Secure Embedded Processing through Hardware-assisted Run-time Monitoring

Divya Arora†, Srivaths Ravi‡, Anand Raghunathan‡ and Niraj K. Jha†
†Dept. of Electrical Engineering, Princeton University, Princeton, NJ 08544
‡NEC Laboratories America, Princeton, NJ 08540
†{divya, jha}@princeton.edu ‡{sravi, anand}@nec-labs.com

***Abstract*— Security is emerging as an important concern in embedded system design. The security of embedded systems is often compromised due to vulnerabilities in "trusted" software that they execute. Security attacks exploit these vulnerabilities to trigger unintended program behavior, such as the leakage of sensitive data or the execution of malicious code.**

In this work, we present a hardware-assisted paradigm to enhance embedded system security by detecting and preventing unintended program behavior. Specifically, we extract properties of an embedded program through static program analysis, and use them as the bases for enforcing permissible program behavior in real-time as the program executes. We present an architecture for hardware-assisted run-time monitoring, wherein the embedded processor is augmented with a hardware monitor that observes the processor's dynamic execution trace, checks whether the execution trace falls within the allowed program behavior, and flags any deviations from the expected behavior to trigger appropriate response mechanisms. We present properties that can be used to capture permissible program behavior at different levels of granularity within a program, namely inter-procedural control flow, intra-procedural control flow, and instruction stream integrity. We also present a systematic methodology to design application-specific hardware monitors for any given embedded program. We have evaluated the hardware requirements and performance of the proposed architecture for several embedded software benchmarks. Hardware implementations using a commercial design flow, and architectural simulations using the SimpleScalar framework, indicate that the proposed technique can thwart several common software and physical attacks, facilitating secure program execution with minimal overheads.

I. Introduction

As embedded systems pervade various aspects of our lives, they are often required to deal with sensitive information or perform critical functions, making security an important concern in embedded system design. Security has been the subject of extensive research in the context of general-purpose computing and communications systems, leading to many advances such as cryptographic algorithms, security protocols, *etc.* [1], [2]. While such "functional" security measures provide a strong basis for securing embedded systems, recent trends in security attacks have made it abundantly clear that most attacks target weaknesses in a system's implementation. It is now well accepted that a secure system implementation is as critical to a system's overall security as the strength of the theoretical security measures employed. Consequently, recent years have seen an increasing awareness that security needs to be considered at various stages of the embedded system design process, including system architecture and hardware/software implementation.

A notable trend in embedded systems has been a drastic increase in embedded software content in order to support increasing end-user functionality and performance requirements (on an average, embedded software content has been increasing by 140% per year — even faster than Moore's law). It is therefore not surprising that the most common security attacks on embedded systems are software-based, or exploit weaknesses in embedded software. Together with increasing complexity, features such as network connectivity and extensibility (ability to extend installed software or download new software onto the embedded system) have only increased the vulnerabilities to software attacks.

In this work, we address the problem of secure software execution in embedded systems. Software security can be compromised in a variety of ways, *e.g.*, through the execution of programs that originate from untrusted or unknown sources, or through the corruption of binaries while they are being downloaded or stored on the embedded system. However, a recurring theme among many recent software security attacks is that they exploit weaknesses in "trusted" code (operating system (OS), middleware, applications) that is already present in the system. For example, 66% of the vulnerabilities reported by CERT are based on exploiting buffer overflow behavior in trusted programs [3]. Such attacks are especially dangerous when they are used to subvert programs that have special privileges, *e.g.*, access to sensitive data or system resources. In this work, we propose a hardware/software solution to address the above problem based on the paradigm of hardware-assisted run-time monitoring.

A. Paper Overview and Contributions

In this paper, we address secure program execution by focusing on the specific problem of ensuring that the program does not deviate from its intended or "permissible" behavior. We address this problem by using the notion of a dedicated hardware monitor that enforces permissible behavior as the program executes. We describe the architecture of a hardware monitor that can be connected to any embedded processor to observe its dynamic execution trace as it executes the program, check whether the trace conforms to the definition of permissible behavior, and flag violations by triggering appropriate response mechanisms.

We propose to define permissible behavior by identifying suitable program properties or invariants that are indicators of untampered execution (*i.e.*, will be violated if a software attack occurs). We identify properties that capture both coarse-grained and fine-grained program behavior in a hierarchical manner, including (i) the inter-procedural control flow of a program, as represented by its function call graph, (ii) the intra-procedural control flow for each function, represented by a basic-block control-flow graph (CFG), and (iii) the integrity of the instruction stream within each basic block.

We propose techniques to extract the above properties from any given program, and automatically synthesize a hardware monitor for it. The hardware monitor can be implemented as a programmable unit that can be configured for each application that executes on the embedded system.

We have evaluated the area and delay overheads associated with the proposed architecture using several embedded software benchmarks. Hardware implementations using a commercial design flow, and architectural simulations using the SimpleScalar framework, demonstrate that the proposed hardware-assisted monitoring technique can eliminate a wide range of common software and physical attacks, facilitating secure program execution with minimal overheads.

The remainder of this paper is organized as follows. We present a survey of relevant past work in Section II. Section III describes a typical software attack and motivates the need for the proposed technique. Section IV presents the details of the proposed hardware-assisted monitoring architecture. Section V describes a systematic methodology to design hardware monitors for any given application. We present our experimental methodology and results in Section VI and conclude in Section VII.

II. Related Work

A wide range of techniques has been proposed to enhance software security in the context of general-purpose computing systems. Most of them address problems such as verifying the identity of the provider of a program, checking the integrity of program binaries, ensuring isolation between different programs running on a system, *etc.* These

Acknowledgments: This work was supported by NSF under Grant No. CCR-0326372.

techniques are complementary to our work, which focuses on eliminating unintended behavior in trusted programs that have already been authenticated, but may potentially contain vulnerabilities that can be exploited for attacks. We examine research related to secure program execution in three categories – static checking, software-based program monitoring, and processor architectures for secure execution.

Static techniques include source code scan tools and code review tools that attempt to strengthen security by eliminating vulnerabilities during the software design phase [4], [5]. Studies have been performed to model vulnerabilities that open doors for security exploits [6]. Although these techniques are useful, the protection they offer is not complete, since they are typically rule-based and attempt to match the target program with specific known classes of vulnerabilities.

Software-based monitoring has also been explored along various dimensions. In [7], the authors provide formal specifications of security-critical programs and check for access rights violations through an analysis of their execution traces. Efficient methods for automatically embedding software security checks in a program, in order to perform security-related actions (*e.g.*, checksumming and repair) during execution are explored in [8]–[10]. In these studies, the focus is on the efficiency and stealth of the security checks. Program shepherding [11] proposes a software-based run-time environment that monitors control flow during program execution, and enforces user-specified security policies, such as restricted code origins and control transfers. The main drawback of software schemes is that the security checks are also pieces of code which are themselves vulnerable to corruption. Also, the granularity of checking is limited due to the overhead imposed by the additional code.

The basic concept of using a hardware unit or co-processor to facilitate secure execution has roots dating back to tamper-resistant cryptoprocessors that were used to store cryptographic keys and execute cryptographic algorithms [12], [13]. However, a significant difference in our work is that the monitor does not execute any program itself — it only ensures that the program running on the host processor does not display unintended behavior. The application of a secure co-processor to perform intrusion detection (to monitor critical OS data structures, check the integrity of files on disk, perform virus scanning, *etc.*) was proposed in [14]. Recently, enhanced processor architectures, such as XOM and AEGIS, have been proposed [15], [16], which attempt to provide code integrity and privacy in the presence of untrusted memory. The authors of XOM use the ideas of eXecute-Only-Memory (allowing instructions to be executed but not modified), ciphered-code execution (decrypting code on-the-fly), and tagged data (associating a process identifier tag with all architectural data) to achieve these goals. However, these techniques do not safeguard an application from its own vulnerabilities. In [17], the authors propose a hardware-supported scheme to track the use of input data that are captured from untrusted input channels, and ensure that such data are never used to affect program control flow. In addition to the above general techniques, a number of *attack-specific* mechanisms have been developed [18]–[20], mostly in response to the increasing number of exploits involving buffer overflows and format string vulnerabilities.

Our work can be differentiated from previous work along several dimensions, including the overall approach to enforcing security, as well as the hardware/software architecture used to implement it:

- Our approach does not require users to explicitly specify security policies — instead, properties indicative of permissible program behavior are extracted from the application itself. The extracted properties are application-specific, yet our approach has wide applicability since the process of deriving the properties can be incorporated into a compiler tool flow.
- Unlike the purely static or purely dynamic approaches described above, we partition the burden of ensuring security between static analysis and dynamic (run-time) monitoring and enforcement.
- The use of hardware-assisted monitoring enables the checking of fine-grained, application-specific properties, resulting in higher levels of security and lower detection latencies, while incurring minimal delay overheads.
- The proposed hardware-assisted monitoring architecture does not require any changes to the embedded processor, and hence can be directly applied to existing embedded systems.

Due to the above factors, we believe that our technique offers a practical approach to counter a broad range of attacks including, for example, buffer overflows, spurious control transfers, and run-time code corruption.

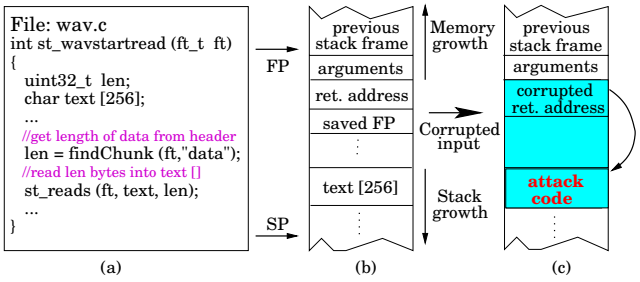


Fig. 1. A simple example of a "stack smashing" security attack

III. Motivation

A large number of security breaches are effected by software exploits that take advantage of weaknesses present in "trusted" code. In this section, we illustrate how a simple vulnerability in a benign application program can be exploited to achieve undesirable side-effects, such as the execution of arbitrary malicious code, motivating the need for solutions such as the one proposed in this paper.

We consider a program that is part of a popular audio format conversion utility, called the SoX (Sound eXchange) toolkit. Fig. 1(a) presents a code snippet from a file that contains functions related to the reading and writing of "wav" files. The function $st\_wavstartread()$ shown in Fig. 1(a) reads len bytes from an input file into a local array $text[\ ]$, where parameter len is read from the file header. Fig. 1(b) shows the layout of the stack frame for function $st\_wavstartread()$, when the function is called during the program's execution. The stack frame contains copies of the function's arguments, the return address in the calling function, as well as storage space for local variables such as the $text[\ ]$ array. In order to execute a successful exploit, an attacker creates an input wav file that contains a payload of malicious code and a large value of len ($len > 256$). When the program is executed with this malicious input file, it causes a buffer overflow for array $text[\ ]$, resulting in corruption of the local variables and the function's return address stored on the program stack. When the function $st\_wavstartread()$ returns, program flow is directed to the corrupted return address. Through appropriate construction of the input file, the corrupted return address can easily be made to point to the start of the malicious code to be executed.

While the vulnerability in the above example was a lack of input validation, vulnerabilities in large, complex programs can be much more subtle and difficult to catch. Numerous variants of similar exploits (return-into-libc, format string attacks, heap overflow) on security-critical programs have been reported in the literature [3], [5]. In addition, embedded systems are also highly susceptible to physical attacks that involve tampering with system properties such as voltage levels, clock frequency, and memory contents. Irrespective of how they originate, most attacks eventually manifest themselves as a subversion of "normal" program execution - violation of control flow behavior, execution of corrupted instruction sequences, *etc.* Instead of trying to block all sources of attack, we concentrate on defining permissible program behavior and monitoring program execution to catch these aberrations. However, the overhead of tracking execution flow at a fine granularity makes a software-based solution infeasible, and presents a compelling case for designing an efficient, hardware-assisted run-time monitor.

IV. Hardware-assisted Monitoring Architecture

In this section, we provide an overview of the proposed hardware-assisted monitoring architecture. We then describe in detail the properties that model permissible program behavior and the design of the corresponding hardware monitors that check them.

A. Architecture Overview

Fig. 2 shows the conceptual block diagram of the proposed hardware-assisted monitoring architecture. For ease of illustration, we depict the embedded processor as an in-order five-stage pipeline[1]. The inputs to the monitor include the program counter (PC) and instruction

[1]The proposed technique is fairly independent of the processor microarchitecture, and can be easily adapted to more complex architectures such as superscalar and VLIW.

register (IR) of the completing instruction, and the pipeline status from the pipeline control unit. Effectively, the monitor is provided with a cycle-by-cycle trace of the executing instructions and their program addresses. The monitor's outputs include a *stall* signal and an *invalid* signal. When the monitor detects a violation of permissible program behavior, it asserts the *invalid* signal, which results in a non-maskable interrupt to the processor. This signal can be used to trigger a response mechanism, such as termination of the program or transfer of the processor to a secure mode. The *stall* signal is asserted if the monitor is unable to keep pace with processor execution (this happens in very rare cases). This is handled as a normal processor stall, and all the pipeline stages are "frozen" until the stall signal is de-asserted. The three sub-blocks within the monitor handle checking of the three different classes of program properties mentioned earlier. The design of these sub-blocks is described in detail later in this section.

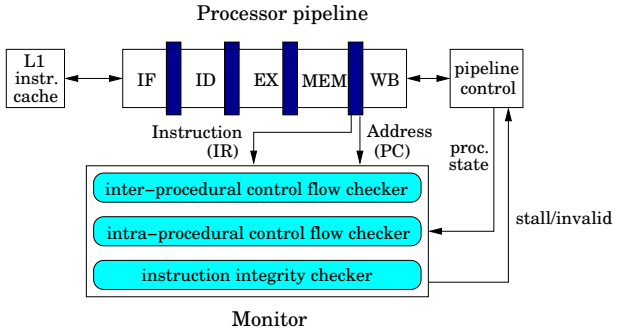


Fig. 2. Proposed hardware-assisted monitoring architecture: Conceptual block diagram

B. Modeling Permissible Program Behavior

There are several factors to be considered when selecting program properties to be monitored. They should be accurate indicators of invalid behavior, *i.e.*, very likely to be violated when system security is compromised. They should also be easily derivable through automatic program analysis for a wide range of programs. They should lend themselves to a concise representation for scalability to large programs. Finally, the hardware overheads involved in checking them at run-time should be reasonable. Based on these considerations, we have chosen three key properties to provide protection at varying granularities, which we describe next.

1) Inter-procedural control flow: At the highest level of granularity, we verify the correctness of a program's inter-procedural control flow. It is common to represent inter-procedural control flow using a *function call graph*. For the purpose of hardware implementation, a function call graph of a program with N functions is translated into a finite-state machine (FSM) with $N+1$ states – one state corresponding to each function in the program, and an additional INVALID state. The functions are associated with a unique index from 1 to N. For convenience, the FSM state corresponding to a function is also labeled with the same index. A transition between two states in the FSM (except the INVALID state) represents a valid control transfer (call or return) between the corresponding functions. Any invalid call or return transitions the FSM to the INVALID state. The input alphabet Z of the FSM consists of two distinct inputs: $Z_0 = \{0, 1\}$, which is set to 0 or 1 depending on whether the executing instruction is a function call or return, and $Z_1 = \{1, 2, \ldots, N\}$, which is the index of the target function of the call/return. Procedure 1 describes how the FSM is automatically extracted from the function call graph.

By default, all user-defined and library functions that are part of the application program are included in the function call graph. Potentially, this can be done for system calls (OS functions) as well. Alternatively, if the kernel is assumed to be secure, system calls can be treated as leaf functions in the function call graph. The extraction of the function call graph should also include indirect function calls with function pointers, by adding an edge from the calling function to each function whose address can be assigned to the function pointer.

2) Intra-procedural control flow: A logical succession to the above scheme is to also track control flow *within* each function in the program. The control flow within a function can be represented using the function's CFG, and translated into an FSM, or equivalently, a basic block information table. Each basic block b_i has a corresponding row

Procedure 1 FSM extraction for inter-procedural control flow checking

1: Inputs: Function Call Graph $G(F, E)$
2: $F \leftarrow$ set of all functions $f_i,\ 1 \le i \le N$
3: $E \leftarrow$ set of directed edges $e_{ij} \in E \iff f_i$ includes a direct or indirect call to f_j
4: Output: FSM (Z, S, T, s, A), where Z: input alphabet, S: set of states, T: $S \times Z \rightarrow S$: transition function, s: initial state, A: accept states
5: $s \leftarrow index(main)$
6: **for all** $f_i \in F$ **do**
7: Add state i to S and A
8: **for all** $f_j : e_{ij} \in E$ **do**
9: Add the following transitions to T:
10: $Z_0 = CALL,\ Z_1 = j \mapsto$ Nextstate $[i] = j$
11: $Z_0 = RETURN,\ Z_1 = i \mapsto$ Nextstate $[j] = i$
12: **for all** i : Nextstate $[i]$ = unassigned **do**
13: Nextstate $[i] = N + 1$ /* INVALID STATE*/

in the table, expressed as a tuple $row_i(index, offset, s_0, s_1)$, where $offset$ is the address offset of b_i from the start of the function and s_0, s_1 are indices of its possible successors. Procedure 2 describes how to create the basic block information table. If the code does not contain any indirect jumps, each basic block can have at most two successors listed in fields s_0, s_1 (corresponding to the branch at the end of the basic block being taken or not taken). The procedure shows how a single level of indirect jumps (*e.g.*, switch statements) is handled. Field s_0 in this case carries a special code to denote an indirect jump and s_1 contains the number of possible jump targets, which appear in the rows immediately following the current row. Hence, a jump to the s_1 fields in any of these rows is considered valid.

Procedure 2 Transition table generation for intra-procedural control flow checking

1: Inputs: Control Flow Graph $G(B, E)$
2: $B \leftarrow$ set of all basic blocks $b_i,\ 1 \le i \le n$
3: $E \leftarrow$ set of directed edges $e_{ij} \in E \iff b_j$ is a successor of b_i
4: Output: $T = \{row_0, row_1, \ldots, row_n\}$ where row_i is a tuple $(index, offset, s_0, s_1)$
5: **for all** $b_i \in B$ **do**
6: $row_i.index = i$
7: row_i.offset = address (b_i) − function start address
8: **if** b_i ends with indirect jump **then**
9: Targets $\leftarrow b_j : e_{ij} \in E$
10: $row_i.s_0 =$ Special code
11: $row_i.s_1 = \|Targets\|$
12: Process all $b_j : b_j \in$ Targets
13: **else if** b_i ends with a direct conditional jump **then**
14: $row_i.s_0 = j : b_j$ is the branch-not-taken target
15: $row_i.s_1 = k : b_k$ is the branch-taken target
16: **else**
17: $row_i.s_0 = NULL$ /*unconditional jump*/
18: $row_i.s_1 = k : b_k$ is the jump target

Nested switch statements are handled by directing the compiler to compile them as a sequence of *if-then-else* statements. To facilitate easy detection of end of basic blocks in hardware, the procedure requires that all basic blocks end with an explicit control transfer instruction. Hence, extra jump instructions (to the next basic block) are appended to fall-through blocks. Our experiments indicate that this does not lead to significant overheads in terms of code size increase ($\le 1\%$). In our investigations, intra-procedural control flow checks are performed for basic blocks in all user-functions. In practice, these fine-grained checks may be restricted to only security-sensitive parts of the application.

3) Integrity of the executed instruction stream: Some security attacks may not result in a control flow violation, *e.g.*, alteration of a basic block in the program code segment during execution. In order to detect such attacks, we explore a complementary approach by checking the integrity of the dynamic instruction stream with the aid of cryptographic hash functions. Given a message x and its cryptographic hash $H(x)$, it is computationally infeasible to find another message y such that $y \neq x$ and $H(y) = H(x)$. Hence, given a sequence of instructions, it is extremely hard for an adversary to find another sequence of instructions such that both hash to the same value. Hash

values of each basic block in the program are computed beforehand, loaded into the hardware monitor when the application is loaded for execution, and subsequently checked during program execution. Hashing is computationally intensive and requires dedicated high-speed hardware to keep up with the processor pipeline. Moreover, most well-known hash algorithms map a variable-sized input to a fixed-sized output (16-20 Bytes). To keep the monitor's hardware requirements low, a user-specified number of bits of the pre-computed hash value are randomly selected at program start-up and loaded into the monitor for checking. It is worth noting that the proposed technique is complementary to static hash checking, since it detects program corruption during execution.

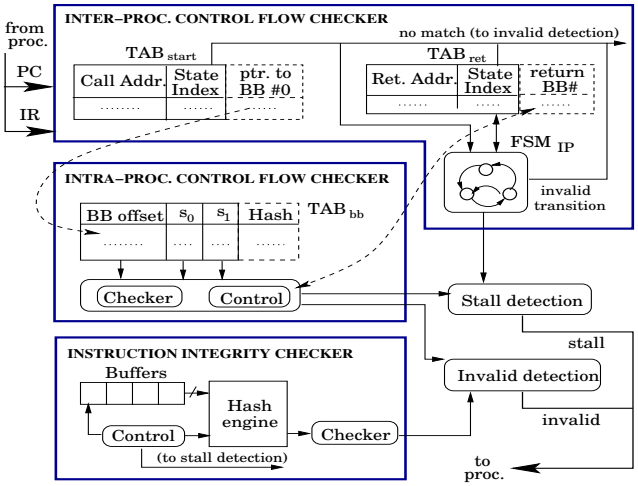


Fig. 3. Architectural details of the run-time monitor

C. Architectural Details

Fig. 3 shows the detailed architecture of the monitor. The monitor is delineated into its three major constituent sub-blocks — the inter-procedural control flow checker, the intra-procedural control flow checker, and the basic block level instruction integrity checker. Although the three checks are logically independent, in the implementation described here, the three sub-blocks share hardware and hence communicate with each other.

1) Inter-procedural control flow checker: The inter-procedural control flow checker consists of two look-up tables, TAB_{start} and TAB_{ret}, that store function start and return addresses, respectively, and an FSM FSM_{IP} that verifies the validity of function calls and returns.

The function start address table, TAB_{start}, takes an instruction address as input. If the instruction address corresponds to the start of a function, it asserts a match signal, and outputs a unique function index. The number of entries in TAB_{start} equals the number of functions in the application program. The contents of this table are generated by enhancing the compiler tool flow and the run-time application loader, as described in Section V.

The return address table, TAB_{ret}, performs a similar mapping from instruction addresses that correspond to return locations, to the index of the function that they are located in (*i.e.*, the calling function). However, there is a notable difference in how its contents are generated. TAB_{ret} is updated dynamically by the monitor hardware as the program executes. At any point in time, it contains an entry corresponding to the return location for each function that has been called but not yet returned. Note that the number of distinct return locations is bounded by the number of function call instances, which is, in general, larger than the number of functions in a program. The return address table is similar to the use of a hardware return address stack on which return addresses are pushed on a call and popped on a return. The only difference is that recursive functions are handled more efficiently — only one entry is maintained for each distinct call site. For example, if the call sequence is $B \rightarrow A \rightarrow A \cdots \rightarrow A \rightarrow B$, the table could contain only two return addresses for function A — one entry for the return address corresponding to the call $B \rightarrow A$, and a second entry for the return address corresponding to the call $A \rightarrow A$.

In hardware, each of the two lookup tables can be directly implemented using a *content addressable memory* (CAM), which is basically a read-writable memory with additional logic circuitry (comparators) to provide fast table look-up.

The function indices output from TAB_{start} and TAB_{ret} are input to the FSM FSM_{IP}, which keeps track of program control flow by executing corresponding state transitions. At each state transition, the FSM checks whether the incoming function index is equal to the index of one of the valid next states. If so, it executes the state transition. The invalid detection circuitry is signaled in case there is no match in address look-up, or if there is an FSM transition into the INVALID state.

2) Intra-procedural control flow checker: The intra-procedural control flow checker consists of a basic block table, TAB_{bb}, which stores the information necessary to track control flow within each function. The basic blocks are grouped according to the function that they belong to. The function start address table, TAB_{start}, which is part of the inter-procedural control flow checker, is augmented with an additional field that is a pointer to a location in TAB_{bb}. This additional field tells the intra-procedural control flow checker where it should start when a function is called. When control enters a function, the function start address is copied into a special register and used to calculate offsets for all future branches within the function. In addition, on each function call, the basic block index within the calling function is also saved in TAB_{ret}, and restored upon return.

3) Instruction integrity checker: In order to check instruction stream integrity, each row in the basic block table, TAB_{bb}, is augmented to contain another field that stores the statically-computed cryptographic hash of the instruction sequence in the basic block. During program execution, the monitor buffers the instruction stream corresponding to a basic block until a branch/jump instruction is encountered. At this point, it switches to an empty buffer and signals the hardware hash unit to compute the hash of the buffered basic block. The computed hash value is then compared against the value stored in TAB_{bb}. When the buffers are full, the processor is stalled in order to allow the instruction integrity checker to catch up.

Many popular hash algorithms, such as MD4, MD5, and SHA-1 [1], incur a high latency, since the input is processed through several rounds wherein the result of the i^{th} round is input to the $i+1^{th}$ round. Data dependencies within a round limit the amount of parallelism that can be achieved by adding more hardware. Moreover, if the input size is greater than 512 bits, the output from the last round is fed back to the first round and included in further computation. We break this feedback loop by imposing a limit l_{max} on the maximum basic block length. Basic blocks that exceed this limit are split into sub-blocks of length $\leq l_{max}$ and an XOR of the hash values of all constituent sub-blocks is checked. This allows us to pipeline the hashing unit at a fine-grained level (each round is a pipeline stage). In practice, this implies that the hardware monitor stalls the processor only in very rare cases, leading to minimal performance impact, as demonstrated in our experimental results.

V. Design Flow

Fig. 4 shows the compilation and execution flow for hardware-assisted run-time monitoring. Given an application and user-specified options, program models, namely the function call graph and CFGs for each function, are extracted during compilation. Depending on the security requirements, the user can specify options to define the granularity of checks (*e.g.*, the depth to which the function call graph is tracked, selected functions for which intra-procedural checking and hash checking are done, *etc.*). Function addresses, basic block offsets, and hashes are computed after linking. The extracted information is translated into the data required to configure the hardware monitor (FSM state table, function start address table, basic block table, hash values), and appended to the program's binary code.

Configuration of the security monitor is performed when the application is loaded for execution. Security technologies such as ARM TrustZone [21] can be used to provide a secure mode during configuration, while the remaining application is run in a non-secure mode. The contents of lookup tables TAB_{start}, TAB_{ret}, and TAB_{bb} are loaded into the monitor, and the FSM in the monitor is configured. There are multiple implementation options for the FSMs in the hardware monitor, including reconfigurable logic (*e.g.,* FPGA cores), or explicit state-table based implementations using memories. After configuration of the monitor, the loader initiates application execution.

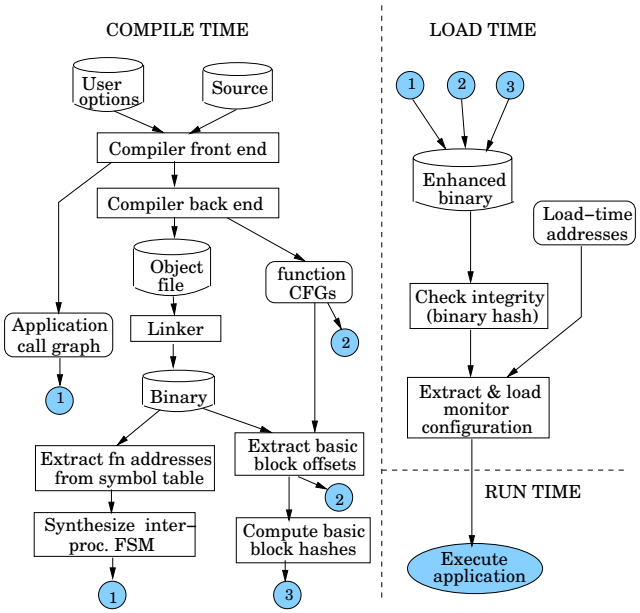


Fig. 4. Design flow for hardware-assisted run-time monitoring

The flow described in this section can be extended to the case when the application uses dynamically-linked libraries. For such applications, addresses of the functions contained in the dynamic libraries are available only when the application is loaded or when the function is invoked ("lazy evaluation"). Addresses of such functions are marked null in the function start address table when the program is compiled. The dynamic loader is responsible for updating these addresses and loading them into the function start address table in the hardware monitor.

VI. Experimental results

In this section, we present the hardware (area) overheads incurred by the proposed hardware-assisted monitoring architecture, as well as the impact of hardware-based monitoring on performance (program execution time). For the purpose of our experiments, we selected applications from the MediaBench [22] and MiBench benchmark [23] suites, which represent typical workloads for embedded processors. The information necessary to generate the hardware monitor was extracted using the GNU compiler `gcc`. We developed tools to post-process this information and generate the contents of the memories in the hardware monitor, as well as Verilog hardware descriptions of the monitor's FSM and control logic.

A. Area Overheads

The key functional blocks in the monitor can be categorized into the FSM and CAMs that are part of the inter-procedural checker, SRAM for the intra-procedural checker, the buffers and hardware hash engine for the instruction integrity checker, and miscellaneous control logic. For estimating the area of hash engines, we implemented Verilog RTL descriptions of two commonly used hash algorithms, MD4 and MD5 [1]. The hash engines and the FSM (for each application) were synthesized into technology-mapped gate-level netlists using Synopsys Design Compiler [24] with NEC's 0.13μ CB-130M CMOS standard cell library. Memory models from [25] were used to estimate the area of the CAM and SRAM. In order to compute area overheads, we used as the base case an ARM920T 32-bit processor core that has separate 16kB instruction and data caches, runs at 250MHz , and occupies an area of $4.7mm^2$ in a 0.13μ technology [26].

Table I reports the area overheads for different benchmarks, for three monitoring scenarios that correspond to increasing levels of security. The second, third, and fourth columns list the number of functions (*#Fns*), the number of function call locations (*#Fn calls*), and the number of basic blocks (*#Basic blocks*) in each benchmark program. The fifth column (labeled *AO% level1*) reports the total percentage area overhead when only the inter-procedural control flow checker is used. The sixth column (*AO% level2*) reports the total percentage area overhead when both inter- and intra-procedural checkers are used, and

TABLE I
Area overheads for hardware-assisted run-time monitoring

Benchmark	#Fns	#Fn calls	#Basic blocks	AO% level1	AO% level2	AO% level3	
						unpipe	pipe
rawdaudio	7	6	27	0.16	0.22	1.46	3.10
epic	71	122	901	0.68	1.65	3.15	4.78
g721decode	24	64	298	0.29	1.83	2.19	3.83
toast (gsm)	137	229	1130	1.20	3.14	4.88	6.52
mpeg2encode	116	223	2114	0.98	5.17	7.43	9.07
pegwit	124	293	1275	1.08	3.06	4.80	6.44
susan	38	64	563	0.41	1.45	2.95	4.59
rasta	111	246	1357	0.96	2.98	4.73	6.36

the last column (*AO% level3*) represents the case when an instruction integrity checker is also employed in addition to control flow checkers. The two minor columns (*unpipe* and *pipe*) under the last column represent the cases when the hash engine within the instruction integrity checker is un-pipelined and pipelined, respectively. The table indicates that for all the benchmarks considered, inter-procedural checking alone can be implemented with a maximum area overhead of 1.20% and an average overhead of 0.72%. Adding intra-procedural checking raises the maximum overhead to 5.17% and average overhead to 2.44%. For the case with the highest hardware requirements (inter- and intra-procedural control flow checking together with pipelined hash checking), the maximum overhead is 9.07% and the average overhead is 5.59%. Clearly, the area overhead is quite low in all cases, when viewed as a percentage of the area of the processor. When employed as part of a system-on-chip that includes an embedded processor and other components, the relative area overheads are likely to be much lower.

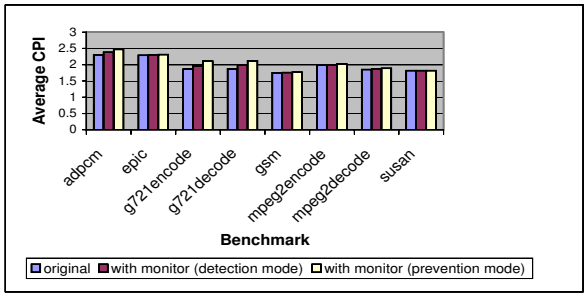
TABLE II
Architectural parameters used in simulations

Parameter	Config.	Parameter	Config.
L1 I-Cache	16kB	L1 D-Cache	16kB
Fetch queue size	8	Issue width	2
Commit width	1	Issue	inorder
Call addr. CAM latency	10 ns	Return addr. CAM latency	5 ns
Intra-procedural lookup latency (On-chip RAM)	5 ns	Intra-procedural lookup latency (Off-chip RAM)	40 ns
Hash engine clock period	4 ns	Processor clock period	10 ns

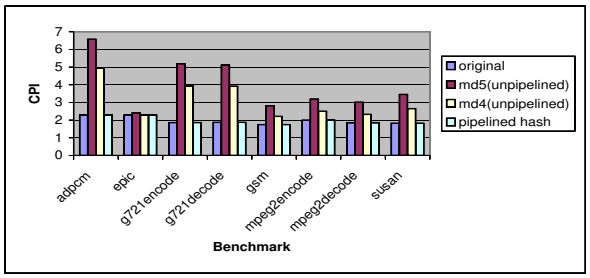
B. Performance Impact

We evaluated the performance impact of the proposed architecture using the SimpleScalar 3.0/PISA architectural simulation toolset [27]. The micro-architectural parameters of SimpleScalar were configured to model a typical embedded processor such as the ARM920T. The parameters used for the simulated processor are shown in Table II. We built our simulator within the framework of the existing cycle-accurate simulator *sim-outorder* and used it to evaluate program execution statistics when the monitor is running in parallel and performing various checks. The clock frequency of hash engines was determined from the hardware implementations, as discussed in the previous subsection. Each benchmark was simulated for five million instructions. We considered two different implementation scenarios for the memories that are part of the hardware monitor — in one case, we assumed that they were implemented as on-chip memories, and in the second case, we assumed that they were implemented off-chip. The main difference between these cases is the access latency.

For inter- and intra-procedural checks, we experimented with two different modes for flagging invalid behavior. In the *detection* mode, the processor is allowed to continue while the monitor is running and is stalled only if a control instruction (call/branch) completes before the previous control instruction has been verified. In the *prevention* mode, no new instruction is allowed to commit after a control instruction until the latter has been checked. For dynamic hashing, the latency of hash computation does not permit stalling the processor for each basic block. Therefore, the processor and hashing units are allowed to run



(a)



(b)

Fig. 5. Performance analysis of (a) intra-procedural control flow checking and (b) dynamic hash checking

in parallel and the former is stalled only if the instruction buffer in the hash unit becomes full.

When on-chip memories were used for the hardware monitor's CAMs and SRAM, performance degradation for inter-procedural checks was found to be negligible ($< 1\%$) for all benchmarks tested, for both detection and prevention modes. This is expected since most applications have sparse call graphs and their FSMs can be synthesized to operate at or above processor speeds. The same is true for intra-procedural checks, assuming an on-chip SRAM is used to store the FSM state table for each function.

When off-chip memories were used, as expected, we observed slightly higher performance penalties. Fig. 5(a) plots the average cycles per instruction (CPI) for all benchmarks, for both the detection and prevention modes. The figure demonstrates that even in this case, the impact is fairly small (average of 1.77% for detection mode and 4.94% for prevention mode).

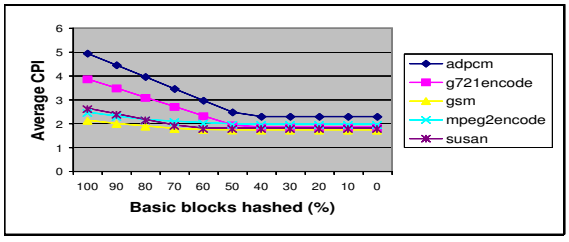


Fig. 6. Performance estimation when the percentage of basic blocks hashed is varied

Fig. 5(b) illustrates the impact of instruction integrity checking on CPI, for different hash algorithms and their implementations. As expected, the performance penalty for an un-pipelined MD5 hash engine, with a latency of 64 clock cycles, is substantial. The performance penalty using an un-pipelined MD4 engine (latency of 48 cycles) is lower, but still considerable. However, with pipelining, the penalty can be virtually eliminated, as shown by the last bar in the graph. In most cases, the original CPI is maintained with the use of pipelined hash engines. With the use of pipelining, there is negligible performance difference between the MD4 and MD5 algorithms.

Fig. 6 shows how the average CPI is affected by varying the number of basic blocks that are hash-checked when an un-pipelined MD4 hash engine is used. In most cases, 50-60% of basic blocks can be hash-checked without any observable decline in performance, even without a pipelined implementation.

In summary, the results demonstrate that the proposed hardware-assisted run-time monitoring technique is viable and imposes minimal area and execution time overheads.

VII. Conclusions

In this paper, we presented a scalable, application-specific methodology to safeguard the execution of programs running on embedded processors. We formulated a hierarchical run-time monitoring framework including program attributes such as inter-procedural control flow, intra-procedural control flow and instruction contents within a basic block, which provides protection at different granularities for applications and systems with diverse security requirements. Run-time monitoring is performed by a configurable hardware monitor to ensure minimal performance degradation. Our studies reveal that the proposed architecture is capable of facilitating secure execution of programs in the face of a wide variety of security threats. We believe that such techniques will be useful in addressing the increasing security concerns in embedded system design.

References

[1] B. Schneier, *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley and Sons, 1996.
[2] W. Stallings, *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 1998.
[3] *Vulnerability Notes Database*. CERT coordination center (`http://www.kb.cert.org/vuls/`).
[4] M. Howard and D. LeBlanc, *Writing Secure Code*. Microsoft Press, 2002.
[5] G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code*. Addison-Wesley, 2004.
[6] S. Chen, Z. Kalbarczyk, J. Xu, and R. K. Iyer, "A data-driven finite state machine model for analyzing security vulnerabilities," in *Proc. Int. Conf. on Dependable Systems & Networks*, June 2003, pp. 605–614.
[7] C. Ko, M. Ruschitzka, and K. Levitt, "Execution monitoring of security-critical programs in distributed systems: A specification-based approach," in *Proc. IEEE Symp. on Security & Privacy*, May 1997, pp. 175–187.
[8] B. Horne, L. R. Matheson, C. Sheehan, and R. E. Tarjan, "Dynamic self-checking techniques for improved tamper resistance," in *Proc. Wkshp. on Security & Privacy in Digital Rights Management*, Nov. 2001, pp. 141–159.
[9] H. Chang and M. J. Atallah, "Protecting software code by guards," in *Revised Papers from the ACM CCS-8 Wkshp. on Security & Privacy in Digital Rights Management*, Nov. 2001, pp. 160–175.
[10] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. Jakubowski, "Oblivious hashing: A stealthy software integrity verification primitive," in *Proc. Int. Wkshp. on Information Hiding*, Oct. 2002, pp. 400–414.
[11] V. Kiriansky, D. Bruening, and S. P. Amarasinghe, "Secure execution via program shepherding," in *Proc. USENIX Security Symp.*, Aug. 2002, pp. 191–206.
[12] M. Kuhn, *The TrustNo 1 Cryptoprocessor Concept*. CS555 Report, Purdue University (`http://www.cl.cam.ac.uk/~mgk25/`), Apr. 1997.
[13] *Secure Coprocessing*. IBM Inc. (`http://www.research.ibm.com/scop/`).
[14] X. Zhang, L. Doorn, T. Jaeger, R. Perez, and R. Sailer, "Secure coprocessor-based intrusion detection," in *Proc. ACM SIGOPS European Wkshp.*, Sept. 2002.
[15] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000, pp. 168–177.
[16] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for tamper-evident and tamper-resistant processing," in *Proc. Int. Conf. on Supercomputing*, June 2003, pp. 160–171.
[17] G. E. Suh, J. Lee, and S. Devadas, "Secure program execution via dynamic information flow tracking," Dept. of EECS, MIT, Tech. Rep., July 2003.
[18] *Non-Executable User Stack*. `http://www.openwall.com/linux`
[19] C. Cowan *et al.*, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. USENIX Security Symp.*, Jan. 1998, pp. 63–77.
[20] J. P. McGregor, D. K. Karig, Z. Shi, and R. B. Lee, "A processor architecture defense against buffer overflow attacks," in *Proc. Int. Conf. on Information Technology: Research and Education*, Aug. 2003, pp. 243–250.
[21] *ARM TrustZone Technology Overview*. `http://www.arm.com/products/CPUs/arch-trustzone.html`.
[22] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *Proc. Int. Symp. on Microarchitecture*, Dec. 1997, pp. 330–335.
[23] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proc. Annual Wkshp. on Workload Characterization*, Dec. 2001, pp. 3–14.
[24] Synopsys. `http://www.synopsys.com`
[25] J. M. Mulder, N. T. Quach, and M. J. Flynn, "An area model for on-chip memories and its application," *IEEE J. Solid-State Circuits*, vol. 26, no. 2, pp. 98–106, Feb. 1991.
[26] *ARM920T Embedded Processor Core*. `http://www.arm.com`
[27] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," Computer Sciences Department, University of Wisconsin, Tech. Rep., June 1997.