Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors

Divya Arora, Srivaths Ravi, *Senior Member, IEEE*, Anand Raghunathan, *Senior Member, IEEE*, and Niraj K. Jha, *Fellow, IEEE*

***Abstract*—Embedded system security is often compromised when "trusted" software is subverted to result in unintended behavior, such as leakage of sensitive data or execution of malicious code. Several countermeasures have been proposed in the literature to counteract these intrusions. A common underlying theme in most of them is to define security policies at the system level in an application-independent manner and check for security violations either statically or at run time. In this paper, we present a methodology that addresses this issue from a different perspective. It defines correct execution as synonymous with the way the program was *intended* to run and employs a dedicated hardware monitor to detect and prevent 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 at run time. The processor architecture is augmented with a hardware monitor that observes the program's dynamic execution trace, checks whether it falls within the allowed program behavior, and flags any deviations from expected behavior to trigger appropriate response mechanisms. We present properties that capture permissible program behavior at different levels of granularity, namely inter-procedural control flow, intra-procedural control flow, and instruction-stream integrity. We outline a systematic methodology to design application-specific hardware monitors for any given embedded program. Hardware implementations using a commercial design flow, and cycle-accurate performance simulations indicate that the proposed technique can thwart several common software and physical attacks, facilitating secure program execution with minimal overheads.**

***Index Terms*—Embedded processors, processor architectures, security and protection, special-purpose and application-based systems.**

I. Introduction

TECHNOLOGICAL innovations in the last few decades have enabled embedded systems to pervade all spheres of our lives. The gamut of applications they target includes everything from commonplace consumer products such as televisions and cell phones, to the most safety critical equipment such as controllers employed in automobiles and nuclear power plants. Also, as they become increasingly widespread and inter-networked, both through wireless means and the Internet, embedded systems open themselves up to security threats and exploits that have traditionally targeted desktop personal computers. Therefore, the question of security in embedded systems is receiving a lot of attention amongst researchers [1].

Security has been the subject of extensive research in the context of general-purpose computing and communication systems, leading to many advances in cryptographic algorithms and security protocols. For an introduction to this, the reader is referred to [2] and [3]. While such "functional" security measures provide a strong basis for securing embedded systems, recent trends 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.

System security can be compromised either 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. Well-known techniques exist for verifying the origin of a program, and for ensuring the integrity of program binaries. However, executing code from a trusted vendor does not guarantee its safe execution. A recurring theme among many security attacks is that "trusted" code is hijacked at run time—so even if the original code is not malicious by intent, it can be manipulated by clever attackers, resulting in malicious behavior.

Security attacks are perpetrated through various channels. Software security exploits take advantage of weaknesses in code [operating system (OS), middleware, applications] that is already present in the system. Amongst these, buffer overflow attacks, which exploit the lack of bounds checking in C/C++ programs, have emerged as one of the most common forms of security violations. Cookbook recipes [4] have furthered their ease of dispatch. An analysis of the advisories published by CERT [5] over a three-month period, January–March 2006, (as shown in Fig. 1) indicates that 23 of the 67 vulnerabilities were pertaining to buffer overflows. In addition to this, many embedded systems are susceptible to physical attacks. Physical attacks, as the name implies, involve tampering with physical properties of the system—voltage levels, clock frequencies, memory contents, etc., and can be used to alter the program executing on a system. Many embedded systems are mobile devices with small form factors that may be passed around in the hands of adversaries for a period of time that is sufficient to launch such attacks. The outcome of both the previous attacks

Manuscript received February 3, 2006; revised March 24, 2006. This work was supported by the National Science Foundation under Grant CCR-0326372.

D. Arora and N. K. Jha are with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA (e-mail: divya@princeton.edu; jha@princeton.edu).

S. Ravi and A. Raghunathan are with the NEC Labs, Princeton, NJ 08540 USA (e-mail: sravi@nec-labs.com; anand@nec-labs.com).

Digital Object Identifier 10.1109/TVLSI.2006.887799

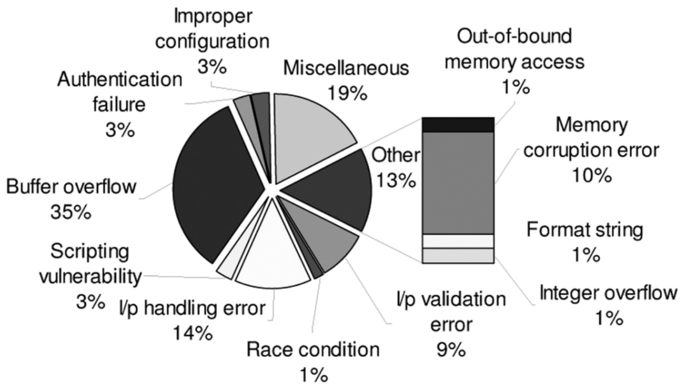


Fig. 1. Breakdown of vulnerabilities reported by CERT for a three-month period.

is especially dangerous when they are used to subvert programs that have special privileges, e.g., access to sensitive data or system resources.

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. A recent report[1] predicted an annual market growth rate of 140% for embedded Linux—even faster than Moore's law. With increasing software complexity and shorter times-to-market, it is not surprising that many software bugs and vulnerabilities go undetected during the design phase. Additionally, with expanding network capabilities, many embedded systems are able to download new software or extend installed software automatically and speedily, thus, exposing themselves to potentially vulnerable programs and becoming easy targets for software-based security attacks.

On the one hand, embedded systems pose a number of unique challenges in terms of resource constraints and susceptibility to physical and side-channel attacks [6]. On the other hand, since they are often targeted at specific applications, they also offer opportunities to constrain the scope of the security problem, and to design security solutions that are optimized to the needs of a specific end system. In this work, we adopt such an approach and apply it to ensuring secure execution of software running on an embedded system.

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 introduce 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 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 the following: 1) the inter-procedural control flow of a program, as represented by its function call graph; 2) the intra-procedural control flow for each function, represented by a basic-block control-flow graph (CFG); and 3) the integrity of the instruction stream within each basic block. We propose techniques to extract these 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.

Our approach has several unique features that differentiate it from previous work and embody our major contributions.

- It 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 the compilation flow.
- Unlike purely static or purely dynamic approaches, 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 application-specific properties at a fine granularity which would not be feasible with a software-protection mechanism. Additionally, it results in lower detection latencies (thus, narrowing the window of opportunity available to the attacker), while incurring minimal delay overheads.
- It uses a specification of intended program behavior and detects deviations from what is specified. It is *attack-independent* in the sense that it does not use information collected from past well-known attacks or the exact mechanism by which they are perpetrated to aid in detection.
- The proposed hardware-assisted monitoring architecture is minimally intrusive to existing embedded processor architectures, and hence, can be easily applied to existing embedded systems.

It is well-known that buffer overflow attacks can be avoided if safe languages such as Java are used. However, the speed, flexibility, and granularity of control imparted by C make it a very popular language for both user applications and systems software. It is not reasonable to expect all developers to switch to a new language in the near future. Our techniques do not place the onus of security on the software developers. Once the architecture and compilation tools are designed, the developer can continue to use C as before and have added security. Due to the previously discussed factors, we believe that our technique

[1]Evans Data Corporation. [Online]. Available: http://www.evansdata.com/n2/index.shtml.

offers a practical approach to counter a broad range of attacks including, for example, buffer overflows, spurious control transfers, and run-time code corruption.

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. Section VI illustrates the applicability of our technique in various attack situations. We present our experimental methodology and results in Section VII and conclude in Section VIII.

II. Related Work

A wide range of techniques have 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 or ensuring isolation between different programs running on a system. These 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 [7], [8]. Some static techniques attempt to solve a much broader problem, namely memory access checking, and prevent any READ/WRITE of a memory location through a pointer or subscripted array that resides outside the scope of the referent. Buffer overflows, which are a subset of this problem (since the violation results when a program writes past a buffer end) are also prevented by these techniques. Some of the recent static analysis tools are BOON [9], SPLINT [10], and CSSV [11]. A comparative study of the effectiveness of five static analysis tools on buffer overflows in open-source software can be found in [12]. An important advantage of static analysis techniques is that bugs can be caught before the code is deployed, thus, eliminating the execution time overhead resulting from extraneous checks. However, they do not guarantee completeness, suffer from a large number of false positives, and do not scale to large programs. Software-based monitoring has also been explored along various dimensions. Techniques, such as those in [13]–[15], explore methods for automatically embedding a network of software security kernels into the program source code. These kernels perform security-related actions such as verifying the checksum of specified sections of the binary and repairing tampered sections at run time. In these studies, the focus is on stealth, on hiding the checking kernels from the adversary, and on computational efficiency. While these works preserve only the binary integrity, software monitoring techniques, such as program shepherding [16], allow greater flexibility by permitting a user to specify a custom security policy which is enforced by a software engine at run time. The engine can restrict execution based on code origin (thereby preventing execution of malicious code) and control transfers based on instruction type. However, the method requires a software user to have complete knowledge and ability to specify a security policy. Additionally, two main drawbacks of these works are that security checks are also pieces of code which are themselves vulnerable to corruption and the granularity of checking they can implement is limited due to the overhead imposed by the additional code.

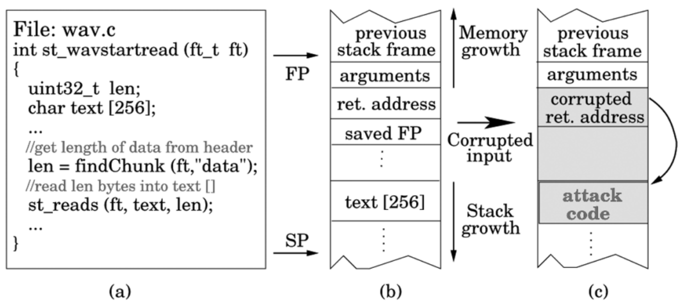
Another branch of software techniques deals with intrusion detection systems [17]–[19] that use learning algorithms to enable the checker to build a database of normal system behavior and detect violations that do not fall within the realm of expected behavior. Although the previously mentioned techniques also characterize "normal behavior" and catch deviations from it, typically, they perform operations at a much higher granularity than our system, such as, tracking resource usage or detecting anomalous system call sequences. Also, they usually result in a large number of false positives due to inherent limitations in the heuristics employed in learning algorithms.

The basic concept of using a hardware unit or coprocessor to facilitate secure execution has roots dating back to tamper-resistant cryptoprocessors that were used to store cryptographic keys and execute cryptographic algorithms [20], [21]. 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 coprocessor to perform intrusion detection (to monitor critical OS data structures, check the integrity of files on disk and perform virus scanning) was proposed in [22].

Recently, enhanced processor architectures, such as XOM and AEGIS, have been proposed [23], [24], 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 prevent an application from being hijacked if its own vulnerabilities are exploited to do so. In [25], 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 these general techniques, a number of *attack-specific* mechanisms have been developed, mostly in response to the increasing number of exploits involving buffer overflows and format string vulnerabilities. An exhaustive listing of these mechanisms is being omitted here for the sake of brevity. However, it should be noted that the problem has been addressed from multiple directions—kernel patches to make the program data/stack segment nonexecutable,[2] compiler extensions to safe-

[2]Nonexecutable User Stack. [Online]. Available: http://www.openwall.com/linux.



```
File: wav.c
int st_wavstartread (ft_t ft)
{
  uint32_t len;
  char text [256];
  ...
  //get length of data from header
  len = findChunk (ft,"data");
  //read len bytes into text []
  st_reads (ft, text, len);
  ...
}
```

Fig. 2. Simple example of a "stack smashing" security attack.

guard the stack [26], hardware-assisted return address protection [27], etc.

III. Motivation

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 Sound eXchange (SoX) toolkit. Fig. 2(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. 2(a) reads `len` bytes from an input file into a local array `text[]`, where parameter `len` is read from the file header. Fig. 2(b) shows the layout of the stack frame for function `st_wavstartread()`, when the function is called during program 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` ($\mathtt{len} > 256$). 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 (which is now stored in `text[]`) to be executed.

While the vulnerability in the previous 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 [5] and [8].

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.

Since the monitoring architecture oversees the execution of the processor and is responsible for safeguarding the whole system in general, its own protection is critical. In this paper, we assume that the communication between the processor and monitor is secure and that the monitor is tamper-resistant (or at least tamper-evident) so that it is not possible for an adversary to disable the monitor without the knowledge of the legitimate system owner. Apart from this, other system components, such as the external memory, on-chip caches, input/output (I/O) devices and the buses connecting these components, are considered insecure. We also assume that the applications to be run within this framework are not malicious by intent.

A. Architecture Overview

Fig. 3 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.[3] The inputs to the monitor include the program counter (PC) and instruction 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 nonmaskable 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 deasserted.

The execution model of a program has a natural hierarchical structure which motivated our behavior model formulation. The monitor has three sub-blocks—which handle program checking at three levels of granularity, namely inter-procedural control flow, intra-procedural control flow, and instruction stream integrity in a straight line code sequence. The design of these sub-blocks is described in detail later in this section.

B. Modeling Permissible Program Behavior

There are several factors to be considered when selecting program properties to be monitored. First and foremost, they should be accurate indicators of invalid behavior, i.e., very likely to be violated when system security is compromised. They should

[3]The proposed technique is fairly independent of the processor microarchitecture, and can be easily adapted to more complex architectures such as superscalar and very long instruction word (VLIW).

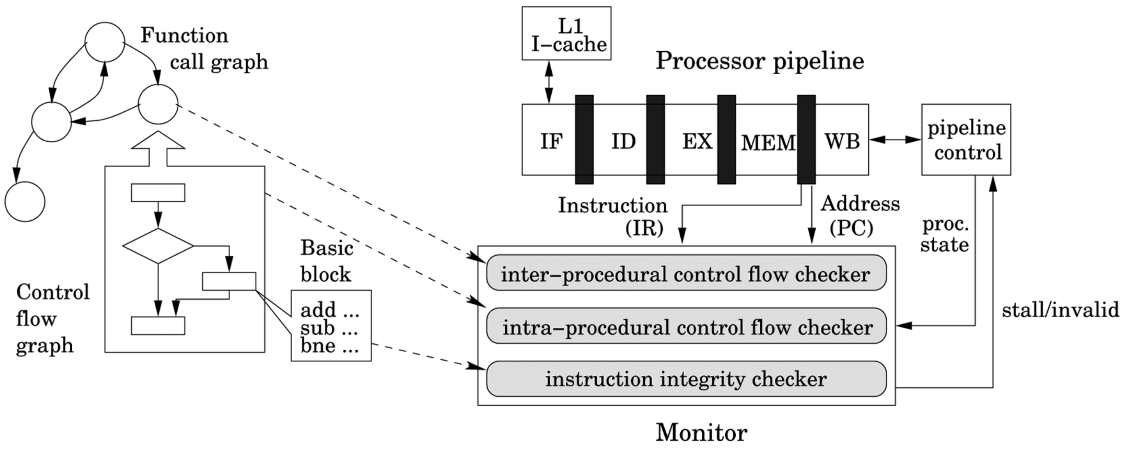


Fig. 3. Proposed hardware-assisted monitoring architecture: conceptual block diagram.

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. States 1 to N are "accepting," i.e., they represent targets of valid transitions. Procedure 1 describes how the FSM is automatically extracted from the function call graph.

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 $f_i$, $1 \leq i \leq N$
 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, $A$: set of 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 */
```

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. However, calls to arbitrary locations resulting from the use of pointer arithmetic are not permitted.

2) Intra-Procedural Control Flow: A logical succession to the previous 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 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) Instruction Stream Integrity: 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, copied onto the monitor tables 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 B). To keep the monitor's hardware requirements low, a user-specified number of bits of the precomputed hash value are randomly selected at

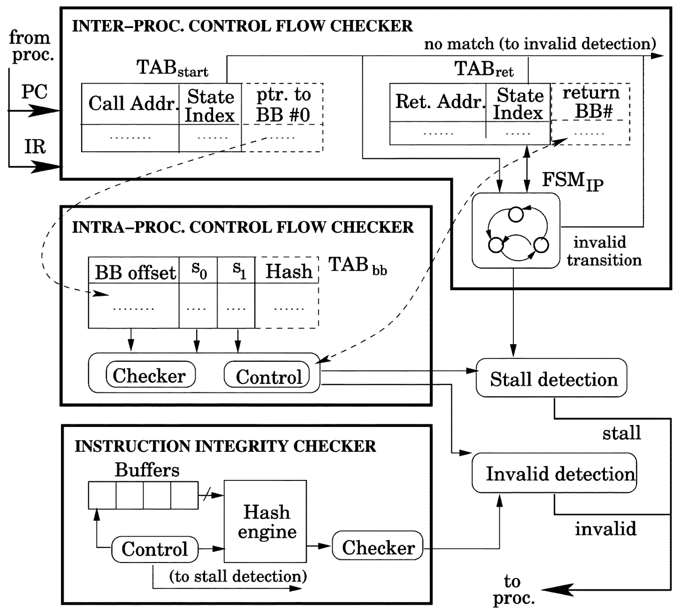


Fig. 4. Architectural details of the run-time monitor.

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.

C. Architectural Details

Fig. 4 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. Also, for $j > i$, monitoring at level j uses information from the level i, i.e., enabling level j checks requires level i checks to also be enabled. For instance, basic block hashes cannot be verified unless the executing basic block is tracked by the intra-procedural control flow checker.

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 to maintain compliance with our formulation, we maintain a mapping from return address to inter-procedural FSM state identifier of the function to which the program returns. Hence, on function return, the state of the inter-procedural FSM is restored to the index stored in this table.

In hardware, each of the two lookup tables can be directly implemented using a *content addressable memory* (CAM). A CAM [28] is basically a read-writable memory circuit that includes additional logic circuitry (comparators) to provide fast table look-up. Small CAMs (of the order of few 100 entries) can achieve a search latency of one cycle. Larger CAMs can be constructed by cascading smaller units and provide multicycle search operations.

The function indices output from $TAB_{\rm start}$ and $TAB_{\rm ret}$ are input to $FSM_{\rm IP}$, 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 $TAB_{\rm bb}$, which stores the information necessary to track control flow within each function. The basic blocks are grouped according to the function they belong to. The function start address table $TAB_{\rm start}$, which is part of the inter-procedural control flow checker, is augmented with an additional field that is a pointer to a location in $TAB_{\rm bb}$. 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 $TAB_{\rm ret}$, and restored upon return.

3) Instruction Integrity Checker: In order to check instruction stream integrity, each row in the basic block table $TAB_{\rm bb}$ 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 $TAB_{\rm bb}$. 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 [2], incur a high latency, since the input is processed through several rounds wherein the result of the ith 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 b, 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.

D. Error Reporting

During execution, if any monitor level detects a violation, it is reported to the processor, which in turn, terminates execution of the current program. The monitor contains special registers for providing feedback to the processor about the location of violation and indicating which of the three monitor sub-blocks flagged the error. In each case, it also reports the PC and IR of the *erroneous* instruction that triggered the alarm.

For inter-procedural control flow violation, the monitor reports include caller/callee state indices and whether the invalid behavior was detected by the FSM or table look-up logic. Depending on IR of the erroneous instruction, this can help in categorizing the violation into three of the following broad categories.

- Violation due to a function making a *call* to an invalid address, i.e., an address that did not correspond to the entry point of any function. This can occur if a function tries to *call* (jump to) some location in the body of another function.
- Violation due to a function making a *call* to a valid but disallowed address—this may occur if a function pointer is set to point to a disallowed callee.
- Violation due to a function making a *return* to an invalid address because of return address corruption in the function stack.

After this, some reverse engineering is required by the person assessing this data to determine the exact cause of violation.

The intra-procedural control flow checker reports the state index of the executing function. This, along with the current PC and jump address, locates the error exactly. However, this only provides information about where the attacker was trying to jump to in the program and not how this was achieved. Corruption of the branch target address may have been achieved either by physical tamper or by execution of a corrupt function that satisfied inter-procedural control flow and was not subjected to intra-procedural control flow checking (in practice, this is very hard to accomplish). Therefore, all security-sensitive functions (e.g., grant access privileges, READ/WRITE sensitive files, etc.) of the program must be subjected to intra-procedural control flow checking.

The instruction integrity checker reports the state index of the executing function, basic block index of the executing basic

TABLE I
STATE ATTRIBUTES FOR FUNCTIONS REQUIRING SPECIAL PROCESSING

Attribute	Handling	Usage
u (unchecked)	Function address is not known before execution, so either it will not be checked or added to the address map the first time the function is invoked	Functions in DLLs
l (leaf node)	The call is checked, but any further function calls are ignored till the function returns	To track the call graph only upto a specified depth
r (recursive)	Entries in TAB_{ret} are checked before their insertion/deletion to avoid multiple entries	recursive functions

block and corrupted basic block, and the buffered contents representing instructions that have been executed but not yet committed. We do not know of any software attacks that can corrupt instruction contents without modifying the control flow of the program. Hence, the aim of this checker is just to prevent such attacks (and possibly collect statistics on likely program targets), since if they are caused by physical tamper, it is not possible to prevent them by software modification.

E. Special Cases

Some functions require special handling, which is implemented by augmenting the state information stored for each function with predefined attributes. The handling capabilities for these attributes are provided in hardware. Therefore, when the program state changes (on a function call/return), the monitor knows if the current function requires special handling. The attributes and their usage scenarios are listed in Table I.

Attribute u (*unchecked*) is required for functions whose addresses are not known statically. For instance, addresses of some functions in dynamically linked libraries (DLLs) may be resolved "lazily" only when a call to the function is made. Calls to such functions are not verified by the monitor. On such a call, the monitor stores the return address in a special register and is disabled from checking any further nested calls until the program returns to the stored return address. Alternatively, the security policy can be defined to add the address entry of functions marked u to the monitor tables the first time such a call is made and check all subsequent calls against this entry.

Attribute l (*leaf node*) allows the developer to stop the checking at any nesting level in the program call sequence. Thus, this is a programmer-defined counterpart to attribute u (which is automatically generated for certain functions). The monitor handles this attribute in the same way, namely by storing the return address when a call to a function marked l is made and suspending all checking till the program returns from that function. This attribute also provides a powerful technique to provide scalability—for instance, a programmer may choose to not check all hook functions called by standard library functions, thus, reducing the amount of configuration information required by the monitor (and the run-time checking) significantly.

Attribute r indicates recursive functions which need to be handled separately since the number of entries in TAB_{ret} is fixed whereas the call depth can be potentially unbounded. We assume an upper bound on the call depth, excluding recursion, for determining the maximum size of the table. Recursive functions are handled more efficiently—only one entry is maintained

```
int flag =0;
void A(args ...)
{
  flag =1;
  B(...);
}
void B(args ...}
{
  if (!flag)
    C( ...);
  else
    ...
}
```

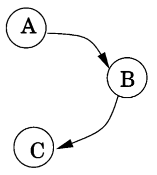


Fig. 5. Example of a violation missed by the inter-procedural checker.

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 $A \rightarrow A$.

Some commonly used C programming constructs present tricky situations for our system. One such construct is function pointers. The specific function that is called on a function pointer dereference is determined at run time, depending on which function address is assigned to the pointer in accordance with some condition being met. We handle function pointers by adding an edge to our static call graph from the function in which the pointer is dereferenced to all possible functions whose address can be assigned to the pointer. In our experience, most applications tend to use function pointers to call a small subset of functions (typically 3–4) and resolve this call at run time. Hence, adding caller–callee relationships in the manner described before does not lead to too many edges in the call graph.

The monitor is disabled during exceptions, i.e., if the processor finds that the committing instruction resulted in an exception, it disables the monitor. The monitor is re-enabled on return from the exception handler. However, sometimes exception handling code can exhibit a call sequence unlike last-in first-out (LIFO). This is also true for constructs such as `setjmp()` and `longjmp()`. Providing support for these constructs is currently under investigation.

F. Limitations

One limitation of the inter-procedural check is that it passes all function calls that are allowed by the static call graph. Therefore, a mere check of the call graph may miss a control flow violation such as the one shown in Fig. 5. Here, function B is permitted to call C but the sequence $A \rightarrow B \rightarrow C$ is not valid.

However, our method does not verify sequences of calls. Similarly, in case of function pointers, if the pointer is somehow assigned to point to one of the other valid functions (than the one that would be called with untampered execution), it would be cleared by the monitor.

On similar lines is the issue of functions in DLLs. As mentioned previously, the addresses of these functions cannot be supplied with the binary. Either such functions are left unchecked or their addresses are added to the monitor tables the first time they are called. Since the function address is not supplied beforehand, it is possible for a corrupted program to call a spurious location the first time a library function is called, and the violation will go undetected by the monitor.

We do not check for data integrity in this work. The underlying assumption is that most of the times, exploits, such as buffer overflow, attempt to corrupt a control structure in the program (return address or function pointers) and would eventually result in a control flow violation. If the intention of the attacker is merely to corrupt a sensitive data location (e.g., name of a file written by the program), without causing a deviation from expected control flow, such an exploit would not be caught by our system.

The purpose of using special purpose hardware in this work is to enable fine grained checks with minimal performance loss and greater security. However, there is a price to be paid for this. The capacity of configuration information that the monitor can store is fixed once the hardware is fixed. Multiple applications can still be loaded with the use of configurable hardware, but the maximum size of an application that can run on this system is bounded. This limitation can be partially alleviated because the extent and granularity of security checks can be traded off, e.g., by checking the application only upto a particular call depth or limiting intra-procedural checks to fewer functions. Also, we do not believe this is a serious limitation in case of embedded systems since their application space and typical application sizes are known *a priori* in most cases.

V. Design Flow

In this section, we describe the sequence of steps by which an application is enhanced to use our monitoring framework, followed by configuration of the monitor to enable application-specified security checks.

Fig. 6 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 the granularity of checks (e.g., the depth to which the function call graph is tracked) and select functions for which intra-procedural checking and hash checking are done. 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. Our run-time monitoring framework primarily aims at protecting programs from violations during execution. However, if the initial binary provided

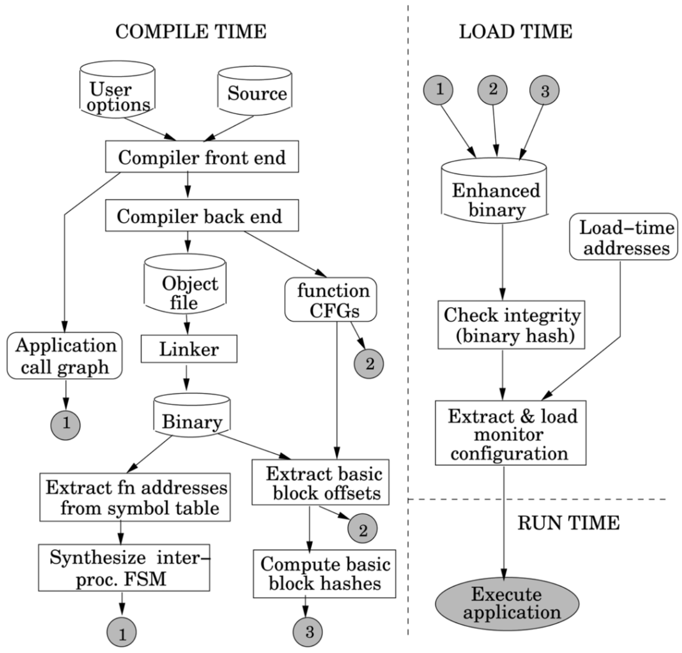
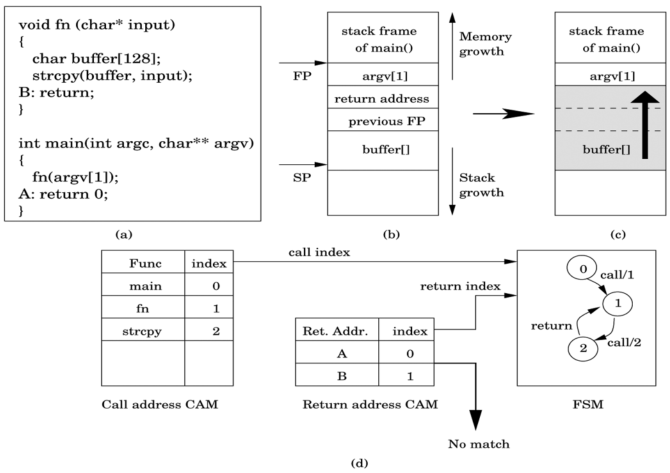


Fig. 6. Design flow for hardware-assisted run-time monitoring.

to the system is already tampered, then security cannot be guaranteed by our system. In such a case, the integrity of the binary is verified before it is loaded by computing its cryptographic hash and comparing it against a prespecified "golden hash." Again, this assumes that golden hashes of the programs, which execute with monitoring enabled, are stored in the secure memory of the processor and cannot be modified by the adversary. If both the golden hash and binary are susceptible to tamper, then a keyed hash or message authentication code (MAC) must be employed. Schemes using similar ideas are described in detail in [23] and [24], although these systems focus both on privacy and integrity and, therefore, use encryption to protect applications from being revealed to unauthorized agents. Essentially, each processor possesses a secret key K_s that is burnt into a nonvolatile memory and forms the identification of the processor. For each binary that is to be run on a particular processor, the vendor computes its keyed-hash using K_s of that processor and appends this hash to the binary. This hash is then verified by the monitoring system before execution. K_s is known only to the processor and trusted software vendors and is shared via secure mechanisms. Therefore, an adversary cannot corrupt a binary and still create a valid hash of the corrupted binary without any knowledge of K_s.

Configuration of the security monitor is performed when the application is loaded for execution. Security technologies, such as ARM TrustZone,[4] can be used to provide a secure mode during configuration, while the remaining application is run in a nonsecure 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., field-programmable gate

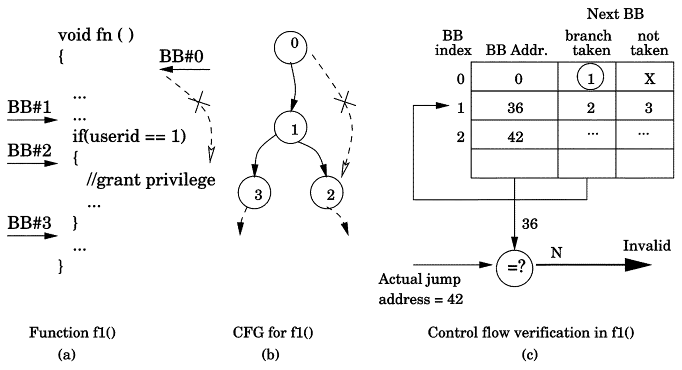
[4]ARM TrustZone technology overview. [Online]. Available: http://www.arm.com/products/CPUs/arch-trustzone.html.



```
void fn (char* input)
{
  char buffer[128];
  strcpy(buffer, input);
B: return;
}

int main(int argc, char** argv)
{
  fn(argv[1]);
A: return 0;
}
```

Fig. 7. Protecting against "stack smashing" attack.



```
void fn ( )
{
  ...
  ...
  if(userid == 1)
  {
    //grant privilege
    ...
  }
  ...
}
```

Fig. 8. Protecting against intra-procedural control flow violation.

array (FPGA) cores], or explicit state-table-based implementations using memories. After configuration of the monitor, the loader initiates application execution. The dynamic loader is also responsible for updating addresses of functions, which are not known at load time, and loading them into the function start address table (TAB_{start}) in the hardware.

VI. Attack Scenarios

As previously mentioned, the techniques presented in this paper were not designed to counter a particular category of exploits or shield certain kinds of vulnerabilities. The rationale behind our approach is that one can statically determine and specify expected program behavior and anything that does not conform to this specification is a plausible security violation. In this section, we present some attack scenarios that can be foiled effectively by our technique.

1) Stack Smashing: An example of such an attack is shown in Fig. 7. The basic mechanism employed in stack smashing exploits [7] involves overflowing a buffer declared on a function's stack frame and overwriting its return address in the process. This example is similar to the one provided earlier in the paper except that here a standard C library function is involved in the overflow.

The stack shows the activation record of function `fn()` when it is called from `main()`. The call to library function `strcpy()` from function `fn()` copies the user-supplied `input` into character array `buffer[]`. `strcpy()` does not perform any bounds checking on `input` and overwrites the function frame pointer (FP) and return address if the size of `input` exceeds the size of `buffer[]`, as shown in Fig. 7(b) and (c). Fig. 7(d) illustrates how this exploit is detected by the inter-procedural checking logic when the same program is run with our protection monitor. When the call to `fn()` is made, return address `A` is stored in the return address CAM along with the index of `main()` and the FSM transitions from state `0` to state `1`. Later, when `fn()` tries to return to a corrupted return address, the monitor will search in TAB_{ret}, find no matching entry in it, and assert output signal `invalid`.

Another violation detection scenario is when a function (that maps to a valid index in TAB_{start}) calls another valid function that it is not permitted to call. In that case, the FSM transitions to the invalid state and signals the processor. Attacks, which reuse pre-existing code through incorrect interfaces (e.g., return-into-libc exploits that call a C library function like `system()` rather than new code injected onto program stack), are caught in this manner.

2) Control Flow Violation: Fig. 8 demonstrates an example where intra-procedural control flow violation checks are necessary. The figure shows a code snippet that grants special access privileges to a user depending on `userid`. The CFG of this code is also shown. If an adversary can somehow insert a jump directly, from say BB_0 to BB_2, he/she can bypass the security checks imposed by the `if` condition. For such security-critical functions, the second level of monitoring strategy should be applied along with inter-procedural checks. In this case, the monitor will expect a jump to the address of BB_1 and will signal an error since the (corrupted) jump is to a different address. Thus, the intra-procedural FSM checker allows jumps to only prespecified branch and jump targets.

3) Instruction Memory Corruption: In the previous example, a violation will not be detected if an attacker inserts a branch on some other condition (which is always true) from BB_1 to BB_2 before the actual `userid` check. Such a transgression will only be caught by dynamic hash checking since the sequence of instructions in BB_1 will be altered. Admittedly, the attacker can still violate program behavior by actually modifying the value of `userid` and setting it to 1. Nevertheless, all these tests significantly reduce the available window of opportunities for attack. Hash checking is also required in cases in which only a portion of the code is modified such that it does not result in any change in the control flow. The high frequency of hash checks provides a greater assurance that any such attempt will be detected during code execution.

VII. 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 [29] and MiBench benchmark [30] suites, which represent typical workloads for embedded processors. The information necessary to generate the hardware monitor was extracted using the GNU

TABLE II
AREA OVERHEADS FOR HARDWARE-ASSISTED RUN-TIME MONITORING

Benchmark	#Fns	#Fn calls	#Basic blocks	AO% level1	AO% level2	AO% level3		
						unpipe	pipe	pipe + 20B hash
C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9
rawdaudio	7	6	27	0.16	0.22	1.46	3.10	3.28
epic	71	122	901	0.68	1.65	3.15	4.78	9.64
g721decode	24	64	298	0.29	1.83	2.19	3.83	6.28
toast (gsm)	137	229	1130	1.20	3.14	4.88	6.52	16.21
mpeg2encode	116	223	2114	0.98	5.17	7.43	9.07	28.41
pegwit	124	293	1275	1.08	3.06	4.80	6.44	16.13
susan	38	64	563	0.41	1.45	2.95	4.59	9.44
rasta	111	246	1357	0.96	2.98	4.73	6.36	16.05

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, buffers, and hardware hash engine for the instruction integrity checker, and miscellaneous control logic.

For estimating the area of hash engines, we implemented Verilog register-transfer level (RTL) descriptions of two commonly used hash algorithms, MD4 and MD5 [2]. The hash engines and FSM (for each application) were synthesized into technology-mapped gate-level netlists using Synopsys Design Compiler[5] with NEC's 0.13-μm CB-130M CMOS standard cell library. Memory models from [31] 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 16-kB instruction and data caches, runs at 250 MHz, and occupies an area of 4.7 mm^2 in a 0.13-μm technology.[6]

Table II reports the area overheads for different benchmarks, for three monitoring scenarios that correspond to increasing levels of security. The second, third, and fourth columns (C_{2-4}) list the number of functions (*#Fns*), number of function call locations (*#Fn calls*), and number of basic blocks (*#Basic blocks*) in each benchmark program. C_5 (*AO% level1*) reports the total percentage area overhead when only the inter-procedural control flow checker is used. C_6 (*AO% level2*) reports the total percentage area overhead when both inter- and intra-procedural checkers are used, and C_{7-9} (*AO% level3*) represent the case when an instruction integrity checker is also employed in addition to control flow checkers. C_7 (*unpipe*) and C_8 (*pipe*) represent the cases when the hash engine within the instruction integrity checker is unpipelined and pipelined, respectively.

C_9 (*pipe + 20 B hash*) reports the area overhead when a pipelined hash engine is employed and the entire 20 B of the hash value is loaded into the monitor tables for integrity checking. This case is similar to C_8, except that there, eight randomly selected bits of the hash value are used for comparison during instruction integrity checking. C_9 essentially shows the tradeoff between security and area overhead made in our scheme. Using only eight bits of the hash value for comparison reduces the security of the scheme since now the adversary needs to guess fewer bits as compared to the case involving comparison of the full hash. Therefore, the chance that the adversary guesses the correct value for a basic block is $1/(C(160, 8) \times 2^8)$ (where $C(n, p)$ represents the number of ways of choosing p elements out of n) rather than $1/2^{160}$. However, it should be noted that the positions of bits that are loaded into the monitor are determined based on the output of a random number generator, each time the program is loaded. Therefore, the probability of success of the adversary does not increase with more trials (if the random number generator is truly random). Additionally, storing a reduced number of bits in the monitor brings the area overhead to within acceptable limits (from a maximum of 28.41% to 9.07%) as shown in the table. Another point to be noted is that this tradeoff does not affect the system performance since the hash engines still need to compute the complete hash before it can be compared against the stored value.

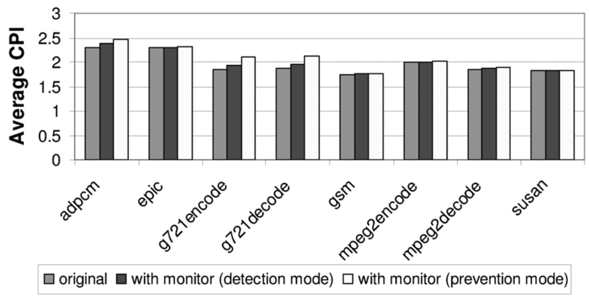
The table indicates that for all the benchmarks considered, inter-procedural checking alone can be implemented with a maximum area overhead of 1.20% and an average overhead of 0.72%. Adding intra-procedural checking raises the maximum overhead to 5.17% and average overhead to 2.44%. For the case with 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.

There are many options for implementing the inter- and intra-procedural FSMs described in this paper. In the area estimates provided before, the inter-procedural FSMs are synthesized to custom hardware while the intra-procedural FSM state information and basic-block hashes are stored in standard DRAM.

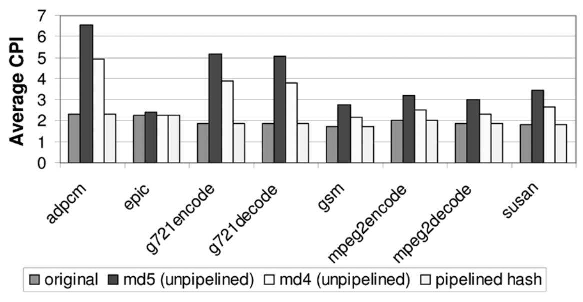
Ultimately, we envision the entire monitor to be implemented in dynamically reconfigurable logic, onto which the program configuration is loaded on application start-up. Programmable platforms (e.g., Virtex-2 Pro from Xilinx) that include embedded processors along with field-programmable gate arrays

[5]Synopsys. [Online]. Available: http://www.synopsys.com.

[6]ARM920T embedded processor core. [Online]. Available: http://www.arm.com.



(a)



(b)

Fig. 9. Performance analysis of (a) intra-procedural control flow checking and (b) dynamic hash checking.

TABLE III
LUT COUNTS FOR INTER-PROCEDURAL FSMS

Benchmark	LUT count	Frequency (MHz)
rawdaudio	17	599
epic	574	162
g721decode	199	265
toast (gsm)	1019	82
mpeg2encode	1215	84
pegwit	270	201
susan	1839	57
rasta	918	75

TABLE IV
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 address CAM latency	10 ns	Return address 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

(FPGAs) have begun to arrive in the market. Some of their reconfiguration capability can be utilized to provide security solutions. To this end, we also evaluated the implementation overhead of inter-procedural FSMs on an FPGA. The FSMs were synthesized using Synopsys FPGA-Compiler with an FPGA from the Xilinx Virtex2P family.[7] The look-up table (LUT) count and corresponding frequencies of the synthesized FSM are presented in Table III. As can be seen from the table, even the largest FSM for these benchmarks requires only 1839 LUTs which is a modest amount, given the capacity of FPGAs in current technology.

B. Performance Impact

We evaluated the performance impact of the proposed architecture using the SimpleScalar 3.0/PISA architectural simulation toolset [32]. The microarchitectural parameters of SimpleScalar were configured to model a typical embedded processor, the ARM920T. The parameters used for the simulated processor are shown in Table IV. We built our simulator within the framework of the existing cycle-accurate simulator *sim-out-order* [32] 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 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. 9(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. 9(b) illustrates the impact of instruction integrity checking on CPI, for different hash algorithms and their implementations. As expected, the performance penalty for an unpipelined MD5 hash engine, with a latency of 64 clock

[7]Xilinx. [Online]. Available: http://www.xilinx.com.

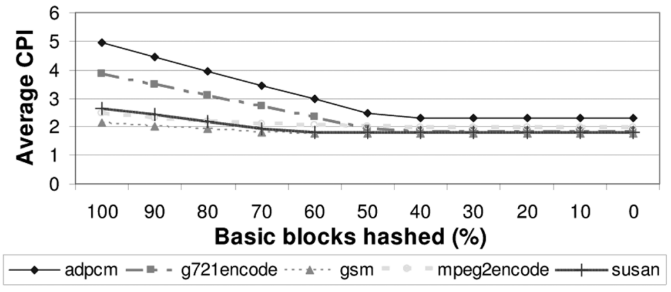


Fig. 10. Performance estimation when the percentage of basic blocks hashed is varied.

cycles, is substantial. The performance penalty using an unpipelined 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. 10 shows how the average CPI is affected by varying the number of basic blocks that are hash-checked when an unpipelined 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.

VIII. Conclusions and Future Directions

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.

There are associated costs—in terms of extra logic and on-chip storage, and it may initially seem improvident to add such functionality to systems which are already resource constrained. However, we believe, in the coming years, security concerns in embedded systems will become serious enough to warrant such a solution. Moreover, our approach is an instance of a broad idea. One direction in which we plan to extend it is to incorporate other program signatures that can be used as evidence of untampered execution. Also, the specification can be developed further to allow more flexibility in run-time behavior of programs. The concept of designing hardware to ensure consistency between static and dynamic aspects of a program is still unexplored and appears to have considerable potential in securing program execution in untrustworthy environments.

References

[1] P. Koopman, "Embedded system security," *IEEE Comput.*, vol. 37, no. 7, pp. 95–97, Jul. 2004.
[2] B. Schneier, *Applied Cryptography: Protocols, Algorithms and Source Code in C*. New York: Wiley, 1996.
[3] W. Stallings, *Cryptography and Network Security: Principles and Practice*. Englewood Cliffs, NJ: Prentice-Hall, 1998.
[4] O. Aleph, "Smashing the stack for fun and profit," *Phrack Mag.*, vol. 7, 1996. [Online]. Available: http://www.phrack.org/archives/49/P49-14
[5] Vulnerability notes database CERT Coordination Center, 2006. [Online]. Available: http://www.kb.cert.org/vuls/
[6] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *ACM Trans. Embedded Comput. Syst.*, vol. 3, no. 3, pp. 461–491, 2004.
[7] M. Howard and D. LeBlanc, *Writing Secure Code*. Redmond, WA: Microsoft Press, 2002.
[8] G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code*. Reading, MA: Addison-Wesley, 2004.
[9] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2000, pp. 3–17.
[10] D. Larochelle and D. Evans, "Statically detecting likely buffer overflow vulnerabilities," in *Proc. USENIX Security Symp.*, 2001, pp. 177–190.
[11] N. Dor, M. Rodeh, and M. Sagiv, "CSSV: Towards a realistic tool for statically detecting all buffer overflows in C," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implement.*, 2003, pp. 155–167.
[12] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," in *Proc. ACM Int. Symp. Foundations Softw. Eng.*, 2004, pp. 97–106.
[13] B. Horne, L. R. Matheson, C. Sheehan, and R. E. Tarjan, "Dynamic self-checking techniques for improved tamper resistance," in *Proc. Workshop Security Privacy Dig. Rights Managem.*, 2001, pp. 141–159.
[14] H. Chang and M. J. Atallah, "Protecting software code by guards," in *Revised Papers ACM CCS-8 Workshop Security Privacy Dig. Rights Managem.*, 2001, pp. 160–175.
[15] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. Jakubowski, "Oblivious hashing: A stealthy software integrity verification primitive," in *Proc. Int. Workshop Inf. Hiding*, 2002, pp. 400–414.
[16] V. Kiriansky, D. Bruening, and S. P. Amarasinghe, "Secure execution via program shepherding," in *Proc. USENIX Security Symp.*, 2002, pp. 191–206.
[17] R. Sekar, M. Bendre, D. Dhurjati, and P. Bolineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proc. IEEE Symp. Security Privacy*, 2001, pp. 144–155.
[18] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *Proc. IEEE Symp. Security Privacy*, 2003, pp. 62–75.
[19] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *J. Comput. Security*, vol. 6, no. 3, pp. 151–180, 1998.
[20] M. Kuhn, "The trust No 1 cryptoprocessor concept," Purdue Univ., (1997) [Online]. Available: http://www.cl.cam.ac.uk/ mgk25/
[21] S. W. Smith and S. H. Weingart, "Building a high-performance, programmable secure coprocessor," in *Proc. Comput. Netw.*, 1999, pp. 831–860.
[22] X. Zhang, L. Doorn, T. Jaeger, R. Perez, and R. Sailer, "Secure coprocessor-based intrusion detection," in *Proc. ACM SIGOPS Eur. Workshop*, 2002, pp. 239–242.
[23] 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. Arch. Support Program. Lang. Oper. Syst.*, 2000, pp. 168–177.
[24] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas, "AEGIS: Architecture for tamper-evident and tamper-resistant processing," in *Proc. Int. Conf. Supercomput.*, 2003, pp. 160–171.
[25] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proc. Int. Conf. Arch. Support Program. Lang. Oper. Syst.*, 2004, pp. 85–96.
[26] C. Cowan *et al.*, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. USENIX Security Symp.*, 1998, pp. 63–77.
[27] J. P. McGregor, D. K. Karig, Z. Shi, and R. B. Lee, "A processor architecture defense against buffer overflow attacks," in *Proc. Int. Conf. Inf. Technol.: Res. Edu.*, 2003, pp. 243–250.

[28] J. H. Shaffer, "Designing very large content-addressable memories" (1992). [Online]. Available: citeseer.ist.psu.edu/shaffer92designing.html.

[29] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. Int. Symp. Microarch.*, 1997, pp. 330–335.

[30] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBnch: A free, commercially representative embedded benchmark suite," in *Proc. Annu. Workshop Workload Characterization*, 2001, pp. 3–14.

[31] 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.

[32] D. Burger and T. M. Austin, "The Simplescalar Tool Set, Version 2.0," Comput. Sci. Dept., Univ. Wisconsin, Madison, 1997.

Divya Arora received the B.Tech. degree from the Indian Institute of Technology, Delhi, India, in 2002 and the M.A. degree from Princeton University, Princeton, NJ, in 2004, both in electrical engineering. She is currently pursuing the Ph.D. degree, also in electrical engineering, from the same university.

Her research interests include architectures and system-level design methodogies for secure embedded systems.

Srivaths Ravi (SM'05) received the B.Tech. degree in electrical and electronics engineering from the Indian Institute of Technology, Madras, India, in 1996, and the M.A. and Ph.D. degrees in electrical engineering from Princeton University, Princeton, NJ, in 1998 and 2001, respectively.

Currently, he is a Research Staff Member with NEC Laboratories America Inc., Princeton, NJ, and also holds a Visiting Research Collaborator Position with the Department of Electrical Engineering, Princeton University. He leads various projects in the areas of embedded security and low-power design. He co-designed the MOSES security processing architecture for NEC's MP211 multiprocessor mobile phone application system-on-chip (SoC). He has also been responsible for developing RTL and C-based power estimation engines in NEC's C-based design flow CYBER. His research interests include the areas of advanced embedded processing architectures, system-level and RTL test technologies, and low-power design. His publications have appeared in leading ACM/IEEE conferences and journals on VLSI/computer-aided design (CAD), including invited contributions and talks at the International Forum on Application-Specific Multi-Processor SoC, *ACM Transactions on Embedded Computing Systems*, International Conference on VLSI Design and Design Automation and Test in Europe Conference.

Dr. Ravi was a recipient of Best Paper Awards at the International Conference on VLSI design in 1998, 2000, and 2003. He received the Siemens Medal from the Indian Institute of Technology, Madras, India, in 1996. He serves in the organizing/program committees of various conferences including VLSI Test Symposium (VTS) and Design Automation and Test in Europe (DATE).

Anand Raghunathan (S'93–M'97–SM'00) received the B. Tech. degree in electrical and electronics engineering from the Indian Institute of Technology, Madras, India, in 1992 and the M.A. and Ph.D. degrees in electrical engineering from Princeton University, Princeton, NJ, in 1994 and 1997, respectively.

Dr. Raghunathan is currently a Senior Research Staff Member at NEC Laboratories America Inc., Princeton, NJ, where he leads research projects related to system-on-chip (SoC) architectures, design methodologies, and design tools. He coauthored *High-level Power Analysis and Optimization* and six book chapters, and has presented several full-day and embedded conference tutorials in the previously mentioned areas. He holds or has filed for 20 U.S. patents in the areas of advanced SoC architectures, design methodologies, and VLSI computer-aided design (CAD).

Dr. Raghunathan was a recipient of Best Paper Awards at the IEEE International Conference on VLSI Design (one in 1998 and two in 2003) and at the ACM/IEEE Design Automation Conference (1999 and 2000), and three Best Paper Award nominations at the ACM/IEEE Design Automation Conference (1996, 1997, and 2003). He received a Patent of the Year Award (an award recognizing the invention that has achieved the highest impact), and a Technology Commercialization Award from NEC in 2001 and 2005, respectively. He was chosen by MIT Technology Review to be among the TR35 (top 35 technology innovators under 35 years) for 2006. He was a recipient of the IEEE Meritorious Service Award (2001) and Outstanding Service Award (2004), and was elected a Golden Core Member of the IEEE Computer Society in 2001, in recognition of his contributions. He has been a member of the technical program and organizing committees of several leading conferences and workshops. He has served as Program Chair for the IEEE VLSI Test Symposium and the ACM/IEEE International Symposium on Low Power Electronics and Design. He has also served as Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN, the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, *IEEE Design and Test of Computers*, and the *Journal of Low Power Electronics*. He is currently the Vice-Chair of the Tutorials and Education Group at the IEEE Computer Society's Test Technology Technical Council.

Niraj K. Jha (S'85–M'85–SM'93–F'98) received the B.Tech. degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, India, in 1981, the M.S. degree in electrical engineering from the State University of New York (SUNY) at Stony Brook, NY, in 1982, and the Ph.D. degree in electrical engineering from University of Illinois at Urbana-Champaign, in 1985.

He is a Professor of Electrical Engineering at Princeton University, Princeton, NJ. He has coauthored *Testing and Reliable Design of CMOS Circuits* (Kluwer, 1990), *High-Level Power Analysis and Optimization* (Kluwer, 1998), and *Testing of Digital Systems* (Cambridge Univ. Press, 2003). He has also authored six book chapters. He has authored or coauthored more than 300 technical papers.

Dr. Jha is currently serving as an Associate Editor of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS II: EXPRESS BRIEFS, the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN, the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, and the *Journal of Low Power Electronics*. He has served as an Editor of the *Journal of Electronic Testing: Theory and Applications (JETTA)* in the past. He has served as the Guest Editor for the JETTA special issue on high-level test synthesis. He served as the Director of the Center for Embedded System-on-a-chip (SoC) Design funded by the New Jersey Commission on Science and Technology. He has received 11 U.S. patents. His research interests include nanotechnology, thermal analysis and optimization, computer-aided design of integrated circuits and systems, digital system testing, and computer security. He is a Fellow of the ACM. He is the recipient of the AT&T Foundation Award and NEC Preceptorship Award for research excellence, NCR Award for teaching excellence, and Princeton University Graduate Mentoring Award. He has co-authored six papers which have won the Best Paper Award at ICCD'93, FTCS'97, ICVLSID'98, DAC'99, PDCS'02, and ICVLSID'03. A paper of his was selected for "The Best of ICCAD: A collection of the best IEEE International Conference on Computer-Aided Design papers of the past 20 years," and another by IEEE Micro as being among the best of 2005 Computer Architecture conference papers. He has served as the Program Chairman of the 1992 Workshop on Fault-Tolerant Parallel and Distributed Systems and the 2004 International Conference on Embedded and Ubiquitous Computing.