Hardware Support for Secure Processing in Embedded Systems

Shufu Mao and Tilman Wolf, *Senior Member*, *IEEE*

Abstract—The inherent limitations of embedded systems make them particularly vulnerable to attacks. We have developed a hardware monitor that operates in parallel to an 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 quantify trade-offs 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.

Index Terms—Embedded system security, processing monitor, security enforcement.

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 several 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 these 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). 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 consider the following important criteria:

- 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 meets these design goals. The main idea behind our design is to analyze the binary code of an embedded system application and derive an augmented control flow graph. During runtime, 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 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 instruction 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: cellphones, 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 [1]. Wood and Stankovic consider a networked scenario where systems are exposed to additional remote attacks [2]. Embedded systems are also susceptible to side-channel attacks (e.g., differential power analysis [3]). 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.

- *The authors are with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003-9284. E-mail: {wolf, smao}@ecs.umass.edu.*

Manuscript received 23 Feb. 2008; revised 14 Nov. 2008; accepted 11 June 2009; published online 29 Jan. 2010.
Recommended for acceptance by S. Fahmy.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-02-0086.
Digital Object Identifier no. 10.1109/TC.2010.32.

have proposed one such in [5]. 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, the system by Arora et al. [6] and the IMPRES system [7] are 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 (in [6]) or per-block encrypted checksum (in [7]), and deviations in the program execution are detected when the hash value or checksum does not match at the end of a basic block. In our work, deviations from the binary can be determined within a single (or a few) instructions. In addition, 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.

The SAFE-OPS system by Zambreno et al. [8] uses information that is collected across multiple executed instructions to determine valid operation. This system can detect errors and attacks at the end of such a sequence, whereas our system may immediately detect the first instruction that deviates.

Abadi et al. [9] also use a control flow graph for monitoring program execution. Nakka et al. [10] introduce integrity checks into the microarchitecture and use special check instructions. Ragel et al. [11] introduce microinstruction to monitor for fault detection, return address checks, and memory boundary checks. The main difference to our work is that these approaches require changes in the machine code to implement the necessary checks, while, in our work, binaries do not need to be modified. 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) [12]. This system requires a much more complex design that needs to be integrated with the processor.

A completely different approach to ensuring secure execution of programs is to identify noninstruction 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. Another approach to defending against buffer overflow attacks is described by Shao et al. in [13], where bound checks are used and function pointers are protected by XORing them with a secret key.

Anomaly and intrusion detection by comparing behavior against a model is also used in other domains (e.g., mobile ad hoc networks [14]). In our case, we have a simpler problem since our model is derived from the actual binary of the application. Thus, there is no guesswork 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 use a monitoring system that verifies that the processor indeed performs the operations that it was intended to. 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.

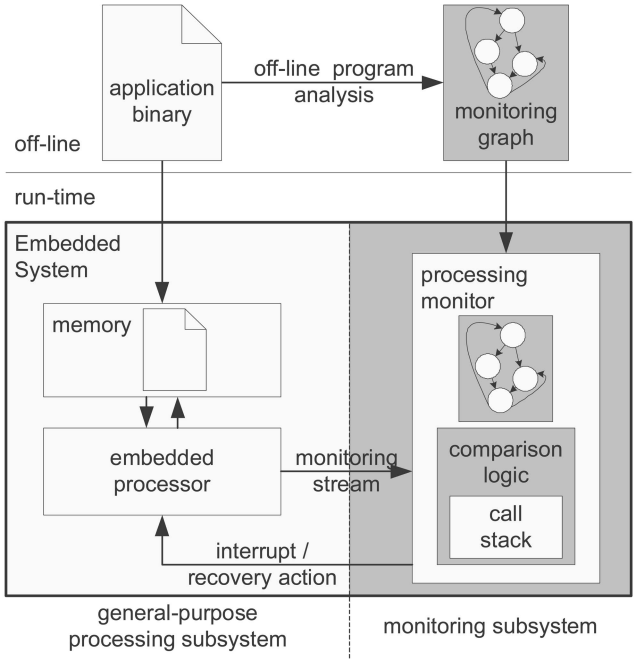

Fig. 1. System architecture for secure embedded processing.

3.1 System Architecture

To detect an attack, we employ the system architecture shown in Fig. 1. The system 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 offline 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 block is also maintained. 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 offline process, where the binary of the application is simulated and analyzed. The simulation is necessary to resolve those branch targets that cannot be determined statically. We assume the simplicity of applications used in embedded systems permits a resolution of all branch targets through simulation. This process only requires the binary and not the source code of the application.

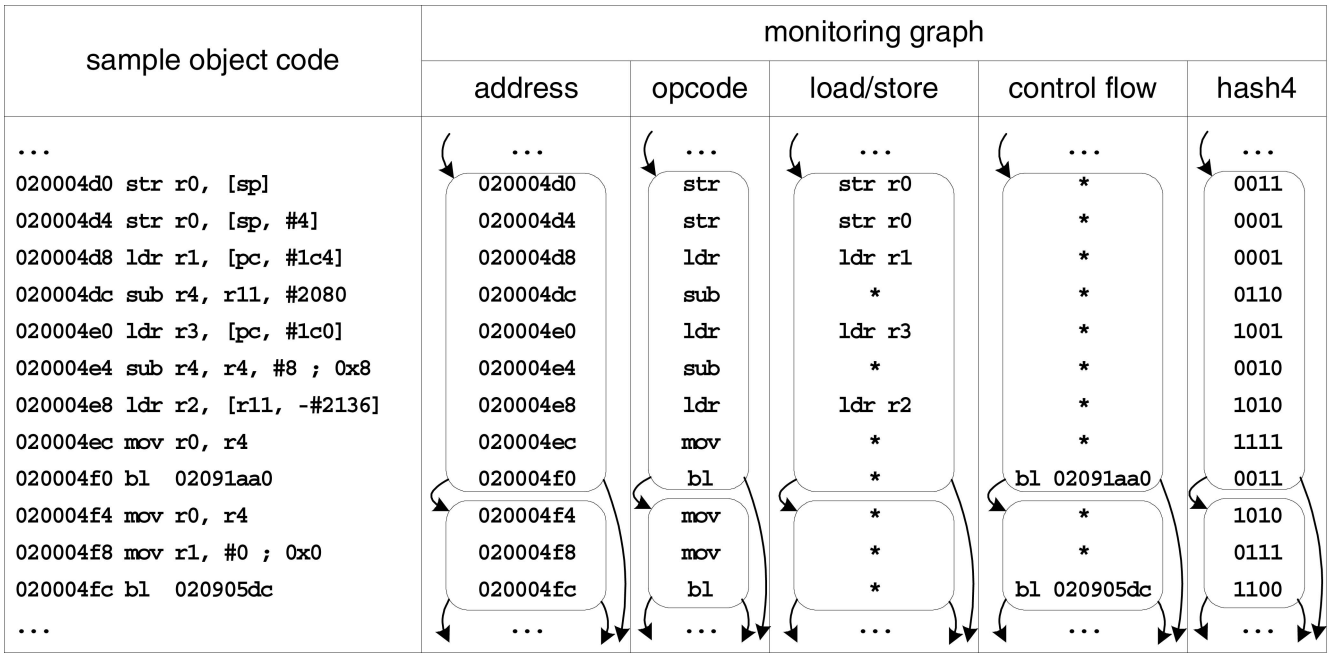

Fig. 2. Examples of monitoring graphs for different information streams.

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. In addition, the offline analysis and the monitoring graph representation discussed in Fig. 1 need to change with different patterns. All of the information required for each pattern can be obtained from the application binary. Fig. 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 to 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 of instructions to monitor processing is the load/store pattern. In this pattern, only load and store instructions and their target registers 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 it allows for dynamic allocation of data structures in memory.
- **Control Flow Pattern:** Another intuitive pattern is the control flow pattern. In this pattern, all control flow operations are stored (e.g., branches, calls, and 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 [6]. 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 use the function name hash$n$ to indicate that an n-bit hash function is used. An attacker would need to generate malicious code that consists of the same sequence of hash values as the original code, which is difficult—even for small values of n. Note that the inclusion of instruction address in the hash limits the program to a fixed location. It is also possible to use an address offset to allow the program to be moved within memory (at the cost of opening a potential attack that changes the base address).

3.3 Monitoring Graph and Comparison Logic

Given the monitoring graph that matches one of the patterns from above and is installed securely on the embedded system at the same time as the application binary, 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 two or more 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 target (e.g., an `add` instruction) or continues with the following instruction (e.g., a `sub` instruction). After validating the branch, the opcode monitors allow 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 (e.g., if the initial sequence of opcodes is the same for both paths). 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. We quantify this ambiguity in Section 4.

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 different sequence calls and returns, these call stacks need to be maintained independently for each monitoring state.

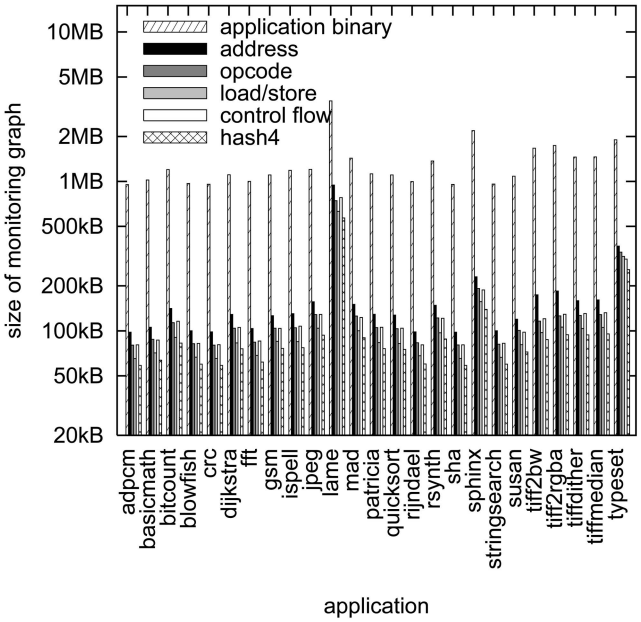

Fig. 3. Size of monitoring graph for different benchmarks and information streams.

4 Results

We present 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 [15] to generate realistic workloads. This suite encompasses over two dozen applications from six different application domains (automotive/industrial, consumer, office, network, security, and telecom). These application domains match very well with the embedded system's complexity that our research targets, and thus, can be considered a representative workload. We employ the SimpleScalar simulator [16] 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.

We pay particular attention to comparing our hash4 monitor to the control flow monitor. The latter is practically identical to the monitoring approach proposed by Arora et al. [6], which is the current state of the art in monitoring in embedded systems.

The first important result is that our implementation of the monitoring system performs application monitoring correctly for all applications. 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.

4.1 Monitoring Graph Size

The size of the monitoring graph that was generated from the binary of each application is shown in Fig. 3. Each pattern is represented in 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

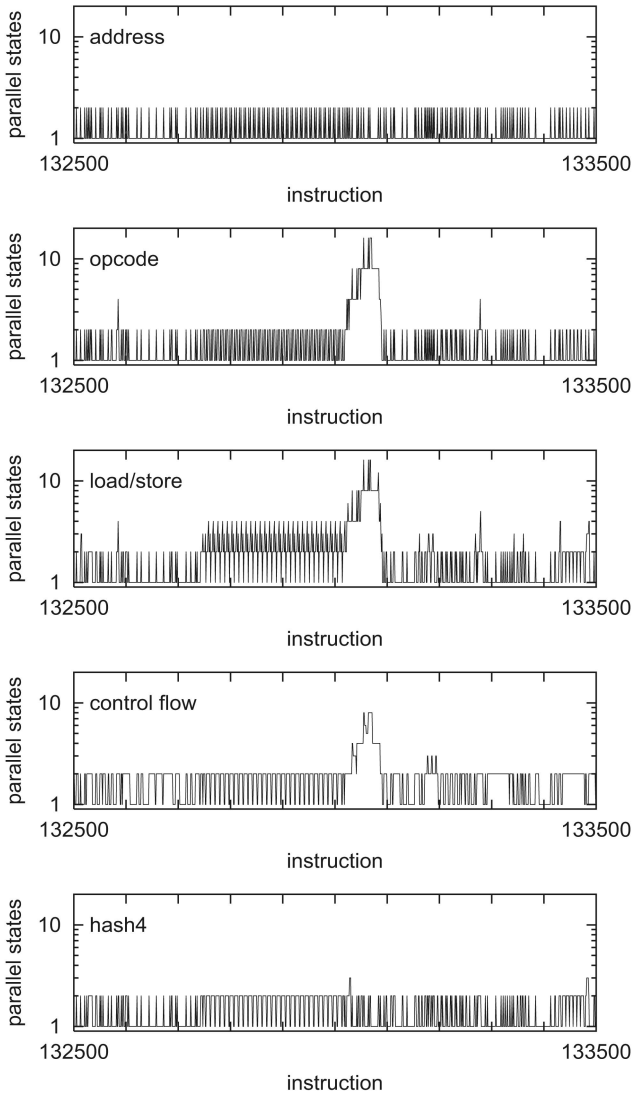

Fig. 4. Snapshot of monitoring of 1,000 instructions in *patricia* application.

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.

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

When comparing the monitoring graph sizes of different monitoring patterns, address is consistently the largest, while hash4 is the smallest. The difference is approximately a factor of 2 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 Fig. 4. In most cases, applications alternate between one and two 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 spawns 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 Fig. 5. The closer the value 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.

The average number of parallel states is only one way of aggregating information about ambiguity. Another way is to consider the length of time that the monitor is in an ambiguous state (measured in number of instructions). This measure is independent of how many ambiguous states are encountered. Instead, it considers how long it takes for the monitor to get back to being certain about the correctness of processing (i.e., being in a single state). Fig. 6 shows the cumulative density function of all applications and all patterns. The benchmarks are ordered by decreasing percentile of ambiguous path length 1 of

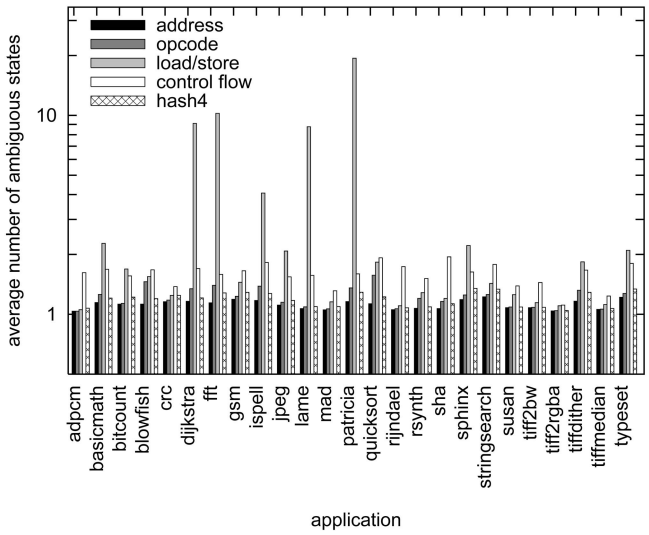

Fig. 5. Average number of ambiguous states for benchmark applications.

hash4 monitoring (i.e., the benchmark where hash4 has most frequently an ambiguous path length of 1 is listed first).

The address pattern achieves the overall shortest durations of ambiguity. In several cases (e.g., *blowfish*, *fft*, *ispell*, and *quicksort*), it is closely followed by the hash4 pattern. The opcode and load/store patterns show similar trends. It is important to note that the average number of ambiguous states in Fig. 5 takes into account how many parallel state occur. In Fig. 6, we show only the duration of the ambiguity (no matter how many parallel states there are). In the control flow pattern, this difference can be observed. Fig. 6 shows that the duration of ambiguity for control flow can be very long (e.g., *sha*, *typeset*, and *rijndael*). This is not visible in Fig. 5 because the ambiguity may only consist of two parallel states. Nevertheless, it is quite apparent that using solely control flow information to monitor the correct execution of a program shows less than desirable performance.

We further compare the control flow monitor and our proposed hash4 monitor in Fig. 7. This figure shows a comparison of the size of the monitoring graph with the 95 percentile of ambiguous path length. The 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. The lines in the graph connect the corresponding data points for all benchmarks. It can be observed that the hash4 monitor is strictly better (i.e., lower monitoring graph size and lower ambiguous path length) for all benchmarks. Again, this indicates that our proposed monitoring approach, which uses a hash pattern to report processor information, is a suitable approach toward ensuring secure processing on an embedded system.

4.3 Evaluation with Attacks on Instruction and Data Memory

To put the above results in context, we show the monitoring performance of our system in scenarios where instruction and data memory are actually modified. It would be very difficult to craft actual exploits for MiBench application (assuming that there even exist vulnerabilities). Instead, we introduce synthetic attacks that target the application binary and the application memory. While these experimental attacks do not lead to useful exploits, they still represent the changes that would occur during a real attack. Thus, it is important to evaluate how quickly a monitoring system can detect such changes.

4.3.1 Attacks on Application Binary

To create an attack on the application binary, we introduce random bit flips into the executable. 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 detected attacks are noticed in Table 1. We find that the hashed pattern has the lowest percentage of undetected bit flips and the fastest possible detection speed. Other patterns cannot 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).

In Table 1, we also compare our results to the performance of control-flow-based monitors as they have been proposed by Arora et al. [6]. 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 six instructions for `gsm` [15]). 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 [6] can detect the same number of attacks as our hash32 pattern, but requires six times as much time to respond.

4.3.2 Attacks on Application Memory

Attacks on application memory often occur when systems are attacked remotely via the network and the attacker does not have access to the application binary. Typical examples are buffer overflow attacks [17], where data structures overwrite portions of the stack. When a return address in the stack is overwritten, the attacker can change the control flow of the program.

To create an attack on application memory, we use one instance of a buffer overflow attack. During runtime, we overwrite a 10-word stack data structure with 20 words of malicious branch instructions, thus causing the return pointer of a function call to be overwritten. When the function returns, control flow is changed to the target of our malicious branch.

Table 2 shows the performance of our monitors when detecting buffer overflow attacks. All monitors detect the stack attack. Both address pattern and hashed pattern can detect the malicious branch in just one instruction. This is expected since the incorrect branch leads to a different address, which affects the address and hash pattern. (Note that for different stack attack scenarios, the hash pattern may not detect the branch on the first instruction with 6.25 percent probability, as explained above.) For opcode and load/store patterns, the attack is not detected until several instructions are executed (in our scenario, two instructions). For

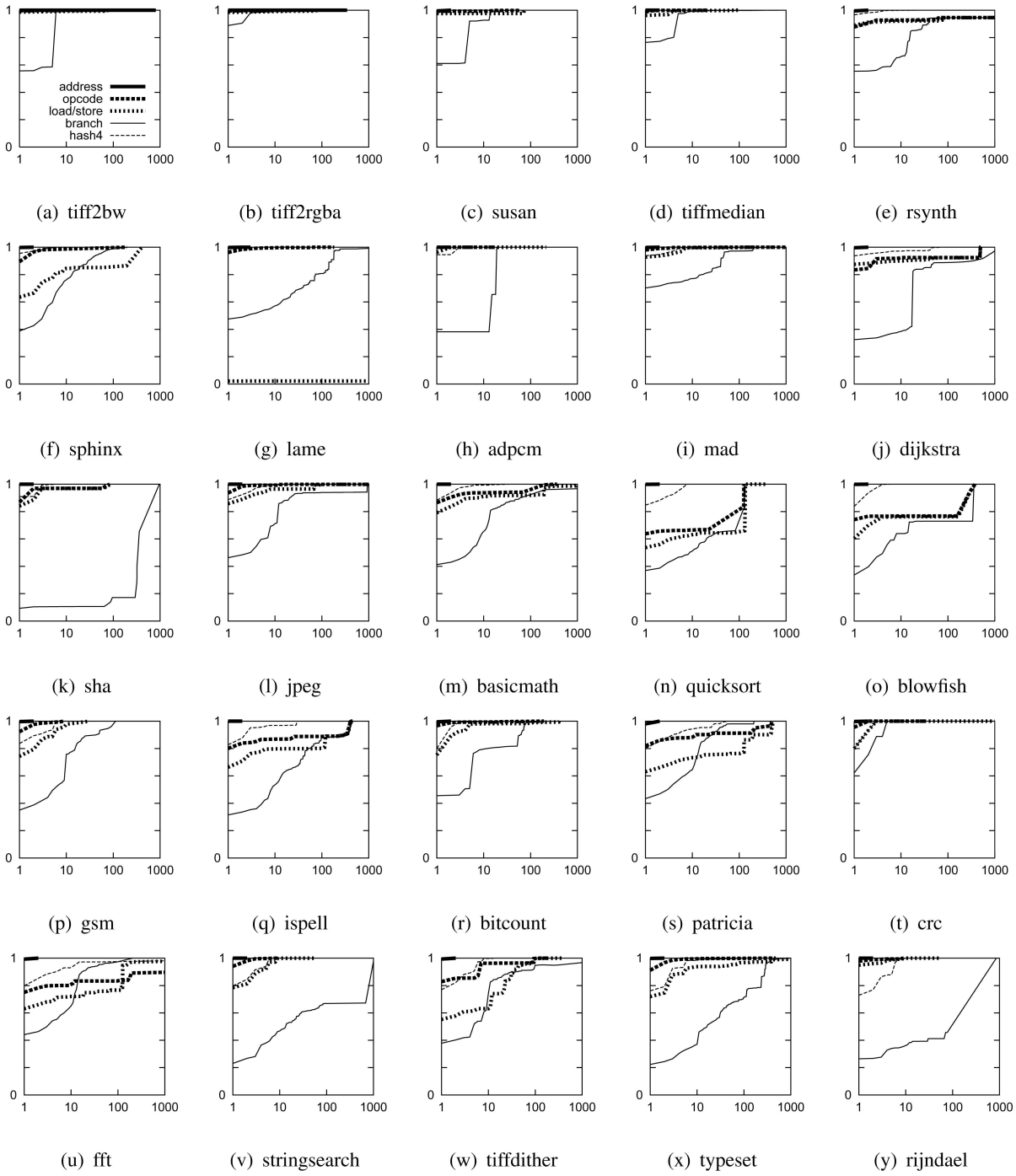

Fig. 6. Cumulative density function of ambiguous execution path length for benchmark applications. The x-axis shows the length of ambiguous paths and the y-axis shows the cumulative density function. The key in the upper right corner is applicable to all figures.

the control flow pattern, the attack can only be detected after the basic block ends (i.e., 10 instructions in our scenario). The monitor in [6] detects this attack within one instruction by maintaining valid branch targets.

The ability of our hash4 monitor to detect attacks within the execution time of a single instruction is an important distinction. Embedded system 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. Therefore, we conclude that this approach outperforms the control flow monitor (which is similar to the monitor proposed in Arora et al. [6]) by requiring less memory (see Figs. 3 and 7) and detecting some attacks faster (see Table 1) and some attacks equally fast (see Table 2).

5 Summary

In this paper, we have presented an 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.

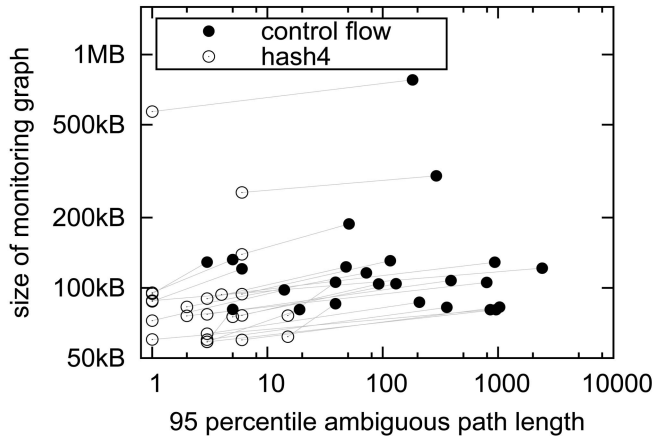

Fig. 7. Monitoring graph size compared to 95 percentile ambiguous path length. All 25 benchmark applications are plotted for each monitoring pattern.

TABLE 1
Performance of Monitor to Detect Bit Flip Attacks on Application Binary

Monitoring pattern	undetected bit flips out of 100 runs	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	0 ($1.5 \cdot 10^{-3}$%†)	1†
hash32	0 ($2.3 \cdot 10^{-8}$%†)	1†
Arora et al. [6]	0 ($2.3 \cdot 10^{-8}$%†)	approx. 6†

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

The results are based on 100 simulations using the `gsm` application.

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 toward providing hardware-based security solutions in embedded systems that address the inherent limitations of these architectures.

ACKNOWLEDGMENTS

This material is based upon work supported by the US National Science Foundation under Grant No. CNS-0447873.

TABLE 2
Performance of Monitor to Detect Buffer Overflow Attacks on Application Memory

Monitoring pattern	stack attack detection	no. of instr. to detection
address	detected	1
opcode	detected	2
load/store	detected	2
control flow	detected	10
hash4	detected	1
Arora et al. [6]	detected	1

REFERENCES

[1] S. Ravi, A. Raghunathan, and S. Chakradhar, "Tamper Resistance Mechanisms for Secure, Embedded Systems," *Proc. 17th Int'l Conf. Very Large Scale Integration Design (VLSI Design '04)*, pp. 605-611, Jan. 2004.

[2] A. Wood and J.A. Stankovic, "Denial of Service in Sensor Networks," *Computer*, vol. 35, no. 10, pp. 54-62, Oct. 2002.

[3] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," *Proc. 19th Ann. Int'l Cryptology Conf. Advances in Cryptology (CRYPTO '99)*, pp. 388-397, 1999.

[4] S. Chari, C.S. Jutla, J.R. Rao, and P. Rohatgi, "Towards Sound Approaches to Counteract Power-Analysis Attacks," *Proc. 19th Ann. Int'l Cryptology Conf. Advances in Cryptology (CRYPTO '99)*, pp. 398-412, 1999.

[5] G. Gogniat, T. Wolf, W. Burleson, J.-P. Diguet, L. Bossuet, and R. Vaslin, "Reconfigurable Hardware for High-Security/High-Performance Embedded Systems: The SAFES Perspective," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 2, pp. 144-155, Feb. 2008.

[6] D. Arora, S. Ravi, A. Raghunathan, and N.K. Jha, "Secure Embedded Processing through Hardware-Assisted Run-Time Monitoring," *Proc. Design, Automation, and Test in Europe Conference and Exhibition (DATE '05)*, pp. 178-183, Mar. 2005.

[7] R.G. Ragel and S. Parameswaran, "IMPRES: Integrated Monitoring for Processor Reliability and Security," *Proc. 43rd Ann. Conf. Design Automation (DAC)*, pp. 502-505, July 2006.

[8] J. Zambreno, A. Choudhary, R. Simha, B. Narahari, and N. Memon, "SAFE-OPS: An Approach to Embedded Software Security," *ACM Trans. Embedded Computing Systems*, vol. 4, no. 1, pp. 189-210, Feb. 2005.

[9] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-Flow Integrity Principles, Implementations, and Applications," *Proc. ACM Conf. Computer and Comm. Security (CCS)*, pp. 340-353, Nov. 2005.

[10] N. Nakka, Z. Kalbarczyk, R.K. Iyer, and J. Xu, "An Architectural Framework for Providing Reliability and Security Support," *Proc. 2004 Int'l Conf. Dependable Systems and Networks (DSN)*, pp. 585-594, June 2004.

[11] R.G. Ragel, S. Parameswaran, and S.M. Kia, "Micro Embedded Monitoring for Security in Application Specific Instruction-Set Processors," *Proc. 2005 Int'l Conf. Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, pp. 304-314, Sept. 2005.

[12] G.E. Suh, J.W. Lee, D. Zhang, and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking," *Proc. 11th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, pp. 85-96, Oct. 2004.

[13] Z. Shao, Q. Zhuge, Y. He, and E.H.-M. Sha, "Defending Embedded Systems Against Buffer Overflow via Hardware/Software," *Proc. 19th Ann. Computer Security Applications Conf. (ACSAC)*, pp. 352-363, Dec. 2003.

[14] G.F. Cretu, J.J. Parekh, K. Wang, and S.J. Stolfo, "Intrusion and Anomaly Detection Model Exchange for Mobile Ad-Hoc Networks," *Proc. Third IEEE Conf. Consumer Comm. and Networking (CCNC '06)*, pp. 635-639, Jan. 2006.

[15] 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," *Proc. IEEE Fourth Ann. Workshop Workload Characterization*, Dec. 2001.

[16] D. Burger and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0," Dept. of Computer Science, Univ. of Wisconsin in Madison, Technical Report 1342, June 1997.

[17] K.-S. Lhee and S.J. Chapin, "Buffer Overflow and Format String Overflow Vulnerabilities," *Software: Practice and Experience*, vol. 33, no. 5, pp. 423-460, Apr. 2003.