of register value that is never read, etc.) and thus may not be detected by some monitoring approaches. Thus it is important to consider what fraction of bit flips can be detected and how long it takes from the execution of the modified instruction to the point where the monitor is aware of the change. For our results, we choose one application (`gsm`) from MiBench and show the fraction of undetected bit flip attacks and the speed at which

Table 1: Performance of Monitor to Detect Bit Flip Attacks. The results are based on 100 simulations using the `gsm` application.

Monitoring pattern	undetected bit flips	avg. no. of instr. to detection
address	87%	49.1
opcode	60%	1.2
load/store	76%	15.8
control flow	74%	23.6
hash4	6%	1
hash16	$1.5 \cdot 10^{-3}$%†	1†
hash32	$2.3 \cdot 10^{-8}$%†	1†
Arora et al. [2]	$2.3 \cdot 10^{-8}$%†	approx. 6†

† Results estimated and not simulated due to extremely small probability of occurrence of event in experimental setup.

detected attacks are noticed in Table 4.3.

We find that the hashed pattern has the lowest percentage of undetected bit flips and the fastest possible detection speed. Other patterns can not detect a larger fraction of the attacks (e.g., the opcode pattern can only detect the attacks if the opcode of the instruction is changed or the control flow is changed) and take a long time until the program execution shows deviation from expected behavior (e.g., due to wildcards). The percentage of undetected attacks in the hash pattern depends on the size of the hash. When only 4 bits are used, the hash has 16 potential values and thus there is a $\frac{1}{16} = 6.25\%$ probability that the hash value does not change despite a bit flip. With larger hashes (e.g., hash16 or hash32 pattern), this probability decreases significantly (at the cost of larger monitoring graphs and more computational overhead).

We compare our results to the performance of control-flow based monitors as they have been proposed by Arora et al. [2]. The monitor used in that work compares the hash of all executed instructions at the end of a basic block. Thus, the detection speed can be as slow as average basic block length (which is reported to be approximately 6 instructions for `gsm` [7]). The probability for not detecting a bit flip attack is again based on the length of the hash used, which is 32 bits. Thus, the monitor in [2] can detect the same number of attacks as our hash32 pattern, but requires six times as much time to respond.

The ability of our monitor to detect attacks within the execution time of a single instruction is an important distinction. Embedded systems attacks can be launched using just a few instructions (e.g., writing a secret key to I/O, modifying or erasing stored data, etc.) and thus immediate response is crucial for successful defenses.

5. SUMMARY

In this paper, we have presented a novel architecture for secure processing in embedded systems. The key idea is to use a monitoring subsystem that operates in parallel with the embedded processor. The monitor verifies that only processing steps are performed that match up with the originally installed application. Any attack would disturb the pattern of execution steps and thus alert the monitor.

We have shown the operation and performance of the proposed monitoring system on the MiBench embedded systems benchmark suite. We have determined the monitoring graph size and monitoring detection speed for five patterns. Our results show that solely relying on control flow information – as it has been done in the past – is not an efficient way of detecting attacks. Instead, we have proposed a hash-based pattern that uses less memory and can detect deviations from intended processing within a single instruction cycle. This novel approach to monitoring processing on an embedded system presents a significant improvement over prior approaches. We believe this work is an important step towards providing hardware-based security solutions in embedded systems that address the inherent limitations of these architectures.

6. REFERENCES

[1] ABADI, M., BUDIU, M., ERLINGSSON, Ú., AND LIGATTI, J. Control-Flow Integrity Principles, Implementations, and Applications. In *ACM Conference on Computer and Communication Security (CCS)* (Alexandria, VA, Nov. 2005), pp. 340–353.

[2] ARORA, D., RAVI, S., RAGHUNATHAN, A., AND JHA, N. K. 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.

[3] BURGER, D., AND AUSTIN, T. M. The SimpleScalar tool set, version 2.0. Tech. Rep. 1342, Department of Computer Science, University of Wisconsin in Madison, June 1997.

[4] CHARI, S., JUTLA, C. S., RAO, J. R., AND ROHATGI, P. Towards sound approaches to counteract power-analysis attacks. In *Proc of the 19th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '99)* (London, United Kingdom, 1999), vol. 1666 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 398–412.

[5] CRETU, G. F., PAREKH, J. J., WANG, K., AND STOLFO, S. J. Intrusion and anomaly detection model exchange for mobile ad-hoc networks. In *Proc. of 3rd IEEE on Consumer Communications and Networking Conference (CCNC 2006)* (Las Vegas, NV, Jan. 2006), pp. 635–639.

[6] GOGNIAT, G., WOLF, T., AND BURLESON, W. Reconfigurable security primitive for embedded systems. In *Proc. of International Symposium on System-on-Chip (SOC)* (Tampere, Finland, Nov. 2005).

[7] GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of IEEE 4th Annual Workshop on Workload Characterization* (Austin, TX, Dec. 2001).

[8] KOCHER, P., JAFFE, J., AND JUN, B. Differential power analysis. In *Proc of the 19th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '99)* (London, United Kingdom, 1999), vol. 1666 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 388–397.

[9] RAVI, S., RAGHUNATHAN, A., AND CHAKRADHAR, S. Tamper resistance mechanisms for secure, embedded systems. In *Proc. of 17th International Conference on VLSI Design (VLSI Design 2004)* (Mumbai, India, Jan. 2004), pp. 605–611.

[10] SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. 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.

[11] WOOD, A., AND STANKOVIC, J. A. Denial of service in sensor networks. *IEEE Computer 35*, 10 (Oct. 2002), 54–62.