# DESIGN AND EVALUATION OF INFORMATION FLOW SIGNATURE FOR SECURE COMPUTATION OF APPLICATIONS

BY

PRATEEK PATEL

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Adviser:

Professor Ravishankar K. Iyer

# ABSTRACT

This thesis presents an architectural solution that provides secure and reliable execution of an application that computes critical data, in spite of potential hardware and software vulnerabilities. The technique does not require source code of or specifications about the malicious library function(s) called during execution of an application. The solution is based on the concept of Information Flow Signatures (IFS). The technique uses both a model-checker-based symbolic fault injection analysis tool called SymPLFIED to generate an IFS for an application or operating system, and runtime signature checking at the level of hardware to protect the integrity of critical data. The runtime checking is implemented in the IFS module. Reliable computation of data is ensured by the critical value re-computation (CVR) module.

Prototype implementation of the signature checking and reliability module on a soft processor within an FPGA incurs no performance overhead and about 12% chip area overhead. The security module itself incurs about 7.5% chip area overhead. Performance evaluations indicate that the IFS module incurs as little as 3-4% overhead compared to 88-100% overhead when the runtime checking is implemented as a part of software. Preliminary testing indicates that the technique can provide 100% coverage for insider attacks that manifest as memory corruption and change the architectural state of the processor. Hence the IFS and CVR implementation offers a flexible, low-overhead, high-coverage method for ensuring reliable and secure computing.

# ACKNOWLEDGMENTS

This project drew upon the efforts and insights of many people, without whom it could not have succeeded. I am grateful for the support and guidance offered by my adviser, Ravishankar K. Iyer. I also deeply appreciate the insights and encouragement offered by Prof. Zbigniew Kalbarczyk. Thanks also to Prof. Nithin Nakka, who played a pivotal role in helping me through the design process; Karthik Pattabiraman, who produced the SymPLFIED framework and LLVM instrumentation utilized by this project during the initial phase; and Flore Yuan, who was instrumental in resolving issues related to evaluation and providing insights on SymPLFIED. Additionally, I would like to thank all the other members of the DEPEND research group for being so cooperative and encouraging me throughout the process. This project could not have produced as the working prototype that exists today without the hard work and contributions of the aforementioned individuals. Finally, thanks to my friends and family, who supported me during this process.

# TABLE OF CONTENTS

# CHAPTER 1

# INTRODUCTION

Memory corruption attacks represent a major part of security attacks reported in recent years [1]. Applications written in unsafe languages like C or C++ are still numerous and are vulnerable to attacks exploiting memory errors like buffer overflow and underflow [2], [3], dangling pointers [4], or double frees [5]. Theft and modification of sensitive information on computer systems contribute to billions of dollars in annual losses and often reflect organized crime.

As for defensive techniques, the most complete solutions often incur either high runtime or memory space overhead (e.g., duplication of critical data). Current software-based security techniques tend to focus on remote vulnerabilities, such as buffer overflow attacks, format string attacks, and other memory corruption attacks, looking either to “plug” every potential hole in the software, or to make sure the statistical likelihood of a successful attack is extremely low. Several hardware-based runtime security techniques have also emerged to prevent similar classes of attacks. These techniques improve upon the performance overheads of the software-based techniques. Techniques to detect anomalous execution based on control flow monitoring have also been proposed to protect against code injection into an application. However, these approaches focus on defending against attacks rather than protecting critical data.

The piecewise nature of current defenses against computer security vulnerabilities continues to allow malicious attackers to gain unauthorized access into computer systems. Hardware defenses against malicious tampering can ensure that attacks are detected reliably and nearly instantaneously. The Reliability and Security Engine (RSE), a low-level interface to the internal signals of a microprocessor, is a generic hardware framework that can be leveraged by a hardware designer to implement powerful mechanisms in modules that improve the reliability and security of a computer system [6]. The RSE presents a standard interface of pipeline signals that can be used to monitor the state of a processor and ensure reliability and security of the executing code. Because the RSE gives low-level access to processor signals, it is possible to develop techniques that can guarantee the integrity of data within a system by ensuring trustworthy execution of code that computes these data. In addition to providing such guarantees, RSE modules can be designed so that they have no effect on performance and little hardware area overheads on the host processor. The Information Flow Signatures (discussed in this thesis) technique for data integrity implemented as RSE modules does exactly this. Using the technique, it is possible for a software developer to select which critical data must be protected within a program or operating system and to have the data automatically protected during runtime. Because the runtime protection happens at a low level in hardware, it is possible to detect security violations with extremely low latencies, thus allowing action to be taken before corruption of data occurs.

## 1.1 Motivation

Insider attacks have been widely reported in the financial and critical infrastructure sectors [7, 8], in which the attackers' goals have been financial gain and disruption of essential services, respectively. Such attacks are becoming increasingly prevalent due to the outsourcing of software components, often spanning geographic boundaries, and the increased distribution of libraries and third-party modules over the internet. Also, as process technologies continue to shrink and margins decrease [9], the likelihood of memory corruption errors, which impact an application's integrity [10], is on the rise. Further, as critical systems increasingly deploy commodity components, there is a greater opportunity for malicious individuals or nation-states to launch such attacks. Therefore, insider attacks are underrepresented in the literature and will become increasingly important in the future.

Application-level insider attacks are particularly hard to detect because third-party modules and library functions are often distributed in binary form, and their source code is not available. Even if the code is available, only a detailed security audit can find logical loopholes in it, and such detailed audits are time- and effort-intensive. Further, the malicious library or module may exhibit its malicious behavior only under exceptional situations triggered by the attacker's input. Such inputs are unlikely to be provided during testing unless exhaustive testing is performed, which is often impractical. Finally, the insider has almost unrestricted privileges to write to any part of the application data space or execute arbitrary code. Under those circumstances, it is extremely difficult to identify a specific module or library as the cause of the attack, so insiders can effectively cover their tracks. Finding these attacks requires extensive instrumentation of the application and incurs high performance and resource overheads or hardware requirements [11]. These overheads represent more than an annoyance; for example, time-critical systems may not be able to accommodate a high runtime overhead, while embedded systems may not be able to accommodate excessive hardware requirements. Moreover, such techniques also detect a number of benign errors (errors that have no detrimental impact on a program's execution) [12] and therefore unnecessarily expend computational resources on their detection. Several hardware-based runtime security mechanisms based on these techniques have also emerged to prevent similar classes of attacks. Hardware-based runtime security techniques, covered in detail in Chapter 2, improve upon the performance overheads of software-based techniques. Techniques to detect anomalous execution based on control flow monitoring have been proposed to protect against code injection into an application [13]. However, these approaches focus on defending against attacks rather than protecting critical data. For example, with the increasing processor power available as a result of from shrinking process technology, computers can now efficiently run many applications simultaneously on the same system. While some of these applications are well-written and secure, others are not. Insecure applications can pose a threat to those that are secure, since current security techniques do not focus on protecting the target of attacks (critical data such as passwords or user authentication variables), but instead look to mitigate classes of attacks. In the face of such vulnerabilities, it is worthwhile to consider security techniques that operate under the assumption that a "hole" in the virtual fence may be found at any time. Thus, by protecting the critical data target of malicious attacks, a technique may prevent both current and future attack vectors.

In order to minimize the performance and area overhead associated with duplication detection mechanisms, a method is proposed whereby only critical variables are considered for detection.

### 1.2 Contributions

This thesis presents a new method for application source code analysis to derive the critical signature along with the hardware-implemented enforcement of Information Flow Signatures. The implementation ensures that, during runtime, security-critical data are computed according to source code semantics, even under the threat of hardware or software vulnerabilities. If an attacker attempts to tamper with the protected program execution or data used by the application, the system will be notified so that it can take appropriate actions. Though this technique can also be used to provide protection against a wide range of memory corruption attacks, we focus on the hardware-based mechanism for protecting security-critical data against malicious tampering due to external and insider attacks.

The specific contributions of this thesis are:

1. Design, implementation, and demonstration of IFS (Information Flow Signatures) that check hardware for trustworthy execution of instructions computing critical data. An FPGA-based prototype has been developed with real-world applications to demonstrate the efficacy of the concept.
2. Augmentation of the Maude model-checker-based SymPLFIED [14] tool to support (i) a prototype (SPARC-Leon3-based) architecture, (ii) exhaustive symbolic fault injection for application analysis to derive the signature of the application, and (iii) application instrumentation to facilitate an interface between the application and IFS checking hardware.
3. Qualitative analysis of the technique's coverage of memory corruption (generic and targeted) attacks and detailed evaluation of hardware resource overhead and performance overhead for real-world applications.

### 1.3 Organization

The rest of the thesis is organized as follows. In Chapter 2 we describe some of the related security work. Chapter 3 addresses the basic concept of the IFS, the attack model, the software implementation, and, briefly, the new proposed analysis method and hardware-based architectural solution. In Chapter 4 we introduce the tool SymPLFIED, explain how it is used for source code analysis, and we describe the extensions made to it to support IFS. The chapter also describes in great detail the limitations of the previous compiler-based static analysis and how symbolic error injection overcomes all these limitations. Chapter 5 describes the Reliability and Security Engine, a hardware-based framework for monitoring of applications, and its new modified architecture for easier coding and extensibility. Chapter 6 gives more details on the architectural features of the IFS module, its runtime checking operation, and how a hardware-based solution removes the vulnerabilities present in the software-based IFS implementation. Chapter 7 details the nature of insider attacks, the importance of preventing such attacks, different attack scenarios, and the efficacy of the IFS method in detecting those attacks. Chapter 8 describes another major contribution of this thesis, which is the design and implementation of a unified framework for security and reliability; hence, we briefly describe the reliability module and how it was integrated with the security module and the main processor on a single fabric. Finally, Chapter 9 presents results that include hardware footprints for different configurations and performance overhead due to IFS, along with conclusions and plans for future work.

# CHAPTER 2

# RELATED WORK

A multitude of techniques have been proposed to address the security concerns discussed in the Introduction. Some of these techniques offer low-level detection or correction but operate in a manner largely indifferent to the needs of the application being protected. To support application-aware detection, several techniques utilize static analysis of application source code or dynamic analysis of application invariants to provide error detection services that are geared to the needs of individual applications. Others utilize hardware and software redundancy; such techniques rely on either software duplication or hardware duplication. All of these methods have their benefits and drawbacks. The properties of a secure software application can be summarized as follows:

1. Instructions cannot be directly stolen, inferred, or modified;
2. Data cannot be directly stolen, inferred, or modified;
3. Execution flow cannot be tampered with or monitored;
4. The application and all of its subcomponents can only be used by authorized entities;
5. Authorized users cannot be denied usage of or access to an application or its subcomponents.

Nearly all direct attacks against software, despite variations in methodology and sophistication, exhibit intentions that fall under one of the previously stated protection properties. While it is generally accepted that no system can completely protect against all classes of threats, the properties listed above provide a general upper bound on protection goals for all systems that attempt to protect software.

### 2.1 Security Architectures

The following sections present an overview of architectures for securing software. Discussions of design issues associated with software protection architectures can be found in [15] and [16]. These design issues are extended in [17], which discusses similar issues arising from the combination of software security and multiprocessors. The purpose, protection, and design assumptions of security architectures range from custom-designed processors to small jump tables.

The concept of using hardware to protect software was first publicly introduced by Robert Best, who patented the idea of a crypto-microprocessor in the early 1970s [18], [19]. The crypto-microprocessor was a microprocessor outfitted with encryption hardware that could decrypt encrypted memory for a single application using a single key. This concept was again revisited in [20] and was realized as a commercial product in the Maxim DS5002FP [21] secure microcontroller, which could access 128kB of encrypted random access memory (RAM).

Dyad [22], [23] was one of the first architectures to rely on a secure micro-kernel. Dyad examined the necessary protocols and procedures for networked hosts utilizing secure coprocessors. Specific applications of interest were audit trails, electronic transactions, and copy protection. The secure coprocessor was responsible for providing a secure compartment for execution and security functionality not available in the OS. The secure coprocessor was not responsible for enforcing security policies, because the OS was booted securely and verified using the coprocessor. Keys and other secret material were assumed to be stored exclusively on the coprocessor's internal, nonvolatile memory. The Dyad architecture was neither simulated nor implemented.

The Cerium [24] architecture created security by utilizing a tamper-resistant CPU in coordination with a micro-kernel. The micro-kernel, much like the secure kernel used in Dyad, was responsible for providing process separation and was specifically responsible for address space separation, protection during cache-evicts, cryptographic processes (e.g., authentication, encryption/decryption), and certificate creation/signing. The Cerium architecture was neither simulated nor implemented. Similarly, the ChipLock [25] architecture relied on a secure micro-kernel, in addition to processor modifications, to provide integrity and confidentiality for both instructions and data. Unlike Cerium, however, ChipLock placed security functions in hardware and not in the micro-kernel. Another security architecture proposed by Gilmont et al. [26], [27] modified traditional, architectural memory management structures; included a hardware cipher unit and a permanent memory for storing keys; and appended TLB, page registers, and an L1 cache interface to allow for prefetching. This architecture moves security farther toward the outside of the processor but still keeps it within the processor boundary. Gilmont's MMU security architecture fails to address the protection of data and ignores issues related to operation in a multitasking environment, such as shared libraries and different security levels. The architecture was evaluated using simulation and showed overhead costs between 2.5% and 6%.

An important aspect of hardware-based security solutions that modify or redesign processor internals is the effect of adding the additional security hardware to the processor. The additional hardware consumes logic resources that would otherwise be used for cache or memory that are essential to performance. Also, the additional security hardware might also impact timing and other requirements. Together the loss of logic resources and possible timing impact could severely hinder performance. These critical system issues have not been addressed in the design and characterization of existing processor redesign software security solutions.

Other recent efforts at placing security features at the chip boundary include the SafeOps architecture by Zambreno et al. [28], [29] and CODESEAL [30], developed by Gelbart et al. The SafeOps architecture attempted to protect software integrity using a combined compiler and FPGA approach. The compiler examined the executable for sequences of register usage and employs this knowledge to check for integrity and consistency. Additionally, the compiler could embed several nonessential instructions to add to the register usage patterns and perform opcode obfuscation. At execution time, instructions were pushed through the FPGA hardware that compared the register sequence to the register sequence established in the static analysis. If discrepancies were detected, the FPGA hardware halted the processor. SafeOps was evaluated using SimpleScalar [31], which showed average performance degradation to be around 20%. The CODESEAL architecture extended the SafeOps architecture to include instruction decryption and hashing of blocks of information. This architecture was also validated using SimpleScalar and demonstrated performance degradation between 4% and 42%, depending on the type of protections incorporated and the benchmark applications.

## 2.2 Software Solutions

Terra [32] introduced the idea of trusted virtual machine monitors that compartmentalize applications or sets of applications using the virtual machine concept. Terra relied on the existence of a tamper-resistant platform. Obfuscation methods that attempt to complicate program analysis have also been explored.

Wang [33] used a compiler implementation that produced obfuscated programs that made static analysis of the program difficult. The obfuscated software is unobfuscated by the operating system using a runtime interpreter. Duvarney et al. [34] presented the idea of extending traditional executable formats, particularly the Executable Linking Format (ELF) [35], to include security information. Their particular application appended standard ELFs with information that was used to perform runtime monitoring, static monitoring, and program transformation. This allowed security information to be added postcompile if necessary. Another platform under development in the commercial sector under the names of the Next Generation Computing Base (NGSCB) [36], Trusted Computing Platform Alliance, and Palladium utilizes a combination of hardware and software techniques to provide protection from software-based attacks. The NGSCB is not designed to prevent physical attacks on the local machine. NGSCB utilizes a small security coprocessor called a *trusted platform module*, which is attached to the motherboard to store and limit access to sensitive information, e.g., keys and certificates.

## 2.3 Hardware-Based Security Solutions

The first class of hardware solutions for protecting execution flow aims to provide execution flow confidentiality. Zhuang et al. [37] utilized a fairly simple obfuscation mechanism, called a *shuffle buffer*, to complicate memory and address access monitoring as a method for gaining information about an executing application. The shuffle buffer is a circular memory of recently accessed memory blocks. When a memory block is accessed, a block is replaced in the shuffle buffer, and the information being replaced in the buffer is written back to memory. This process attempts to create a “random” memory access pattern. The tracking of memory blocks throughout memory as a method for breaking the obfuscation method was not addressed.

Another similar architecture by Zhuang et al., named HIDE [38], also examined the protection of instruction access sequences, likely obtained (leaked) by monitoring memory or disk accesses, using hardware. The HIDE architecture attempted to make instruction accesses independent of program execution using multilevel prefetch buffers in addition to a hot function set. As hot functions were identified and added to the set, the entire function block was prefetched and stored in the prefetch buffers. As the functions were requested, the prefetch buffers were accessed to retrieve the appropriate portions of the function. This required that the size of the function be added to the executable description such that the prefetch buffer hardware would know the appropriate amount of memory to allocate. Also, the prefetch controller was required to encrypt and decrypt memory for writing blocks back that have been encrypted with a different key such that blocks cannot be tracked as they are moved within memory. Evaluation of the HIDE architecture method was performed using the SimpleScalar [31] simulation platform and the SPEC benchmarks [39]. Performance degradation was shown to be approximately 38%.

Barrantes et al. [40] sought to prevent code injection attacks (that could potentially alter execution flow) by implementing a form of program obfuscation called *instruction set* *randomization*. They proposed the production of a machine-specific executable using a pseudorandom sequence loaded at runtime that is XORed with instructions by an interpreter that resides in the process's virtual address space. The method was evaluated using an emulator and demonstrated a 5% performance degradation. Another approach for preventing invalid code injection was dynamic information flow tracking [41] that utilized the operating system to mark sources of information flow as possibly malevolent. The classification of information was used by a custom processor, which included architectural support at the TLB and registers, to tag information from these possibly malevolent sources. Tags were propagated to the data produced, if the data consumed contained tags. The SimpleScalar [31] was used to evaluate the architecture, which demonstrated modest performance degradation of around 5% and an approximate storage overhead of about 5%. The architecture does, however, require a custom processor and was not implemented.

The next two hardware architectures, basic block signatures [42] and runtime execution monitoring [43], used signature methods for verifying execution flow. Basic block signature verification attempted to prevent execution of unauthorized code by verifying runtime block signatures calculated by hardware with signatures created during the install process. The signatures were created during the install process using secret coefficients and the relative address of the basic block. Because this method verified the integrity of instruction sequences, access to the instruction pipeline hardware was necessary. The method was evaluated using traces from the SPEC [39] benchmarks for an Alpha processor. Analysis of the method compared the number of misses in the basic block signature table to the number of instruction cache misses. The results indicated that if the number of sets in the basic block signature is sufficiently large, basic block signature misses are relatively small compared to the number of cache misses and should not contribute greatly to system performance degradation. Specific performance degradation numbers were not cited.

Runtime execution monitoring protected execution flow integrity by comparing precomputed hashes of basic program blocks after they were loaded into L1 caches. If the hashes calculated on instructions and data passing through to the fetch and load store units did not match the original hashes, an exception was raised. This method required the addition of hash compute and comparison hardware to the processor (behind caches) as well as a hash pointer instruction to the instruction set architecture (ISA). The concept was evaluated using the SimpleScalar simulation framework and SPEC2000 benchmarks. The additional hash information increased application size by 40% to 100%. Performance reduction ranged from 6.4% to 40% depending on hash block size, cache sizes, and memory bandwidth consumed retrieving hashes from memory.

### 2.4 Formal Security Analysis

The goal of formal systems is to verify the operation of a subsystem or system, or interaction within a collection of systems, using formal methods. Formal methods have generally fallen into two categories: theorem proving and model checking [44]. Several generally accepted rules-of-thumb relating to the use and application of formal methods can be found in [45]. In the security domain, formal methods have been utilized to support claims concerning security. One of the initial uses of formal methods in the security domain was cryptographic protocol analysis [46], [47], which uncovered an attack on the Needham-Schroeder exchange protocol [48]. Other variations of formal protocol analysis [49], [50], [51] examine abstract state machines and team automata as methods for examining cryptographic protocols. Lowe [52] made one of the first attempts at automating the protocol analysis process by creating a compiler, called *Casper*, which would generate process algebra descriptions automatically from a more abstract description. The analysis was completed by processing the algebras with a standard model checker.

Butler et al. [53] examined the use of formal methods for validating and modeling trust. Another area of interest is secure systems. With the arrival of the secure coprocessor (as discussed in Section 2.2.1), much effort was focused on formal verification of these systems, such as in [54]. A practical example of formal security analysis of a secure coprocessor, specifically the Infineon SLE88, can be found in [55]. The analysis utilizes a combination of model checking and theorem proving, called *interacting state machines*, that was created and applied to the Infineon SLE88 [56], [57], [58], [59]. Recently, several custom architectures have been proposed as methods to provide security functionality. These systems, like other hardware and/or software [60], are subject to functional verification and are also subject to security verification. For example, the security features of the XOM architecture [61], [62] were analyzed and verified using model checking on a scaled-down model; this was one of the few efforts to apply formal methods to security hardware.

# CHAPTER 3

# INFORMATION FLOW SIGNATURE FOR DATA INTEGRITY

## 3.1 Principles

The Information Flow Signatures (IFS) technique is used to protect the integrity of critical data within an application or operating system. The data may be selected as the highest-priority security variables, such as variables that hold information about user authentication in an SSH application or structures containing information about processes currently running within an operating system.

The crux of the approach is to allow programmers to specify critical data in applications (i.e., data that are important for the application's secure execution) and ensure that the data cannot be corrupted by untrusted third-party libraries or modules. This enforcement is done at runtime using the Information Flow Signatures (IFS) of the critical data. The IFS represents the sequence of legitimate instructions that are allowed to write to the critical data from within the trusted module. If an instruction that is not part of the IFS attempts to write to the critical data, it is considered an attack. Similarly, if an instruction that is part of the IFS writes to the critical data in a different order than determined by the IFS, it is also considered an attack. Thus, only instructions within the IFS can write to the critical data, and only in the order in which they appear in the IFS.

### 3.2 Attack Model

In this work, we consider application-level memory corruption attacks that alter the integrity of an application. This section further explains the above definition.

**Application-level attacks:** The goal of an application-level attack is not to corrupt the behavior or gain control of the whole system (e.g., launch a root shell), but to alter the normal behavior of the attacked application (corrupted output, login with wrong password, etc.). Application-level attacks can be achieved by exploiting vulnerabilities (e.g., buffer overflow or format string), or by changing the code of an application (e.g., trojan). We focus on application-level attacks conducted by memory corruption.

**Memory corruption attacks:** All attacks that corrupt either data or the control flow of an application (or both) by overwriting data or addresses in memory are considered memory corruption attacks. They include, for example, the usual buffer overflow, integer overflow, and format string attacks, but also logic bombs or third-party libraries that modify memory locations, which they are not allowed to modify semantically.

In this work, we assume that the attacker's goal is to alter the integrity of data produced by the application, i.e., to corrupt data produced by the application. Therefore, we do not consider attacks targeting the confidentiality of data, where the goal is to steal information from the application. Furthermore, we assume in this work that the attacker does not want to crash the application. Any attacks that result in a system exception being raised or a system crash are considered to be "detectable" attacks and thus are not considered. In general, the IFS technique described here can be used to prevent data corruption attacks. The aim of the technique is to preserve data integrity rather than its confidentiality. Hence, the technique does not address side-channel attacks. The threat model assumes that the attacker can execute arbitrary code and overwrite program variables stored in both memory and processor registers. We assume all malicious memory accesses are visible to the processor pipeline. Thus, malicious DMA transfers are *not* covered by this threat model. We also assume that the IFS hardware is initialized prior to the attack conditions; thus, the technique protects against runtime attacks, but only if the correct signatures have been loaded into the hardware.

### 3.3 Software Implementation of IFS

In this section we briefly describe the initial design and implementation of the Information Flow Signature Module. In the proposed approach, the portion of the program that manipulates the critical data (i.e., the trusted module) is statically analyzed and instrumented to ensure that runtime modifications of the critical data follow the language-level semantics of the application. This corresponds to static extraction of the backward slice of the critical data [63], and ensures that only the instructions within the slice can modify the critical data, and only in accordance with their execution order as specified by the program code. A third-party module or a memory corruption attack that overwrites the critical data or violates the established dependencies in the slice is detected. Because the approach ensures that information flows to the critical data in accordance with the program source code, we call it the *Information Flow Signatures (IFS)* technique.

**Invariants:** The instrumentation added by the IFS technique ensures that the following invariants are maintained at runtime.

1. Only the instructions that are allowed to write to data operands in the backward slice of the critical data (according to the static data dependencies) in fact do so.

2. The instructions in the backward slices of the critical data are executed in the order of their occurrence along a valid acyclic path in the program.

3. Either all the instructions in the backward slice along a control path are executed, or no instruction along the path is executed.

### *3.3.1 Phase 0: Identification of Critical Data (Carried Out by the Programmer)*

The critical data in a program can refer to both program variables (i.e., local and global variables) and dynamically allocated memory objects on the heap. The programmer identifies critical data in the program through annotations in the source code. In the case of program variables (local or global), the annotations are placed on the definitions of the variables. In the case of dynamically allocated memory objects, the annotations are placed on the allocation sites (i.e., the instruction that makes the function call) in the program (e.g., calls to *malloc)*. It is assumed that all objects allocated at an annotated *malloc* call are critical.

### *3.3.2 Phase 1: Static Analysis (Carried Out by Our Enhancements to the Compiler)*

1. Extract intra-procedural backward slice of the critical data by identifying all instructions within a function in the program that can influence critical data. It is assumed that the function that manipulates the critical data is trusted, and its source code is available for the analysis.

2. For each instruction in the backward slice, insert an encoding operation after the instruction and pass the value computed by the instruction as an argument to the encoding operation.

3. Before every use of an operand that has been encoded in the backward slice, insert a call to the decoding operation and pass it the value returned by the encoding operation.

4. Generate the sequences of instructions for each acyclic control-flow path in the function. Do this for each function in the program.

5. Since we consider intra-procedural paths, add instrumentation functions at the beginning and end of function calls in order to push and pop the current state of the function's state machine on to a runtime stack (see phase 3). (State machines track the sequence of calls to the encoding functions and check if the program control flow is valid.)

### *3.3.3 Phase 2: Code Generation (Carried Out by Custom Programs)*

6. Generate finite-state machines to encode the sequences of calls to the encoding operations within a function for all control paths identified in step 5. Mark the final state of each state machine as an accepting state.

7. Generate the encoding and decoding operations to check the data-values of the program as it executes (see phase 3). Also, check the validity of the program control-flow using the state machines derived in step 6.

8. Track the runtime path based on the state machines generated in step 6. If the path does not correspond to a valid path, raise an alarm and stop the program.

9. Encode data values depending on where the encoding operation is called in the original program. Check the value for consistency by decoding it before its use, i.e., check if the decoded value matches the encoded operand. If the value is inconsistent, raise an alarm and stop the program.

**Slicing Algorithm:** The backward slices of the critical data are computed on a path-specific basis, i.e., each execution path in the function is considered separately for slice extraction. This is based on our earlier work on extracting backward slices for detecting transient errors in programs [64].

**State machines:** The state machines track the sequence of calls to the encoding functions and check if the program control flow is valid. The state is tracked on a per-function basis, since we consider only intra-procedural slices. A separate stack is maintained at runtime to push and pop the current state of the state machine (for the function) at the beginning and end of function calls. At the entry point to a function, the corresponding state machine is reset to the start state. Similarly, the state-machine state is checked just before the function returns to ensure that the state machine is in an *accepting state*, i.e., the state machine has accepted the observed sequence of encoding calls (invariant 3). Finally, we check that every encoding call executed by the program corresponds to a valid state transition from the current state of the state machine (invariant 2).

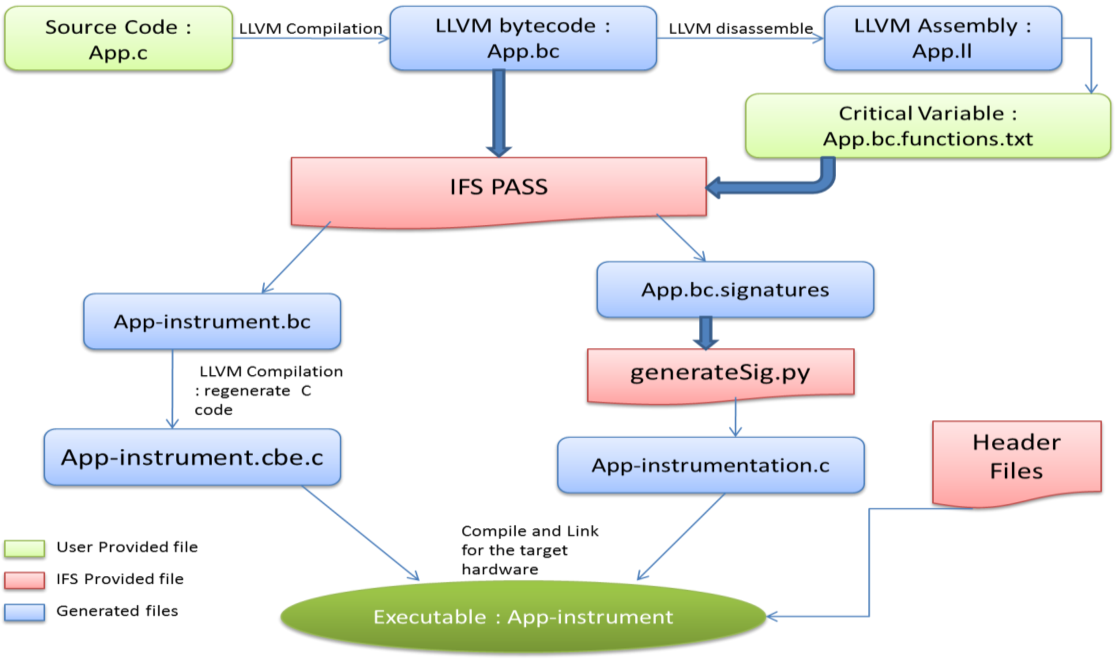


Fig. 3.1: Tool chain for software implementation of IFS

Figure 3.1 describes the tool chain for implementing IFS checks in software. There are a number of limitations and shortcomings in this approach which we overcome with our new proposed method, for which a prototype has been developed to demonstrate efficacy. The major limitations come from the preprocessing method of the code, which is based on compiler static analysis; it is discussed in detail in Chapter 4, where the new analysis method based on model checking and error injection has been described and contrasted with the compiler-based analysis. The approach involves major performance degradation and vulnerability due to the inherent problems of handling everything (encoding and decoding operations) in software. The instrumentation is done at a level of intermediate code that is still high-level C code, hence providing a vulnerable window for attackers to attack the code successfully without being detected. By implementing the IFS module in hardware, we reduce this vulnerable window to 0 for any practical purpose and also have a huge performance boost, which is described in Chapter 6 (in the discussion on advantages of hardware-based IFS) and Chapter 8 (performance results).

### 3.4 New Approach and Algorithm

*Critical data* is defined as any variable or memory object that, if corrupted by an attacker, can lead to security compromise of the application. The programmer identifies critical data through type annotations in the code. The approach consists of two phases:

- A compile time phase to extract the trusted signature of the application source code. This is done completely in software. As described (Fig. 3.2), the application source code has to go through SymPLFIED analysis to extract the application signature, which is used to instrument the application source code.
- A runtime phase, in which the IFS module enforces the integrity of critical data using the trusted signature of the application configured in the hardware itself. It does so at the hardware level. The instrumented source code is loaded onto the processor, which, upon execution, configures the IFS hardware with the application’s instruction and data signature. Later, during the course of execution, the IFS module enforces the integrity of critical data using the signature initially loaded.

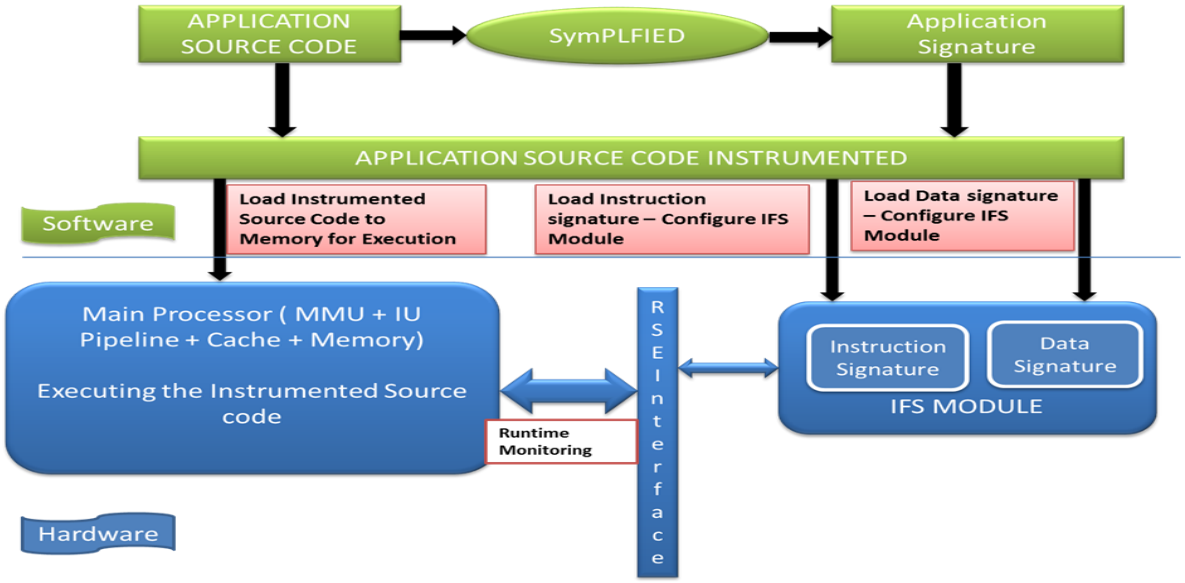

APPLICATION SOURCE CODE
SymPLFIED
Application Signature
APPLICATION SOURCE CODE INSTRUMENTED
Software
Load Instrumented Source Code to Memory for Execution
Load Instruction signature – Configure IFS Module
Load Data signature – Configure IFS Module
Main Processor ( MMU + IU Pipeline + Cache + Memory)
Executing the Instrumented Source code
Runtime Monitoring
RSE Interface
Instruction Signature
Data Signature
IFS MODULE
Hardware

Fig. 3.2: Conceptual design flow: IFS

### *3.4.1 Compile Time Phase*

We leverage a formal tool, SymPLFIED [14], to find the minimum yet exhaustive set of critical data locations that one needs to protect. SymPLFIED uses symbolic execution and fault injection to comprehensively enumerate all possible memory errors that can be corrupted by malicious instructions to cause a program failure.

SymPLFIED runs on top of the Maude model checker and models applications' execution at the assembly level using rewriting logic. SymPLFIED-supported symbolic fault injection introduces a single error per execution in one memory location and propagates the error's consequences using symbolic execution. The results from the symbolic fault injections are then used to determine the complete set of memory locations and the corresponding critical stores that need to be protected. This set of memory locations and critical store instructions forms the signature for an application. The original program is instrumented to load the signature in the IFS module during the runtime. More details on the compilation and extraction of signatures for an application's source code are provided in the next two chapters, which address the SymPLFIED analysis and loading of signatures to the hardware module.

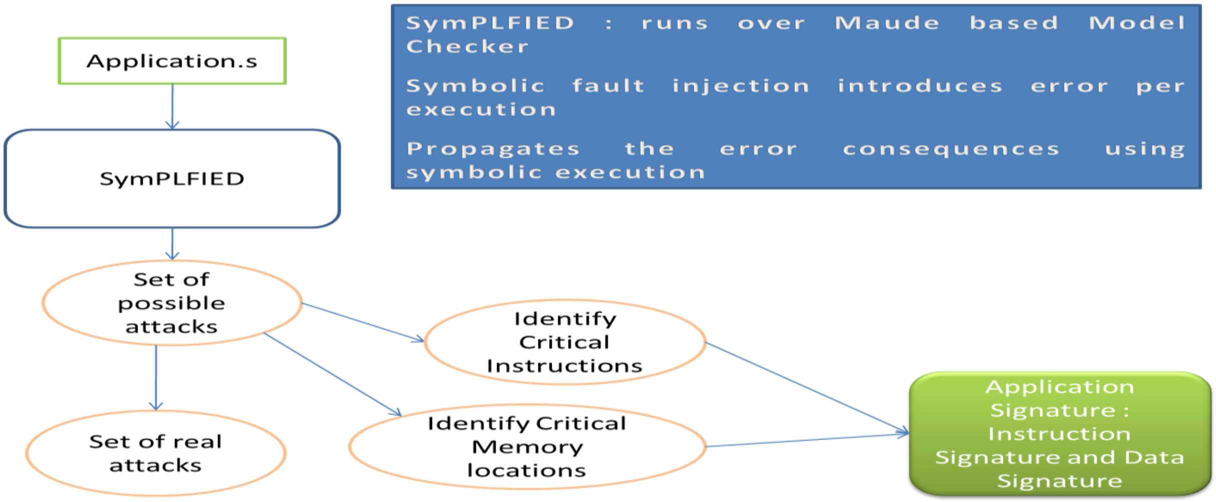

SymPLFIED : runs over Maude based Model Checker
Symbolic fault injection introduces error per execution
Propagates the error consequences using symbolic execution
Application.s
SymPLFIED
Set of possible attacks
Identify Critical Instructions
Set of real attacks
Identify Critical Memory locations
Application Signature : Instruction Signature and Data Signature

Fig. 3.3: Proposed compilation phase framework

Figure 3.3 shows the conceptual flow of the proposed compilation phase framework. SymPLFIED [14] exhaustively simulates and analyzes the processor state at instruction-level granularity and hence helps in identifying all possible sources of vulnerabilities that can be exploited to achieve the attacker's goal, namely, compromise of the integrity of critical data being computed by the application. The set of real-world attacks is a subset of all the possible attack scenarios analyzed, and hence we claim that the signature extracted by the proposed framework (by analyzing and parsing the successful attack cases) can theoretically provide 100% coverage. The obtained coverage is verified with simulations and error injection experiments discussed later in this thesis.

### *3.4.2 Runtime Phase*

The instrumented code is loaded onto the processor. The coprocessor instructions (in the SPARC instruction set) are modified to act as an interface between the application running on the main core and the signature-checking hardware. Hence, the checking is done in parallel with minimum performance overhead due to instrumented code. The instrumented code also configures the IFS hardware to load the corresponding critical instruction and data signatures during the initialization of the program. The IFS module then enforces the integrity of the critical data to be maintained during the execution of the application.

The instrumentation added by the IFS technique ensures that the following invariants are maintained at runtime.

1. Only the instructions that are allowed to write to data operands in the backward slice of the critical data (according to the static data dependencies) in fact do so.

2. The instructions in the backward slices of the critical data are executed in the order of their occurrence along a valid acyclic path in the program.

# CHAPTER 4

# SYMPLFIED

In this chapter we discuss the formal model-checker-based tool called SymPLFIED and how it is extended to support the analysis required for extracting application-specific signatures to configure the Information Flow Signature module and hence monitor the application during the execution of the instrumented source code.

SymPLFIED [14] is a program-level formal framework that runs on top of the Maude model checker. It has been designed to enable reasoning about application vulnerabilities using symbolic execution and model checking. We leverage the symbolic fault-injection capabilities of SymPLFIED along with the flexible design of the machine model in order to design, model, and verify detection mechanisms. Critical variables along with associated critical code are identified using SymPLFIED's symbolic fault injector.

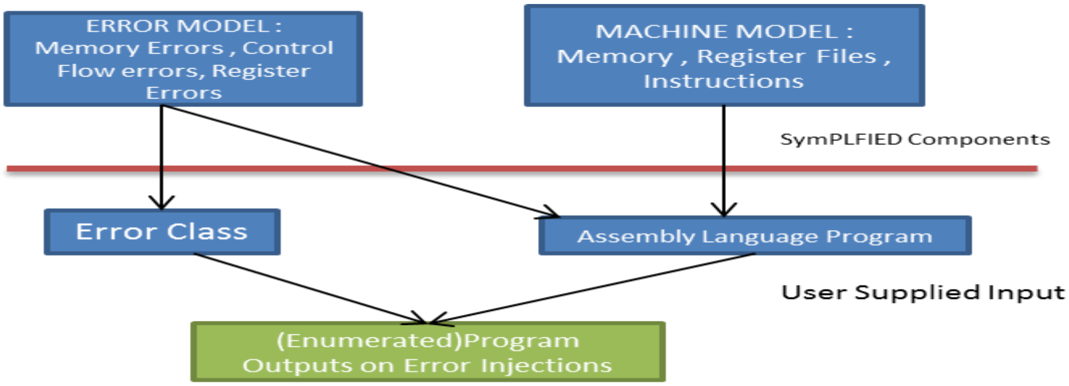

ERROR MODEL : Memory Errors , Control Flow errors, Register Errors
MACHINE MODEL : Memory , Register Files , Instructions
SymPLFIED Components
Error Class
Assembly Language Program
User Supplied Input
(Enumerated)Program Outputs on Error Injections

Fig. 4.1: Conceptual design flow of SymPLFIED

SymPLFIED is implemented using rewriting logic. Equations are used to model deterministic actions such as the program execution or memory and register lookups. Rules are used to introduce nondeterministic transitions when errors are injected. Figure 4.1 describes the conceptual design components used in the Maude-based model checker SymPLFIED. Error model describes the class of errors to be analyzed; "machine model" describes the architectural state of the hardware processor, and the assembly of the source code is the input to the processor.

A machine state describes the states of the registers, memory, program counter, etc., during the execution of an application. The whole application's execution consists of going from one machine state to another through transitions. In other words, transitions modeling the fault-free execution of an application are written with equations, and transitions involving errors are written with rules.

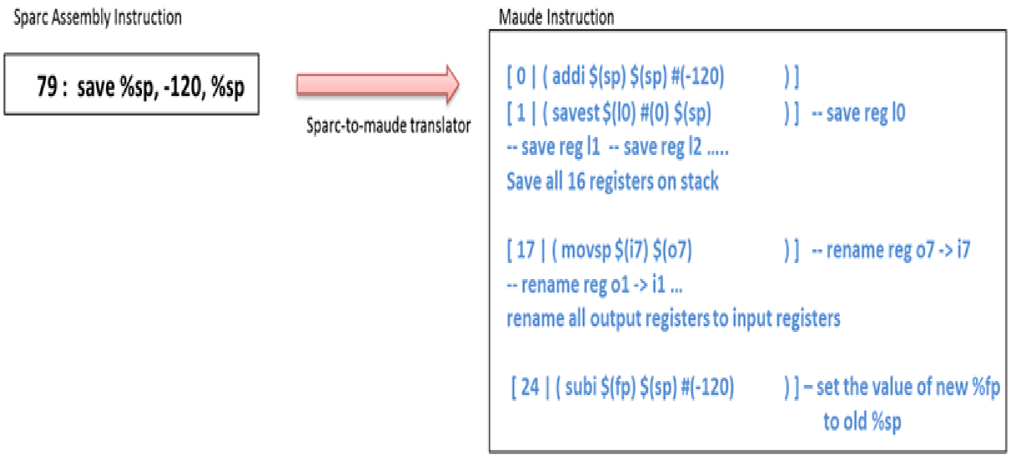
### 4.1 SymPLFIED: Extended Implementation for SPARC

The existing tool SymPLFIED emulated MIPS ISA. We extended the tool to support the SPARC instruction set architecture because our experimental platform is based on the open-source Leon3 processor, to which we attached the custom-designed IFS hardware to support the runtime error checking of the application.

The extension of the existing tool consisted of two implementation phases:

1. We modified the front end of the SymPLFIED tool to support all the instruction formats of SPARC V8. The execution behavior of the new instructions, which determine the states of the application and the processor after a particular instruction is emulated, was implemented as re-writing rules for Maude. The dependency and define list for the SPARC instructions that define the resulting values (on execution), the initial state (of the processor), and the dependency of an instruction on registers and memory locations was re-implemented. For the new Maude instructions defined, we implemented the state transition functions that would define the subsequent states after the execution of a particular instruction by adding rules to the constraint checker for Maude.
2. We implemented a translator that interprets the SPARC V8 assembly instructions and maps them to the corresponding Maude formalism to enable symbolic execution of instructions. A Maude instruction representation mimics the behavior of the corresponding assembly instruction in SPARC. To incorporate all the unique features of the SPARC-based Leon3 architecture, a single instruction can map to multiple Maude instructions. Unlike MIPS, SPARC uses register windows to minimize access to memory. For example, to support function calling conventions, for instructions like *save* and *restore*, Leon 3 passes the top 6 arguments through the output registers. These registers overlap with the set of input registers for the function called. Consequently, before the function is called, a *save* instruction is executed to increment the register window where the new sets of input windows overlap with the previous set of output windows. At the same time, in the stack frame layout, the top 64 bytes are reserved for pushing the register values onto the stack in case of register spilling, which happens once the processor runs out of new sets of register windows. To mimic this behavior, we intentionally push the

registers onto the stack every time a save instruction is executed (as described in Figure 4.2) and reload the registers when the *restore* instruction is executed, which in the processor decrements the window pointer and returns the access to the previous set of register windows. Pushing registers onto the stack frame, even when the processor has available register windows that can be used, increases the memory locations that must be protected, which is fine because in an actual system (with kernel and other processes running simultaneously), the resource usage for a processor at any particular time is unpredictable. Thus the methodology proposed in this thesis should abstract away such concerns while the application’s source code is being instrumented.

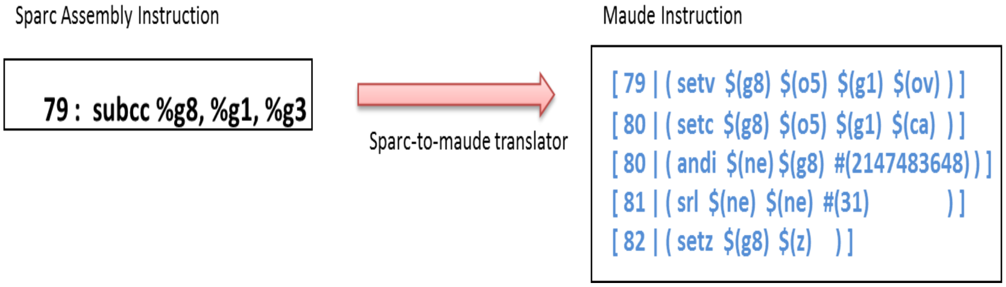


```
79 : save %sp, -120, %sp
```

```
[ 0 | ( addi $(sp) $(sp) #(-120)        ) ]
[ 1 | ( savest $(l0) #(0) $(sp)         ) ]  -- save reg l0
-- save reg l1  -- save reg l2 .....
Save all 16 registers on stack

[ 17 | ( movsp $(i7) $(o7)              ) ]  -- rename reg o7 -> i7
-- rename reg o1 -> i1 ...
rename all output registers to input registers

 [ 24 | ( subi $(fp) $(sp) #(-120)      ) ] – set the value of new %fp
                                                 to old %sp
```

Fig. 4.2: Example of mapping a single SPARC instruction to multiple Maude instructions to accurately simulate the architectural state of the processor

Another case that requires mapping of a single instruction to multiple Maude instructions is the instruction that sets condition code registers. Unlike the MIPS architecture, which does not have condition code registers and uses *slt* instructions and a comparator to make branching decisions, the SPARC architecture uses four condition code registers to make branching decisions. Mapping of a complex instruction to multiple Maude instructions gives us better granularity in cases where we can set breakpoints, even within an atomic instruction, and makes it easier to debug and monitor the architectural state of the processor. Also, it offers a better way to extend Maude to different instruction set architectures, as opposed to redesigning complex Maude instructions to map to specific complex instructions, which would increase the verification complexity of the model checker. While mapping a single instruction to multiple instructions in cases like the one in Figure 4.3, one has to be careful to avoid including any redundant memory locations as a part of the backward slice.



```
79 :  subcc %g8, %g1, %g3
```

```
[ 79 | ( setv  $(g8)  $(o5)  $(g1)  $(ov) ) ]
[ 80 | ( setc  $(g8)  $(o5)  $(g1)  $(ca)  ) ]
[ 80 | ( andi  $(ne) $(g8)  #(2147483648) ) ]
[ 81 | ( srl  $(ne)  $(ne)  #(31)           ) ]
[ 82 | ( setz  $(g8)  $(z)    ) ]
```

Fig. 4.3: Example SPARC instruction mapped to multiple Maude instructions

### 4.2 Stack Memory Management in SymPLFIED

In order to emulate the SPARC architecture behavior, the *save* instruction is translated to multiple *save-store* instructions in Maude, which push the set of input registers onto the stack space allocated to the called subroutine. Similarly, on the *restore* instruction, we pop the values from the stack into the corresponding registers in the reverse order before tearing down the stack frame for the called function. The stack frame layout after every subroutine call is shown in Fig. 4.4.

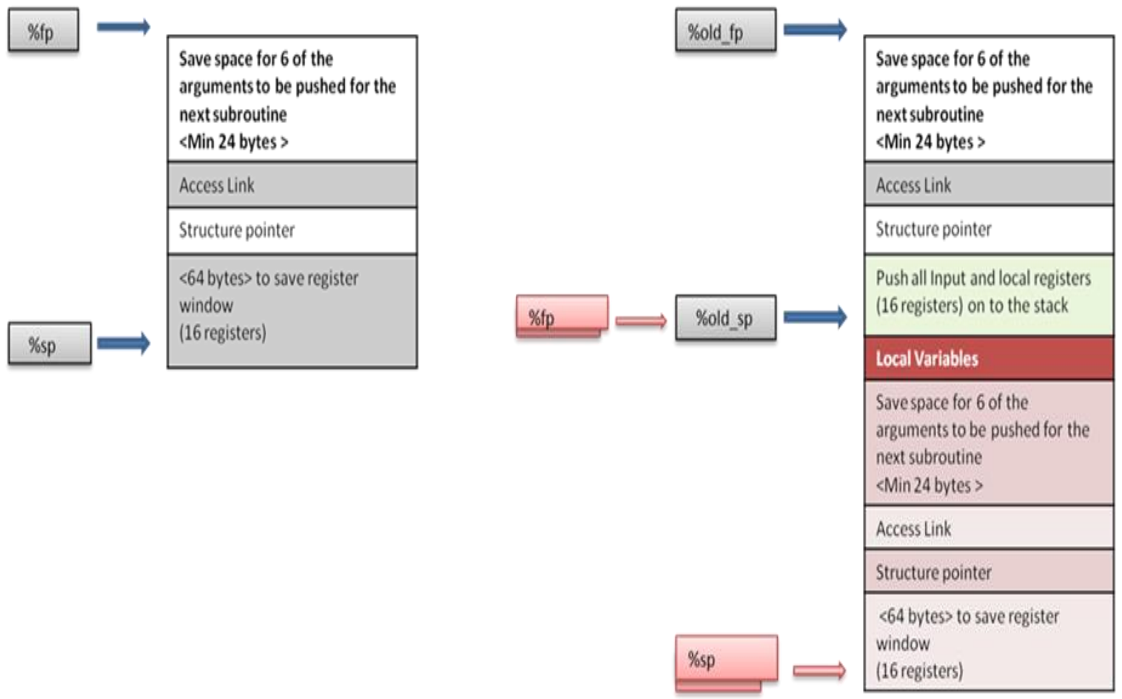

%fp
%sp
Save space for 6 of the arguments to be pushed for the next subroutine <Min 24 bytes >
Access Link
Structure pointer
<64 bytes> to save register window (16 registers)
%old_fp
%fp
%old_sp
%sp
Save space for 6 of the arguments to be pushed for the next subroutine <Min 24 bytes >
Access Link
Structure pointer
Push all Input and local registers (16 registers) on to the stack
Local Variables
Save space for 6 of the arguments to be pushed for the next subroutine <Min 24 bytes >
Access Link
Structure pointer
<64 bytes> to save register window (16 registers)

Fig. 4.4: Stack memory frame layout before and after subroutine call in SymPLFIED

### 4.3 Register File Management for SPARC Architecture in SymPLFIED

When a function call is made, all the input and local registers are pushed onto the stack frame. The save instruction, which is mapped to multiple movsp instructions to emulate a behavior similar to the overlapping of registers, is executed; i.e., the values in the output registers are moved into the input registers. Similarly, on a restore instruction, the values of the input registers are moved to the output registers, and values of the frame pointer and stack pointer register are restored to tear down the called function’s stack frame. Figure 4.5 describes register overlapping before and after a save instruction is executed on a function call.

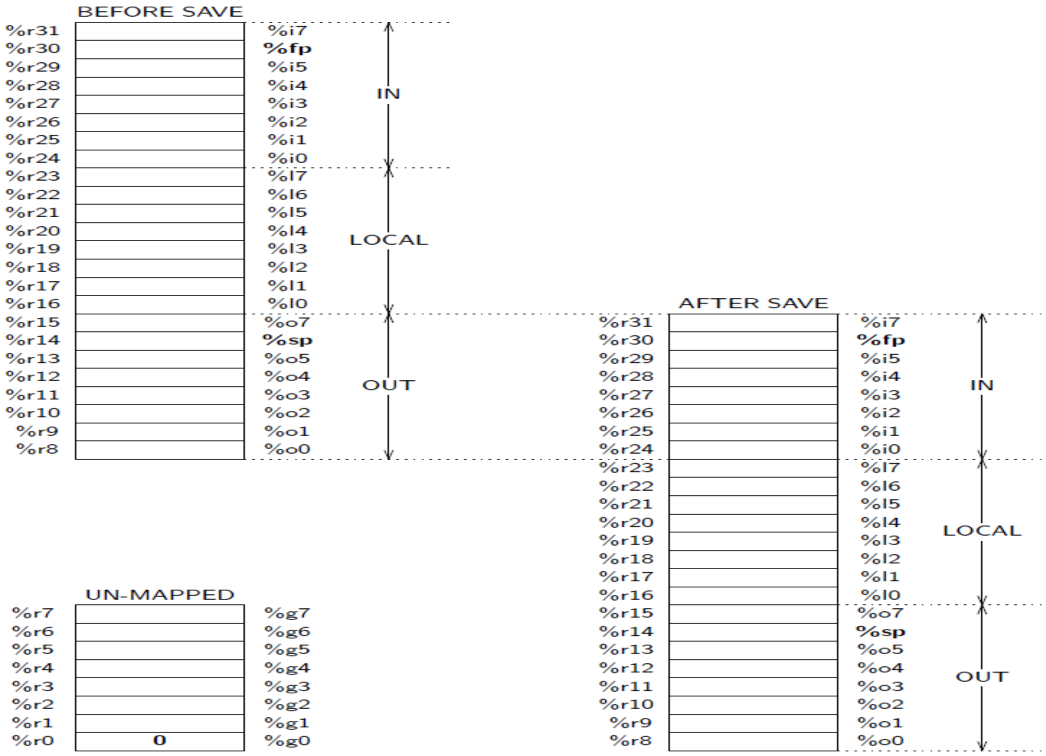


Fig. 4.5: Register overlapping in a SPARC ISA-based architecture

That behavior is not exactly what SPARC processors do (because SPARC processors push the oldest set of used registers onto the stack frame). But the current design does not introduce any false negatives (i.e., it does not compromise an application's critical signature, and hence no critical memory locations are eliminated) because register spilling is forced on every function call. In a very conservative analysis, the bottom 64 bytes of every new stack frame allocated must be included in the application signature and then invalidated as soon as the stack frame is torn down. In most critical applications, objects allocated on stack memory usually are associated with temporary calculations and are not that critical to the application's security. If an object is indeed a critical variable, then it is guaranteed to be included in the application's signature until the function stack frame is alive; after returning to the caller, the stack frame (for the called function) is free to be used. In usual programming practice, data are never accessed (by reference) from a lost stack frame; hence the correct design is to prevent such memory locations from being part of the application’s signature.

## 4.4 Error Injection

The error injection mechanism mimics the behavior of existing fault injectors such as NFTAPE [65]. SymPLFIED uses symbolic fault injection. A breakpoint is set at a given application’s execution point. When the breakpoint is reached, the *err* symbol is injected at one memory location. Then, the application’s execution resumes propagating the error.

### *4.4.1 Error Injection Mechanism*

At each run, the user is able to specify where to inject the symbolic error (into the register, memory or the program counter). In this technique, as we wish to model memory corruption attacks, we only consider memory errors. The user can also specify when to inject the error by setting breakpoints. By default, SymPLFIED exhaustively and successively injects errors at every available memory location and at every possible breakpoint. Figure 4.6 illustrates the exhaustive fault injection process. Within each state, $a_i$ denotes memory location $i$ and $a_i = err$ denotes the fact that the memory location $a_i$ contains the error symbol. Note that there is at most one error injected on each path of the state diagram (Figure 4.6).

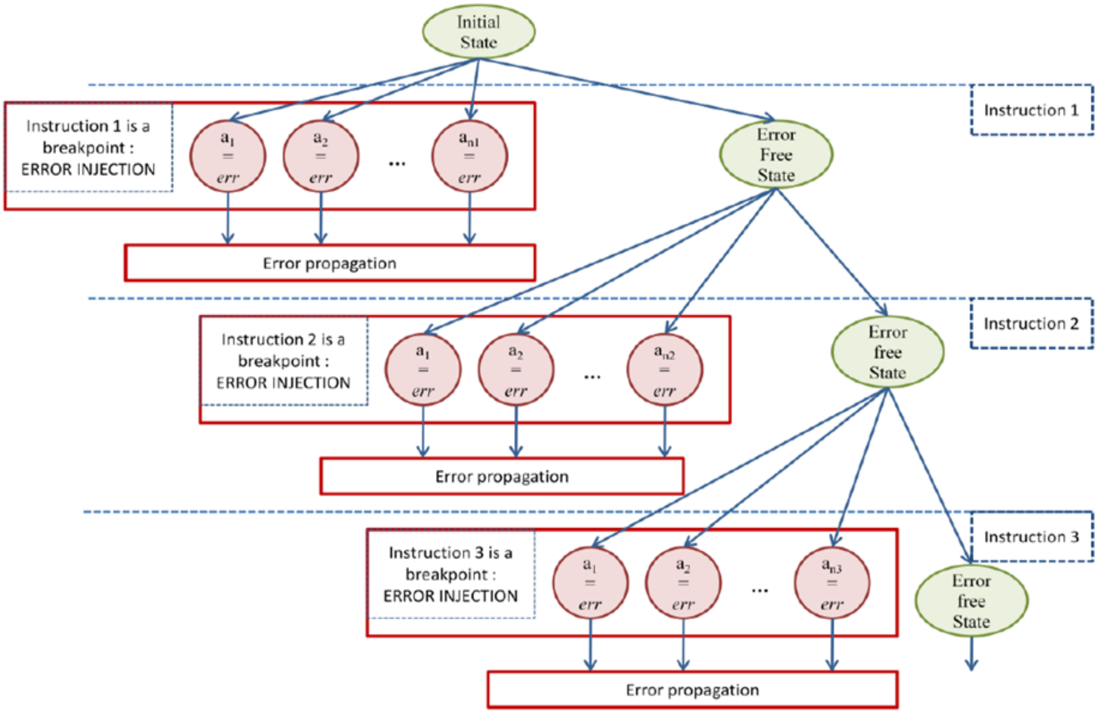
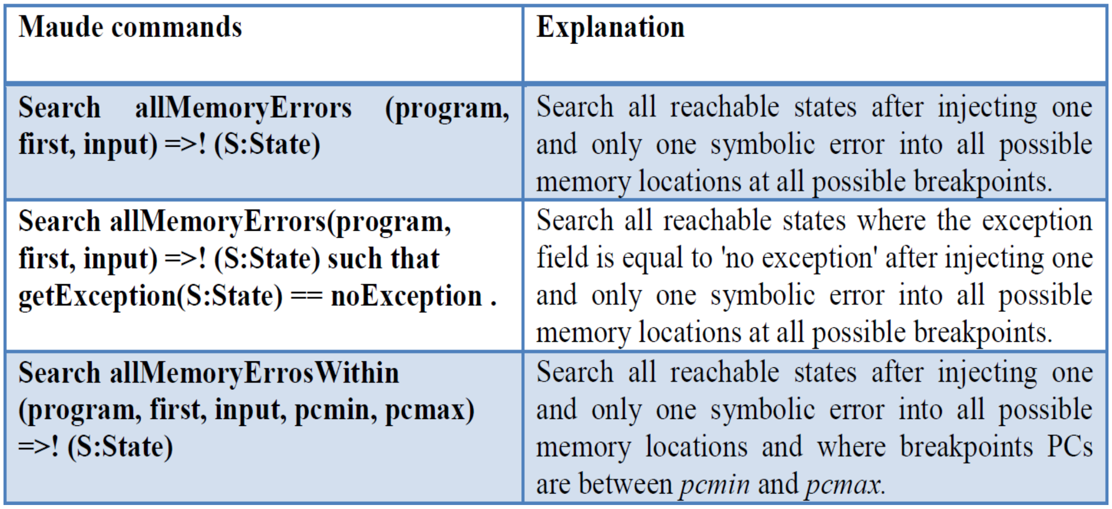


Fig. 4.6: State diagram of error injection of all memory addresses at all possible breakpoints

The initial state is a state in which no instruction has been executed yet. After instruction *1* is executed, *n1 + 1* states can be reached from the initial state (*n1* being the number of memory locations available). In the *n1* state, a unique symbolic error *err* is injected. In the first state, an error is injected into memory location 1, and in the *i*th state, an error is injected into memory location *i*. Once injected, each error is propagated independently from the others. The *(n1 + 1)*th state is an error-free state from which new states are forked after the execution of the second instruction. Table 4.1 gives the injection commands defined in SymPLFIED. The “Search” command defined in Maude enables us to explore all reachable states from the initial state given the specified injections.

Table 4.1: Basic commands to inject memory errors in SymPLFIED



| **Maude commands** | **Explanation** |
|---|---|
| **Search allMemoryErrors (program, first, input) =>! (S:State)** | Search all reachable states after injecting one and only one symbolic error into all possible memory locations at all possible breakpoints. |
| **Search allMemoryErrors(program, first, input) =>! (S:State) such that getException(S:State) == noException .** | Search all reachable states where the exception field is equal to 'no exception' after injecting one and only one symbolic error into all possible memory locations at all possible breakpoints. |
| **Search allMemoryErrosWithin (program, first, input, pcmin, pcmax) =>! (S:State)** | Search all reachable states after injecting one and only one symbolic error into all possible memory locations and where breakpoints PCs are between *pcmin* and *pcmax.* |

### *4.4.2 Error Propagation*

Once the error has been injected into the program, the error is propagated. Every operation involving *err* will return error expressions merging *err* symbols (e.g., *err + 2 = err*, *err + err = err*).

The interesting part of the error propagation is the way the tool handles branching or comparison involving the *err* symbol. When branching and comparison are not involved, merging expressions into *err* is sufficient. But if we need to evaluate a predicate in order to determine how to branch, merging of expressions into *err* is no longer possible. SymPLFIED uses rules to fork the execution when evaluating a predicate that involves the *err* symbol; in other words, a predicate containing an *err* symbol is evaluated to both true and false.

### *4.4.3 SymPLFIED Outputs*

SymPLFIED logs the result of the error injection experiments. The results enumerate all the cases that match the search criteria, for example, attacks that were successful in modifying the critical variable. The logged results describe the precise instructions and the memory locations at which the errors were injected; hence, one can identify the critical instructions and the corresponding memory locations with which they are associated. The critical instruction signature and data signature for the application are formed by the set of instructions and memory locations that upon error injection were successful in exploiting the application’s vulnerability, for example by changing the system password in the SSH program. The check instructions encoding this information are used to instrument the original application, which, upon execution, configures the IFS hardware module and enforces runtime checking.

SymPLFIED considers the effect of all possible transient hardware errors on computation. Our focus in this thesis is on memory-corruption-related errors when a program is being executed under a specific input. SymPLFIED uses symbolic execution and model-checking to exhaustively reason about the effect of the error on the program. The key innovation of SymPLFIED is that it combines an entire set of errors into a single abstract class and symbolically reasons about the effects of the error class as a whole. This grouping effectively collapses into a single state the entire set of errors, in turn greatly enhancing the scalability of SymPLFIED compared to exhaustive fault injection. However, the scalability is obtained at the cost of accuracy, as this abstraction can lead to false positives, i.e., erroneous outcomes that occur in the model but not in the real system. However, the loss in accuracy is acceptable in practice, as the detectors can be conservatively overdesigned to protect against a few false positives.

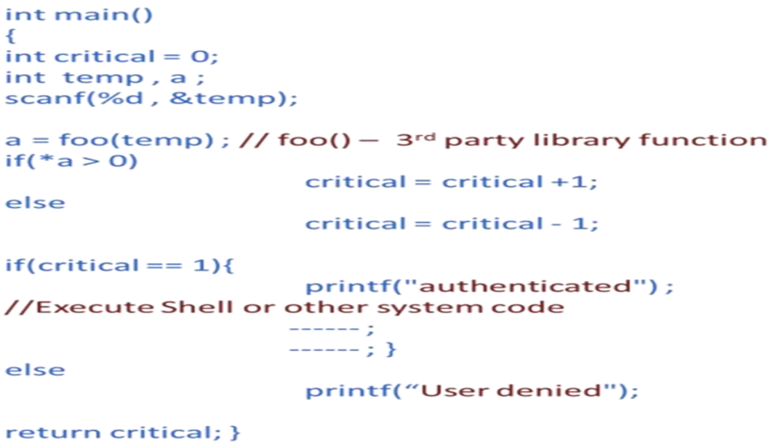
### 4.5 SymPLFIED vs. Compiler Static Analysis

Compared to previous work (namely software implementation, as discussed in Chapter 3), computation of the backward slice of a critical variable in the program based on static analysis has certain limitations, such as intra-procedural slicing, conditional dependencies of a control flow path that computes critical variables and difficulties in Pointer analysis. The proposed method of having exhaustive symbolic fault injection running over model checker overcomes most of those limitations, as discussed below.

1. **Intra-procedural slicing:** The IFS technique considers only intra-procedural slices; i.e., it truncates the slice at the beginning of functions. Hence, any corruption of the slice prior to the function call will not be detected by the technique. There are different ways of mitigating the impact of this using function in-lining, which is not practical for large pieces of code. The user can choose critical variables in each function, but again that is not practical for millions of lines of application software. From the perspective of SymPLFIED analysis, the function boundaries are transparent. The analysis is done at the abstraction level of instructions that modify memory locations. Function boundaries are transparent to such a design approach and hence obviate the limitation presented by the static compiler analysis methodology.
2. Annotation of critical instructions at the RTL level gives finer granularity and precision than the backward slices given by static analysis of compiler-generated intermediate code representation of an application's source code. Even the intermediate representation used for analysis of routines is a relatively high level of code and does not exactly represent the change in the architectural state of the processor during execution of an application. To ensure the integrity of critical data, it is important that no window of vulnerability be

left open during runtime checking; hence, it is better to have more fine-tuned analysis, such as checking at the instruction level. The RTL-level-checking approach described in this thesis presents a better picture of the architectural state of a processor and is more suited to solving the problem of eliminating all windows of vulnerability.

3. The analysis of an application or a part of the application that is the trusted module using SymPLFIED is complete in the sense that it includes all conditional dependencies for which a static analysis using the compilers cannot account. The control flow fault is the backdoor for most of the crafted attacks and hence requires a method that can include all possible dependencies for a critical variable.



```
int main()
{
int critical = 0;
int  temp , a ;
scanf(%d , &temp);

a = foo(temp) ; // foo() —  3rd party library function
if(*a > 0)
                        critical = critical +1;
else
                        critical = critical - 1;

if(critical == 1){
                        printf("authenticated") ;
//Execute Shell or other system code
                       ------ ;
                       ------ ; }
else
                        printf(“User denied");

return critical; }
```

Fig 4.7: Example code exposing the limitation of compiler static analysis approach

In the piece of code in Figure 4.7, neither the value of variable *a* nor the memory location of the variable *a* is a part of the backward slice computed for the critical variable. Hence it is easy to craft an attack by corrupting the memory location *a*. Such dependencies are not represented in the backward slice generated by compiler analysis techniques.

4. Another big limitation of the backward slices produced using compiler techniques is the level of indirection of pointers supported (points-to analysis) and pointer alias analysis.

Even a compiler with advanced pointer analysis capabilities, like IMPACT or LLVM, cannot guarantee memory safety with a given backward slice.

- **Pointer-alias analysis**: *(critical) and *(xyz_variable) (where xyz_variable is any random pointer variable) can both point to the same critical object, but there can be cases where xyz_variable is never a part of the backward slice or even a part of the source code (e.g., xyz_variable is a part of the dynamically loaded shared library), as seen in the example in Figure 4.7.
- **Points-to analysis**: The variables (a, critical: in the piece of code in Figure 4.7) can have multiple levels of indirection (which is very common in industry software written in C++). One can compromise the integrity of the system by corrupting the memory location at any level of indirection of those pointers; the pointers may or may not be a part of the backward slice, depending on the level of indirection supported by the compilers.

We overcome all these limitations because symbolic fault injection gives us all the possible memory locations that can compromise an application’s integrity, and hence we can annotate all the critical instructions that are accessing a particular memory location and need to be protected. All the memory locations that we need to protect are determined by SymPLFIED analysis. Also, the set of attack scenarios simulated by SymPLFIED is a superset of the real-world attacks.

# CHAPTER 5

# RELIABILITY AND SECURITY ENGINE (RSE)

The RSE used to evaluate the IFS technique is implemented on the Leon 3 soft processor [6]. This chapter introduces the Leon 3 processor we used as the RSE development platform, in this thesis, and describes the RSE implementation for the Leon 3, including the details of the CHK instruction used by an application to communicate with RSE modules.

## 5.1 The Leon 3 Processor

The Gaisler Research Leon 3 processor was chosen as the new RSE development platform because of its clear and concise documentation, easy-to-use tools, preconfigured FPGA synthesis examples, operating system support, large development community, and active mailing list. The GRLIB peripheral library included with this processor has an Ethernet controller, video graphics array (VGA) controller, several types of memory controllers, and a significant number of device controllers and interfaces. The AMBA bus standard is used for all peripherals in GRLIB. The hardware description of the entire GRLIB was written in very high-speed integrated circuit hardware description language (VHDL) using a special two-process state machine format developed at Gaisler research [66]. The Leon 3 processor implements the SPARC v8 instruction set. It is a seven-stage pipelined processor with configurable caches, a floating point unit. The seven stages of the processor and their functionalities are as follows:

- **Fetch:** The first stage of the processor controls the instruction cache and does program counter calculations, including loading of branch addresses and changing of addresses for exception handling.
- **Decode:** This stage decodes the instructions into microcode to be used by the processor as control signals, determines branch instructions, and loads the addresses for register access.
- **Register file access:** Operands to each instruction are loaded by the Register File Access stage. This stage handles register bypassing from the execution unit and retrieves and pads immediate values from the instruction to be used as operands.
- **Execute:** Arithmetic and logical operations are processed in the Execute stage. Floating point instructions and addressing for the data cache and branch instructions are calculated are also calculated at this time.
- **Memory:** The Memory stage generates control signals for the data cache and generates the appropriate interrupt control signals for the following stage.
- **Exception:** Traps for interrupts and exceptions are generated in the Exception stage. The final address calculations are made here before they are sent back to the Fetch stage. This stage also generates the signals used by the Writeback stage to control the register file.
- **Writeback:** Results of instructions are written back to the register file in this stage.

### 5.2 The Reliability and Security Engine

The Reliability and Security Engine (RSE) [6] is a framework that provides a standard interface between a processor pipeline and hardware modules that implement reliability and security services for the executing application. Figure 5.1 provides a block diagram of the RSE connected to the Gaisler Research Leon 3 open-source VHDL processor pipeline. The modules are running alongside the host processor, monitoring the behavior of the executing application. Although shown with the Leon 3, the RSE framework is general enough to be integrated with any general-purpose processor. Additionally, the RSE is nonintrusive to the main processor pipeline, as it only needs to monitor execution behavior. It does so by inserting probes into the pipeline of the host processor that continuously transfers selected state information to the RSE modules.

Reconfigurable hardware slices similar to those in an FPGA may be used to implement the modules, allowing for instantiation of the desired modules based on the requirements of the current application. Alternatively, a module can be implemented as an ASIC IP core, which is imported by a processor designer to suit the demands imposed by the customers and the market. In our design, we override the SPARC v8 instruction set architecture’s co-processor operation instruction (CPOP1), converting it into a CHK instruction. This CHK instruction is used for communication between an instrumented application and the RSE modules. It is ignored by the main pipeline of the processor and considered a NOP. The CHK instructions are uniquely identified to specify the modules for which they are intended. For example, during application initialization, the Information Flow Signatures technique uses CHK instructions to convey the signatures of the trusted instructions and critical data that are required to enforce the checks in hardware.

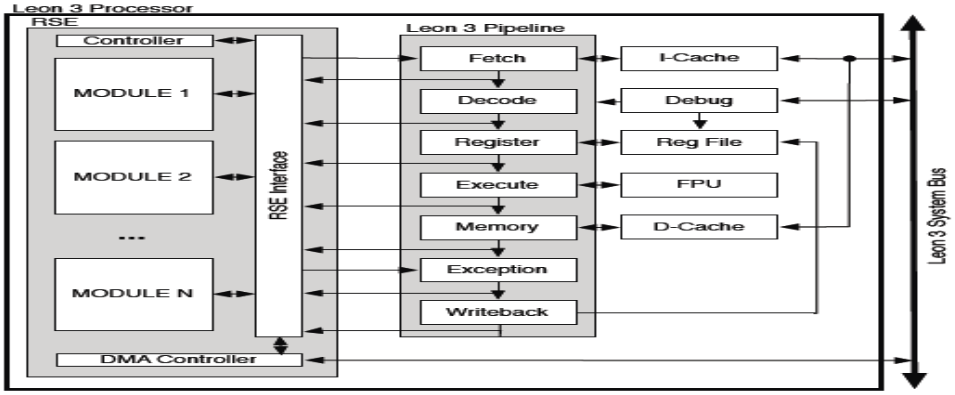


Fig. 5.1: RSE framework

The RSE also contains a DMA controller connected to the system bus in parallel with the processor. With the RSE, module designers need not worry about making direct changes to a processor pipeline. This allows one to focus on the performance of only the components being added, rather than their interaction with the rest of the processor. Additionally, the RSE minimizes the intrusiveness of developing new techniques, which decreases the chances of RSE module designers introducing errors that affect the functionality of the chip in unexpected ways.

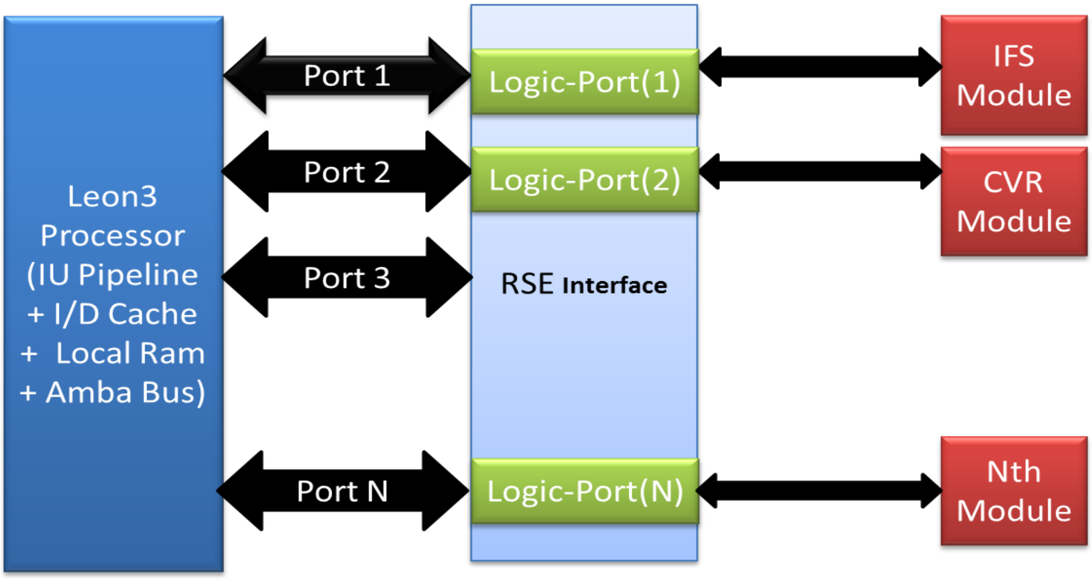


Fig. 5.2: New RSE architecture

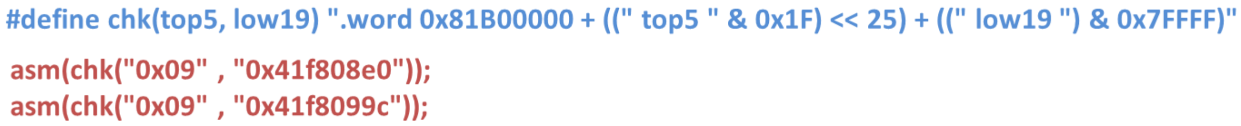
The concept of RSE as an interface to tap in to the pipeline signals remains the same, but the architectural interface has been modified to make it easier to extend the RSE interface to incorporate new modules for different kinds of monitoring tools without requiring any changes in the original source code of the hardware. Figure 5.2 describes the modified architecture for the RSE framework. Hence, for a new user who wants to add a module, it is easier to code the module without making any changes in the original RSE interface code.

The advantage of having an interface like RSE is that it is very easily extensible. RSE has been implemented such that it is possible to add customized ports by concatenating the bits of required control signals from the pipeline to form a customized bundle of signals for a particular port. Each individual port can have its own associated logic to control a custom module designed for monitoring purposes. This design methodology also eases the use of the interface with a new monitoring tool where the user does not have to modify any of the original code and instead just needs to define its own port and custom control logic associated with the tool. As a result, the RSE interface does not add any performance overhead in the execution of the main application. Communication with the RSE interface depends on the instrumented CHECK instructions. Each module has its own check decoders and handlers, which allows the user to conveniently switch modules on and off at will. A program not instrumented with CHECK instructions executes as if it were oblivious to the existence of such a monitoring interface. The current implementation of RSE controls two different modules: IFS for secure computation of the critical data, and CVR (critical value recomputation) [67] for reliable computation of data. More details on integrating the reliability module (CVR) with the security module to achieve both secure and reliable computing are presented in Chapter 8.

# CHAPTER 6

# INFORMATION FLOW SIGNATURE MODULE: ARCHITECTURE

The Information Flow Signature (IFS) technique is implemented using a combination of hardware and software. The hardware implementation requires the addition of a special CHK instruction to the instruction set of the processor, but no other direct modifications to the pipeline. The critical signature is loaded from an instrumented program at load time by using the CHK instruction (recall that the program load path is trusted). CHK instructions are interpreted as no-operations by the main processor, and thus are to be used only by the checking module as it snoops on signals from the main processor. The processor's caches are not modified. Instead, a content addressable memory (CAM) outside of the processor's pipeline is used to store the critical signature of the application being executed. The check is implemented in hardware, which guarantees that the check does not add performance overhead and is performed on every executed instruction. The IFS module itself is implemented as pipelined stages similar to those of Leon3 to maintain coherency with the main integer unit pipeline. This module snoops required signals from the Leon 3 processor pipeline through a generic interface (RSE) to control its own pipelined stages. These signals include 1) the current instruction and its pointer, 2) pipeline stall and flush, and 3) cache control. Signals are used directly from the processor's pipeline without modifications. Outputs from the checking module trigger an error signal if a critical variable (that is included in the application's critical data signature) is corrupted, which in turn raises the integer unit trap signal to stall the propagation of the error to the register files and other memory locations. The Check instructions through which the IFS module is configured from the main processor pipeline are implemented as C macros, as shown in Figure 6.1. These C macros contain the critical information about the signature needed to configure the instruction signature and data signature modules within IFS that enforce runtime checking of the security-critical variables. These CHK instructions are overwritten coprocessor instructions provided by the Leon3-SPARC V8 ISA; after the decode stage of the main pipeline they are transformed to NOP instructions and hence do not modify the processor state during the execution.

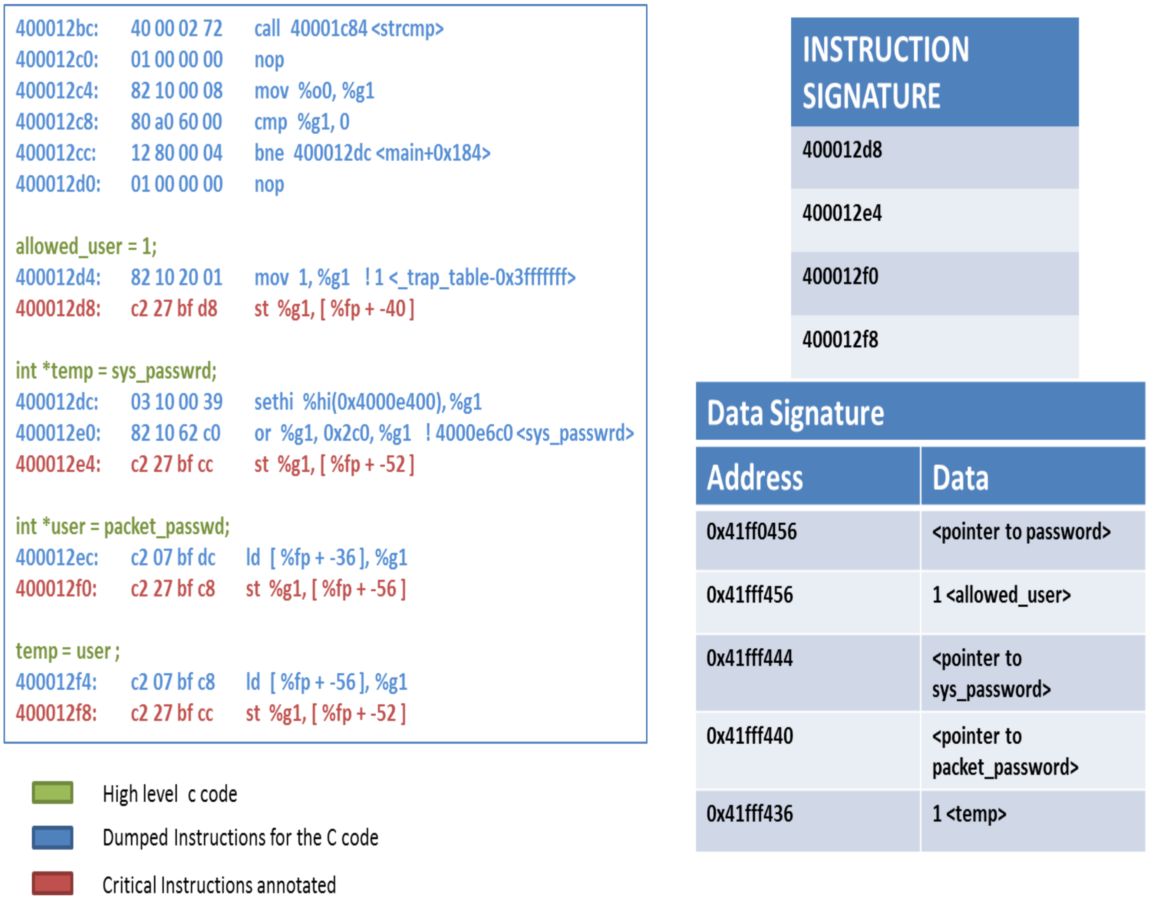


```
#define chk(top5, low19) ".word 0x81B00000 + ((" top5 " & 0x1F) << 25) + ((" low19 ") & 0x7FFFF)"
 asm(chk("0x09" , "0x41f808e0"));
 asm(chk("0x09" , "0x41f8099c"));
```

Fig. 6.1: C Macro for the CHECK instruction and the example CHECK instruction

### 6.1 Critical Signature

In this section, we discuss the critical signature for an application source code. The critical signature has two components: the *instruction signature* and the *data signature*. CHK instructions have been implemented to configure the instruction signature and data signature during the initial loading of the program. The *instruction signature* is fully configured at the beginning of the program by means of a dump file that marks the critical program counters that are virtual addresses. The design has been implemented such that one need not bother about the address translation mechanism by the MMU. The *data signature*, on the other hand, can be configured statically using check instructions to protect global and initialized memory locations that contain critical global data that are required for the application and can be found in the dump files. (Dump files are produced by generating an assembly dump from the source code, from which one can find the relative offsets and relative addresses for critical variables. These relative addresses are then used to configure the critical data signature in the IFS module.)  Since the application computes and produces critical data during execution and stores them onto the stack or heap, the data signature is also configured by the critical instructions annotated during the runtime. Consequently, the data signature contains statically initialized global data and dynamically computed critical data. This is illustrated in Figure 6.2, where the first entry of the data signature is global data configured using a check instruction, while the next four entries correspond to the dynamically computed address and data computed during the execution of the four corresponding critical instructions.



```
400012bc:   40 00 02 72   call  40001c84 <strcmp>
400012c0:   01 00 00 00   nop
400012c4:   82 10 00 08   mov  %o0, %g1
400012c8:   80 a0 60 00   cmp  %g1, 0
400012cc:   12 80 00 04   bne  400012dc <main+0x184>
400012d0:   01 00 00 00   nop

allowed_user = 1;
400012d4:   82 10 20 01   mov  1, %g1   ! 1 <_trap_table-0x3fffffff>
400012d8:   c2 27 bf d8   st  %g1, [ %fp + -40 ]

int *temp = sys_passwrd;
400012dc:   03 10 00 39   sethi  %hi(0x4000e400), %g1
400012e0:   82 10 62 c0   or  %g1, 0x2c0, %g1   ! 4000e6c0 <sys_passwrd>
400012e4:   c2 27 bf cc   st  %g1, [ %fp + -52 ]

int *user = packet_passwd;
400012ec:   c2 07 bf dc  ld  [ %fp + -36 ], %g1
400012f0:   c2 27 bf c8  st  %g1, [ %fp + -56 ]

temp = user ;
400012f4:   c2 07 bf c8  ld  [ %fp + -56 ], %g1
400012f8:   c2 27 bf cc  st  %g1, [ %fp + -52 ]
```

- 🟩 High level  c code
- 🟦 Dumped Instructions for the C code
- 🟥 Critical Instructions annotated

| INSTRUCTION SIGNATURE |
|---|
| 400012d8 |
| 400012e4 |
| 400012f0 |
| 400012f8 |

Data Signature

| Address | Data |
|---|---|
| 0x41ff0456 | <pointer to password> |
| 0x41fff456 | 1 <allowed_user> |
| 0x41fff444 | <pointer to sys_password> |
| 0x41fff440 | <pointer to packet_password> |
| 0x41fff436 | 1 <temp> |

Fig. 6.2: Example code and corresponding instruction and data signature

### 6.2 Instruction Signature

The instructions that write to critical memory locations are critical to the application, and hence the integrity of these instructions must be assured at runtime. Every such instruction has a unique identifier (PC) stored in the instruction signature module, which is implemented as a fully associative look-up table. Criticality of a fetched instruction is resolved only after comparison with the signature stored in the fetch stage of the IFS module.

### 6.3 Data Signature

A Content Addressable Memory (CAM) embedded within the IFS module is used to hold the addresses of critical data in the program. The use of the CAM to store the critical data and address obviates the need to use system RAM to mark every address's criticality. This technique also eliminates the need to add an extra bit to the main bus width within the main processor [68], or tag caches with extra information [41]. Such approaches, which have been used for different processor-level security enhancements, require significant effort in modifying and revalidating the design of the processor and call for changing architectural characteristics, such as bus widths, of current systems.

In order to alleviate any storage limitations that may arise from a fixed-size table, we also propose modifications to the TLB that allow for unlimited critical instructions and data. However, the currently implemented fixed sized of the instruction and data signature is adequate for medium-sized applications.

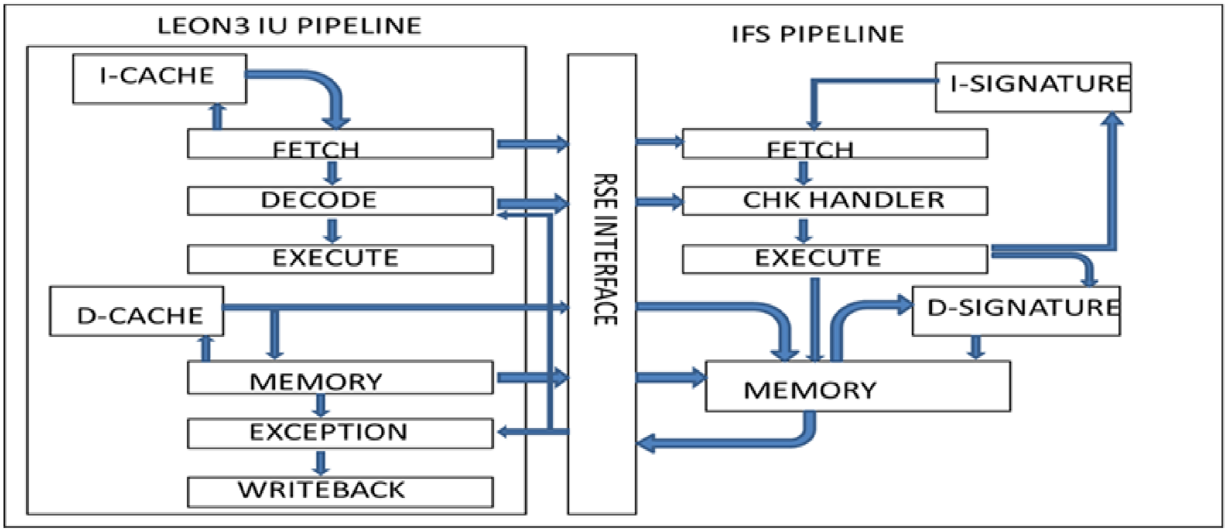


Fig. 6.3: IFS architecture

### 6.4 IFS Pipeline Stages

Figure 6.3 illustrates the hardware architecture for the IFS module hooked up to the Leon3 processor via the RSE interface.

**Fetch:** In the fetch stage, the instructions fetched in the main processor IU pipeline are passed on to the IFS module either to configure the IFS module or to resolve whether the instruction is critical or not.

**Check handler:** This stage is the same as the decode stage (of IFS). The instructions fetched into the IFS pipeline are decoded to identify the different kinds of check instructions that can be handled in the IFS module. There are three different kinds of checks: one for configuring the instruction signature and two for configuring the data signature (critical memory address and critical memory data). The handler also identifies whether the instruction is a check instruction and reorganizes the bits to form a uniform pattern to be fed into the execute stage.

**Execute stage:** This stage behaves as the controller for all check instructions, critical instructions, and noncritical instructions. It is responsible for generating the control signals corresponding to the check instructions in order to configure the correct signature module. For the critical instructions, it also generates signals that are needed in order to synchronize with the main processor pipeline when memory is being accessed to load or store critical data into the caches or memory.

**Memory Stage:** In this stage, the critical data being stored or loaded are also stored and loaded from the data signature, to ensure the integrity of the critical data and hence detect a potential memory corruption attack. The memory stage (in IFS) overlaps with the memory stage, exception stage, and writeback stage of the main pipeline. The reason is that on a cache miss, long latency stalls during traps, or annulled instructions, the correct data from memory are fetched into the main pipeline during exception or write back stages (bypassing the memory stage). So in order to be synchronized with the main pipeline, the memory stage in IFS has to work concurrently with the memory, exception, and writeback stages. Once the instruction in the main pipeline has reached the final writeback stage, the final comparison to detect the attack is done in the memory stage of the IFS pipeline, because it is only then that the pipeline has the correct data loaded from the memory on a load request.

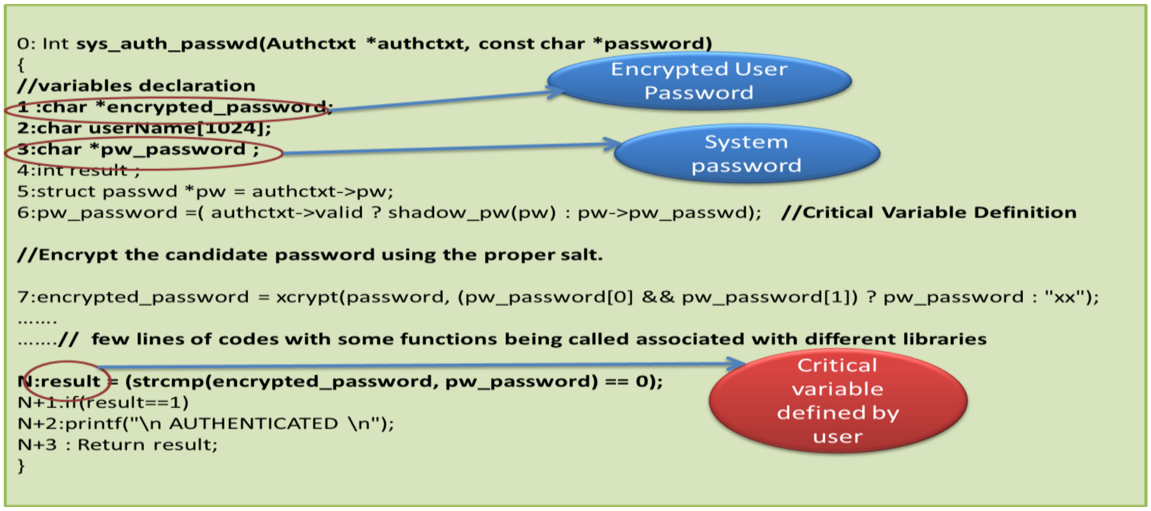
### 6.5 IFS Runtime Operation

1. *Program initialization*: RSE CHK instructions from the main processor pipeline enter the IFS module in which the CHK handler in the decode stage generated the control signal to initialize the critical instruction and data signatures.
2. Runtime: The fetch stage of the IFS module checks if the instruction fetched in the main processor pipeline is a part of the critical signature. The instruction is read in through the RSE interface.

3. Critical instructions have their criticality propagated through the IFS pipeline in the subsequent clock cycles. Instructions flowing through the IFS pipeline are synchronized with the main processor pipeline using the pipeline control signals through the RSE.
4. In the memory stage, every instruction is checked as to whether it is a load instruction using the cache control signals or a part of the critical data signature.
5. If the instruction is a critical instruction reading critical data, the data are fetched from the CAM in the memory stage simultaneous with the retrieval of the actual data from the data cache. Hence the integrity of the critical data is resolved in the memory stage without allowing the error to be propagated to the next clock cycle.

### 6.6 Software vs. Hardware (IFS)

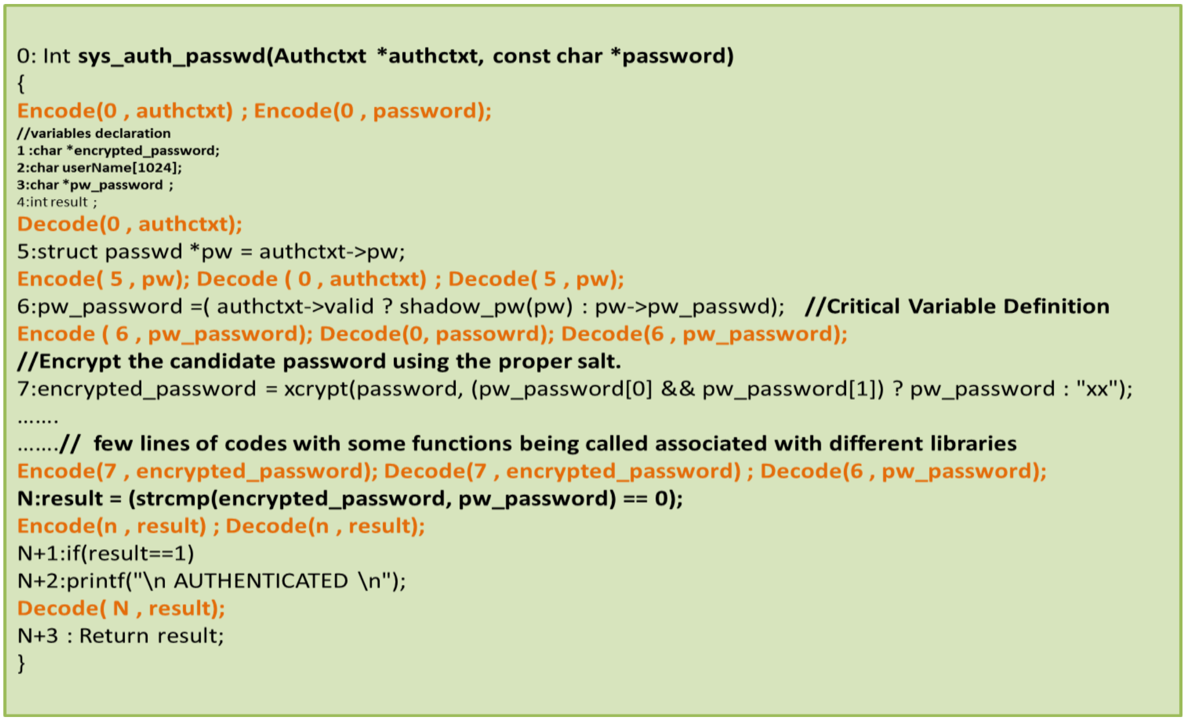
In this section, we will use an example code to discuss the comparative advantage of a hardware-based implementation of the Information Flow Signature (IFS) module over a software-based design.



```
0: Int sys_auth_passwd(Authctxt  *authctxt, const char *password)
{
//variables declaration
1 :char *encrypted_password;
2:char userName[1024];
3:char *pw_password ;
4:int result ;
5:struct passwd *pw = authctxt->pw;
6:pw_password =( authctxt->valid ? shadow_pw(pw) : pw->pw_passwd);   //Critical Variable Definition

//Encrypt the candidate password using the proper salt.

7:encrypted_password = xcrypt(password, (pw_password[0] && pw_password[1]) ? pw_password : "xx");
.......
.......//  few lines of codes with some functions being called associated with different libraries

N:result = (strcmp(encrypted_password, pw_password) == 0);
N+1:if(result==1)
N+2:printf("\n AUTHENTICATED \n");
N+3 : Return result;
}
```

Fig. 6.4: SSH stub

Figure 6.4 shows a stub (which forms the authentication module) of the SSH application. In a software version of IFS, the code is re-instrumented, and the encoding and decoding operations are as shown like in Figure 6.5.



```
0: Int sys_auth_passwd(Authctxt  *authctxt, const char *password)
{
Encode(0 , authctxt) ; Encode(0 , password);
//variables declaration
1 :char *encrypted_password;
2:char userName[1024];
3:char *pw_password ;
4:int result ;
Decode(0 , authctxt);
5:struct passwd *pw = authctxt->pw;
Encode( 5 , pw); Decode ( 0 , authctxt) ; Decode( 5 , pw);
6:pw_password =( authctxt->valid ? shadow_pw(pw) : pw->pw_passwd);   //Critical Variable Definition
Encode ( 6 , pw_password); Decode(0, passowrd); Decode(6 , pw_password);
//Encrypt the candidate password using the proper salt.
7:encrypted_password = xcrypt(password, (pw_password[0] && pw_password[1]) ? pw_password : "xx");
…….
…….//  few lines of codes with some functions being called associated with different libraries
Encode(7 , encrypted_password); Decode(7 , encrypted_password) ; Decode(6 , pw_password);
N:result = (strcmp(encrypted_password, pw_password) == 0);
Encode(n , result) ; Decode(n , result);
N+1:if(result==1)
N+2:printf("\n AUTHENTICATED \n");
Decode( N , result);
N+3 : Return result;
}
```

Fig. 6.5: Instrumented SSH stub: software IFS

### 6.7 Advantages of Using Custom Hardware over Software Implementation

1. Development of custom logic for runtime checking of application does not suffer from the time-of-check-time-of-use (TOCTOU) vulnerability, because the check on the critical signature is performed in parallel with the original computation in the main pipeline, without insertion of new software instructions. A software implementation cannot solve the TOCTOU problem, as new instructions introduce windows of vulnerability that can be exploited by malicious attackers, whereas in hardware, the annotated instructions can be checked in parallel without time overhead. Of course, extra instructions are required to load the critical signature for an application, but after hardware initialization, the window

of vulnerability for a critical memory corruption error is practically zero. Hardware operates at the level of assembly instructions, so it gives much more granularity to enforce integrity than do high-level software checks (which are composed of multiple instructions). Also, hardware does not suffer from the limitation of compiler-based static analysis (discussed in Chapter 4).

2. Given a sequence of instructions and the critical data that need to be protected, using the proposed approach, one can show that the system never enters a compromised state (where a compromised state is defined as one in which the critical data can be influenced by an untrusted instruction). Attacks are represented as the modification of the destination of an instruction in a given instruction sequence to a different object (or variable) than the one specified by program semantics.

3. One of the most important assumptions that can be relaxed by enforcing checking in hardware is the assumption that instructions cannot be modified during execution. We only assume that a critical store instruction cannot be modified to any another instruction. Since we have a unique identifier for each instruction that is part of the instruction signature, changing one instruction to another critical instruction would not compromise the data or instruction signature. For example, if any instruction (other than a critical store) changes to a store instruction, because of the unique identifier, it would not be treated as a critical instruction, and the integrity of the signature would be preserved. Consequently, when any of the following instructions access the data from the same critical memory location, the memory corruption error will be detected. In the reverse case, when a critical store instruction is modified to some other noncritical instruction

(which can be a noncritical store) the signature would not be updated and can be a potential hole for a well-crafted attack. If the instruction is not a store instruction, its identifier is definitely not one of the critical instruction signature identifiers, so in any case such modifications cannot harm the critical signature integrity.

4. The compiler-based software techniques check the validity of the program's control flow using state machines derived by static analysis of the code. The state machines track the sequence of calls to the IFS functions and check if the program's control flow is valid. The state is tracked on a per-function basis, since we consider only intra-procedural slices. Similarly, the state-machine state is checked just before the function returns to ensure that the state machine is in an *accepting state*, i.e., the state machine has accepted the observed sequence of function calls. During the execution of the program, it is verified that every function call executed by the program corresponds to a valid state transition from the current state of the state machine. Loops in the backward slice of critical variables are represented as cycles in the state machine. However, the state machines do not include information about the number of iterations executed by a loop. That could be exploited by an attacker, who might make the loop execute for more or fewer iterations than allowed by the source program. (The attacker's intent might be to bypass security checks performed in loop iterations, or to introduce a semantic violation.) Detecting such attacks requires timing/semantic information about the execution counts of program loops. Also, it is possible for an attacker to replace a sequence of IFS function calls with a legal but invalid sequence in the program (i.e., valid for some other input). In order to avoid detection, the attacker must replace the sequence in its entirety with another valid sequence, and this is difficult. Shifting the design paradigm to hardware

alleviates this problem, since the preprocessing of the source code uses a model-checker-based instruction-level simulator to analyze the architectural state of the processor and embed the checks, which are dynamic in the sense that they monitor memory locations during the runtime and do not depend on the static state machines. Crafting attack-based loopholes in the state machine would not suffice, as the hardware itself will act as a monitor to the set of exhaustive critical memory locations; hence, any invalid jumps within the code that arise because of  memory corruption attacks would be detected.

5. A software implementation of the IFS would not protect against attacks launched by the operating system (OS) or another process with higher privilege. The reason is that the instrumentation is inserted at the granularity of instructions, and consequently it is possible for the OS or another process to corrupt the critical data by setting breakpoints immediately before and after the update of critical data in the program. The IFS technique does not detect these attacks, as it is implemented entirely in software with no hardware support. An approach to addressing these attacks would be to store the signature in the hardware and prevent even the OS from being able to modify the signatures once the application has been loaded. Hence, hardware-based support for IFS is much more secure against attacks launched through kernel debuggers or super-users. The added advantage of moving IFS to hardware is that the window of vulnerability is practically zero, as IFS hardware executes in parallel; hence, setting breakpoints between instructions to craft any kind of attack would not work and would be detected during the execution of the application.

# CHAPTER 7

# QUALITATIVE ANALYSIS: INSIDER ATTACK

Insider attacks are attacks in which a privileged entity (“insider”) abuses its privilege to attack the system. Insider attacks can be mounted at the hardware, virtual machine, operating system, and network levels. Various techniques have been applied at different abstraction levels to foil insider attacks. For example, there are hardware techniques that protect against exploit through backdoors in hardware during the design [69] and synthesis stages [70]. Network-based security techniques that assume that the attacker controls the nodes in a network have been deployed using enhanced intrusion detection systems to look for anomalous patterns in network traffic [71] or using attack graph analysis [72]. Operating-system-based techniques that are commercially used include the Rootkit-based detection system, which relies on memory access patterns and inspecting data structures [73, 74]. Such techniques ensure that the insider cannot infiltrate the system at the level in which the technique is deployed. However, none of these techniques address the problem of application-level insiders. The reason is that an application-level insider does not need to change the hardware, operating system, virtual machine, or network [75], and hence does not trigger the detection mechanisms deployed at these levels.

Further, extension of techniques deployed at lower levels to the application level is not straightforward. The reason is that applications are diverse, and hence the technique must be generic enough to support a majority of applications. In addition, the technique must be automated in order to be widely deployed, because, unlike operating systems or virtual machines, applications come in a wide variety, and it is not possible to modify each application individually to incorporate the technique.

The rest of this section discusses security techniques deployed at the application level that protect the application from malicious attackers. We classify techniques for protecting applications from security attacks into three broad categories: (1) techniques to protect against external attackers (e.g., memory safety checking, taintedness detection, address-space/instruction-set randomization, and system-call-based checking); (2) techniques to protect against internal attackers, including application-level insiders (e.g., privilege separation, remote audit, code attestation, and oblivious hashing); and (3) techniques to protect critical data in applications such as *Samurai* and *redundant data diversity*.

External security techniques such as memory safety checking [76] are designed to protect applications from memory-corruption attacks (e.g., buffer-overflow or format string). This class of techniques is not effective for thwarting insider attacks, as insiders do not need to exploit memory-corruption vulnerabilities in order to assume control of the application. Taintedness detection techniques [77, 78] and [79] prevent application input from influencing high-integrity data in applications such as pointers. However, insiders can directly influence security-critical data without going through the inputs, as they are already within the application. Further, it is almost impossible to prevent an internal module of the application from writing to generic program objects such as pointers without incurring a very high false positive rate [80].

Security techniques such as randomization [81, 82] attempt to obscure a program layout or instruction set from attackers. However, randomization can be bypassed by insiders who are themselves subject to the same randomization as the application. For example, a malicious module in an application can examine the address of its local variables and calculate the absolute address of a stack variable in a different module, even when both modules are randomized. Unlike an external attacker who requires multiple attempts to bypass the randomization [83, 84], an insider needs only a single attempt to bypass it.

System-call, based checking is a technique that monitors the sequence of system calls made by an application and checks if the sequence corresponds to an allowable sequence as determined by static analysis [85]. The main assumption made by system-call checking is that the attacker seizes control of the application by executing unwanted or malicious system calls such as *exec*, or by changing the argument of system calls, e.g., *seteuid*. The reason is that system calls provide a conduit for attacking other applications on the same system as well as the operating system (OS) itself. However, an application-level insider goal is to subvert the execution of the attacked application, and not to attack other applications or the OS. Hence, system-call-based detection techniques may not protect against insiders who overwrite only the attacked application data and/or control.

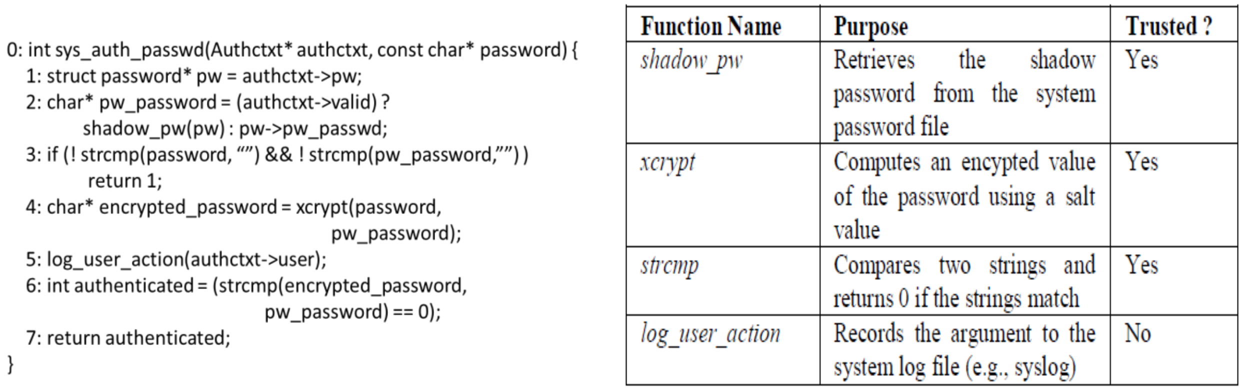
Techniques such as oblivious hashing [86, 87] and code attestation [88] detect malicious modifications of the application executable code after it has been generated (by the linker). However, these techniques do not protect from insider attacks in which the application developer links the application with an untrusted third-party library prior to its distribution. A technique that offers limited protection from insider attacks is remote audit [89], which ensures that computationally intensive portions of the application code are not skipped at runtime. However, remote audit does not protect against malicious modifications of the application data by an application-level insider.

The only known technique that can effectively thwart application-level insider attacks is privilege separation [90]. In this technique, the application is divided into separate processes, and each module executes in its own process. A module can share data with another module only through the OS's inter-process communication (IPC) mechanisms. This prevents an untrusted module from overwriting data in the application address space, unless the application explicitly shares the data with the untrusted module (through an IPC call). However, privilege separation incurs performance overheads of up to 50% when deployed in real applications [91] (measured as a fraction of the entire application execution time). Further, a trusted module may load an untrusted library function in its address space, thereby annulling the technique's security guarantees.

Finally, Samurai [92] and redundant data diversity [93] protect critical data in applications from accidental and malicious corruptions, respectively. However, they both require the programmer to manually identify read/write operations on the critical data, which can be cumbersome. Further, redundant data diversity requires replication of the entire process and execution of it in lock-step, even though the critical data may constitute only a small portion of the application data. This leads to unnecessary overheads and wasted resources.

Thus, there exists no technique that can detect insider attacks on program data without requiring considerable intervention on the part of the programmer or incurring unacceptably high performance overheads. In this thesis, we try to demonstrate the efficacy of the IFS concept in handling insider attacks while incurring a very low performance overhead.

**Example Attack:** This section illustrates the IFS technique using a code fragment drawn from the OpenSSH application. The section also considers example attacks on the application's code and discusses how IFS detects the attacks. In order to determine the critical data, we assume that the goal of the attacker is to subvert the authentication mechanism by making the function return 1 in spite of the user's password being invalid. Therefore, we designate the value returned by the function as the critical variable (i.e., the *authenticated* variable, defined in line 6).



```
0: int sys_auth_passwd(Authctxt* authctxt, const char* password) {
   1: struct password* pw = authctxt->pw;
   2: char* pw_password = (authctxt->valid) ?
            shadow_pw(pw) : pw->pw_passwd;
   3: if (! strcmp(password, “”) && ! strcmp(pw_password,””) )
            return 1;
   4: char* encrypted_password = xcrypt(password,
                                         pw_password);
   5: log_user_action(authctxt->user);
   6: int authenticated = (strcmp(encrypted_password,
                                  pw_password) == 0);
   7: return authenticated;
}
```

| Function Name | Purpose | Trusted ? |
|---|---|---|
| *shadow_pw* | Retrieves the shadow password from the system password file | Yes |
| *xcrypt* | Computes an encypted value of the password using a salt value | Yes |
| *strcmp* | Compares two strings and returns 0 if the strings match | Yes |
| *log_user_action* | Records the argument to the system log file (e.g., syslog) | No |

Fig. 7.1 Example code fragment from the SSH program and functions called from the code fragment

### 7.1 Attacks

To illustrate the IFS technique, we consider different attack scenarios on the untrusted *log_user_action* function as follows.

### *7.1.1 Generic Attacks*

An attacker has full control over the log_user_action source code and provides a malicious library to be used with the authentication module of the SSH application. The attacker can either exploit the format string vulnerability in the *log_user_action* function by crafting an appropriate input to the program, or manipulate the source code. For the purpose of illustrating these attacks, it is assumed that OS can overwrite any variable(s) and jump anywhere in the program. The attacker is oblivious to the implementation of the IFS technique while creating these attacks. Since the malicious function is an external library function, the IFS technique does not require any knowledge of the function, since the function is not supposed to manipulate or compute critical variables.   We consider different kinds of generic attacks as discussed below.

**Attacker executes system call to overwrite critical data:** Assume that the attacker launches a system call by overwriting the return address on the stack with the address of a system call instruction. The attacker also sets up the frame-pointer on the stack such that the system call is executed with the parameters specified by the attacker. The IFS technique by itself does not prevent the attacker from launching the system call, nor does it prevent the attacker from overwriting the return address (as it does not belong to the backward slice of the critical variable). However, once the system call has been executed, any attempt by the attacker to overwrite the critical data from within the system call will be detected. The reason is that the store instruction that modifies the critical instruction within the system call was never a part of the application signature of the critical store instructions. Hence, when the store instruction executes, it changes the critical memory location, but the integrity of the signature is still intact. So when the application tries to read the critical data, there will be a mismatch, as critical data stored in the IFS module were never changed by to the system call, and the attack will be detected.

**Attacker overwrites function-pointers/return address to impact the program's critical data:** Here we assume that the attacker overwrites the return address on the stack from within the log_user_action function. The goal of the attacker is to make the function return directly to line 7, in effect bypassing the initialization of the authenticated variable in line 6 (see Figure 7.1). Let us further assume that the authenticated variable is assigned a non-zero value prior to line 6. This allows the attacker to falsely authenticate herself/himself to the system. The IFS technique detects the attack as follows. The return address that can be modified to change the control flow and authenticate the user is a part of the critical signature. The reason is that SymPLFIED does an exhaustive error injection in all the possible valid memory locations and hence identifies the return address as one of the critical memory locations that can be modified to achieve the attacker's objective. Because the malicious store instruction (within log_user_action) that changes the return address is not a part of the critical signature, if executed, it will result in a mismatch in the critical data stores in the IFS module and on the stack, and the attack will be detected.

**The log_user_action function modifies the contents of encrypted_password:** Since the attacker has control over the malicious function, there is a malicious store instruction such that the value of the pointer variable encrypted_password is assigned to the address of pw_password (or vice versa). This would cause the strcmp function in line 8 to return the value 0, and hence the authenticated variable will assume the value of 1, which is the attacker's goal. Since the IFS technique does not require the source code of third-party libraries and the function is not supposed to manipulate the critical data, any change made to critical data will be detected, because the malicious store instruction would not compromise the integrity of the application's signature.

### *7.1.2 Targeted Attacks*

An attacker may be aware of the implementation of the IFS technique and hence craft intelligent attacks in order to avoid detection.

**Attacker corrupts a data value produced that is stored in the memory:** If the corrupted data value in any way changes the control flow of the program or directly influences the result of the computed critical data, hence leading to a successful attack, the corresponding store instruction that stores this data value to memory will be a part of the critical signature for the application. Since SymPLFIED does an exhaustive error injection to all possible memory locations, it will also identify and annotate the corresponding store instruction during the pre-processing of the application, to be included in the signature for the application. If the data value is changed due to a memory corruption attack (which would be possible only after the data value has been stored in the memory), runtime signature checking will detect the attack, since the signature of the application is still correct.

If the data value is corrupted due to register error before it is stored into the memory, the attack will not be detected, as the signature will also store the corresponding corrupted value during the execution of the store instruction. Register errors are more of a reliability concern and can be detected using other modules, such as CVR attached to the RSE interface. The IFS technique can detect all the memory corruption attacks that do not compromise the integrity of the signature. Attacks that do not manifest themselves as memory errors, for example register errors, would not be detected by the IFS technique as such. From an attacker's perspective, crafting such an attack by manipulating register errors is very complex, because the attacker has to identify the registers being used and also has to create a breakpoint at a precise point during the execution of the program. Such an attack using high-level language is very difficult, while memory corruption attacks are comparatively easier to create at this level of coding.

**The attacker modifies an instruction in the instruction cache or local instruction memory**

**Case 1: Attacker modifies a noncritical instruction to another noncritical instruction:** Such an attack will not achieve the attacker's goal because in order to craft a successful attack, the attacker has to change application code; if that change indeed results in success, that has to be a part of the critical signature.

**Case 2: Attacker modifies a critical store instruction:**

a) **Critical store instruction changes to a modified critical store instruction with a changed data value:** The correctness of the signature would not be preserved, as it would contain the wrong value. Hence, this kind of attack would go undetected.

b) **Critical store instruction is modified to represent another store with a different memory location:** The signature would be incorrect in this case, but would be detected if the actual critical memory location were indeed protected by more than just one critical store instruction (as would happen in most modern security applications, as frequent computational changes and updates are made to critical memory locations during the execution of the application). Thus the attacker would be provided with only a small window of vulnerability, that is, until the second critical store instruction that uses the actual memory location that was to be protected. Protecting a memory location that is not critical would just increase the security layer for the application without any performance or hardware overhead.

c) **Critical store instruction is modified to another noncritical instruction:** This case is the same as the one discussed in the previous point, except that there is no noncritical memory location being protected.

**Case 3: Attacker modifies a noncritical instruction to a critical store instruction:** Such an attack would not be detected, but modifying a noncritical instruction would not subvert the system. The IFS would just be protecting noncritical code with all other critical signatures without any performance overhead. This kind of an attack is very complex, since it cannot be crafted just by flipping some bits. It would be necessary to change the control flow of the program and corrupt the architectural state of the processor, which would usually lead to an application crash unless it was done at a precise breakpoint with accurate values. The IFS module does not prevent such an attack, but the success of this attack requires other memory locations to be corrupted, which can be detected by IFS since SymPLFIED’s exhaustive analysis protects those memory locations.

**Case 4: Attacker modifies instructions that configure the IFS modules:** All these attacks would definitely compromise the integrity of the signature, but the implications would be as follows.

1. **Attacker modifies a check instruction to any other instruction:** The IFS module would not detect such an attack, but this case is similar to the case in which the attacker modifies a critical store instruction to a noncritical instruction and hence would result in similar implications; that is, it would result in a small window of vulnerability that extends until the critical store that updates the same critical memory location. To corrupt this particular memory location, the attacker must change all the critical stores that modify this memory location, which cannot be done using any static analysis of the code or during the compile time. The attacker requires access to hardware debuggers to monitor particular memory locations. The other option for the attackers would be to modify all the check instructions that configure the IFS hardware, which again is not possible using just a static analysis of the code.
2. **Attacker modifies a check instruction such that it is protecting the wrong instruction:** If the wrong instruction with which the IFS is configured is not a critical store instruction, the IFS hardware has been designed and implemented such that it can identify such a modification and would behave as if the application had not been instrumented with that particular check instruction. Thus, the IFS would be protecting the corresponding noncritical memory location. From the attacker's perspective, this case is similar to a modified critical store instruction for which there is a small window of vulnerability until the critical memory location is protected by another store or check instruction.

# CHAPTER 8

# RELIABILITY AND SECURITY: UNIFIED FRAMEWORK

In this chapter, we discuss the reliability module called *CVR* (critical value recomputation). We will then talk about the design and implementation of a unified framework that brings the IFS module for security and CVR for reliability on a single on-chip system. A prototype has already been developed on an FPGA fabric. This section shows how the Reliability and Security Engine is used to interface two different modules with different monitoring purposes, integrating them to work on a single platform and hence allowing secure and reliable computation at the same time. The hardware overhead corresponding to these modules and the performance overhead are discussed in Chapter 9.

## 8.1 Critical Value Recomputation

This design method presents a flaw detection technique aimed at offering high levels of coverage with a minimal impact on overhead. This module implements the hardware components of the statically derived error detectors. It consists of a path-tracking sub-module and a microcontroller-based checking sub-module. The path-tracking sub-module keeps track of the program control-flow path, and the checking sub-module executes the checking expressions corresponding to the path determined by the path-tracking sub-module. The microcontroller is used as an extension of the RSE [6], which is coupled with the Leon3 processor. This design is proposed to provide the following solutions:

1. Application-aware reliability and security solutions that mask or detect malicious activities and accidental errors with minimal cost.
2. Processor-level solutions that achieve low-cost, high-performance, scalable security and reliability checking in the same framework.

Like the IFS module, the application source code is statically analyzed to extract the backward slice for the critical data and then instrumented with the check instructions to maintain the integrity of the information.

The whole procedure for Critical Value Recomputation can be summarized as follows:

1. Critical variables for an application are determined according to the fanout metric by using static analysis. Variables with a higher fanout are deemed more critical than variables with a lower fanout. The most critical variables are then selected for protection.
2. Once a critical variable has been determined, the backward slice of the critical variable is derived. The backward slice consists of instructions that are legally (according to C language semantics) allowed to contribute to the value of critical variables.
3. Given the backward slice of each critical variable, optimal placement of a detector as well as operations necessary for recomputation are established. Operations required for recomputation are determined by optimizing each path, which contributes to the computation of a critical variable. (Figure 8.1 describes derivation of detectors from code semantics.) Each path is examined, because the computation of variables depends on the path being executed. This step produces a textual representation of path transitions contained in the backward slice as well as an expression list that describes the operations

necessary to recompute a critical variable. Figure 8.2 shows how an injected error on a particular control flow path is detected by the CVR module.

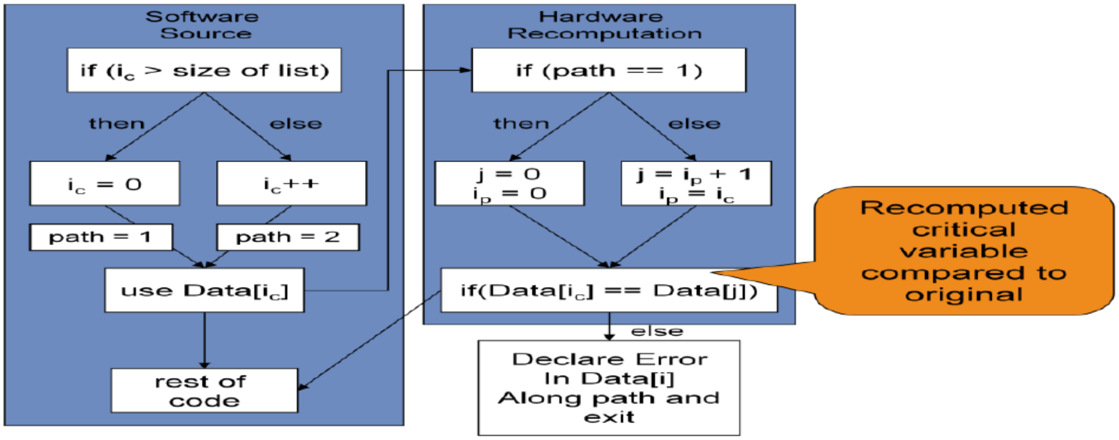


Fig. 8.1: Derivation of detectors (hardware recomputation: microcontroller)

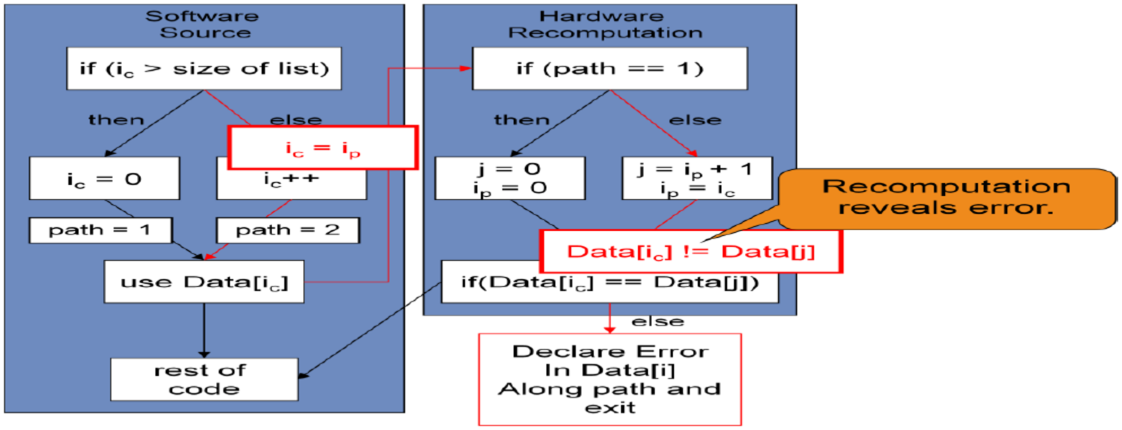


Fig. 8.2: Detection of injected error using CVR module

## 8.2 Overall Unified Framework

This section describes the design and approach to building dependable (reliable and secure) prototype systems using the notion of application-aware dependability. Application properties are automatically extracted using compiler-based static and SymPLFIED-based symbolic error injection analysis techniques, and are converted to error and attack detectors. The detectors are implemented using programmable hardware as a part of the Reliability and Security Engine

(RSE), which is a hardware framework for executing application-aware checks. The RSE functionality is as follows:

1. It derives application-aware error and attack detectors through compiler analysis and a symbolic error injection simulator like SymPLFIED.
2. It implements the derived detectors in a common, programmable hardware framework.

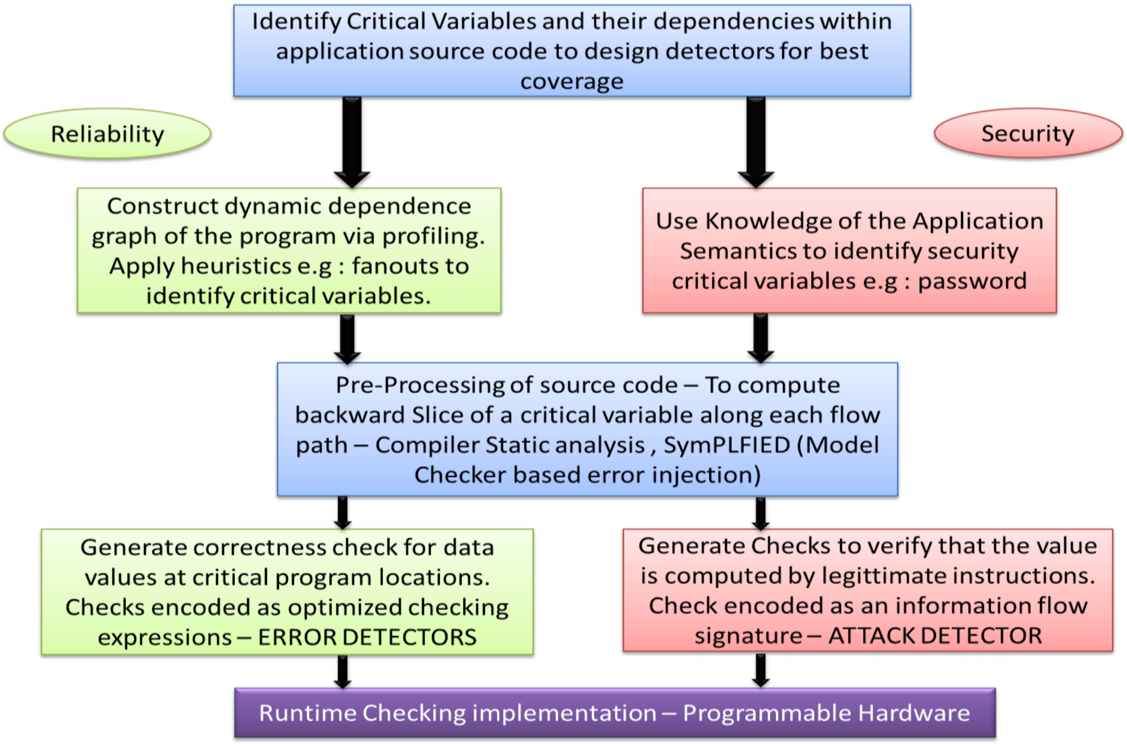

Identify Critical Variables and their dependencies within application source code to design detectors for best coverage
Reliability
Security
Construct dynamic dependence graph of the program via profiling. Apply heuristics e.g : fanouts to identify critical variables.
Use Knowledge of the Application Semantics to identify security critical variables e.g : password
Pre-Processing of source code – To compute backward Slice of a critical variable along each flow path – Compiler Static analysis , SymPLFIED (Model Checker based error injection)
Generate correctness check for data values at critical program locations. Checks encoded as optimized checking expressions – ERROR DETECTORS
Generate Checks to verify that the value is computed by legittimate instructions. Check encoded as an information flow signature – ATTACK DETECTOR
Runtime Checking implementation – Programmable Hardware

Fig 8.3: Conceptual unified framework for security and reliability

Figure 8.3 shows the components of the framework. The left side of the figure shows the process for derivation of error detectors, while the right side shows the process for derivation of security attack detectors.

The major steps in the framework are as follows:

1. Identification of critical variables: From a reliability perspective, these are variables that are highly sensitive to errors in the application. From a security perspective, these are

variables that are desirable targets for an attacker who wishes to take over the application. For reliability, it is possible to automate the selection of sensitive or critical variables through error propagation analysis. This can be done based on analysis of the dynamic dependencies in the application. For security, we require the programmer to identify security-critical variables in the application through annotations based on knowledge of the application semantics. An example of a security-critical variable is a Boolean variable that indicates whether the user has been authenticated, as overwriting the variable can lead to authentication of a user with an incorrect password.

2. Extraction of backward program slice: Once the critical variables and the program points at which checks must be placed have been identified, the next step is to derive the properties of these variables from the application code. These properties can be computed based on the backward program slice of the critical variable from the check placement point. The backward program slice of a variable at a program point is defined as the set of all program statements that can potentially affect the value of the variable at that program point. The slice is computed through static analysis for all legitimate program inputs. For error detection, we are interested in re-executing the statements in the slice of the critical variable to ensure that the value of the critical variable computed at the check placement point is correct; hence, the slice of the critical variable computed for error detection needs to preserve the execution order of program statements. For attack detection, we are only interested in checking whether the statements/instructions that can modify the critical variable in any possible way (e.g., including the conditional

dependencies within a backward slice) in fact write to the critical variable (directly or indirectly) at runtime.

3. Encoding of slice: The third step is to encode the slice computed for the critical variable in the form of a runtime check. For error detection, the check takes the form of an executable expression that re-computes the critical variable, whereas for attack detection, the check takes the form of a signature that contains the addresses of the instructions that can write to the critical variable (directly or indirectly). The compiler inserts calls to the checks (expressions or signatures) into the executable file and configures the hardware monitors with the checks at application load time.

4. Runtime checking: The final step is performed at runtime. The application is monitored (using hardware or software), and the checks inserted by the compiler are executed at the appropriate points in the execution. In the case of error detection, the checks compare the value of the critical variable computed by the original program with the value of the expression derived using static analysis. A value mismatch indicates an error. In the case of attack detection, the checks compare the signature derived using static analysis with the signature computed at runtime based on the instructions that write to the critical variable (directly or indirectly). A signature mismatch indicates an attack. In both cases, the execution of the program is stopped, and a suitable recovery action is taken.

## 8.3 Hardware Prototype

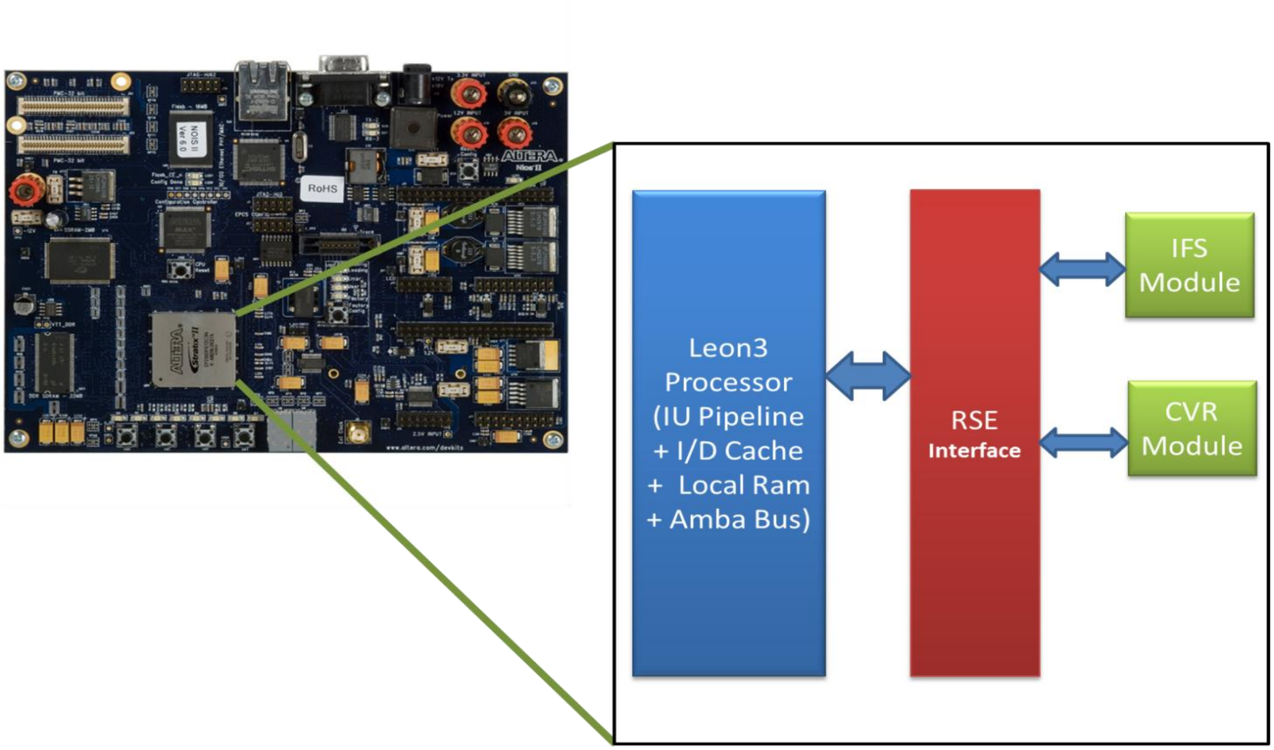

Leon3 Processor (IU Pipeline + I/D Cache + Local Ram + Amba Bus)
RSE Interface
IFS Module
CVR Module

Fig. 8.4: Final prototype of the unified framework for security and reliability

Figure 8.4 shows the final hardware prototype that has been developed and demonstrated to execute security and reliability checks. The platform used for prototyping the unified framework for reliability and security is a Stratix II Altera board. Chapter 9 gives more details on the hardware overhead.

# CHAPTER 9

# RESULTS AND CONCLUSIONS

## 9.1 Experimental Evaluation

The RSE and IFS (along with CVR) have been implemented alongside the Leon 3 processor, which includes a 7-stage in-order pipeline, split 16 kB L1 instruction and data caches, a branch prediction unit, and a DDR2 memory controller. Synthesis, mapping, translation, and place & route were done with Quartus 7.1. The prototype system is implemented on an Altera board utilizing the Stratix II FPGA chip with a nominal clock speed of 80 MHz.

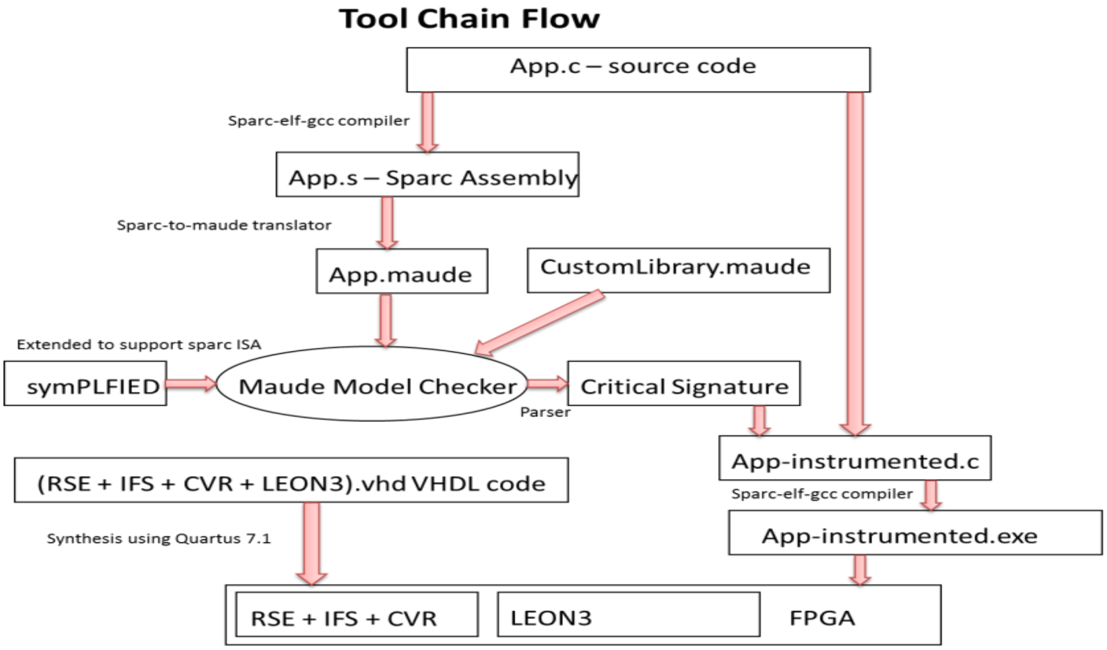


Fig. 9.1: Tool chain flow for IFS

The software tool chain (Fig. 9.1) is based upon the Maude-based model checker, which is used for symbolic-fault-injection-based analysis of an application's source code. The source code is translated to SPARC assembly using the sparc-elf-gcc cross-compiler, which is then translated to the corresponding Maude equations using a Perl-based SPARC-to-Maude translator. A custom library has also been implemented and provided for use with Maude-based applications to successfully run and analyze the whole application. After the exhaustive symbolic fault injection experiment, the result is automatically parsed to get the signature for the application. Through the use of the signature, the original source code of the application is instrumented with the necessary check instruction that will configure the IFS hardware and enforce runtime checking. The IFS module (along with the reliability module, CVR) is implemented as a part of the RSE (Reliability and Security Engine) and is interfaced with the main integer unit pipeline of the Leon3 processor. The code base is VHDL, which is synthesized, placed, and routed onto a Stratix II FPGA fabric. The ELF (executable and linkable) format can be run as a standalone code on this FPGA board to make performance measurements.

### 9.2 Benchmarks

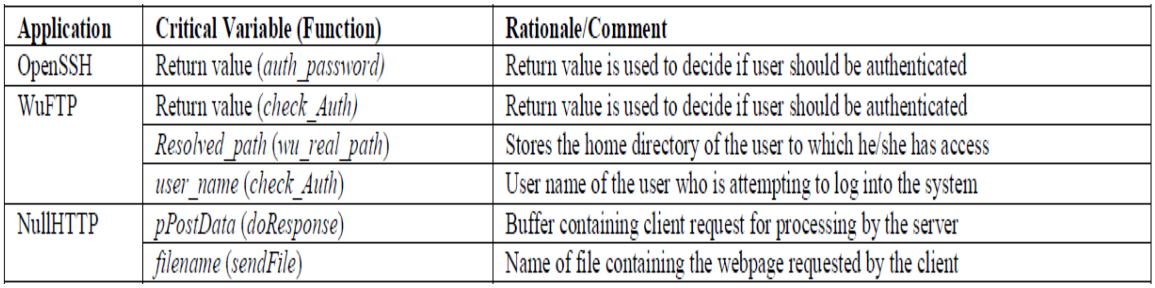
We demonstrated the IFS technique on server applications. We chose server applications because (1) they are typically executed with superuser privileges, which makes them extremely attractive targets for attackers; (2) they are often organized as separate software modules, each of which performs a specific function in the program (for example, the authentication module is responsible for ensuring that only legitimate users are able to gain access to the system); and (3) they consist of different modules executing in a single address space, which allows a malicious module to infiltrate security-critical modules in the application. The server applications considered are as follows:

1. **OpenSSH:** Implementation of the Secure Shell (SSH) protocol. Consists of over 50000 lines of C code.
2. **WuFTP:** Implementation of the File Transfer Protocol (FTP), consisting of over 25000 lines of C code.
3. **NullHTTP:** A small and efficient multithreaded HTTP server consisting of about 2500 lines of C code.

## 9.3 Critical Variables

In each of the target applications, we chose critical variables based on possible insider attacks that may be launched against the application. The insider attacks considered are as follows: OpenSSH: The insider allows a colluding user to be authenticated in spite of providing the wrong password. WuFTP: The insider allows spoofing of a user identity in order to access the user files/directories and perform malicious activities so that the user is blamed for the activities. NullHTTP: The insider modifies the client request or the response in order to send malicious or unintended content to the user. Table 9.1 [94] shows the critical variables in each application that are chosen (manually) to correspond to the above attacks.

Table 9.1: Benchmark applications with corresponding critical variables



| Application | Critical Variable (Function) | Rationale/Comment |
|---|---|---|
| OpenSSH | Return value (*auth_password*) | Return value is used to decide if user should be authenticated |
| WuFTP | Return value (*check_Auth*) | Return value is used to decide if user should be authenticated |
| | *Resolved_path* (*wu_real_path*) | Stores the home directory of the user to which he/she has access |
| | *user_name* (*check_Auth*) | User name of the user who is attempting to log into the system |
| NullHTTP | *pPostData* (*doResponse*) | Buffer containing client request for processing by the server |
| | *filename* (*sendFile*) | Name of file containing the webpage requested by the client |

## 9.4 Experimental Results

**Performance Overhead:** In order to measure the performance overhead of the IFS technique, we executed the original, noninstrumented program, the instrumented version of the program for software implementation of IFS, and the instrumented version of the program that is supported by the hardware module of the RSE interface. The measurements were conducted on two different platforms on a 2.0 GHz Pentium 4 single-core X86 system with 2 GB of RAM and on a Leon3 processor (SPARC v8 ISA) with a nominal clock speed of 80 MHz.

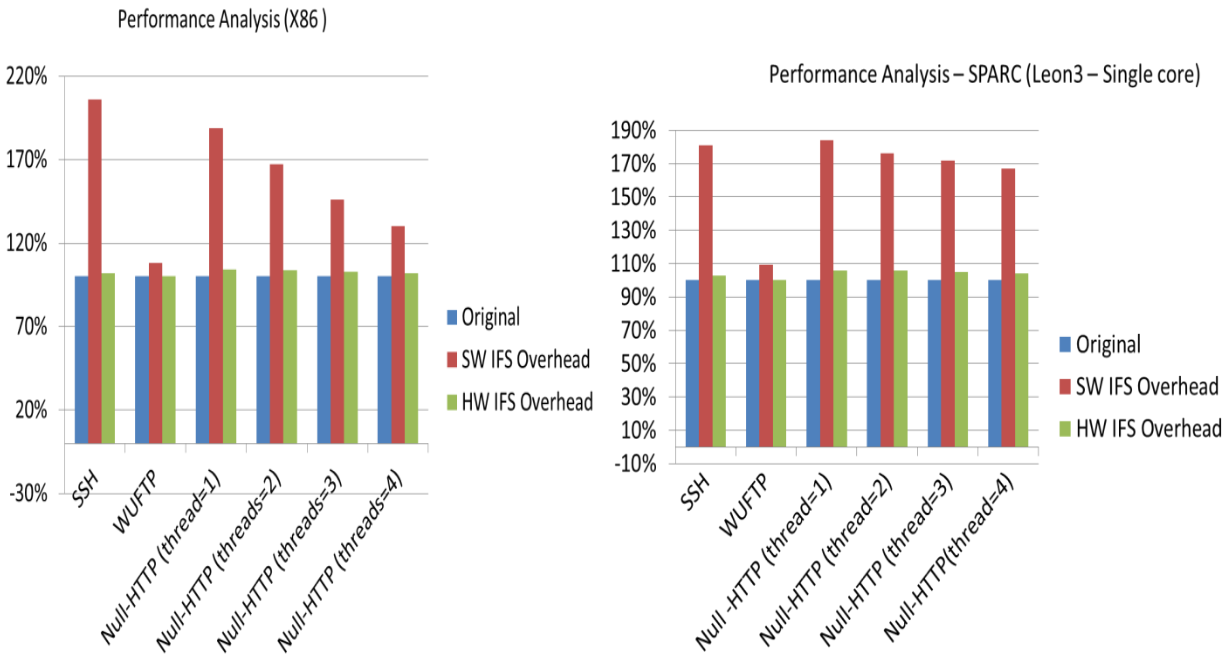


Fig. 9.2: Performance overhead results for benchmarks

From the results in Figure 9.2, it can be concluded that the performance overhead for the software implementation of IFS is highly dependent on the nature of the application and the choice of critical data. The reason is that the software IFS implementation is dependent on the number of IFS function calls, critical variables, and backward slices computed. For the SSH application, this overhead was as high as 106%, but FTP introduced only 8.5% overhead. For the Null-HTTP server application, the software overhead was 89% for a single thread, but decreased gradually to 30% when the number of threads was increased to 4. The reason was not the increased parallelism but the increasing number of threads on the single core, causing false sharing of data structures that lead to cache line invalidations. That increased the time it took the application to execute, which made the software version of IFS look more efficient even when the overhead was as high as 85% on average. On average, the performance overhead for the software implementation varied from 88% to 100% for the benchmark application. From the graphs, one can observe that the hardware-based IFS for all the applications has a performance overhead of as low as 3% to 4%, even when the whole stub was guarded by IFS, which resulted in a more conservative measurement than we would get from protecting only a few critical variables in a function. The overhead is as low as 0.08% for WuFTP, 1.8% for SSH, and 3.7% for the http server application. Similar observations can be made when the three different versions of the application are run on a platform such as Leon3 processor-based hardware. For the software implementation, the overhead varied from 81% for SSH, to 9.4% for WuFTP, to 67-84% for the nullHTTP. The hardware-based IFS performance overheads were very low for these applications: 2% for SSH, 0.06% for WuFTP, and 5% for nullHTTP server application. The performance overhead for the hardware-based IFS is due to the check instructions that configure the IFS hardware during the initial loading of the program; enforcing runtime checks does not hinder the performance, as the checking is done in parallel to the instruction execution without any stalling of the pipeline.

**Hardware Overhead:** In this section, we present the hardware resource usage for the implementation of the RSE with reliability (CVR) and security (IFS) modules, and we compare it to that of other processor units such as cache and MMU.

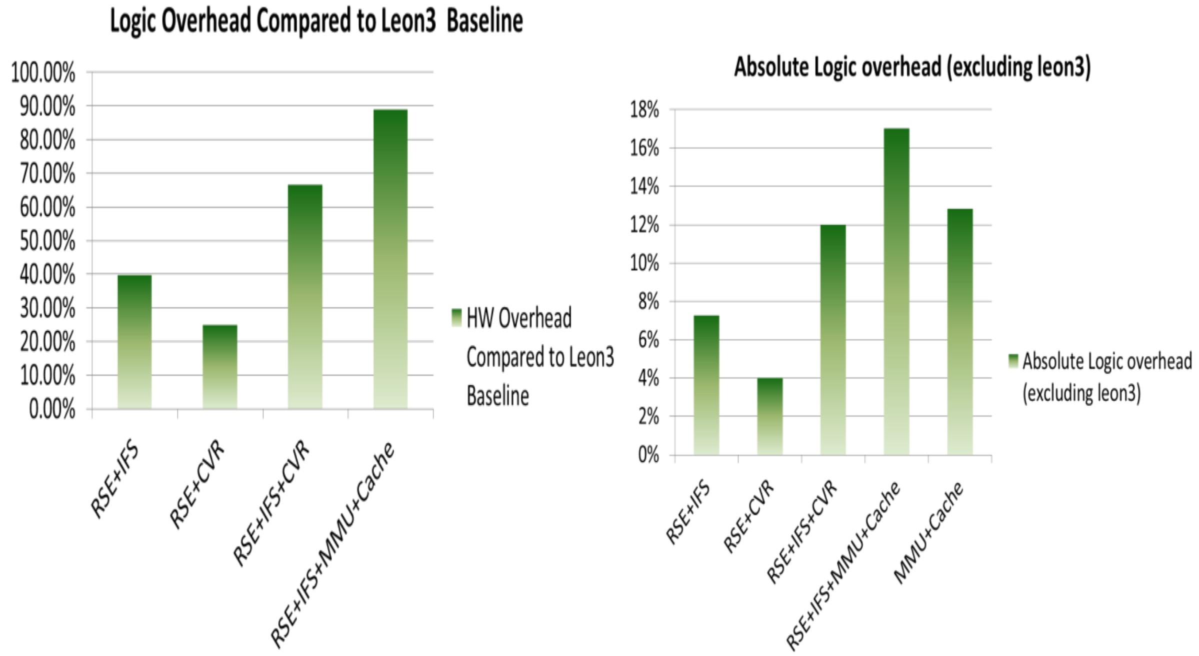


Fig. 9.3: Hardware logic overhead results of RSE modules

Figure 9.3 depicts the resource usage of the RSE interface along with the CVR and IFS modules relative to the Leon3 processor and resources available on the FPGA board. The RSE along with the IFS module takes up about 39.8% of the resources used by the Leon3 processor, which represents about about 7% of the resources available on the board. RSE with both IFS and CVR modules incurs 66.67% of the resource overhead incurred by to the Leon3 processor, which is equivalent to 12% of the FPGA board.

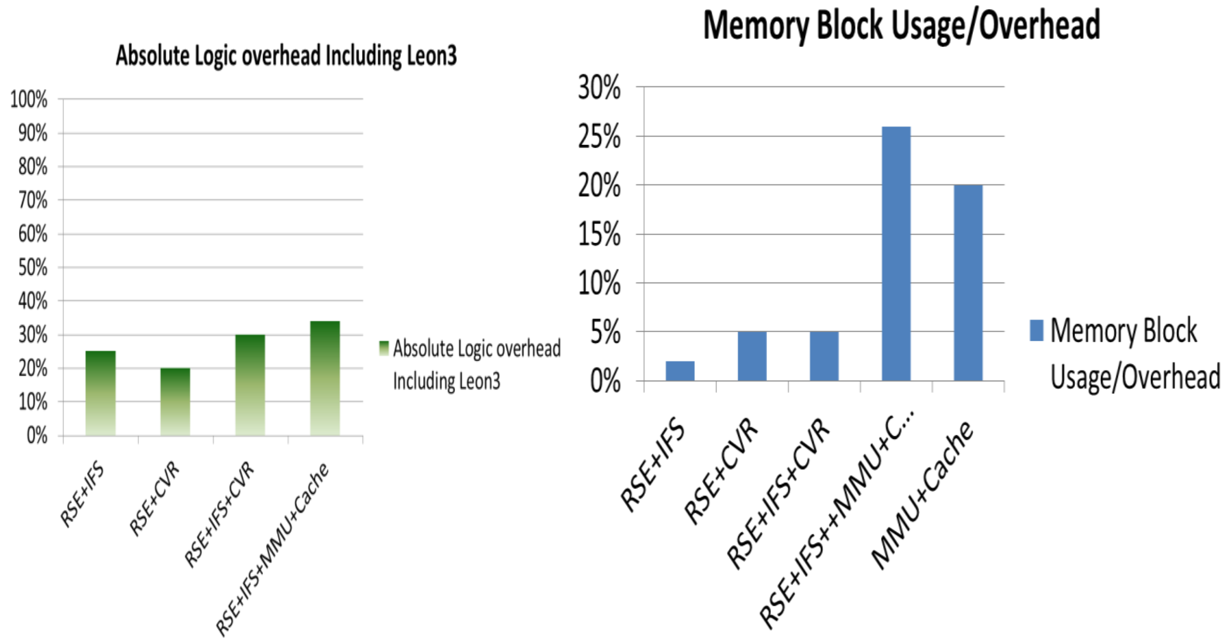


Fig. 9.4: Hardware (including Leon3) and memory logic overhead results of RSE modules

Figure 9.4 shows that the full-blown processor along with the RSE interface and the IFS and CVR modules consumes at most 30% of the board's logic units. The processor and the IFS module consume one out of four logic units available. If we look at the memory units usage, the IFS module itself uses almost 0 memory blocks, and the 2% shown in the graphs is due to the other memory components of the processor. The CVR module has its own memory and uses up to 5% of the board resources. Other units, like cache and MMU (memory management unit), take up much larger chunks of memory on the boards than the IFS and CVR modules do, indicating the potential benefit of mapping more components to memory blocks and thereby further reducing logic resource use. Also, the availability of large memory blocks can be leveraged to protect a larger portion of the application source code and increase the size of the signature. To minimize delay in runtime checks, the current implementation of the IFS uses a configurable stack frame of registers to store the signature; which could easily be extended to map to local on-chip memory components of the boards, having more space to store signatures of multiple applications running on top of the operating system.

### 9.5 Discussion and Future Work

**IFS on superscalar processors:** Interestingly, the complexity introduced by superscalar processor architecture is helpful to the Information Flow Signatures technique. For example, because of the reorder buffer and store buffers present on these processors, the technique can actually trigger an alarm *before* an instruction or memory operation is committed. Incorporation of control signals from these structures into the RSE is straightforward, and has been demonstrated previously on a DLX superscalar processor [95].

**Multicore architectures:** Designing the RSE for many-core architectures presents several challenges. In order for the Information Flow Signatures technique to work fully, instructions executed on each core must be checked against a global view of the signatures. Is this best implemented through a single Information Flow Signatures module that has instruction and data signature modules containing all running applications from each core? If multiple modules protecting each core separately are used, what is the best method for maintaining coherence among them? These questions raise topics similar to some found in Intel processor errata. For example, erratum AH39 for the Core 2 Duo Centrino architecture states, “Cache Data Access Request from One Core Hitting a Modified Line in the L1 Data Cache of the Other Core May Cause Unpredictable System Behavior.” Such bugs beg a more general question: If a component of a processor is faulty, to what extent can we rely on processor to provide information needed for security protection?

**Operating Systems**: For multiple applications to take advantage of the security provided by the IFS technique, modifications to the kernel code base are needed, because the same virtual address can refer to different instructions across different applications running at the same time. The operating system on a context switch should also switch the signature of the application to the extended task structure and reconfigure the IFS with the stored signature when the scheduler schedules the application again. This would allow multiple tasks with different signatures to execute at the same time. For embedded applications without MMU support, one needs to export the object code assembly generated by an ELF binary dump to create a flat binary, because SymPLFIED analysis is done on standalone application source code and not on the whole operating system. Running millions of lines of code on model checkers incurs the state explosion problem, although there are ways to minimize the time consumed for this analysis. Such systems run on OSes that do not have virtual memory implemented and hence would require a small modification to the IFS hardware. The current version of IFS works in the absolute addressing mode, which is sufficient for most of the modern processors with MMU support and OSes that have virtual memory management implemented. Modifying the hardware to work with relative addressing mode can easily be done just as the address translation mechanism. The complexity introduced by the address translation mechanism can be further reduced with a little software support that would introduce slight modification in the SymPLFIED analysis but result in the same hardware overhead and almost nil performance overhead. The operating system running on this embedded system would still require the added ability to reconfigure the IFS with signatures on a context switch.

### 9.6 Conclusions

Because of the current trend towards increasing processor design complexity, more bugs are being introduced in new architecture. The security implications of processor errata are significant, and we suggest that the current paradigm of protecting a system using a virtual fence will not suffice in the near future, as unknown vulnerabilities will be present in the processor or computer system.

In this thesis, we presented the hardware architecture used to enforce the Information Flow Signatures security technique. This combined hardware-software technique allows for trustworthy execution of instructions that influence security-critical data, even in the face of vulnerabilities that exist within a system. The technique detects any deviation from the behavior of the application described by the source code. By using the Reliability and Security Engine as an abstraction of signals in the pipeline, we are able to implement the Information Flow Signatures checking module without modification to the processor pipeline. The module itself proves to have a small footprint of less than 7% of all the resources present, and has no effect on the performance of the processor. Future extensions to the hardware can lower the performance overhead introduced by the software portion of the technique.

## REFERENCES

[1] CERT. [Online]. www.cert.org

[2] Aleph1. (1996, Nov.) Smashing the stack for fun and profit. *Phrack Magazine* http://www.phrack.com/issues.html?issue=49&id=14

[3] J. Pincus and B. Baker, “Beyond stack smashing: Recent advances in exploiting buffer overruns,” *IEEE Security and Privacy*, vol. 2, no. 4, pp. 20-27, 2004.

[4] J. Afek and A. Sharabani, “Dangling Pointer: Smashing the pointer for fun and profit,” Watchfire white paper, Aug. 2007.

[5] Advanced Doug Lea's Malloc exploits. *Phrack Magazine*. (2003, Sept.) http://www.phrack.com/issues.html?issue=61&id=6

[6] N. Nakka, Z. Kalbarczyk, R. K. Iyer, and J. Xu, “An architectural framework for providing reliability and security support,” in *International Conference on Dependable Systems and Networks*, 2004, pp. 585-594.

[7] M.R. Randazzo et al., *Insider Threat Study: Illicit Cyber Activity in the Banking and Finance Sector*. U.S. Secret Service and CERT Coordination Center/Software Engineering Institute: Philadelphia 2004, PA. p. 25.

[8] M. M. Keeney et al., Insider Threat Study: Computer System Sabotage in Critical Infrastructure Sectors. 2005, CERT/CC: Philadelphia, PA.

[9] A. Dixit, R. Heald, and A. Wood, “Trends from ten years of soft error experimentation,” in *IEEE Workshop on Silicon Errors in Logic – System Effects*, March 2009.

[10] I. Lee and R. K. Iyer, “Software dependability in the tandem GUARDIAN system,” *IEEE Transactions on Software Engineering*, vol. 21, no. 5, pp. 455-467, 1995.

[11] T. J. Siegel, E. Pfeffer, and J. A. Magee, “The IBM eServer z990 microprocessor,” *IBM Journal of Research and Development*, vol. 48, no. 304, pp. 295-309, 2004.

[12] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer, “Automated derivation of application-aware error detectors using static analysis,” in *Proceedings of the International Online Testing Symposium*, July 2007, pp. 211-216.

[13] T. Zhang, X. Zhuang, S. Pande, and W. Lee, “Anomalous path detection with hardware support,” in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2005, pp. 43-54.

[14] K. Pattabiraman, N. Nakka, Z. Kalbarczyk and R. K. Iyer, *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2008. www.crhc.illinois.edu/DEPEND/pubs/papers/SymPLFIED-final.pdf

[15] W. Shi, H.-H. S. Lee, C. Lu, and M. Ghosh, “Towards the issues in architectural support for protection of software execution,” *SIGARCH Comput. Archit. News*, vol. 33, no. 1, pp. 6–15, 2005.

[16] W. Shi, H.-H. S. Lee, C. Lu, and T. Zhang, “Attacks and risk analysis for hardware supported software copy protection systems,” in *DRM ’04: Proceedings of the 4th ACM workshop on Digital Rights Management. ACM Press*, 2004, pp. 54–62.

[17] W. Shi, H.-H. S. Lee, M. Ghosh, and C. Lu, “Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems,” in *PACT’04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques.* Washington, DC, USA: IEEE Computer Society, 2004, pp. 123–134.

[18] R. Best, “Crypto-microprocessor that executes enciphered programs,” United States Patent 4,433,901, August 1984.

[19] R. Best, “Preventing software piracy with crypto-microprocessors,” in *Proceedings of the IEEE Spring COMPCON* 80, 1980, p. 146.

[20] S. Kent, “Protecting externally supplied software in small computers,” PhD dissertation, Laboratary of computer Science, Massachusetts Inst. Od technology, 1980.

[21] D. S. Maxim, “Ds5002fp secure microprocessor chip,” http://www.maxim-ic.com/quick view2.cfm/qv pk/2949

[22] J. Tygar and B. Yee, “Dyad: A system for using physically secure coprocessors,” in *IP Workshop Proceedings*, 1994. www:http://www.cni.org/ docs/ima.ip-workshop/www/Tygar.Yee.html

[23] J. D. Tygar and B. S. Yee, “Strongbox: A system for self securing programs,” *in CMU Computer Science: 25th Anniversary Commemorative. ACM*, 1991.

[24] B. Chen and R. Morris, “Certifying program execution with secure processors,” in *HotOS*, 2003, pp. 133–138.

[25] T. Kgil, L. Falk, and T. Mudge, “Chiplock: Support for secure microarchitectures,” *SIGARCH Comput. Archit. News*, vol. 33, no. 1, pp. 134–143, 2005.

[26] T. Gilmont, J.-D. Legat, and J. Quisquater, “Enhancing security in the memory management unit,” in *EUROMICRO*, 1999, p. 1449.

[27] T. Gilmont, L. Legat, and J. Quisquarter, “Hardware security for software privacy support,” *Electronic Letters*, vol. 35, no. 24, pp. 2096–2098, 1999.

[28] J. Zambreno, A. Choudhary, R. Simha, B. Narahari, and N. Memon, “Safe-ops: An approach to embedded software security,” *Trans. on Embedded Computing Sys*., vol. 4, no. 1, pp. 189–210, 2005.

[29] J. Zambreno, A. Choudhary, R. Simha, and B. Narahari, “Flexible software protection using hardware/software codesign techniques,” in *DATE ’04: Proceedings of the Conference on Design, Automation and Test in Europe. Washington, DC, USA: IEEE Computer Society*, 2004, p. 10636.

[30] O. Gelbart, P. Ott, B. Narahari, R. Simha, A. Choudhary, and J. Zambreno, “Codesseal: A compiler/fpga approach to secure applications,” in *In Proceedings of the IEEE International Conference on Intelligence and Security Informatics (ISI)*, 2005, pp. 530–535.

[31] D. Burger and T. Austin, “The simplescalar toolset 4.0.” http://www.simplescalar.com/

[32] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, “Terra: A virtual machine-based platform for trusted computing,” in *SOSP ’03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles. New York, NY, USA: ACM Press*, 2003, pp. 193–206.

[33] C. Wang, “A security architecture for survivability mechanisms,” 2000. citeseer.ist.psu.edu/wang00security.html

[34] D. C. DuVarney, V. N. Venkatakrishnan, and S. Bhatkar, “SELF: A transparent security extension for ELF binaries,” in *NSPW ’03: Proceedings of the 2003 Workshop on New Security Paradigms. ACM Press*, 2003, pp. 29–38.

[35] ELF: Executable Linking Format, Unix System Laboratories: Tool Interface Standard, version 1.1.

[36] “Next-generation secure computing base,” Microsoft.com, 2004

[37] X. Zhuang, T. Zhang, H.-H. S. Lee, and S. Pande, “Hardware assisted control flow obfuscation for embedded processors,” *in CASES ’04: Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems. ACM Press*, 2004, pp. 292–302.

[38] X. Zhuang, T. Zhang, and S. Pande, “Hide: An infrastructure for efficiently protecting information leakage on the address bus,” in *ASPLOS*, 2004, pp. 72–84.

[39] S. P. E. Corporation, “Spec benchmark suite.” [Online]. Available: http: //www.spec.org/

[40] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, “Randomized instruction set emulation to disrupt binary code injection attacks,” in *CCS ’03: Proceedings of the 10th ACM Conference on Computer and Communications Security. ACM Press*, 2003, pp. 281–289.

[41] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, “Secure program execution via dynamic information flow tracking,” *SIGPLAN Not.*, vol. 39, no. 11, pp. 85–96, 2004.

[42] M. Milenkovi, A. Milenkovi, and E. Jovanov, “A framework for trusted instruction execution via basic block signature verification,” in *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference. ACM Press*, 2004, pp. 191–196.

[43] Fiskiran and R. B. Lee, “Runtime execution monitoring (rem) to detect and prevent malicious code execution.” in *ICCD*, 2004, pp. 452–457.

[44] E. M. Clarke, J. M. Wing, R. Alur, R. Cleaveland, D. Dill, A. Emerson, S. Garland, S. German, J. Guttag, A. Hall, T. Henzinger, G. Holzmann, C. Jones, R. Kurshan, N. Leveson, K. McMillan, J. Moore, D. Peled, A. Pnueli, J. Rushby, N. Shankar, J. Sifakis, P. Sistla, B. Steffen, P. Wolper, J. Woodcock, and P. Zave, “Formal methods: State of the art and future directions,” *ACM Computing Surveys*, vol. 28, no. 4, pp. 626–643, 1996.

[45] J. P. Bowen and M. G. Hinchey, “Ten commandments of formal methods,” *IEEE Computer*, vol. 28, no. 4, pp. 56–63, 1995.

[46] J. Baek, “Construction and formal security analysis of cryptographic schemes in the public key setting,” Ph.D. dissertation, School of Network Computing Monash University, January 2004.

[47] S. Older and S.-K. Chin, “Formal methods for assuring security of protocols,” *Computer Journal*, vol. 45, no. 1, pp. 46–54, 2002

[48] C. J. F. Cremers, S. Mauw and E. P. De Vink “Formal methods for security protocols: Three examples of the black-box approach,” *NVTI newsletter*, vol. 7, pp. 21–32, newsletter of the Dutch Association for Theoretical Computing Scientists.

[49] G. Bella and E. Riccobene, “A realistic environment for crypto-protocol analyses by ASMs,” in *Proceedings of the 28th Annual Conference of the German Society of Computer Science.* Technical Report, Magdeburg University, 1998.

[50] V. Shmatikov and J. Mitchell, “Finite-state analysis of two contract signing protocols,” *Theoretical Computer Science* vol. 283, no. 2, Pages 419-450, 2002.

[51] S. Chen, Z. Kalbarczyk, J. Xu, and R. K. Iyer, “A data-driven finite state machine model for analyzing security vulnerabilities.” in *DSN,* 2003, pp. 605–614.

[52] Lowe, “Casper: A compiler for the analysis of security protocols,” in *PCSFW: Proceedings of The 10th Computer Security Foundations Workshop. IEEE Computer Society Press,* 1997.

[53] M. Butler, M. Leuschel, S. L. Presti, and P. Turner, “The use of formal methods in the analysis of trust (position paper),” in *iTrust*, 2004, pp. 333–339.

[54] S. W. Smith and V. Austel, “Trusting trusted hardware: Towards a formal model for programmable secure coprocessors,” in *3rd USENIX Workshop on Electronic Commerce. Boston, Massachusetts, USA: USENIX Association*, August 1998.

[55] D. Oheimb, G. Walter, and V. Lotz, “A formal security model of the infineon SLE 88 smart card memory management,” in *Proc. of the 8th European Symposium on Research in Computer Security (ESORICS), ser. LNCS*, vol. 2808. Springer, 2003.

[56] D. Oheimb and V. Lotz, “Formal Security Analysis with Interacting State Machines,” in *Proc. of the 7th European Symposium on Research in Computer Security (ESORICS)*, vol. 2502. Springer, 2002, pp. 212–228.

[57] D. Oheimb and V. Lotz, “Generic Interacting State Machines and their instantiation with dynamic features,” in *Formal Methods and Software Engineering (ICFEM)*, vol. 2885. Springer, Nov. 2003, pp. 144–166.

[58] D. Oheimb, “Interacting State Machines: a stateful approach to proving security,” in *Formal Aspects of Security, ser. LNCS*, vol. 2629. Springer, 2002, pp. 15–32.

[59] D. Oheimb, “Information flow control revisited: Noninfluence = Noninterference + Nonleakage,” in *Computer Security – ESORICS* 2004, vol. 3193. Springer, 2004, pp. 225–243.

[60] S. Chen, Z. Kalbarczyk, J. Xu, and R. K. Iyer, “Formal reasoning of various categories of widely exploited security vulnerabilities using pointer taintedness semantics,” *Proceeding of the IFIP International conference on Information Security (SEC)*, 2004.

[61] D. J. Lie, A. Chou, D. Engler, and D. Dill, “A simple method for extracting models from protocol code,” *ISCA*, pp. 192–203, 2001

[62] D. Lie, J. Mitchell, C. A. Thekkath, and M. Horowitz, “Specifying and verifying hardware for tamper-resistant software,” *in SP ’03: Proceedings of the 2003 IEEE Symposium on Security and Privacy. Washington, DC, USA: IEEE Computer Society*, 2003, p. 166.

[63] F. Tip, “A survey of program slicing techniques,” *Journal of Programming Languages*, 1995, vol. 3, no. 3, pp. 121-189.

[64] K. Pattabiraman et al. “Automated Derivation of Application-Aware Error Detectors using Static Analysis,” in *Internation Online Testing Symposium (IOLTS)*. IEEE 2007.

[65] D. T. Stott, B. Floering, Z. Kalbarczyk, and R. K. Iyer, "A framework for assessing dependability in distributed systems with lightweight fault injectors," in *Proceedings of the 4th International Computer Performance and Dependability Symposium,* Washington, DC, USA, 2000, p. 91

[66] J. Gaisler, Gaisler Research AB, “A structured VHDL design method,” http://www.gaisler.com/doc/vhdl 2proc.pdf, 2006.

[67] G. Lyle, S. Chen, K. Pattabiraman, Z. Kalbarczyk and R. K. Iyer “An End-to-end Approach for the Automatic Derivation of Application-aware Error Detectors,” *Proceedings of the International Conference on Dependable Systems and Networks (DSN)* , Estoril, Portugal, 2009.

[68] M. Dalton, H. Kannan, C. Kozyrakis, “Raksha: A Flexible Information Flow Architecture for Software Security,” in *Proc. of the ACM Intl. Symp. on Computer Architecture*, June 9–13, 2007, San Diego.

[69] S. T. King et al. “Designing and implementing malicious hardware,” 2008. *USENIX Association Berkeley*, CA, USA.

[70] Y. M. Alkabani et al., “Active hardware metering for intellectual property protection and security,” *Proceedings of 16th USENIX Security Symposium*, 2007.

[71] S. Upadhyaya, “Real-Time Intrusion Detection with Emphasis on Insider Attacks,” *Lecture Notes in Computer Science*, 2003. 2776, p. 82-85 .

[72] O. Sheyner, et al., Automated Generation and Analysis of Attack Graphs, in *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. 2002, IEEE Computer Society.

[73] J. Rhee et al., “Defeating Dynamic Data Kernel Rootkit Attacks via VMM-based Guest-Transparent Monitoring,” *International conference on Availability, Reliability and Security*, 2009.

[74] Baliga et al. “Automatic Inference and Enforcement of Kernel Data Structure Invariants,” *IEEE Computer Society Washington*, DC, USA. 2008.

[75] E. Skoudis et al. *Malware: Fighting Malicious Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[76] D. Lie, J. Mitchell, C. A. Thekkath, and M. Horowitz, “Specifying and verifying hardware for tamper-resistant software,” in *SP ’03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2003, p. 166.

[77] D. Lie, C. A. Thekkath, and M. Horowitz, “Implementing an untrusted operating system on trusted hardware,” in *SOSP ’03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM Press, 2003, pp. 178–192.

[78] 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 *Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 168–177.

[79] Stanford, “Simos: The complete machine simulator.” [Online]. Available: http://simos.stanford.edu/

[80] J. Yang, Y. Zhang, and L. Gao, “Fast secure processor for inhibiting software piracy and tampering,” in *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society,* 2003, p. 351.

[81] W. Shi, H. Lee, C. Lu, and M. Ghosh, “High speed memory centric protection on software execution using one-time-pad prediction,” in *PACT: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2004, pp.123–134 2004.

[82] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, “Efficient memory integrity verification and encryption for secure processors,” in *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2003, p. 339.

[83] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, “Caches and hash trees for efficient memory integrity verification,” in *HPCA ’03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2003, p. 295.

[84] W. A. Arbaugh, D. J. Farber, and J. M. Smith, “A secure and reliable bootstrap architecture,” in *SP ’97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 1997, p. 65.

[85] Sovarel et al., “Where's the FEEB? the effectiveness of instruction set randomization,” in *Proceedings of the 14th Conference on USENIX Security Symposium* – Vol. 14, 2005.

[86] Y. Chen et al., “Oblivious Hashing: A Stealthy Software Integrity Verification Primitive,” in *Revised Papers from the 5th International Workshop on Information Hiding*. 2003, Springer-Verlag.

[87] M. Jacob et al., “Towards integral binary execution: implementing oblivious hashing using overlapped instruction encodings,” in *Proceedings of the 9th Workshop on Multimedia Security*. 2007, ACM: Dallas, Texas, USA.

[88] Seshadri et al., “Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems,” in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. 2005, ACM: Brighton, United Kingdom.

[89] F. Monrose et al. “Distributed Execution with Remote Audit,” in *ISOC Network and Distributed System Security Symposium*, 1999.

[90] N. Provos et al., “Preventing privilege escalation,” in *Proceedings of the 12th Conference on USENIX Security Symposium* – Vol. 12. 2003, USENIX Association: Washington, DC.

[91] D. Brumley et al., “Privtrans: Automatically partitioning programs for privilege separation,” *Proceedings of the 13th USENIX Security Symposium*, 2004.

[92] K. Pattabiraman et al., “Samurai: protecting critical data in unsafe languages,” in *Proceedings of the 3$^r$ ACM SIGOPS/EuroSys European Conference on Computer Systems* 2008. 2008, ACM: Glasgow, Scotland UK.

[93] Nguyen-Tuong et al. “Security through Redundant Data Diversity,” in *IEEE International Conference on Dependable Systems and Networks (DSN).* IEEE 2008.

[94] K. Pattabiraman and R.K. Iyer, “Automated Derivation of Application-aware Error Detectors using Compiler Analysis,” *Technical Report UILU-ENG-07-2203*, Univ. of Illinois (Urbana Champaign), Jan. 2007.

[95] N. Nakka et. al., “An Architectural Framework for Providing Reliability and Security Support,” in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, 2004.