AUTOMATED DERIVATION OF APPLICATION-AWARE ERROR AND ATTACK DETECTORS

BY

KARTHIK PATTABIRAMAN

B.Tech., University of Madras, 2001
M.S., University of Illinois at Urbana-Champaign, 2004

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Abstract

As computer systems become more and more complex, it becomes harder to ensure that they are dependable i.e. reliable and secure. Existing dependability techniques do not take into account the characteristics of the application and hence detect errors that may not manifest in the application. This results in wasteful detections and high overheads. In contrast to these techniques, this dissertation proposes a novel paradigm called “Application-Aware Dependability”, which leverages application properties to provide low-overhead, targeted detection of errors and attacks that impact the application. The dissertation focuses on derivation, validation and implementation of application-aware error and attack detectors.

The key insight in this dissertation is that certain data in the program is more important than other data from a reliability or security point of view (we call this the critical data). Protecting only the critical data provides significant performance improvements while achieving high detection coverage. The technique derives error and attack detectors to detect corruptions of critical data at runtime using a combination of static and dynamic approaches. The derived detectors are validated using both experimental approaches and formal verification. The experimental approaches validate the detectors using random fault-injection and known security attacks. The formal approach considers the effect of all possible errors and attacks according to a given fault or threat model and finds the corner cases that escape detection. The detectors have also been implemented in reconfigurable hardware in the context of the Reliability and Security Engine (RSE) [1].

To my parents and teachers

Acknowledgements

First and foremost, I would like to thank my advisor Professor Ravishankar Iyer, for his support and encouragement throughout this dissertation. Ravi constantly encouraged me to explore new ideas and push established boundaries. I would also like to thank Dr. Zbigniew Kalbarczyk, who has been a mentor, colleague and friend through my PhD. Zbigniew provided active feedback and advice, and but for his patience and support this dissertation would not have been possible. I would like to thank the members of my dissertation committee, namely Prof. Vikram Adve, Prof. Wen-mei Hwu and Prof. Grigore Rosu, for their advice and support during various stages of this dissertation.

I have also been fortunate to have a wonderful set of colleagues in the DEPEND group, many of whom were collaborators in various joint projects. In particular, Nithin Nakka, Giacinto Paulo Saggesse, Daniel Chen, William Healey, Peter Klemperer, Paul Dabrowski, Shelley Chen, Galen Lyle and Flore Yuan have all collaborated with me at various times in developing the ideas in this dissertation. Special thanks to my office mate Long Wang for patiently listening to my trials and triumphs during the course of my PhD. I would like to especially thank Shuo Chen, who was a great source of inspiration and helped me publish my first research paper at Illinois. I would also like to thank Heidi Leerkamp who helped with many day-to-day administrative tasks and support.

I also owe a lot to Dr. Benjamin Zorn, who has been an active mentor during various internships and visits at Microsoft Research during the course of my PhD. My interactions with him have helped shape many of the ideas in this dissertation.

I would like to thank my wife Padmapriya Kandhadai, who has been a never-ending source of emotional support during the tumultuous course of a PhD. But for her, I would probably have not stuck it out till the finish line. Thanks also to my parents for patiently waiting without asking the inevitable question of when (if ever) I would finish.

Finally, a big thank you to all my friends at UIUC and elsewhere for helping me not take myself too seriously. A special thanks to the “potluckers gang” (you know who you are) for all the endless hours we spent talking about life. But for you guys, this dissertation would have been done a lot sooner, but I would be all the poorer for the experience.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

1.1 MOTIVATION

The increasing complexity of computer systems and their deployment in mission- and life-critical applications are driving the need to build reliable and secure computer systems. Compounding the situation, the Internet's ubiquity has made systems much more vulnerable to malicious attacks that can have far-reaching implications on our daily lives. Traditionally, reliability has meant expensive mainframe computers running in lock-step and security has meant access control and cryptography support. However, the Internet's phenomenal growth has led to the large-scale adoption of networked computer systems for a diverse cross section of applications with highly varying requirements. In this all-pervasive computing environment, the need for reliability and security has expanded from a few expensive, proprietary systems to something that is a basic computing necessity. This new paradigm has important consequences:

- Networked systems stretch the boundary of fault models from a single application or node failure to failures that could propagate and affect other components, subsystems, and systems, and
- Attackers can exploit vulnerabilities in operating systems and applications with relative ease. Due to the complex interlinking of systems, attacks on even a single component of the system can lead to a compromise of the entire system.

Users ultimately want their applications to continue to operate without interruption, despite attacks and failures, but as systems become more complex, this task becomes more difficult. The traditional one-size-fits-all approach to security and reliability is no longer sufficient or acceptable from the end-user's perspective. Spectacular system failures due to malicious tampering or mishandled accidental errors call for novel, application-specific approaches. This dissertation proposes the concept of application-aware dependability as an alternative to traditional heavyweight dependability approaches such as duplication and cryptography.

Application-aware dependability extracts application's characteristics and presents it to the underlying system, so that the system can tune itself to provide the optimal level of reliability and security to the application. This fits in with the idea of *utility computing [2, 3]*; or *cloud computing [4, 5]*, in which large computing farms configure themselves to execute complex applications for long periods of time with guaranteed performance and dependability. In this environment, the reliability or security of the physical hardware on which the application executes is less important than the dependability of the application. Further, as more and more computing shifts to the cloud, the value of a cloud-computing platform is governed more by the services provided to the application (be they for enhancing the application's performance, reliability or security) than the platform itself.

Hardware-based techniques have the advantage of low performance overheads because the hardware modules can perform security and reliability checking in parallel with the application. Because these techniques can detect errors close to their points of occurrence, low levels of detection latency are possible. This in turn ensures speedy recovery before errors and attacks can propagate in the system [2].

Application-aware techniques also expose knowledge of the underlying hardware platform to the application, so that the application can invoke the services exposed by the hardware at critical points in its execution to request reliability and security support. This allows the protection obtained and the performance overheads incurred to be configured based on the application's needs and characteristics. Clearly, it is very hard for the application-developer or system administrator to coordinate this complex interaction with the hardware. Therefore, it is important to develop automated techniques that can (1) Extract application properties and expose them to the underlying hardware, (2) Configure the hardware-based checks based on the extracted properties and (3) Instrument the application's code to invoke the hardware-based checks at strategic points in its execution. Further, it is necessary to validate the derived checks and evaluate their efficacy against both accidental and malicious errors.

The research question we address in this dissertation is as follows*: How do we automatically extract and validate application properties to provide low-latency, high-coverage error and attack detection using a combination of programmable hardware and software?* We first provide an overview of the reliability techniques and security techniques developed in this dissertation. We then provide an overview of the fault- and attack- models considered in this dissertation and outline its main contributions. Finally, we detail the overall frameworks developed in this dissertation for derivation, implementation and validation of application-aware error and attack detectors.

1.2 PROPOSED RELIABILITY TECHNIQUES

1.2.1 Introduction

Reliability techniques may be broadly classified into fault-avoidance and fault-tolerance techniques. Fault-avoidance techniques attempt to eliminate errors at software development time, prior to its deployment. Examples include program testing and static analysis techniques. Typically, fault-avoidance techniques target specific classes of errors (e.g. memory errors, uninitialized variables). Although these methods have been applied extensively, studies have shown that subtle software defects such as timing and synchronization errors persist in applications, and lead to application failures in operational settings [6-8].

In contrast to fault-avoidance techniques, fault-tolerance techniques provide detection of (and recovery from) general hardware and software errors. By far the most widely deployed fault-tolerance technique is duplication, which involves running two or more copies of a program and comparing their outputs. While duplication has been successfully deployed on selected commercial systems such as IBM mainframes and Tandem Non-stop computers [9], it has not found wide acceptance in Commodity Off-the-Shelf systems (COTS). This is because duplication incurs high performance overheads (up to 100 %), and may require the provision of special-purpose hardware to alleviate the performance overheads. However, the special hardware requires chip area (up to 33 % in the IBM Mainframe G5 processor [10]) and increases the complexity of the overall design. Further, the errors detected by duplication-based approaches that may not ultimately matter to the application, due to significant fault masking at the device level (80-90 %) [11] and at the architectural level (50-60 %) [12].

Failure-oblivious computing [13] takes the view that most errors do not affect the application’s execution, and hence does not recover from or correct errors as long as the system operates within its acceptability envelope. The acceptability envelope is defined as the set of acceptable (but not necessarily correct) behaviors of the system. For example, a web-server is considered to be operating within its acceptability envelope if it processes a request without writing to an undefined memory location. An aircraft controller is operating within its acceptability envelope as long as it does not lead to the aircraft accelerating beyond a certain threshold. While failure-oblivious computing is a promising approach if the acceptability envelope is well-defined, in practice it is hard to isolate the range of acceptable behaviors for a system. Further, failure-oblivious computing allows errors to stay undetected and propagate, which in turn can lead to massive failures. Hence, the failure-oblivious approach may not be well-suited for applications that exhibit high degrees of error propagation before crashing (if they crash).

This dissertation proposes a novel, low-overhead approach for providing high reliability to applications. It proposes insertion of error detectors (runtime checks) in the application’s code based on the application’s properties. This is achieved by extracting application properties using compiler-driven static and dynamic analysis, and converting the extracted properties into runtime checks. The properties are obeyed in any error-free execution of the program, but not in an erroneous execution. As a result, the checks can detect general hardware and software errors that impact program correctness and are not confined to particular types of faults.

While the detectors are application-specific and are derived on a per-application basis, the method for deriving and implementing detectors can be applied to any application. The method is completely automated and requires no intervention from the programmer.

1.2.2 Detector Placement

Studies have shown that undetected error propagation leads to extended system downtimes [14-16]. It is therefore essential, that errors are detected before they propagate and cause application failure. An effective error detection mechanism must necessarily limit the extent of error propagation and preempt application failure in order to enable speedy and sound recovery (after the error is detected).

The error detectors derived in this dissertation are placed at strategic locations in the application in order to prevent error propagation and preempt application failures (crashes). The locations encompass both the program variable that must be checked as well as the program point at which the check must be performed. The locations are chosen based on the application's dynamic dependence graph, which is constructed using the application's execution profile under representative inputs. For example, for a large application such as *gcc*, the detector placement methodology identifies a small number of strategic locations (10-100), at which placing (ideal) detectors can provide high coverage (80-90%) for errors leading to application failure [17].

1.2.3 Detector Derivation

Once the detector placement points and variables have been identified, error detectors are derived for the program variables (*critical variables*) at the identified points. The error detectors for critical variables are arithmetic and logical expressions that check whether the value of the critical variable was computed correctly i.e. according to the applications' code and/or semantics. Two approaches to derive error detectors are proposed as follows:

1. Based on dynamic execution traces of the application, gathered by instrumenting the values of critical variables and executing the application under representative inputs. An automatic approach learns the characteristics of the variable(s) based on pre-defined template patterns, and embeds the learned patterns as runtime checks in the application. The runtime checks are implemented in a programmable hardware framework, and are invoked through special instructions embedded in the application code at the detector placement points.

2. Based on the statically-generated backward program slice [18] of the critical variables at the detector placement points. The backward slice is specialized for each control-flow path in the application by the detector derivation technique. This specialization allows the compiler to optimize the backward slice aggressively and derive a minimized symbolic expression for the slice (called the *checking expression*). Programmable hardware is used to track control-paths at runtime and choose the checking expression corresponding to the executed path. The checking expression

recomputes the value of the critical variable and flags any deviation from the original as an error.

1.2.4 Detector Validation

Fault-injection is a commonly used approach to evaluate the efficacy of fault-tolerance mechanisms [19]. Fault-injection involves perturbing the code or data of the system (for example, by flipping a single bit) and studying the behavior of the system under the perturbation. We have evaluated the derived detectors through fault-injections in application data, and have shown that the detectors provide nearly duplication-levels of error-detection coverage for errors that matter to the application (at a fraction of the corresponding overheads). Because fault-injection is statistical in nature, it is not guaranteed to expose all errors under which the detector may fail. In order to ensure that the errors missed by the derived detectors do not lead to catastrophic consequences in safety- or mission- critical systems, it is important to evaluate the derived detectors exhaustively under all possible errors. However, exhaustive fault-injection often incurs considerable time and resource overheads.

Formal verification is a complementary approach to fault-injection that can exhaustively enumerate the effects of errors on fault-tolerance mechanisms (such as. detectors) and expose corner case scenarios that may be missed by traditional fault injection. We build a formal verification framework, SymPLFIED, to comprehensively enumerate all errors that evade detection and cause the program to fail. SymPLFIED operates directly on the assembly language representation of the program, and uses symbolic execution and model-checking to systematically consider the effect of all possible transient errors on the program according to a given fault-model. For each error, SymPLFIED finds whether the error was detected and if not, whether the error led to a failure in the application.

1.3 PROPOSED SECURITY TECHNIQUES

1.3.1 Introduction

Many existing approaches for security are piece-meal approaches, in the sense that they either protect from very specific types of attacks (e.g. Stackguard, which protects from certain types of stack-buffer overflow attacks [20]) or they suffer from high false-positive rates (e.g. system-call based intrusion detection [21]).

Techniques such as memory-safety checking [22-24] and taintedness [25-27], while providing comprehensive protection from security attacks, incur high performance overheads when done in software, which in turn limits their deployment in operational settings. When done in hardware, they high-false positive rates thereby necessitating traps to software, and in turn incur high performance overheads. Further, they require the entire application's code to be available for analysis, which is often not the case. Thus, they leave open the possibility that an untrusted third-party module may be used to attack the application (i.e. insider attacks).

Randomization is a low-overhead technique that has been used to protect programs from targeted attacks. By randomizing the layout of the stack, heap or static data items in a program [28-30], it is possible to obscure potential targets of an attacker, and hence foil the attack. The randomization can be carried out transparently to the application, with minimal modifications to the hardware or operating system. However, randomization based techniques can be broken by repeated undetected attacks on the application [31], or by carrying out targeted attacks through information-leaks in the program. Further, randomization techniques may not be effective against attacks launched by trusted insiders, as an insider may be able to determine the seed value used for randomization and hence identify the locations of the target objects.

Thus, we see that existing security techniques either incur high-performance overheads or are ineffective against trusted insiders in the same address space as the application. In contrast to these techniques, we propose a technique called *Information-Flow Signatures* (IFS) to protect critical data in applications from both external and insider attacks. The technique extracts the properties of the critical data based on the application's source language semantics, and enforces the extracted properties through runtime monitoring in software. Because the monitored properties are based on the inherent properties of the application, the technique incurs *no* false-positives. Further, by focusing on a subset of application data (*critical data*), the technique is able to ensure the integrity of the data with modest performance overheads.

1.3.2 Information-flow Signatures

Information-flow Signatures (IFS) encapsulate the dependencies among the instructions that are allowed to influence the value of the critical variables as per its source-level semantics. The reason for memory-corruption and insider attacks is the gap between a program's source-level semantics and its runtime execution semantics [32]. Hence, the proposed technique derives the Information-flow signature of the program's critical variables (identified by programmer using annotations) from its source-level semantics and checks the program at runtime for conformance to the signature. It is assumed that attackers will attempt to influence the critical variable by introducing new code in the system (e.g. code-injection attacks and insider attacks) or by overwriting the critical variable through instructions that are not allowed to write to the critical variable legitimately (e.g. memory corruption attacks). Both categories of attacks will cause the runtime behavior of the program to deviate from its statically derived Information-Flow Signature, and will hence be detected.

The proposed technique extracts the information-flow signatures of the program based on the backward slice of the critical variables in the program. This is similar to the static detector derivation technique in section 1.2.3 (Table 2 presents the main differences).

1.3.3 Formal Validation

The formal methodology for verification of error detectors has also been extended to verify security attack detectors. Similar to the SymPLFIED tool for evaluating error detectors, we developed an automated tool SymPLAID, to systematically enumerate all security attacks that evade detection and allow the attacker to achieve his/her goals. The attacks considered by SymPLAID include both memory corruption attacks as well as insider attacks. Given the application's code (in assembly language) and a set of attacker goals (in first-order logic), SymPLAID automatically identifies all possible attacks (value corruptions) that will allow the attacker to achieve his/her goals. However, unlike

SymPLFIED, SymPLAID precisely tracks the propagation of corrupted values in the program, thus identifying the value that must be corrupted by the attacker and the precise value that must be used to replace the original value in order to carry out the attack.

1.4 FAULT AND ATTACK MODELS

This section summarizes the fault- and attack- models used in this dissertation. The goal is to provide a broad overview of all faults and attacks that can be addressed using the techniques developed in this dissertation, rather than to provide a detailed characterization of the coverage of individual techniques (these are discussed in the relevant chapters).

The error and attacks can be classified into four broad categories as follows:

1. **Transient hardware errors**: These include soft-errors caused by radiation, single-event upsets due to timing and electrical defects or (in rare cases), faults due to design bugs in the processor that manifest only in exceptional or stressful circumstances.

2. **Transient software errors**: These include (1) memory-corruption errors caused by pointers writing outside their memory intended region (and corrupting other data), (2) race conditions and synchronization errors which may leave a data item in an inconsistent or corrupted state, and (3) errors due to missing or incorrect initialization of data. These are caused by software defects and may not be repeatable unless the environment and inputs to the program are replicated

exactly, which is hard to achieve in practice. Hence, their behavior is similar to the behavior of hardware transient errors.

3. **Control and data attacks:** These include memory corruption attacks such as buffer overflows and format-string attacks, which overwrite the program's control-flow and data to achieve a malicious purpose (e.g. executing a root shell).

4. **Insider attacks:** Insider attacks are those in which parts of the application and/or the operating system may be malicious and overwrite the application's data or alter its control-flow for malicious purposes. These also include code-injection attacks and hardware-based attacks (e.g. smart-cards).

Table 1 shows the coverage of the different techniques considered in this dissertation for each category of error or attack. As can be seen from the table, there is no one technique that can cover all errors/attack categories, yet together, the techniques cover all categories of errors and attacks considered. *Thus, the techniques in this dissertation address a wide range of both random errors as well as malicious attacks that impact the application and cause system failure or compromise.*

Table 1: Coverage of techniques for different error/attack categories

Fault/Attack Category	**Dynamically-derived detectors**	**Statically derived Detectors**	**Information-flow Signatures**
Transient hardware errors (e.g. soft errors, timing errors, logic bugs)	Yes	Yes	No
Transient software errors (memory errors, race conditions, uninitialized variables)	Yes	Yes, except for uninitialized variables	Yes for memory corruption errors
Control and data attacks (e.g. buffer overflow, format-string)	No	No	Yes
Insider attacks (e.g. malicious third-party libraries)	No	No	Yes

1.5 OVERALL FRAMEWORK

This dissertation proposes an approach to building dependable (reliable and secure) systems using the notion of *application-aware dependability*, which uses the application's properties to detect errors and security attacks that matter to the application. Application properties are automatically extracted using compiler-based static and dynamic analysis techniques, and are converted to error and attack detectors. The detectors are formally validated using model-checking and symbolic execution. The detectors are implemented efficiently using programmable hardware as a part of the Reliability and Security Engine (RSE), which is a hardware framework for executing application-aware checks [1].

The main contribution of this dissertation is a unified approach to reliability and security. By treating reliability and security as two sides of the same coin and proposing joint solutions for them, it is possible to achieve significant gains in the economy and efficiency of the solutions. The dissertation proposes unified frameworks for the following.

1. Deriving application-aware error and attack detectors through compiler analysis,
2. Validating the efficacy of the derived detectors using formal verification methods,
3. Implementing the derived detectors in a common, programmable hardware framework

The first two frameworks are unique contributions of this dissertation, while the third framework is based on the RSE framework proposed in prior work [33]. The rest of this section provides an overview of each of the above frameworks.

1.5.1 Unified Framework for Detector Derivation

This section describes the unified framework for derivation of error and attack detectors, which presents a way of unifying the techniques in Sections 1.2 and 1.3.

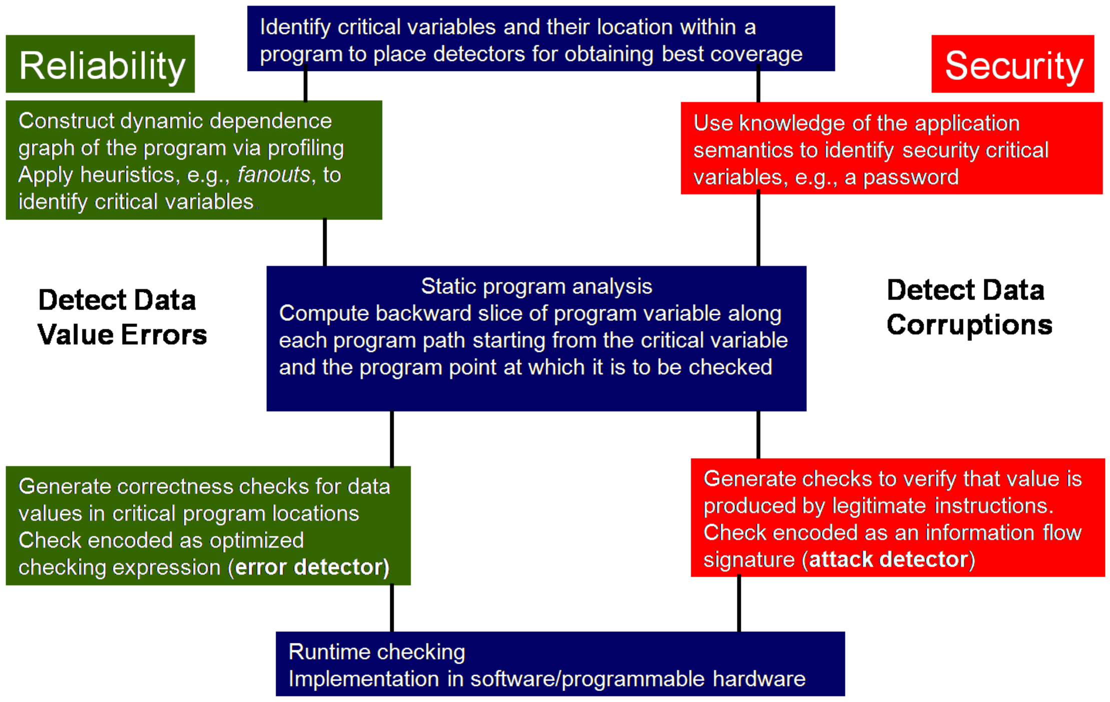

Identify critical variables and their location within a program to place detectors for obtaining best coverage
Reliability
Security
Construct dynamic dependence graph of the program via profiling Apply heuristics, e.g., fanouts, to identify critical variables
Use knowledge of the application semantics to identify security critical variables, e.g., a password
Detect Data Value Errors
Static program analysis Compute backward slice of program variable along each program path starting from the critical variable and the program point at which it is to be checked
Detect Data Corruptions
Generate correctness checks for data values in critical program locations Check encoded as optimized checking expression (error detector)
Generate checks to verify that value is produced by legitimate instructions. Check encoded as an information flow signature (attack detector)
Runtime checking Implementation in software/programmable hardware

Figure 1: Conceptual unified framework for reliability and security

Figure 1shows 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 middle of the figure shows the common steps in both processes.

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 for taking 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 dependences in the application and is described in [17]. 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[18]. 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, and 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 that only the statements/instructions in the static program slice of the critical variable, 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 recomputes 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 where 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 suitable recovery action for the error or attack.

Table 2 summarizes the differences between the derivation of error and attack detectors for each of the steps shown in Figure 1.

Table 2: Differences in the derivation process for error and attack detectors

Step	**Error Detectors**	**Attack Detectors**
Choosing critical variables	Automatically done based on error propagation analysis	Manually selected based on knowledge of security semantics
Extraction of backward slice	Needs to preserve execution order of the slice to generate a checking expression	Only needs to preserve instruction-level dependences to generate signatures
Encoding of slice	Encoded as an expression that captures the computation of the critical variable – Checking expression	Encoded as a signature that captures the dependences – Information-flow Signature
Runtime checking	Recomputation of critical variable by the checking expression to check the computation in the original program	Tracking of instruction dependencies to check whether they conform to the statically-extracted information-flow signature

The error and attack detectors have both been derived through the introduction of new passes in the LLVM compilation framework [30]. Currently, the two design flows are independent of each other, but it is possible to combine them into a single, unified flow.

1.5.2 Unified Framework for Detector Validation

This section describes the unified framework to formally validate the application-aware error and attack detectors using formal verification techniques. To the best of our knowledge, the framework is the first of its kind to use formal verification to validate the properties of *arbitrary* detectors in *general-purpose* programs, and can be used to identify corner cases of errors and attacks that evade detection. Figure 2 shows a conceptual view of the formal framework.

The input to the framework is an assembly language representation of the program with embedded error and/or attack detectors. The advantage of using assembly language is that it is possible to represent a wide variety of errors and attacks at the assembly language level. This is because the assembly language representation of the program includes (1) the source-level characteristics of the program, (2) runtime libraries that are linked with the program, and (3) runtime support code that is added by the compiler (e.g. function prologs and epilogs). Thus, the assembly language representation of the program is closest to the form that is executed in hardware, and consequently can express both software and hardware errors. The program is augmented with special instructions to express error and attack detectors in line with its code.

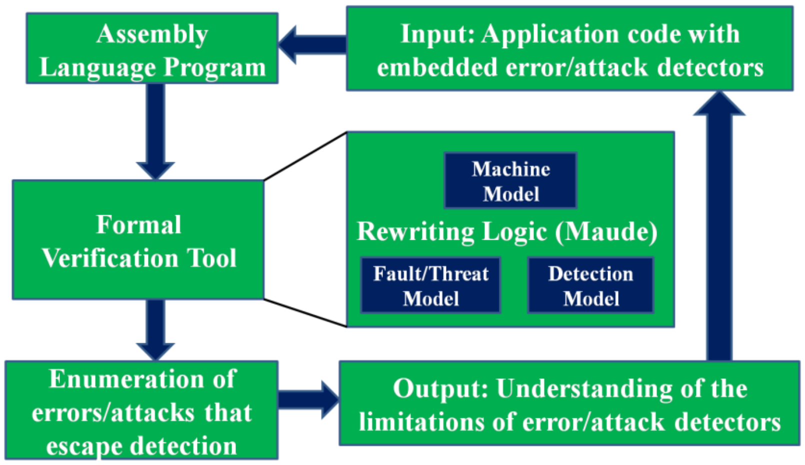

Assembly Language Program
Input: Application code with embedded error/attack detectors
Formal Verification Tool
Machine Model
Rewriting Logic (Maude)
Fault/Threat Model
Detection Model
Enumeration of errors/attacks that escape detection
Output: Understanding of the limitations of error/attack detectors

Figure 2: Unified formal framework for validation of detectors

The framework identifies for each error (attack) in the fault (threat) model, whether the error (attack) leads to application failure (compromise) before it is detected. If so, the error (attack) is printed along with a detailed trace of how the error (attack) propagated in the application. This can help the application developer improve the coverage of the detectors if desired. The main advantage of using formal verification is that it can enumerate all errors (attacks) that evade detection and cause failure (compromise). This can help expose rare corner cases that may be missed by the detectors, which are hard to find through manual inspection alone.

The formal framework consists of the following key structural components: (1) Machine model, which specifies the execution of instructions in the processor, (2) Detection model, which specifies the semantics of detectors, and the (3) Fault/threat model, which specifies the impact of errors and attacks on the program's execution. All three models are expressed in rewriting logic and implemented using the Maude system [34]. The framework has been implemented in the form of two tools – SymPLFIED for verifying error detectors, and SymPLAID for verifying attack detectors. These are described briefly as follows:

SymPLFIED considers the effect of all possible transient hardware errors on computation, memory and registers when a program is being executed under a specific input. It uses symbolic execution and model-checking to exhaustively reason about the effect of the error on the program. The key innovation in SymPLFIED is that it groups 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 that would be considered by an exhaustive injection approach. This in turn greatly enhances the scalability of SymPLFIED compared to exhaustive fault-injection. However, the scalability is obtained at the cost of accuracy, as the abstraction can lead to false-positives i.e. erroneous outcomes that occur in the model but not in the real system. Nevertheless, the loss in accuracy is acceptable in practice as the detectors can be conservatively over-designed to protect against a few false-positives.

SymPLAID considers the effects of insider attacks on the execution of a program. An insider is assumed to corrupt one or more elements of a program's data at runtime in order to achieve his/her malicious goals. Similar to SymPLFIED, SymPLAID tracks corruptions of data values in applications using symbolic execution, and exhaustively considers the effects of data corruptions using model-checking. However, the difference is that SymPLAID tracks each data corruption individually rather than abstracting multiple corruptions into a single class. This is because security attacks are mounted by an intelligent adversary (in contrast to randomly occurring errors) and it is important to identify the exact steps leading to the attack for effective prevention. Further, unlike random errors, an attacker is limited both in the places where the attack may be launched as well as in the values used for the attack. This in turn limits the number of (unique) attacks that may be launched by an attacker. As a result, SymPLAID emphasizes accuracy in tracking individual value corruptions over scalability in terms of the number of corruptions that can be tracked. It does this by precisely tracking the dependencies among corrupted values using error expressions and solving them at decision points (e.g. branches and loads and stores).

Thus, both SymPLFIED and SymPLAID represent different points in the accuracy versus scalability spectrum of formal modeling techniques. Both tools are implemented using a common framework and differ only in the details of the implementation. They can be combined to jointly reason about errors and attacks on programs.

1.5.3 Unified Framework for Detector Implementation

The detectors derived by the technique in Section 1.5.1 are implemented as a part of the Reliability and Security Engine (RSE), which is a processor level framework for application monitoring and error detection [1]. The RSE was proposed as part of Nithin Nakka's dissertation [33] at the University of Illinois at Urbana-Champaign.

The RSE interface taps into the processor's pipeline and exposes signals to the various reliability and security modules. This allows the modules to be oblivious of the processor's internals and for the processor designer to be unencumbered by the implementation details of the RSE modules. A module implements a specific reliability or security mechanism using the signals exposed to it by the RSE interface. The RSE has been implemented on the LEON-3 processor [35] supporting the SUN SPARC instruction set .

The error and attack detectors derived in this dissertation are implemented as RSE modules. Figure 3 shows how the detectors fit into the RSE framework. The left side of Figure 3 shows the security modules and the right side shows the reliability modules. The figure shows a five-stage in-order pipeline with the signals tapped by the RSE interface.

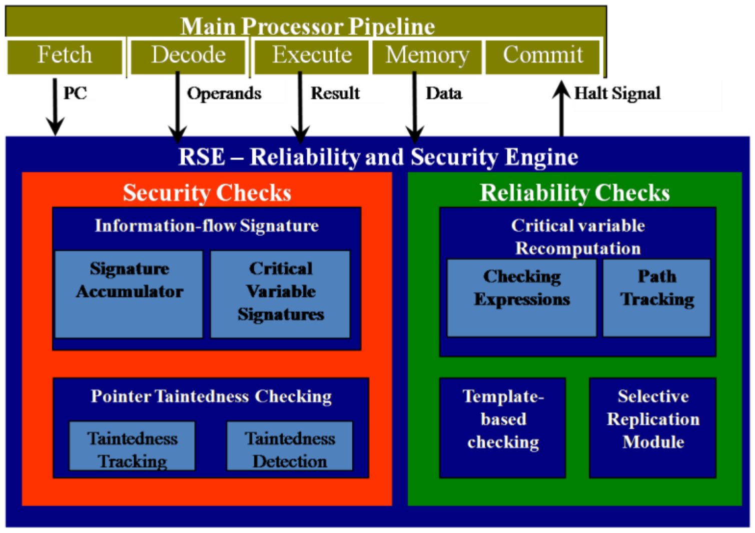


Figure 3: Hardware implementation of the detectors in the RSE Framework

We summarize the RSE modules that implement the derived detectors here.

1. **Information-flow Signatures Module**: This module implements the hardware-side of the information-flow signature tracking scheme outlined in Chapter 7. It consists of a signature accumulator to track the signatures at runtime, as well as a critical variable signature map to store the statically derived signature for comparison with the accumulated signature.

2. **Critical Variable Recomputation**: This module implements the hardware components of the statically derived error detectors described in Chapter 4. It consists of the path-tracking sub-module and the checking sub-module. The path-tracking sub-module keeps tracks of the program's control-flow path and the checking sub-module executes the checking expressions corresponding to the path determined by the path-tracking sub-module.

3. **Template-based Checking**: This module implements the template-based checks based on the dynamic execution of the program. The template based checks are pre-configured into the RSE framework. The method for deriving these checks is described in Chapter 3.

The other two modules shown in Figure 3, namely Pointer Taintedness checking [26] and Selective Replication [12] were not developed in this dissertation but are closely related to the ideas developed in this dissertation. We hence omit detailed description of these modules.

1.6 CONTRIBUTIONS

In addition to the three frameworks described in Section 1.5, this dissertation makes the following contributions:

1. Introduces a methodology to *place* error detectors in application code to preemptively detect errors that result in application failures. The proposed placement method can provide 80-90% error detection coverage with relatively few ideal detectors placed at the identified locations (Chapter 2).

2. Derives error detectors based on *dynamic* characteristics of the application using pre-defined rule-based templates. The templates are customized to application requirements based on dynamic learning over representative inputs to the application and embedded as runtime checks in the code (Chapter 3).

3. Derives error-detectors based on *static characteristics* of the application. Compiler - based static analysis is used to extract the backward program slice of critical variables in the program. The slices are specialized based on the executed control path to derive optimized checking expressions that recompute the value of the critical variable at the detector placement points - *Critical Variable Recomputation* (Chapter 4).

4. Introduces a formal-verification framework to *validate the coverage of the derived error detector*s and find corner-cases in which the derived detectors may be unable to detect the error. The framework uses symbolic execution and model-checking to enumerate *all* failure-causing errors (according to a given fault-model) that evade detection (Chapter 5).

5. Extends the formal verification framework to *automatically discover security attacks* that evade detection in applications. This includes both memory corruption attacks and insider attacks. Memory corruption attacks are usually launched by an external attacker, while Insider attacks are launched by a malicious part of the application itself (Chapter 6).

6. Extends the methodology for derivation of error detectors to *derive detectors for security attacks* in applications (also based on static analysis). The proposed methodology uses *Information Flow Signatures* to detect both memory-corruption attacks and insider attacks. (Chapter 7).

1.7 SUMMARY

Existing techniques for reliability and security are "one-size-fits-all" techniques and incur considerable overheads. In contrast to these techniques, this dissertation proposes "application-aware dependability", in which reliability and security checkers exploit application-specific properties to detect errors and attacks. The dissertation proposes a methodology to extract, validate and implement application-aware error and attack detectors.

The dissertation proposes unified frameworks for reliability and security in order to

1. Derive detectors using compiler-based static and dynamic analysis for critical variables in the application. The detectors are expressed as runtime checks at strategic places in the application.

2. Validate detectors using symbolic execution and model-checking on the assembly code of the application with the detectors embedded in the application. This can be used to improve the coverage of the detectors.

3. Implement the derived detectors as modules in the Reliability and Security Engine (RSE) which is a hardware framework for application-aware detection. The detectors are executed in parallel with the application to provide concurrent error and attack detection with low runtime overheads.

The dissertation shows that by extracting application properties using automated techniques and configuring the properties into reconfigurable hardware, it is possible to detect a wide variety of errors and security attacks in the application at a fraction of the cost of traditional techniques such as duplication.

The rest of this dissertation is organized as follows: Chapter 2 presents a technique to strategically place error detectors in application code, while Chapter 3 and Chapter 4 present respectively the dynamic and static techniques to derive error detectors. Chapter 5 presents the formal technique to validate error detectors, while Chapter 6 presents the formal technique to validate attack detectors for insider attacks. Chapter 7 present techniques to derive attack detectors for insider attacks, and Chapter 8 concludes.

CHAPTER 2 APPLICATION-BASED METRICS FOR STRATEGIC PLACEMENT OF DETECTORS

2.1 INTRODUCTION

This chapter presents a technique to insert detectors or checks into programs to prevent/limit fault propagation due to *value errors.* Value errors are errors that can cause a divergence from the program values seen during the error-free execution of the application. These errors can lead to application crash, hang or fail-silent violations (when the program produces an incorrect result). It is a common assumption that crashes are benign and that there is a mechanism in a system that ensures that when the program encounters an error (that ultimately leads to a crash), the application will crash instantaneously (crash-failure semantics). Data from real systems has shown that while many crashes are benign, severe system failures often result from latent errors that cause undetected error propagation [36]. These latent errors can cause corruption of files [14], propagate to other processes in a distributed system [37] or result in checkpoint corruption [38] prior to the system crash (if indeed the error leads to a crash).

To guarantee crash-failure semantics for a program, we need some form of checking mechanisms in the system. Such support can take many forms including protection at multiple levels and duplication both in hardware and software. Recent commercial examples of such approaches include: (i) IBM G5, which, at the processor level, employs two fully duplicated lock-step pipelines to enable low-latency detection and rapid recovery [10] and (ii) HP NonStop Himalaya, which, at the system level, employs two processors running the same program in locked step. Faults are detected by comparing the output of the two processors at the external pins on every clock cycle [39]. Although these are very robust solutions, due to their high cost and significant hardware overhead, their deployment is restricted to high-end mainframes and servers intended for mission-critical applications.

The detector's coverage depends on two factors: (i) the effectiveness (coverage) of the placement of the detectors, i.e., how many errors manifest at the location where the detector is embedded and (ii) the effectiveness (coverage) of the detector itself, i.e., what fraction of errors manifested at the detector's location are captured.

This chapter introduces metrics to guide strategic placement of detectors and evaluates (using fault injection) the coverage provided by ideal detectors[1] at program locations selected using the computed metrics. Results show that a *small number of detectors, strategically placed, can achieve a high degree of detection coverage.* The issues of development of actual detectors and performance implications of embedding the detectors into the application code are not addressed in this study. Examples of potential detectors are consistency checks on the values in the program, such as range-checks and instruction sequence-checks[40]. In this chapter,

1. The program's code and dynamic execution is analyzed and an abstract model of the data-dependences in the program called the Dynamic Dependence Graph (DDG) is built.

[1] An ideal detector is one that detects 100 % of the errors that are manifested at its location in the program.

2. Several metrics such as *fanout* and *lifetime* are derived from the DDG and used to strategically place/embed (i.e., to maximize the coverage) detectors in the program code.
3. The coverage of ideal detectors placed according to the above metrics is evaluated using fault-injection experiments.

The key findings from this work are:

- A single detector placed using the *fanouts* metric can achieve 50 to 60 % crash-detection coverage for large benchmarks (*gcc* and *perl*).
- A small number of detectors placed using the *lifetimes* metric can achieve high coverage for large benchmarks. For example, it is possible to achieve about 80 % coverage with 10 detectors and 90 % coverage with 25 detectors embedded in the *gcc* benchmark.
- Although the placement of detectors is geared towards providing low-latency detection and preventing propagation by preemptively detecting potential crashes, the placed detectors are also effective at detecting fail-silence violations (i.e., the application terminates normally but produces incorrect results) (30% to70%) and hangs (50% to 60%).

2.2 RELATED WORK

In the recent years, several studies addressed the issue of strategic placement of detectors in application code. Hiller et al [40] uses Error Propagation Analysis (EPA) to determine where detectors or checks should be inserted in an embedded control system. It is assumed that the checks have ideal coverage (100%) and are inserted at points (signals) at which error detection probability is the highest. Voas [41] proposes the “avalanche paradigm”, which is a technique to place assertions in programs before faults in the program propagate to critical states. Goradia [42] evaluates the sensitivity of data values to errors, from a software testing perspective.

Daikon [43] is a dynamic analysis system for generating likely program invariants to detect software bugs. Narayanan et. al. [44] use the invariants produced by DAIKON to detect soft errors in the data cache. DAIKON places assertions at the beginning and ends of loops and procedure calls. However, this may not be sufficient to provide low-latency error detection as the application/system may misbehave long before the assertion point is reached. Benso et. al. [45] presents a compiler technique to detect critical values in a program. The criticality of a variable is calculated based upon the lifetime of the variable and how many other variables it affects. This technique can protect against faults that originate in the critical variable and propagate to other variables, but does not protect against faults that are propagated to the critical variable from other locations in the program.

2.3 MODELS AND METRICS

This section presents the computation model, crash model and fault-model used in the technique. It also considers metrics derived from the models for detector placement.

2.3.1 Computation Model – Dynamic Dependence Graph (DDG)

The computation is represented in the form of a Dynamic Dependence Graph (DDG), a directed-acyclic graph (DAG) which captures the dynamic dependences among the values produced in the course of the program execution. In this context, *a value is a dynamic definition (assignment) of a variable or memory location used by the program at runtime*. *A value may be read many times but it is written only once*. If the variable or location is rewritten, it is treated as a new value. Thus a single variable or memory location may be mapped onto multiple values.

A node in the DDG represents a value produced in the program, and is associated with the dynamic instruction that produced the value. In the DDG, edges are drawn between nodes representing the operands of an instruction and nodes representing the value produced by the instruction. The edge represents the instruction; the source node of the outgoing edge corresponds to an instruction operand and the destination node to the value produced by the instruction. Figure 4 shows a sample code fragment and its corresponding DDG. The code computes the sum of elements of an array *A* of 5 integers (denoted by *size*) and stores the sum in the variable *sum*. The table in the figure shows the mapping between the DDG nodes and the instructions, as well as the effect of executing the instructions. Not all nodes in the DDG correspond to the instructions, e.g., nodes 1, 3, 8, 13, 23, and 28 represent memory locations used by the code fragment.

Code Fragement	Explanation	Nodes in DDG
ADDI R1, R0, 0	R1 ← R0	6
LW R2, [size]	R2 ← [size]	2
ADDI R4, R0, 0	R4 ← R0	0
LOOP: LW R3, R1[A]	R3 ← A[R1]	5, 10, 15, 20, 25
ADD R4, R4, R3	R4 ← R4 + R3	4, 9, 14, 19, 24
ADDI R1, R1, 1	R1 ← R1 + 1	6, 11, 16, 21, 26
BNE R1, R2, LOOP	If (R1!=R2) then goto Loop	7, 12, 17, 22, 27
SW [Sum], R4	[Sum] ← R4	28

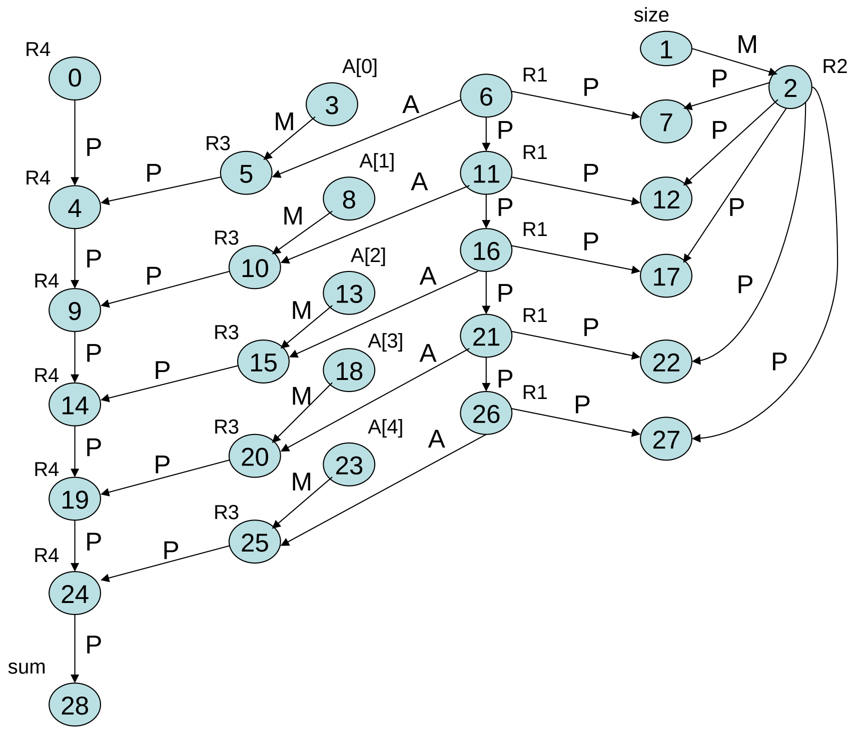


Figure 4: Example code fragment and its dynamic dependence graph (DDG)

The following observation can be made based on the DDG:

- Every value-producing instruction has a corresponding node in the DDG (shown by an arrow from the instruction to its node label in the DDG)
- Memory locations are represented as DDG nodes when they are first read or written e.g., in Figure 4, Nodes 1 and 28 represent memory locations *size* and *sum* respectively and nodes 3, 8, 13, 18 and 23 represent the array locations A[0] to A[4].

Constants are not represented in the DDG (e.g., 0 and 1 are not represented in the DDG, though they appear as instruction operands). Similarly, register names and memory addresses are not stored in the DDG (though they are shown in the figure for convenience).

- The same register/memory location can be mapped onto multiple nodes in the DDG just as a given register or memory location can have multiple value instances during the execution, e.g., in Figure 4, value produced in register R1 is mapped onto nodes 6, 11, 16, 21, 26, one for each loop iteration.
- Each edge of the DDG is marked with the letter, which represents the role of the operand in the instruction: *M* – a memory operand, *A* – an address operand, *P* – a regular operand, *B* – an operand used as a branch target, *F* – a function address operand and *S* – a system call operand.
- The data dependences resulting from control transfer instructions are directly stored in the DDG. In Figure 4, the program executes a jump statement and control is transferred to the location *LOOP* at the end of a loop iteration. The data dependences across loop iterations are represented directly in the DDG, without storing the fact that they are dependent upon the control transfer instruction.

Function calls and returns are also represented in the DDG (not present in the example in Figure 4). Most of the semantics of function calls such as setting up and tearing down of the stack frame and parameters passing already present in the assembly code are automatically included as part of the DDG. However, calling conventions cannot be extracted from the machine code and are explicitly specified in the DDG. For example, in the SPARC architecture, the register *R2* is used to store the return value of a function and this must be incorporated in the DDG to analyze dependences across function calls and returns. The DDG also incorporates dependences caused by system calls (not present in the example in Figure 4).

In this study, the method used to construct the DDG is similar to the one proposed in [46]. The reader interested in techniques for DDG generation can refer to [47].

2.3.2 Fault Model

This study considers the impact of faults in data values produced during the course of a program's execution. Our fault model assumes that any dynamic value in a program can be corrupted at the time of the value's definition. This corresponds to an incorrect computation of the value mainly due to transient (or soft) errors and includes all values written to memory, registers and the processor cache. Note that the assumed fault model also covers errors that arise due to some categories of software faults, e.g., *assignment/initialization* (an un-initialized or incorrectly initialized value is used) and *checking* (a check performed on the variable fails, which is the equivalent of an incorrect value of a variable being used) [48].

2.3.3 Crash Model

Since the ultimate goal is to ensure crash-failure semantics for an application, we first introduce a crash model. It is assumed that crashes can occur as a result of: (i) illegal memory references (*SIGBUS* and *SIGSEGV*), (ii) divide-by-zero and overflow exceptions

(*DIVBYZERO*, *OVERFLOW*), (iii) invocation of system calls with invalid arguments, and (iv) branch to an incorrect or illegal code (*SIGILL*). These four categories can be represented in the Dynamic Dependence Graph (DDG) described in the previous section as follows:

1. A value used as an address operand in a load or store instruction is corrupted and causes the reference to be misaligned or outside a valid memory region.
2. A value used in an arithmetic or logic operation is corrupted and causes a divide-by-zero exception or arithmetic overflow.
3. A value used as a system call operand is incorrect or the program does not have the permissions to perform a particular system call.
4. An operand used as the target of a branch or as the target address of an indirect function call is corrupted, causing the program to jump to an invalid region or to a valid (part of the application) but incorrect (from the point of view of the application semantic) region of code.

Usually, corruption of pointer data is much more likely to cause a crash than non-pointer data, as shown by earlier studies, e.g.Kao [49],. Therefore, this study considers only crashes due to: (i) corruption of values used as address operands of load/store instructions (the first category) and (ii) corruption of values used as targets of branches and function calls (the last category discussed above). While the model does not consider corruption of system call operands and operands of arithmetic and logic instructions, we found that in practice (i.e., in real programs), the percentage of crashes missed by the model is small.

Analysis of error propagation. The dynamic execution traces provided by DDG are used to reason about error propagation from one value to another. It is assumed that a fault originating in a node (value) of the DDG can potentially propagate to all nodes that are successors of this node in the DDG.

2.3.4 Metrics Derived from the Models

In order to strategically place detectors, we develop a set of metrics for selecting locations in the program which can provide high crash detection coverage. The metrics are derived based on the DDG of the program. In order to enable placement of detectors in the code, a notion of *static location of a value* is introduced. *The static location of a value is defined as the address of the instruction that produces the value*. Metrics employed are as follows:

1. *Fanout*: The fanout of a node is the set of all immediate successors of the node in the DDG. In terms of values, it is the set of uses of the value represented by the node. The fanout of a node indicates how many nodes are directly impacted by an error in that node.

2. *Lifetime*: The lifetime of a node is the maximum distance (in terms of dynamic instructions) between the node and its immediate successors. In terms of values, it is the maximum dynamic distance between the *def* and *use* of a value. The lifetime evaluates the reach of the error in the program's execution, as typically values with a long lifetime are global variables or global constants, and an error in these values can affect values that are distant from the current execution context of the program.

3. *Execution*: The execution of a node is the number of times the static location (program counter) associated with the value is executed. Execution reflects the intuition that locations that are executed more frequently are a good place to embed a detector.

4. *Propagation*: The propagation of a node is the number of nodes to which an error in this node propagates before causing a crash. The *propagation* is somewhat similar to the *fanout*, but while the *fanout* considers only the first level of error propagation, the *propagation* metric characterizes error propagation across multiple levels.

5. *Cover*: The cover of a node is the number of nodes from which an error propagates to a given node before causing a crash. Nodes with a high *cover* usually have many error-propagation paths passing through them and consequently, these nodes are a good location for placing detectors to enable preemptive crash detection.

Since detectors are placed in the static code of the program, each node selected (based on the computed metrics) to place a detector must be mapped onto the static locations in the program. Note that multiple nodes in the DDG can be mapped onto a single static location. Consequently, aggregation functions must be defined to compute overall metrics corresponding to a given static program location based on the metrics of the nodes that map onto this location. In the case of *fanout, propagation and cover metrics, set union operation is used to compute the aggregate set and the cardinality of the aggregate set is calculated as the aggregate fanout, propagation and cover of that* *location. For lifetimes and execution, the aggregate value of the metric at a location is computed as the average of the metric values of the nodes that map onto this location.*

For the example in Figure 4, nodes 6, 11, 16, 21, 26 map onto the value produced by the static instruction *ADDI R1, R1, 1*. The instruction has the following metric values:

- The *aggregate fanout* of the instruction is the cardinality of the union of the set of immediate successors of 6, 11, 16, 21 and 26, namely the cardinality of the set which is equal to 15.

- The *aggregate lifetime* of the instruction is the average of the lifetimes of the nodes 6, 11, 16, 21, and 26. The lifetime of each of these nodes is 4 dynamic instructions (the length of a loop iteration), except for 26 for which it is only one dynamic instruction (the last loop iteration). Therefore, the aggregate lifetime of the instruction is 4.25.

- The *aggregate execution* value for the instruction is 5, as the loop is executed 5 times.

For computing the *propagation* and *cover* metrics, we need to locate the points at which the program can crash. The *crash-set* of a node in the DDG is the set of all nodes at which a crash can potentially occur due to an error in that node. The *crash-point* of a node is the earliest point in the error's propagation (not to be confused with the Propagation metric) at which a crash can occur because of a pointer corruption or corruption of a branch/function call target address[2]. For each node N in the DDG, we

[2] This follows from the crash model defined in Section 4, in which only corruptions of pointers and function/branch targets are assumed to cause crashes.

denote by *Crash(N)* the *crash-point* of N[3]. In case there is no crash due to a fault at N, we assume that *Crash(N)* is nil. For the example in Figure 4, the crash-points of nodes 6, 11, 16, 21 and 26 are nodes 5, 10, 15, 20, 25 respectively as these are used as address operands in the instruction *LW R3, A(R1)*.

The crash-distance of a node is the distance between the node and its crash-point in the DDG and can be defined in terms of the successor nodes.

$$CrashDist(N) = \min_{m \in Succ(N)} \left\{ \begin{array}{ll} weight(edge(M,N)) + CrashDist(M); & if\, EdgeType(M,N) \text{ is not } F, B, \text{ or } A \\ wight(edge(M,N)); & otherwise \end{array} \right\}$$

Once the crash-distance is computed, the *propagation* of a node/location N can be computed.

$$\begin{array}{l} Propagation(N) = \{x | x \in S(N) \wedge dist(x,N) \le CrashDist(N)\} \cup \{N\} \\ where \quad S(N) = \bigcup_{s \in Succ(N)} Propagation(s) \end{array}$$

The aggregate *propagation* of a location can be computed as the cardinality of the union of the propagation sets of the nodes in the DDG, which map onto this location. For the example in Figure 4, the aggregate *propagation* of the node corresponding to the instruction *ADDI R1, R1, 1* is 10, as the union of the *propagation* sets of its DDG nodes 6, 11, 16, 21, 26 is the set of nodes {6, 11, 16, 21, 26, 5, 10, 15, 20, 25}. Note that although the nodes 7, 12, 1, 22, 27 are successors of the nodes 6, 11, 16, 21 and 26, they

[3] In the rare case a node has multiple crash points, we arbitrarily pick one of them to be Crash(N)

do not appear in the *propagation* sets as their distance from these nodes (4) is greater than the crash-distance of the nodes (2).

Once the propagation metric is computed, the cover metric can be computed as follows: *A node M is in the cover of N if and only if N belongs to the propagation of M.* This is because any fault in N must propagate to M before causing a crash if M belongs to the Cover of N (by definition). In the example in Figure 4, the *aggregate cover* of the node corresponding to the instruction *LW R3, R1(A)* is the cardinality of the union of the *cover* sets of its nodes in the DDG, namely 5, 10, 15, 20 and 25. This is the set {6, 11, 16, 21, 26}, as the nodes 5, 10, 15, 20 and 25 collectively appear in the propagation sets of nodes 6, 16, 11, 21 and 26. Hence, the aggregate cover is 5, which is the cardinality of the set.

2.4 EXPERIMENTAL SETUP

This section describes the experimental infrastructure and application workload used to evaluate the model and the metrics. The experiment is divided into three parts:

- *Tracing*: The application program is executed and a detailed execution trace is obtained containing all the dynamic dependences, branches and load/store instructions.

- *Analysis:* The trace is analyzed, the dynamic dependence graph (DDG) constructed and the metrics for placing detectors are computed; this part is done offline.

- *Fault-injection*: Fault-injections are performed to evaluate the choice of the detector points. A fault is injected at random into a value used in the program. The values at the detector points are recorded and compared with the corresponding values in the *golden* (error-free) run of the application. Any deviation between the values in the golden run and the faulty run indicates successful detection of the error.

2.4.1 Infrastructure

The tracing of the application and the fault-injections are performed using a functional simulator in *SimpleScalar* family of processor simulators [50]. The simulator allows fine-grained tracing of the application without perturbing its state or modifying the application code and provides a virtual sandbox to execute the application and study its behavior under faults.

We modified the simulator to track dependences among data values in both registers and memory by shadowing each register/location with four extra bytes[4] (invisible to the application) which store a unique tag for that location. For each instruction executed by the application, the simulator prints (to the trace file) the tag of the instruction's operands and the tag of the resulting value to the trace. The trace is analyzed offline by specialized scripts to construct the DDG and compute the metrics for placing detectors in the code. The top hundred points according to each metric are chosen as locations for inserting detectors.

The effectiveness of the detectors is assessed using fault injection. Fault locations are specified randomly from the dynamic set of tags produced in the program. In this mode, the tags are tracked by the simulator, but the executed instructions are not written to the

[4] This allows upto 2^32 unique tags or IDs to be tracked simultaneously, which was sufficient for the programs in our experiments.

trace. When the tag value of the current instruction equals the value of a specified fault location, a fault is injected by flipping a single-bit in the value produced by the current instruction. Once a fault is injected, the execution sequence is monitored to see if a detector location is reached. If so, the value at the detector location is written to a file for offline comparison with the golden run of the application. Table 3 shows the errors detected by the simulator and their mapping into consequence in a real system. It also explains the detection mechanism in the simulator.

Table 3: Types of errors detected by simulator and their real-world consequences

Type of error detected	**Consequence in a real system**	**Simulator detection mechanism**
Invalid Memory Access	Crash (SIGSEGV)	Consistency checks on address range
Memory alignment Error	Crash (SIGBUS)	Check on memory address alignment
Divide-by-Zero	Crash (SIGFPE)	Check before DIV operation
Integer Overflow	Crash (SIGFPE)	Check after every integer operation
Illegal Instruction	Crash (SIGILL)	Check instruction validity before decoding
System Call Error	Crash (SIGSYS)	None, as simulator executes system calls on behalf of application
Infinite loops	Program Hang (live-lock) Program continuously issues instructions and never terminates	Program executes of a double number of instructions as compared with the golden run
Indefinite wait due to blocking system calls or interminable I/O	Program Hang (deadlock) Program stops issuing instructions and never terminates	Program execution takes substantially longer time (five times in our experiments) than the golden run
Incorrect Output	Fail-Silent Violation (silent data corruption)	Compare outputs at the end of the run

2.4.2 Application Programs

The system is evaluated with four programs from the Siemens suite [51] and two programs from the SPEC95 benchmark suite . These benchmark applications range from a few hundred lines of code (Siemens)[5] to hundreds of thousands of lines of code (SPEC95). A brief description of benchmarks is given in Table 4.

[5] *tcas* from the Siemens suite is omitted as it is very small program (less than 200 lines of C code) and there was insufficient separation among the different metrics used in the study.

Table 4: Benchmarks and their descriptions

Benchmark Name	**Suite**	**Description**
Replace	Siemens	Searches a text file for a regular expression and replaces all occurrences of the expression with a specified string
Schedule2	Siemens	A priority scheduler for multiple job tasks
Print_tokens	Siemens	Breaks the input stream into a series of lexical tokens according to pre-specified rules
Tot_info	Siemens	Offers a series of data analysis functions
Gcc95	SPEC95	The gcc compiler, compiled with gcc (optimization level 0)
Perl	SPEC95	The perl interpreter, compiled with gcc (optimization level 0)

Each of these applications is executed for three inputs. For the Siemens programs, the inputs are chosen from the provided set of inputs. For *gcc95* and *perl*, we created inputs of reduced size (as compared to the original SPEC workloads) since our analysis scripts were unable to handle the extremely large dynamic traces of the SPEC workloads. Also, for the SPEC benchmarks, infrequently executed dynamic control paths that contributed to less than 20 % of the cumulative execution time are removed from the DDG (this constitutes 80 % of program paths).

For each program, the dynamic trace from one of the inputs is chosen to build the DDG and to perform the analysis to choose detector points (the top 100 locations according to each metric). Fault-injections are then performed at randomly-chosen values in the application's execution for all three inputs. For each application, input, and metric used to choose the detector points, faults are injected at 500 random locations, randomly flipping a single bit of a value. This is done 10 times for each location leading to a total of 5000 fault injections for each combination of application, input and metric. One fault is injected per run to eliminate the possibility of latent errors due to earlier injected faults.

2.5 RESULTS

The results obtained from the experiments are analyzed with the objective to answer the following questions:

- What is the detection coverage provided by individual detectors placed according to a given metric?
- What is the rate of benign errors of individual detectors placed according to a given metric?
- What is the detection coverage provided jointly by multiple detectors placed according to a given metric?
- What is the rate of benign errors of multiple detectors placed according to a given metric?

2.5.1 Detection Capability of Metrics for Single Detectors

This section evaluates the detection coverage provided by individual detectors placed according to different metrics. All results represent the average calculated for each application across three inputs. The detector points that registered a value deviation for an injection are associated with the outcome of the injection. The results for each outcome category (crash, hang, fail-silent violation, success) are normalized across the total number of errors observed under that category (for each benchmark-metric combination) and are shown in Figures 5, 6, 7 and 8 for crash, successes, fail-silent violations, and hangs, respectively. The following results can be concluded from the graphs:

Detectors placed according to the *fanout* and *propagation* metrics are the best at detecting crashes. They are followed by detectors placed according to the *cover* metric. Random detector placement is the worst in detecting crashes across all benchmarks (see Figure 5). The maximum coverage provided by *fanouts* and *propagation* detectors is more than 90 % for the Siemens benchmarks (with the exception of *tot_info*). For the SPEC benchmarks (and for *tot_info*), the coverage is between 50% and 60 %.

The percentage of benign errors is relatively small – less than 2 % for all benchmarks except *replace* (see Figure 6). The higher false positive rates for *gcc95* and *perl* are registered by detectors placed using *fanout* (1.5%) and *propagation* (2 %) metrics.

Although the detector points were chosen to support crash-detection, they also detect a significant percentage of fail-silent violations (30% to 70 % for detectors placed using *fanout* and *propagation* metrics as shown in Figure 7).

Hangs are best detected by detectors placed using the *fanout* and the *propagation* metric for all benchmarks except *tot_info* (Figure 8). The coverage is 80% to 90 % for the Siemens benchmarks and 50% to 60 % for the SPEC95 benchmarks.

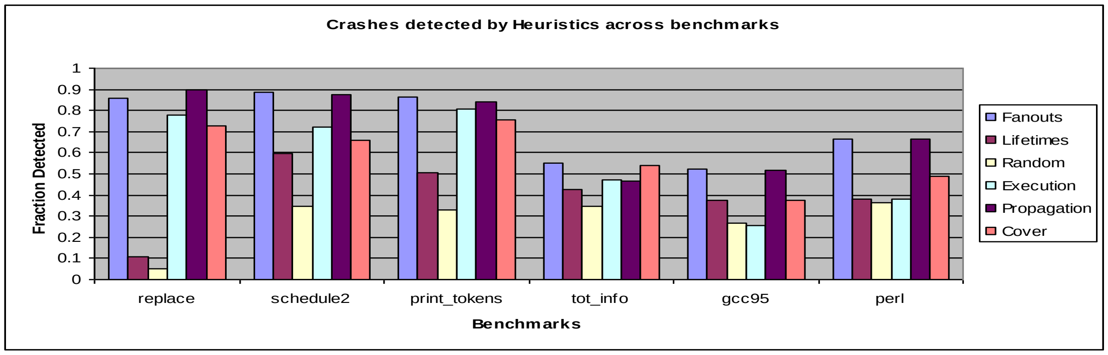


Figure 5: Crashes detected by metrics across benchmarks

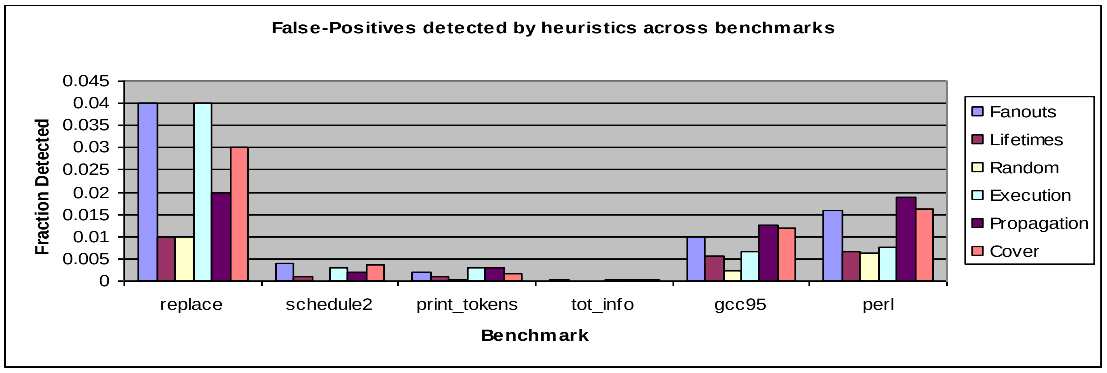


Figure 6: Benign errors detected by metrics across benchmarks

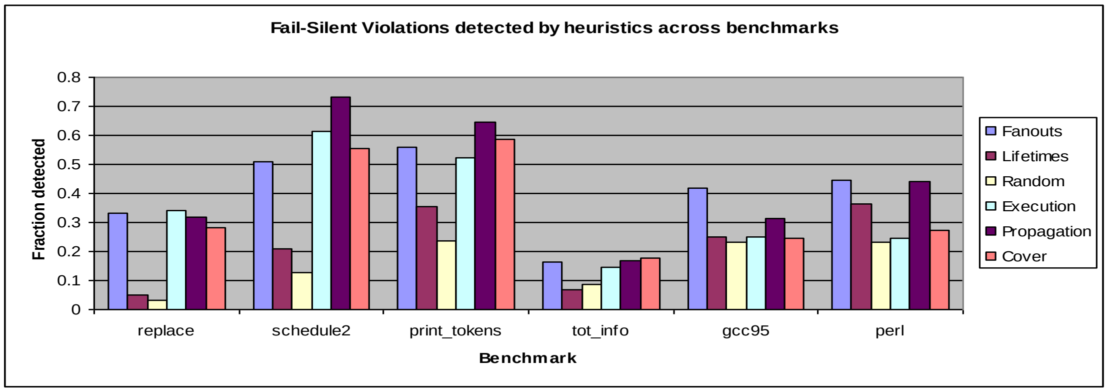


Figure 7: Fail-silent Violations detected by metrics across benchmarks

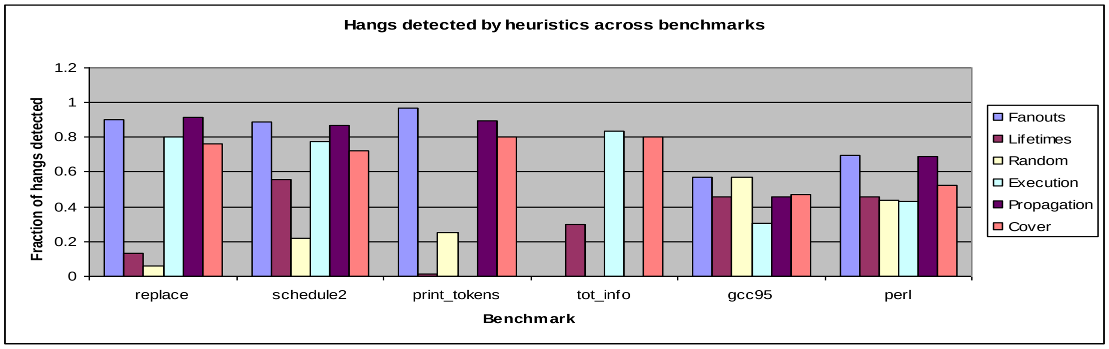


Figure 8: Hangs detected by metrics across benchmarks

2.5.2 Discussion

Locations having high *fanouts* and *propagation* are responsible for propagating errors to a large number of places in the DDG, and it is likely that at least one of the propagated errors causes a crash. Detectors placed using *fanouts* are marginally better than those inserted using *propagation*. The key reasons for the differences are (i) *propagation* relies on the accuracy of the crash model in deciding on the further propagation of the error while *fanouts* does not take the crash model into account and is more conservative and (2) locations with a high *fanout* are often stack or frame pointers. These locations are frequently accessed by the program and hence, an error is likely to crash the program.

The *execution* metric is a good indicator for placing detectors in the Siemens benchmarks where infrequently executed paths are not pruned. The same metric, however, does not perform well in the SPEC benchmarks where paths that contribute to less than 80 % of the execution time are already removed.

The SPEC benchmarks are more complex that the Siemens benchmarks and execute more than 1 million dynamic instructions, while the Siemens benchmarks typically execute less than 100,000 dynamic instructions (only *tot_info* in the Siemens suite executes between 100000 and a million instructions). As a result, the probability of the error reaching the detector is higher in the case of the Siemens benchmarks than for the SPEC95 benchmarks. Hence, the detection coverage for *replace*, *schedule2* and *print_tokens* ranges between 80% and 90 % as compared with 50% to 70 % for *gcc*, *perl* and *tot_info*.

Detectors placed using the *lifetime* metric do not have high crash-detection coverage as the error is likely to remain latent for a long time in a high lifetime node and a crash is unlikely to occur due to this error.

The lower effectiveness of detectors placed using the *cover* metric as compared to *propagation* and *fanout* stems from the fact that *cover* aims at placing detectors along paths leading to potential crash-points while *propagation* and *fanouts* place detectors along paths that can potentially spawn errors in many nodes. Typically, the number of locations with high *fanouts* or *propagation* is small (these metrics follow a *Pareto-Zipf* law like distribution) while the number of potential crash-points of the application is much larger. This result shows that *it is more beneficial to place detectors to protect these few highly-sensitive values, rather than place detectors along the paths that lead to potential crash points.*

The false-positive rate for all metrics is less than 2 % for all benchmarks except *replace*. A false positive means that the error was detected by a detector point, but the program completed successfully (and produced correct output). The number of instructions executed by replace is around 10000, and hence the probability of an error reaching the detector is high even if the error does not trigger a failure. For *gcc* and *perl*, the benign error detection rates are higher than *schedule2*, *print_tokens* and *tot_info* as hot-paths are considered for these two programs.

2.5.3 Detection Capabilities of Metrics for Multiple Detectors

The previous section considered the detection provided by placing a single detector in each of the benchmark programs. For the Siemens benchmarks (except *tot_info*), this was sufficient to provide a coverage of 90 %. However, for applications such as *gcc* and *perl*, a single detector could achieve up to 60 % coverage. In this section, we evaluate the coverage provided jointly by multiple detectors placed in the *gcc95* and *perl* applications.

The top hundred detector locations selected by each metric are grouped into *bins* of a predefined size and the cumulative coverage of detectors placed at locations indicated by a bin is evaluated. For example, to evaluate the coverage of the *fanout* metric with a bin size of 10, the top 100 locations with the highest *fanouts* are arranged in decreasing order by their *fanout* value. The top 10 locations are then grouped into a *bin 1*, the next ten locations into a *bin 2* and so on up to a *bin 10*. The crash-detection coverage of each bin as a whole is evaluated and the average coverage of the 10 bins is the crash-detection coverage for the *fanout* metric with the bin size of 10. The results for crash detection, benign error detections, fail-silent violations and hangs are shown in Figures 9 to 14 as a function of the bin size. The results for *gcc95* are summarized below, and similar trends are observed for *perl*.

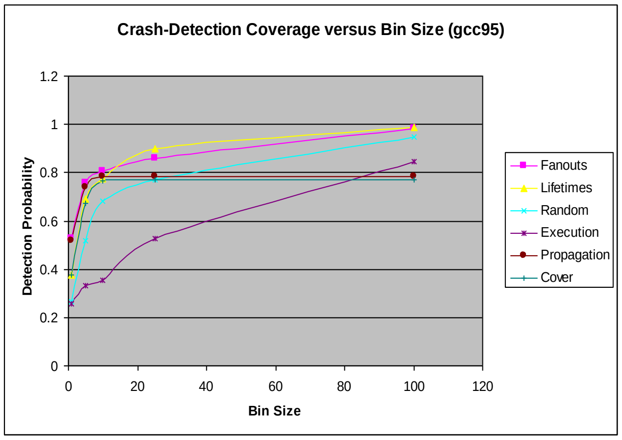


Figure 9: Effect of bin size on crash detection coverage for gcc

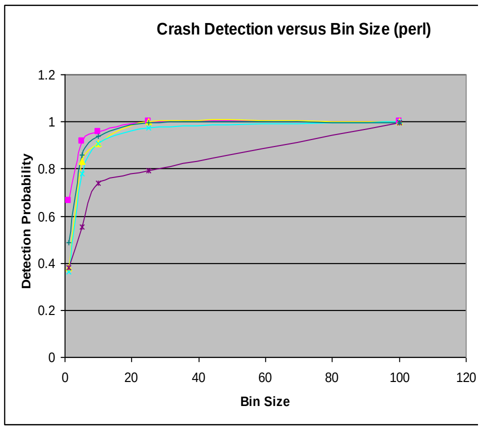


Figure 10: Effect of bin size on crash detection coverage for perl

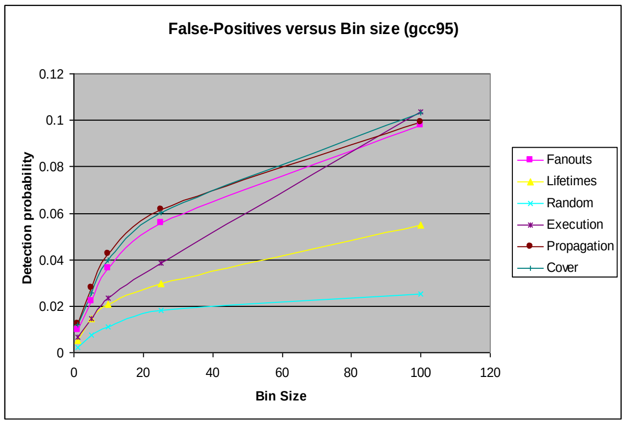


Figure 11: Effect of bin size on benign error detection rate for gcc

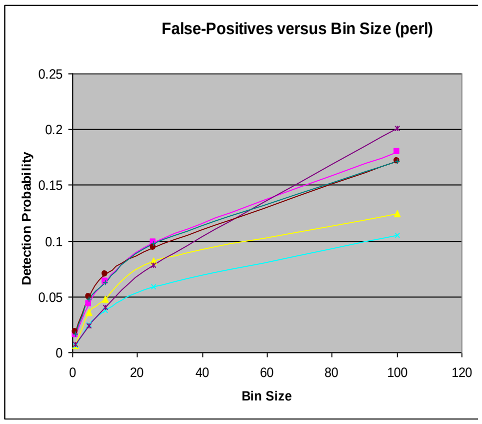


Figure 12: Effect of bin size on benign error detection rate for perl

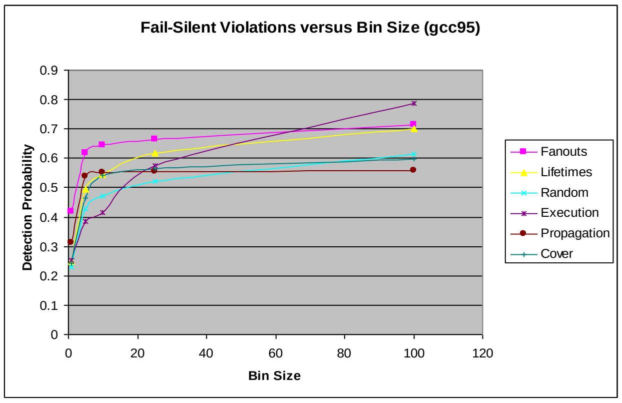


Figure 13: Effect of bin size on fail-silent violation coverage for gcc

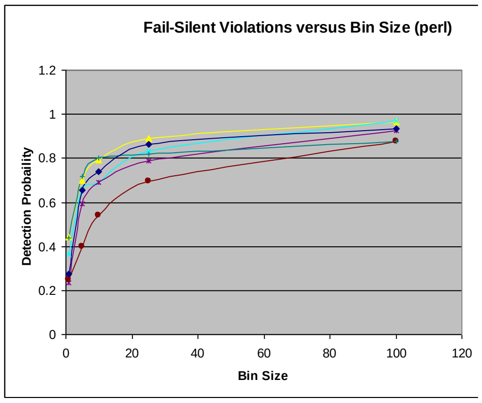


Figure 14: : Effect of bin size on fail-silent violation coverage for perl

For detectors placed using *fanouts* and *propagation*, the crash-detection coverage is less than 60 % when the bin size is 1 (as discussed in Section 8.1). Increasing the bin size to 10 improves coverage to 80% (see Figure 9).

For a bin size of 1, the coverage provided by detectors placed according to *lifetime* is less than 40 %. However, for a bin size of 10, the coverage is almost equal to the one provided by detectors placed according to *fanout* and *propagation* metrics. For a bin size of 25 and 100, it even surpasses the coverage of detectors placed using *fanouts*, providing coverage values of 90 % and 99 %, respectively (see Figure 9).

The percentage of benign error detections also increases with increasing bin-size, but not as much as the crash-detection coverage. For example for detectors placed using the *fanout* metric, the coverage is around 80% when the bin size is 10, but the number of benign error detections remains around 5% (see Figure 11).

The increase in the benign error detection rate for *lifetimes* is much lesser than *fanouts*. The benign error detection percentage for *lifetimes* is only 5 % for a bin size of 100 compared to 10 % for *fanouts* for the same bin size. When 10 or more detectors are considered, placement based on the *lifetime* metric provides the best coverage and the lowest rate of benign error detections (see Figure 11).

Random detector placement provides coverage of 95 % (see Figure 9) when the bin size is 100. Further, it has the smallest percentage of benign error detections (2.5 %; see Figure 11), making random placement of multiple detectors a good choice when minimizing benign error detections is critical.

The fail-silent violation coverage is the highest for detectors placed using the *fanout* metric (70 % for a bin size of 10, see Figure 13). For a bin size of 100, detectors placed using the *execution* metric surpass the detectors placed using *fanout.*

2.5.4 Discussion

For all metrics, the coverage increases with increase in the bin size as the number of detector points increases. The increase in the coverage however flattens out as the bin size increases, as there is considerable overlap among the multiple detector points in detecting crashes. For example, for detectors placed using the *fanout* metric, grouping detectors into bins of size 5 increases the coverage to 75 % (from the 60% coverage provided by individual detectors). However, the increase in coverage is lesser when the bin size increases to 10 (coverage 80%).

Detectors at locations with a high *lifetime* provide limited coverage individually, but several of them jointly achieve very high coverage. This is because each detection point covers a different set of errors. Closer analysis of the results indicates that there is usually one *hot-detector* in each bin, which detects the majority of errors covered by that bin, and the other detectors complement the coverage by detecting errors that escape the hot-detector. These errors are also not easily detectable by the detectors placed using other metrics

2.5.5 Summary of Results

This section summarizes the results from the previous two sections as follows:

Detectors placed using the *fanouts* metric have the best coverage in the program, when single detectors are considered. The coverage provided is 90 % for the Siemens benchmarks and 50-60 % for the SPEC benchmarks. The percentage of benign error detections detected by the detectors is less than 5 % for all the programs considered.

When multiple detectors are placed using the *fanouts* metric, the coverage increases to 97 % by inserting detectors at less than 1 % of the hot-paths (and to 80 % at less than 0.1 % of the hot-paths). There is considerable overlap in the detection capabilities of assertions which leads to the diminishing increase in coverage as the number of assertions is increased. The knee of the curve seems to be about 25 detectors.

In the multiple detector case, the coverage provided by the detectors placed using the *lifetimes* metric is higher than the coverage provided by detectors placed using the *fanouts* metric (when 10 or more detectors are inserted). Further, the percentage of false positives for detectors placed using *lifetimes* is smaller than the percentage of false positives for detectors placed using *fanouts*.

2.6 CONCLUSIONS

This chapter explores the problem of detector placement in programs to preemptively detect crashes arising due to errors in data values used within the program. A model for error propagation and crashes is developed and metrics for placing detectors are derived from the model. The metrics are evaluated on six applications, including two SPEC95 benchmarks. It is found that strategic placement of detectors can increase crash coverage by an order-of-magnitude compared to random placement.

CHAPTER 3 DYNAMIC DERIVATION OF ERROR DETECTORS

3.1 INTRODUCTION

This chapter presents a technique to derive and implement error detectors that protect programs from *data errors*. These are errors that cause a divergence in data values from those in an error-free execution of the program. Data errors can cause the program to crash, hang, or produce incorrect output (fail-silent violations). Such errors can result from incorrect computation, and they would not be caught by generic techniques such as ECC (in memory).

Many static and dynamic analysis techniques (Prefix [52], LCLint [53], Daikon [43]) have been proposed to find bugs in programs. However, these techniques are not geared toward detecting runtime errors as they do not consider error propagation. To detect runtime errors, we need mechanisms that can provide high-coverage, low-latency (rapid) error detection to: (i) preempt uncontrolled system crash/hang and (ii) prevent propagation of erroneous data and limit the extent of the (potential) damage. Eliminating error propagation is essential because programs, upon encountering an error that could eventually lead to a crash, may execute for billions of cycles before crashing [14]. During this time, the program can exhibit unpredictable behavior, such as writing corrupted state to a checkpoint [38] or sending a corrupted message to another process [37], which in turn could result in extended downtimes [8].

It is common practice for developers to write assertions in programs for detecting runtime errors. For example, Andrews [54] discusses the use of executable assertions (checks for data reasonableness) to support testing and fault-tolerance. Assertions are usually specific to the application require considerable programmer effort and expertise to develop correctly. Further, placing assertions in the wrong places could hinder their detection capabilities [55].

Hiller et al. propose a technique to derive assertions in an embedded application based on the high-level behavior of its signals [56]. They facilitate the insertion of assertions by means of well-defined classes of signal patterns. In a companion paper, they also describe how to place assertions by performing extensive fault-injection experiments[40]. However, this technique requires that the programmer has extensive knowledge of the application. Further, performing fault-injection may be time-consuming and cumbersome for the developer. Therefore, it is desirable to develop an automated technique to derive and place detectors in application code.

Our goal is to devise detectors that preemptively capture errors impacting the application and to do so in an automated way without requiring programmer intervention or fault-injection into the system. In this chapter, the term "detectors" refers to executable assertions used to detect runtime errors. This chapter contributes with the following techniques:

1. Derivation of error detectors based on the dynamic execution traces of the application instrumented at strategic points

2. Synthesis of custom hardware (VHDL code) to implement the derived detectors, in order that they can be executed in parallel with the execution of the application
3. Evaluation of the coverage of the derived detectors using fault-injection experiments,
4. Evaluation of the overhead of the detector hardware through synthesis of VHDL code

3.2 *APPROACH AND FAULT-MODELS*

The derivation and implementation of the error detectors in hardware and software encompasses four main phases as depicted in Figure 15. The analysis and design phases are related to the derivation of the detectors, while the synthesis and checking phase are related to the implementation and deployment of the derived detectors at run-time respectively.

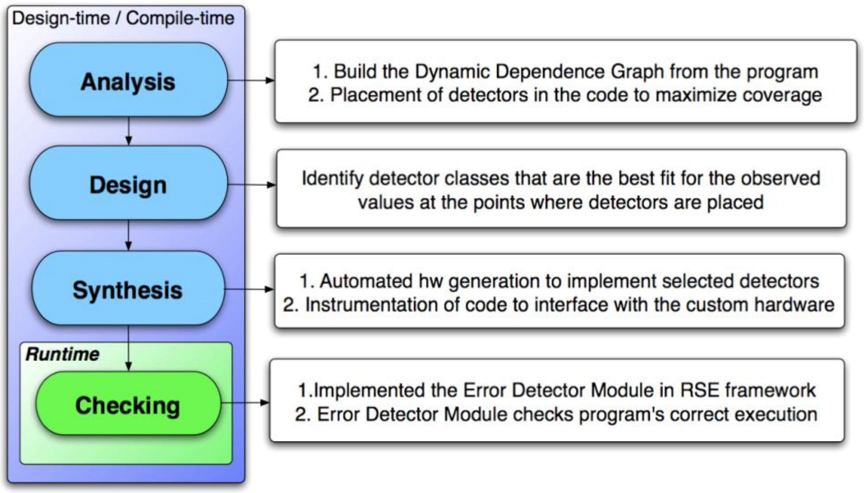

Design-time / Compile-time
Analysis
Design
Synthesis
Runtime
Checking
1. Build the Dynamic Dependence Graph from the program
2. Placement of detectors in the code to maximize coverage
Identify detector classes that are the best fit for the observed values at the points where detectors are placed
1. Automated hw generation to implement selected detectors
2. Instrumentation of code to interface with the custom hardware
1.Implemented the Error Detector Module in RSE framework
2. Error Detector Module checks program's correct execution

Figure 15: Steps in detector derivation and implementation process

During the *analysis* phase, the program locations and variables for placing detectors to maximize coverage are identified, based on the Dynamic Dependence Graph (DDG) of the program. Fault-injections are not required to choose the detector variables and locations. We choose the locations for detector placement based on the *Fanouts* heuristic[17].

The program code is then instrumented to record the values of the chosen variables at the locations selected for detector placement. The recorded values are used during the *design* phase to choose the best detector that matches the observed values for the variable, based on a set of pre-determined generic detector classes (Section 3.3).

After this stage, the detectors can either be integrated into application code as software assertions or implemented in hardware. In this chapter we consider a hardware implementation of the derived detectors. The *synthesis* phase converts the generated assertions to a HDL (Hardware Description Language) representation that is synthesized in hardware. It also inserts special instructions in the application code to invoke and configure the hardware detectors. This is explained in Section 3.5. Finally, during the *checking* phase, the custom hardware detectors are deployed in the system to provide low-overhead, concurrent run-time error detection for the application. When a detector detects a deviation from the application's behavior learned during the design phase, it flags an error.

Fault Model - The fault model covers *errors* in the data values used in the program's execution. This includes faults in: (1) the instruction stream that result in the wrong op-code being executed or in the wrong registers being read or written by the instruction, (2) the functional units of the processor which result in incorrect computations, (3) the instruction fetch and decode units, which result in an incorrect instruction being fetched or decoded (4) the memory and data bus, which cause wrong values to be fetched or written in memory and/or processor register file. Note that these errors would not be detected by techniques such as ECC in memory, as they originate in computation.

The fault-model also represents certain types of *software errors* that result in data-value corruptions such as: (1) synchronization errors or race conditions that result in corruptions of data values due to incorrect sequencing of operations, (2) memory corruption errors, e.g., buffer-overflows and dangling pointer references that can cause arbitrary data values to be overwritten in memory, and (3) use of un-initialized or incorrectly initialized values, as these could result in the use of unpredictable values depending on the platform and environment.

3.3 DETECTOR DERIVATION ANALYSIS

In this chapter, an *error detector* is an assertion based on the value of a single variable[6] of the program at a specific location in its code. A detector for a variable is placed immediately *after* the instruction that writes to the variable. Since a detector is placed in the code, it is invoked each time the program location at which the detector is placed is executed.

[6] In this chapter, the term variable refers to any register, cache or memory location that is visible at the assembly-code level.

Consider the sample code fragment in Table 5. Assume that the detector placement methodology has identified variable *k* as the critical variable to be checked within the loop. Although this example illustrates a simple loop, our technique is general and does not depend on the structure of the source program. In the code sample, variable *k* is initialized at the beginning of the loop and incremented by 1 within the loop. Within the loop, the value of *k* is dependent on its value in the previous iteration. Hence, the rule for *k* can be written as "either the current value of *k* is zero, or it is greater than the previous value of *k* by 1." We refer to the current value of the detector variable *k* as k_i and the previous value as k_{i-1}. Thus, the detector can be expressed in the form: $(k_i - k_{i-1} == 1)$ *or* $(k_i == 0)$.

Table 5: Example code fragment

```
void foo() {
        int k = 0;
        for (; k<N; k++) {
      ….
      }
}
```

As seen from the above example, a detector can be constructed for a target variable by observing the dynamic evolution of the variable over time. The detector consists of a rule describing the allowed values of the variable at the selected location in the program, and an exception condition to cover correct values that do not fall into the rule. If the detector rule fails, then the exception condition is checked, and if this also fails, the detector flags an error. Detector rules can belong to one of six generic classes and are parameterized for the variable checked. The rule classes are shown in Table 6.

Table 6: Generic rule classes and their descriptions

Class Name	Generic Rule (a_i , a_{i-1})	Description
Constant	$(a_i == c)$	The value of the variable in the current invocation of the detector is a constant given by parameter *c*.
Alternate	$((a_i == x \wedge a_{i-1}== y)) \vee (a_i == y \wedge a_{i-1}== x)$	The value of the variable in the current and previous invocations of the detector alternates between parameters *x* and *y* respectively.
Constant-Difference	$(a_i - a_{i-1} == c)$	The value of the variable in the current invocation of the detector differs from its value in the previous invocation by a constant *c*.
Bounded-Difference	$(min <= a_i - a_{i-1} <= max)$	The difference between the values of the variable in the previous and current invocations of the detector lies between *min* and *max.*
Multi-Value	$a_i \epsilon \{ x, y, \ldots \}$	The value of the variable in the current invocation of the detector is one of the set of values *x, y,*
Bounded-Range	$(min <= a_i <= max)$	The value of the variable in the current invocation of the detector lies between the parameters *min* and *max*.

These rule classes are broadly based on common observations about the behavior of variables in the program. Note that, in all cases, the detector involves only the values of the variable in the current invocation (a_i) and/or the previous invocation ($a_{i-1)}$ in the same execution.

The exception condition involves equality constraints on the current and previous values of the variable, as well as logical combinations (*and*, *or*) of two of these constraints. The equality constraints take the following forms: (1) $a_i == d$, where d is a constant parameter; (2) $a_{i-1}== d$, where d is a constant parameter; and (3) $a_i==a_{i-1}$. However, not all combinations of the above three clauses are logically consistent. For example, the exception condition ($a_i==1$ *and* $a_i==2$) is logically inconsistent, as a_i cannot take two different values at the same time. Of the twenty seven possible combinations of the clauses, only eight are logically consistent.

For the example involving the loop index variable k, discussed at the top of this section, the rule class is *Constant-Difference* of 1, and the exception condition is ($k_i == 0$). This was derived automatically using the procedure detailed in this section.

3.4 DYNAMIC DERIVATION OF DETECTORS

This section describes our overall methodology for automatically deriving the detectors based on the dynamic trace of values produced during the application's execution. By automatic derivation, we mean the determination of the rule and the exception condition for each of the variables targeted for error detection. The basic steps are as follows:

The program points at which detectors are placed (both variables and locations) are chosen based on the Dynamic Dependence Graph (DDG) of the program as shown in [17].

The program is instrumented to record the run-time evolution of the values of detector variables at their respective locations, and executed over multiple inputs to obtain dynamic-traces of the checked values. We refer to the sequence of values at a detector location as a value stream for that location.

The dynamic traces of the checked values obtained are analyzed to choose a set of detectors (both rule class and exception condition) that matches the observed values.

A probabilistic model is applied to the set of chosen detectors to find the best detector for a given location. The best detector is characterized in terms of its tightness and execution cost of the detector. These terms are explained in the next subsection.

3.4.1 Detector Tightness and Execution Cost

A qualitative notion of *tightness* of a detector was first introduced in [57]. However, we define tightness in a precise, mathematical sense as the probability that a detector detects an erroneous value of the variable it checks. In mathematical terms, the tightness is the probability that the detector detects an error, given that there is an error in the value of the variable that it checks. The *coverage* of the detector, on the other hand, is the probability that the detector detects an error given that there is an error in any value used in the program. Hence, in addition to the tightness, coverage also depends on the probability that an error propagates to the detector variable and location in the first place. The estimation of this probability is outside the scope of our technique.

In order to characterize the tightness of a detector, we need to consider both the rule and the exception condition (introduced in section 3.3) as the error will not be detected if either passes. The tightness also depends on the parameters of the detector and the distribution of the observed stream of data values in a fault-free execution of the program. For an incorrect value to go undetected by a detector, either the rule or the exception condition or both must evaluate to true. This can happen in one of four mutually exclusive ways, as Table 7 shows.

Table 7: Probability values for computing tightness

Symbol	**Explanation**
$P(R \mid R)$	Probability that an error in a value that originally satisfied the rule (in a correct execution) also causes the incorrect value to satisfy the rule.
$P(R \mid X)$	Probability that an error in a value that originally satisfied the exception condition (in a correct execution) causes the incorrect value to satisfy the rule.
$P(X \mid R)$	Probability that an error in a value that originally satisfied the rule (in a correct execution) causes the incorrect value to satisfy the exception condition.
$P(X \mid X)$	Probability that an error in a value that originally satisfied the exception condition (in a correct execution) causes the incorrect value to satisfy the exception condition.

The tightness of a detector is defined as $(1 - P(I))$, where P(I) is the probability of an incorrect value passing undetected through the detector. This probability can be expressed using the terms in Table 7 as follows:

$$P(I) = P(R) [P(R \mid R) + P(X \mid R)] + P(X) [P(R \mid X) + P(X \mid X)] \qquad (1)$$

where, *P(R)* is the probability of the value belonging to the rule, and the *P(X)* is the probability of the value belonging to the exception condition.

The computation of tightness can be automated, since there are only a limited number of rule-exception pairs[7]. These probabilities can be pre-computed as a function of the detector's parameters as well as on the frequency of elements in the observed data stream for each rule-exception pairs. We will not list all the probabilities, but instead illustrate with an example.

Example. Consider a detector in which the rule belongs to the class *Bounded-Range* with parameters $min = 5$ and $max = 100$ and the exception condition is of the form $(a_i==0)$.

We make the following assumptions about errors in the program.

(1) The distribution of errors in the detector variable is uniform across the range of all possible values the variable can take (say, N),

(2) An error in the current value of the variable is not affected by an error in the previous value of the variable, and

(3) Errors in one detector location are independent of errors in another detector location.

These are optimistic assumptions, and hence the estimation of tightness is an upper bound on the actual value of detector tightness (and hence coverage). Relaxing these assumptions may require apriori knowledge of the application and error behavior in the application.

[7] There are six types of rule classes and eight types of exception conditions, leading to a total of 48 rule-exception pairs.

Table 8 shows the pre-computed probability values for this detector in terms of N and the detector's parameters. Substituting these probability values in equation (1), we find:

$P(I) = P(R) [95/N + 1/N] + P(X) [96/N + 0]$

$= (96/N)[P(R) + P(X)] = \mathbf{96/N}$

The above derivation uses the fact that $P(R) + P(X) = 1$,since the value must satisfy either the rule or the exception in an error-free execution of the program.

Now, assume that the rule belongs to the *Constant* class (with parameter 5). Let us assume that the exception condition is the same as before. For this new detector,

$P(R|R) = 0, P(R|X) = 1/N,$

$P(X|X) = 0$ *and* $P(X|R) = 1/N$

Substituting in equation (1), yields the following expression for *P(I).*

$P(I) = P(R) [0 + 1/N] + P(X) [1/N + 0]= (1/N)[P(R) + P(X)]= \mathbf{1/N}$

Note that the probability of a missed error in the first detector is 96 times the probability of a missed error in the second detector. Hence, the tightness of the first detector is correspondingly much less than the tightness of the second detector (which is intuitive based on the detectors).

The above model is used only to compare the relative tightness of the detectors, and not to compute the actual probabilities (which may be very small). The range of values for the detector variable represented by the symbol *N* gets eliminated in the comparison among detectors for the same variable and does not influence the choice of the detector.

Execution Cost. The execution cost of a detector is the amortized additional computation involved in invoking the detector over multiple values observed at the detector point. The execution cost of a detector is calculated as the number of basic arithmetic and comparison operations that is executed in a single invocation of the detector. An operation usually corresponds to a single arithmetic or logical operator. Note that the computation of the execution cost assumes an error-free execution of the program.

Table 8: Probability values for detector "Bounded-Range (5, 100) except: *(a_i==0)*"

Symbol	Probability Value	Explanation
P (R | R)	*(95 / N)*	Each rule value can turn into any of the other 95 rule values with equal probability.
P (R | X)	*(96 / N)*	An exception value can turn into one of 96 rule values with equal probability
P (X | R)	*(1 / N)*	A rule value can incorrectly satisfy the exception condition if it turns into 0.
P (X | X)	0	An exception value cannot change into another exception value, as there is only one value permitted by the exception condition (in this example).

3.4.2 Detector Derivation Algorithm

For each location identified by the detector placement analysis, the following steps are executed by the algorithm for detector derivation.

1. To derive the detector, the rule class corresponding to the detector is chosen and the associated exception condition is formed. The algorithm to derive a detector for a particular variable and location is given below. We refer to the evolution of a program variable over time as the *stream of values* for that variable.

2. To derive the rule, the rule classes in Table 6 are each tried in sequence against the observed value stream to find which of the rule classes satisfy the observed value stream. The parameters of the rule are learned based on appropriate samples (for each rule class) from the observed stream. For the same location, it is possible

to generate multiple rules that are considered as candidates for exception derivation in the next step.

3. For each rule derived, the associated exception condition is derived based on the values in the stream that do not satisfy the rule. Each of the values that do not satisfy the rule is used as a seed for generating exception conditions for that rule. If it is not possible to derive an exception condition for the observed value as per the conditions in section 3.2, the current rule is discarded and the next rule is tried from the set of rules in step 2.

4. For each rule-exception pair generated, the tightness and execution cost of the detector is calculated. The detector with the maximum tightness to execution cost ratio is chosen as the final detector for that location and is embedded as an assertion in the program’s code

3.5 HARDWARE IMPLEMENTATION

In this chapter, we discuss the hardware implementation of the derived error detectors in context of the Reliability and Security Engine (RSE) framework [1]. The RSE is a reconfigurable processor-level framework that can provide a variety of reliability features according to the requirements and constraints imposed by the user or the application. The RSE Framework hosts (1) *RSE modules*, providing reliability and security services and (2) the *RSE Interface* that provides a standard, well-defined and extendible interface between the modules and the main processor pipeline. The interface collects the intermediate pipeline signals and converts it to the format required by the hardware modules. The application interfaces with the RSE modules using special instructions called CHECK instructions.

The detectors are implemented as a separate module of the RSE called the Error Detector Module (EDM). The detectors are invoked through the CHECK instructions.

3.5.1 Synthesis of Error Detector Module

The output of the algorithm to derive detectors in Section 3.4.2 is a list of detectors, one for each location. This list is used to synthesize hardware modules that interface with the RSE. The hardware implementation of error detectors chosen in the design stage encompasses two steps: (i) instrumentation of the target software application with special instructions to invoke the hardware checkers, and (ii) generation of the Error Detector Module (*EDM*), a piece of customized hardware to check at run-time the execution of the program, and flag a signal when one of the detectors fires. These two phases are carried out at compile time.

Each detector in the *list of detectors* derived in the design phases is characterized by the following attributes: (1) location of the detector in terms of the Program Counter (PC) value at which it is to be invoked, (2) processors' registers to check and (3) detector class and exception parameters. Special instructions are used to load the detectors into the EDM, one for each word of the detector. Figure 16 shows the format of each detector. As can be observed, each detector spans 6 words, and hence requires 6 instructions to be loaded into the EDM.

PC	Rule Class				Exception Condition				
	Class	Logical Register	Param1	Param2	Combination Rule	Class1	Class2	Exception Param1	Excepti on Param2
32 bit	3 bit	5 bit	32 bit	32 bit	2 bit	2 bit	2 bit	32 bit	32 bit

Figure 16 - Format of each detector and bit width of each field

In our current deployment, the *application code* is in the form of assembly code. The header of the code is instrumented with CHECK instruction loading all the detectors needed for the execution of the entire code. This solution minimizes the performance overhead but requires larger storing units in hardware, as explained in Section 7.1. After the instrumentation, the modified code is assembled and converted (*Assembling/Linking* phase) into an executable.

Figure 17 shows the automated design flow starting from the application code to the hardware. Given the application code (in the form of assembly code or program binary), the design flow delivers the instrumented application code and the hardware description of the Error Detector Module tailored for the target application. The *target processor description* (a DLX-like processor in the current implementation [58]) and the *configuration information* are used to extract (from the main pipeline of the processor) the signals that are needed by the EDM.

The output of the *Error Detector Module generation* phase in Figure 17 is a VHDL representation of the EDM. The synthesis procedure then instantiates hardware components from the VHDL representation. These are considered in detail in Section 3.5B.

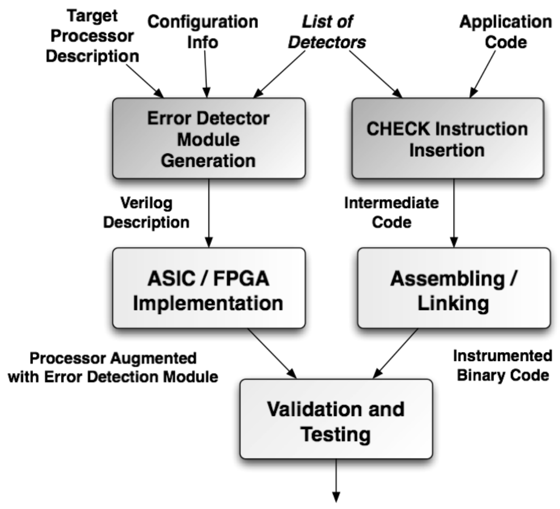

Target Processor Description
Configuration Info
List of Detectors
Application Code
Error Detector Module Generation
CHECK Instruction Insertion
Verilog Description
Intermediate Code
ASIC / FPGA Implementation
Assembling / Linking
Processor Augmented with Error Detection Module
Instrumented Binary Code
Validation and Testing

Figure 17: Design flow to instrument application and generate the EDM

3.5.2 Structure of Error Detector Module

Figure 18 shows the overall architecture of the Error Detector Module (EDM). As mentioned before, the EDM is implemented as a module in the Reliability and Security Engine (RSE).

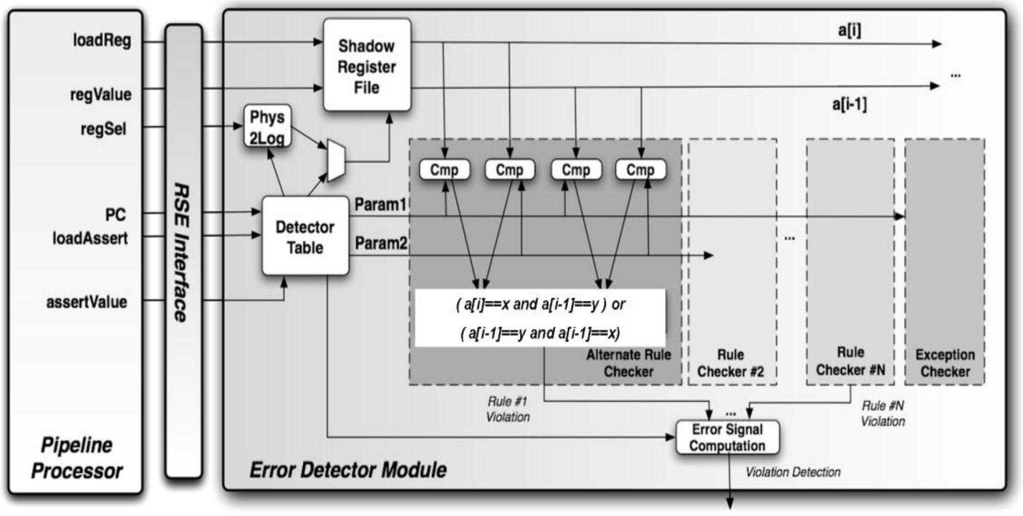


Figure 18: Architectural diagram of synthesized processor

The main components of the EDM are as follows:

Shadow Register File (SRF) – keeps track of current and last values of the microprocessor's registers checked by the detectors (i.e., a_i and a_{i-1}, where a can be any architectural register). This component delivers the required values a_i and a_{i-1} when a detector is executed as required by the expressions in Table 1. When a new value *regValue* is written at time i by the processor in the register R of the processor file (based on the value *regSel*), a copy of the new value R_i is stored in the SRF. The old value R_{i-1} is also retained. Since not all the registers of the processor architecture have to be checked by the detectors, a mapping between the physical addresses of the microprocessor registers and the logical addresses of the corresponding registers in the SRF is kept in the block *Phys2Log*.

Detector Table – stores the information needed for a detector. The size of the Detector Table grows linearly with the number of detectors needed by an application. It is implemented by the following component: (1) comparators checking the current PC against the PCs of the detectors and triggering them if necessary; (2) a RAM hosting the parameters of rules and exceptions. When a detector is triggered by the current PC, the *Detector Table* selects (1) the register R that has to be checked from the SRF forcing the values R_{i-1} and R_{i-1} to be placed on the dual data-path busses, and (2) activates the *Rule* and *Exception Checkers* to compute the detector conditions. The *Error Signal Computation* flags the Violation Detection signal to indicate a detected error.

Rule and Exception Checkers – are the actual data-paths used to carry out the computation of the detector rules and exception conditions. A number of checker components are instantiated to perform the required computations according to the rule classes and exceptions needed by an application. Note that the set of checkers instantiated is equal to the number of detector classes and exceptions (at most forty eight) rather than to the number of detectors inserted in an application (which are essentially unbounded).

Architectural Extensions for High-performance Processors – We are currently working on extending our work for processors where a larger amount of speculation and parallelism is present. This requires enhancing the current architecture of the Error Detector Module. Example extensions are discussed below: (1) Targeting a CISC architecture requires the Error Detector to access the memory bus of the main processor, since some instructions can use memory operands. In the current implementation we assume a load/store RISC architecture, which means that only register operands can be used, and it is sufficient that the Error Detector checks only the content of the processor register file; (2) The use of multiple execution units requires the execution of several checks concurrently and hence the need for (i) multi-ported Detector Table and Shadow Register file, and (ii) independent execution data-path units in the Error Detector; and (3) The use of branch and value speculation requires the ability to execute detectors speculatively and a tighter coupling of the Error Detector Module with the reservation station to keep track of the issued, ready and committed instructions.

3.6 EXPERIMENTAL SETUP

This section describes the experimental infrastructure and application workload used to evaluate the coverage and overheads of the derived detectors. We use fault-injection to evaluate the coverage and implementation on FPGA hardware to evaluate the overheads.

3.6.1 Application Programs

The system is evaluated with six of seven programs from the Siemens suite[8] of programs [51]. These programs are comprised of a few hundred lines of C code, and are extensively used in software testing and verification. A brief description of benchmarks is given in Table 9.

Table 9: Benchmarks and their descriptions

Benchmark	**Description**
Replace	Searches a text file for a regular expression and replaces the expression with a string
Schedule, Schedule2	A priority scheduler for multiple job tasks
Print_tokens, Print_tokens2	Breaks the input stream into a series of lexical tokens according to pre-specified rules
Tot_info	Offers a series of data analysis functions

3.6.2 Infrastructure

The tracing of the application's execution and the fault-injections are performed using a functional simulator in *SimpleScalar* family of processor simulators [50]. The simulator allows fine-grained tracing of the application without modifying the application code and provides a virtual sandbox to execute the application and study its behavior under faults.

[8] *tcas* from the Siemens suite is omitted as it is very small and had insufficient separation among the different metrics in the study

We modified the simulator to track dependences among data values in both registers and memory by shadowing each register/location with four extra bytes (invisible to the application) which store a unique tag for that location. For each instruction executed by the application, the simulator prints (to the trace file) the tag of the instruction's operands and the tag of the resulting value to the trace. The trace is analyzed offline by specialized scripts to construct the DDG and compute the metrics for placing detectors in the code according to the procedure in Chapter 2.

The effectiveness of the detectors is assessed using fault injection. Fault locations are specified randomly from the dynamic set of tags produced in the program. In this mode, the tags are tracked by the simulator, but the executed instructions are not written to the trace. When the tag value of the current instruction equals the value of a specified fault location, a fault is injected by flipping a single-bit in the value produced by the current instruction. Once a fault is injected, the execution sequence is monitored to see if a detector location is reached. If so, the value at the detector location is written to a file for offline comparison with the derived detectors for the application. The above process is continued till the application ends. Note that only a single fault is injected in each execution of the application.

3.6.3 Experimental Procedure

The experiment is divided into four parts as follows:

1. **Placement of detectors and instrumentation of code.** The dynamic instruction trace of the program is obtained from the simulator and the Dynamic Dependence Graph

(DDG) is constructed from the trace. The detector placement points (both variables and locations) are chosen based on the technique described in [17]. For each application, up to 100 detector points are chosen by the analysis, which corresponds to less than 5% of static instructions in the assembly code of the benchmark programs (excluding library functions).

2. **Deriving the detectors based on training set.** The simulator records the values of the selected variables at the detector locations for representative inputs. The dynamic values obtained are used to derive the detectors based on the algorithm in Section 3.4. The training set consists of 200 inputs[9], which are randomly sampled from a test suite consisting of 1000 inputs for each program. These test suites are provided as part of the Siemens benchmark suite [51].

3. **Fault-injections and coverage estimation.** Fault-injection experiments are performed by flipping single bits in data-values chosen at random from the set of all data values produced during the course of the program's execution. After injecting the fault, the data values at the detector locations are recorded and the outcome of the simulated program is classified as a crash, hang, fail-silent violation or success (benign). The values recorded at the detector locations are then checked offline by the derived detectors to assess their coverage. The coverage of a detector is expressed in terms of the type of program outcome it detects i.e. *a detector is said to detect a*

[9] The rationale for the choice of 200 inputs is explained in Section 3.7.3

program crash if the program would have crashed had the detector not detected the error. In case the detector does not detect the error at all, its coverage is counted as zero for all four outcome categories.

For the fault-injection experiments, each application is executed over 10 inputs chosen at random from those used in the training phase. For each input, 1000 locations are chosen at random from the data values produced by the application. A fault-injection run consists of a single bit-flip in the one of the 1000 locations. For each application-input combination, five runs are performed, which corresponds to a total 50,000 fault-injection runs per application.

4. **Computation of false positives.** The application code instrumented with the derived detectors is executed for all 1000 inputs, including the 200 inputs that were used for training. No faults are injected in these runs. If any one of the derived detectors detects an error, then that input is considered to be a false positive (as there was no injected error).

3.7 RESULTS

3.7.1 Detection Coverage of Derived Detectors

The coverage of the detectors derived using the algorithm in Section 3.4 is evaluated using fault-injections as described in Section 3.6.3. Figure 19, Figure 20 and Figure 21 show the coverage for crashes, fail- silence violations (fsv) and hangs obtained for the target applications (in percentages) as a function of the number of detectors placed in each application (ranging from 1 to 100). Figure 22 shows the percentage of total manifested errors that are detected by the derived detectors. .The coverage for each type of failure increases as the number of detectors increases, but less than linearly, as there is an overlap among the errors detected by the detectors. The individual error coverage of the derived detectors depends on the type of failure (crash, FSV, hang).

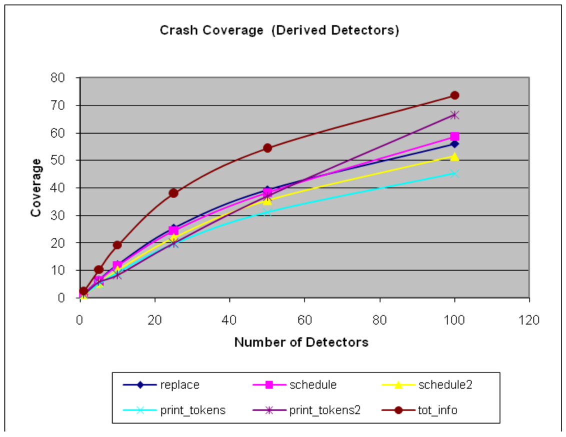


Figure 19: Crash coverage of derived detectors

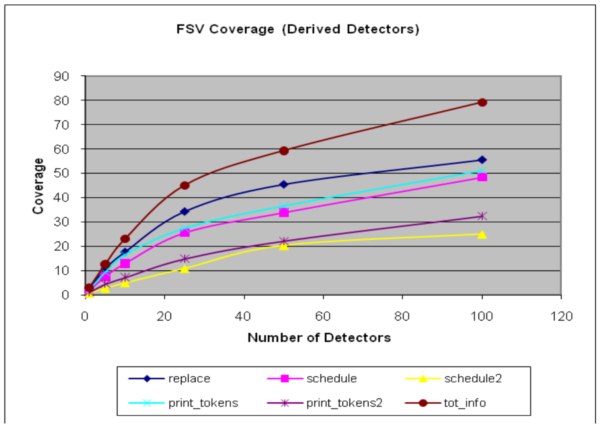


Figure 20: FSV coverage of derived detectors

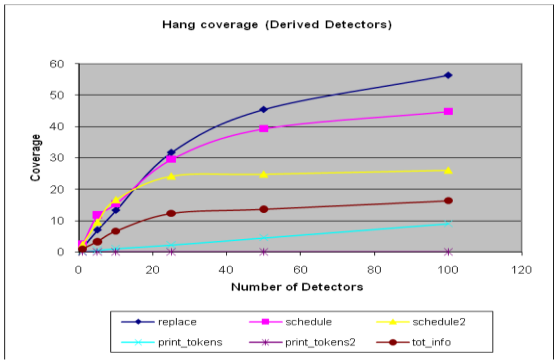


Figure 21: Hang coverage of derived detectors

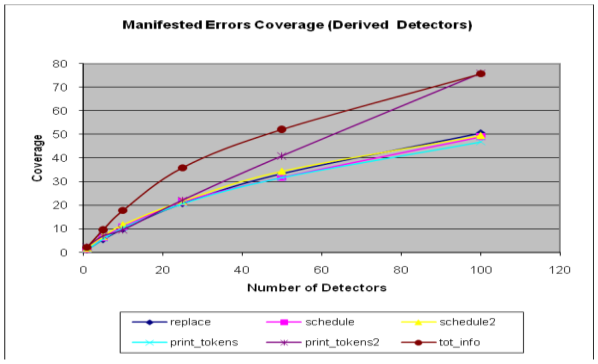


Figure 22: Total error coverage for derived detectors

Table 10: Average detection coverage for 100 detectors

Type of Failure	Minimum Coverage	Maximum Coverage
Program Crash	45% (*print_tokens)*	65% (*tot_info*)
Fail-Silent Violation (FSV)	25% (*schedule2*)	75% (*tot_info*)
Program Hang	0% (*print_tokens2*)	55% (*replace*)
Program Failures	50 % (replace, schedule2, print_tokens, tot_info)	75 % (schedule, print_tokens2)

The coverage obtained for each type of failure is summarized in Table 10 when 100 detectors are placed in each the application. *The derived detectors can detect 50% to 75% of the errors that manifest in the application.* This is because the majority of errors that manifest in an application are crashes (70-75%) and the rest are fail-silent violations (20-30%) and hangs (0-5%).

The results for coverage correspond to any error that occurs in the data values used by the program, and not just for errors that occur in the detector locations. *For example, if even a single bit-flip occurs in a single instance of any data value used in the program, and this error results in a program crash, hang or fail-silence violation, then one of the 100 detectors placed will detect the error 50-75 % of the time.* As mentioned in Section 3.6.1, 100 detectors correspond to less than 5% of program locations in the static assembly code of the benchmark programs.

To put these results in perspective, Hiller et al.[56] obtain a coverage of 80% with 7 assertions for (random) errors that cause failure in an embedded system application. However, in their study about 2000 errors are injected into the system during a short period of 40 seconds, and if one of their executable assertions detects one of the errors in this period, it is considered a successful detection. In contrast, *we inject only a single error* in each run. Furthermore, 7 out of 24 signals are targeted for detection in the embedded system considered in their paper, whereas we place detectors in just 5% of the instructions in the applications considered.

3.7.2 False Positives

False positives can occur when a detector flags an error even if there is no error in the application. A false positive for an input can occur when the values at the detector points for the input do not obey the detector's rule and exception condition learned from the training inputs (because the training was not comprehensive enough).

The training set for learning the detectors consists of 200 inputs and the false positives are computed across all 1000 inputs for each application. No faults were injected in these runs. *If even a single detector detects an error for a particular input, then the entire input is treated as a false positive even if no other detector detects an error for the input.*

Figure 23 presents the percentage of false positives for each of the target applications across 1000 inputs. Across all applications the false positives are no more than 2.5% (with 100 detectors). For the *replace, schedule2, print_tokens* and *print_tokens2* applications, the false positives observed are less than 1%. For the *schedule* and *tot_info* application, the false positive rate is around 2%. While the number of false positives increases as the number of detectors increases, it reaches a plateau as the number of detectors is increased beyond 50. This is because a false positive input is likely to trigger multiple detectors once the number of detectors passes a certain critical threshold (in our case, this critical threshold is 50). However, no such plateau was reached for the coverage results in Figure 22. This suggests that inserting more detectors in the application can increase coverage without increasing the percentage of false positives.

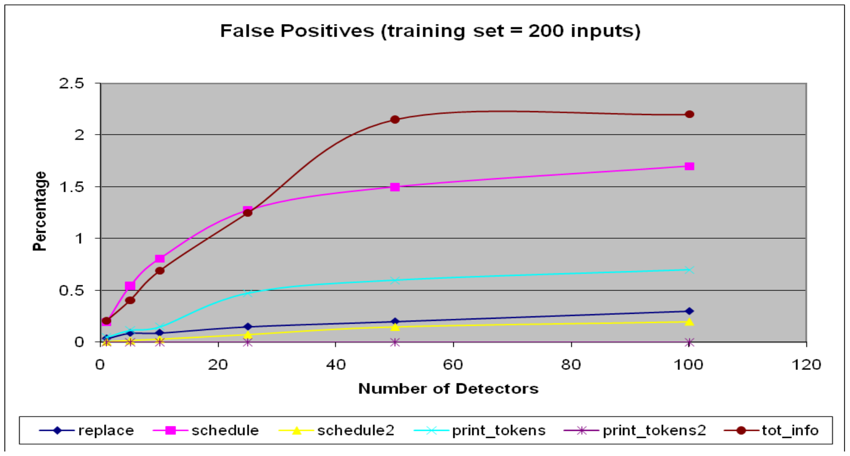


Figure 23: Percentage of false positives for 1000 inputs of each application

When a detector raises an alarm, we need to determine whether an error was really present or whether it is a false-positive. If the error was caused by a transient fault (as we assume in this chapter), then it is likely to be wiped out when the program is re-executed [22]. If on the other hand, the detection was a false positive and hence, a characteristic of the input given to the program, the detector will raise an alarm again during re-execution. In this case, the alarm can be ignored, and the program is allowed to continue. Thus, the impact of a false positive is essentially a loss in performance due to re-execution overhead. Since the percentage of false positives is less than 2.5%, the overhead of re-execution is small. It is possible to reduce the overhead further using checkpointing and restarting scheme as done in Wang and Patel [59].

3.7.3 Effect of Training Set Size

The results reported so far for coverage and false positives of the derived detectors used a training set of 200 inputs from a total of 1000 inputs for each benchmark application. In this section, we consider the effects of varying the size of the training set from 100 inputs, 200 inputs and 300 inputs. In these experiments, the number of detectors is fixed at 100 and the error-detection coverage and false positives are evaluated for each application. The results are shown in Figure 24, Figure 25, Figure 26 and Figure 27.

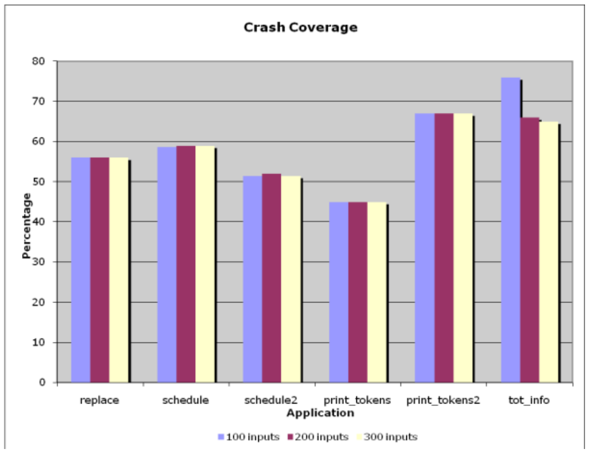


Figure 24: Crash coverage for different training set sizes

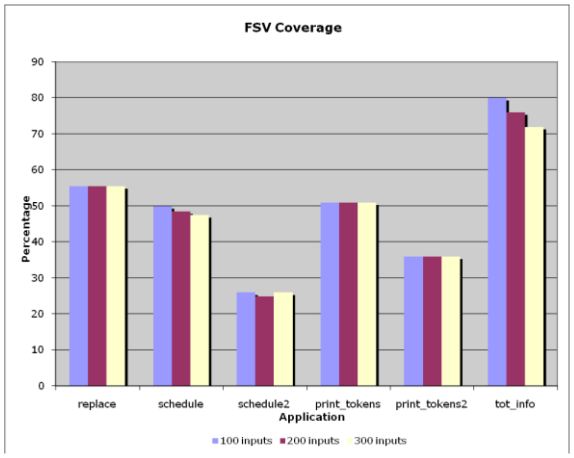


Figure 25: FSV coverage for different training set sizes

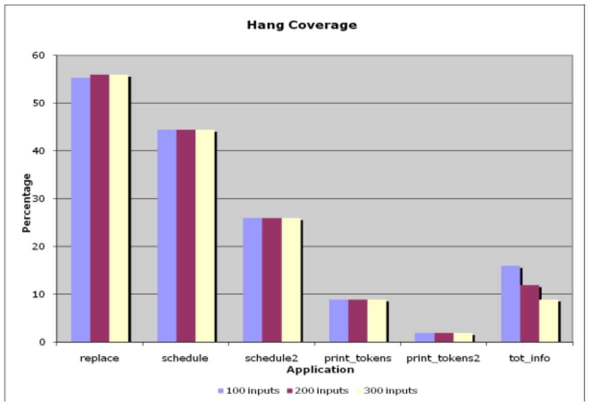


Figure 26: Hang coverage for different training set sizes

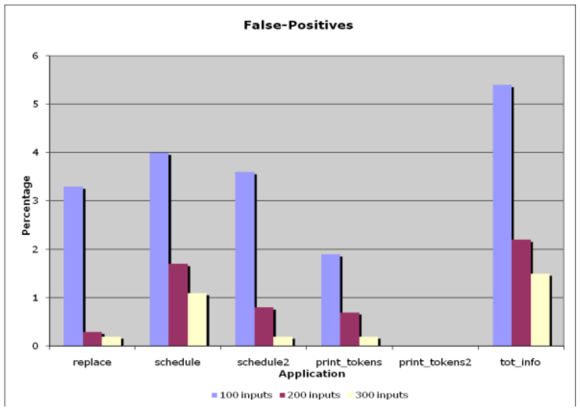


Figure 27: Benign errors for different training set sizes

The following trends may be observed from the graphs:

- The false positives decrease from 5% to 2% as the training set size is increased from 100 inputs to 200 inputs, and to less than 1% for 300 inputs, except *tot_info* (1.5%.).

- The coverage for crashes and hangs remain constant as the training set size increases (Figure 8, Figure 10), except in the case of *tot_info* where the coverage first decreases from 100 to 200 inputs and then remains constant from 200 to 300 inputs (for crashes and hangs).

- The coverage for fail-silent violations decreases marginally as the size of the training set increases from 100 inputs to 300 inputs (Figure 9). This decrease in fail-silent violations is less than 2% for all benchmarks except *tot_info* (5%).

For the applications studied, increasing the training set size from 100 to 200 decreases the false positives significantly, while increasing it from 200 to 300 does not have as large an impact on false positives. The impact on coverage from increasing the training set size is minimal. This suggests that the detectors, once learned, are relatively stable across different inputs, and that their detection capabilities are not affected by the input (beyond a certain number of training inputs). Hence, in this chapter we choose a training set size of 200, which corresponds to 20% of the inputs used for each program.

3.7.4 Comparison with Best-value Detectors

As seen in Section 3.7.1, the derived detectors detect about 45-65% of crashes and 25-80% of fail-silent violations in a program. This section investigates why the remaining errors are not detected and how the detectors can be improved. To form the basis of the discussion, we consider a hypothetical detector that keeps track of the entire history of data values observed at a detector location and uses this knowledge to flag an error. We call these *best-value detectors*, as they represent the maximum coverage that can be obtained by a value-based detector.

The best-value detector may not be achievable in practice, as in addition to requiring enormous space and time overheads (to store the entire history of values), it assumes apriori knowledge of all possible inputs to the program. Nevertheless, the coverage of the *best-value* detector provides an upper bound on the coverage that can be obtained with data-value based detectors such as the detectors considered in this chapter[10]. We build the best-value detector by executing the program under a specific set of inputs and storing the entire sequence of values observed at each location where a detector is placed. This fault-free execution is referred to as the golden run of the program. In this study, we fix the number of best-value detectors in the program to be 100. For each application both the *best-value* detectors and the derived detectors are placed at the same variables and locations. The program is executed under the same set of inputs that were used to derive the best-value detectors. The same set of faults is injected in both cases.

Figure 28, Figure 29, Figure 30 and Figure 31 compare the coverage of the derived detectors with coverage of the *best-value* detectors for crashes, fail-silent violations (FSV), hangs and manifested errors. The results are summarized below.

Crashes - the coverage of the derived detectors is between 75% (*replace*) and 100% (*schedule2, print_tokens2*) of the coverage that can be obtained by the *best-value* detectors (Figure 28)

[10] Note that the *best-value* detectors are different from the *ideal* detectors we introduced in Chapter 2. An *ideal* detector makes use of complete timing and data information to detect an error in a variable, whereas the *best-value* detector employs only data information.

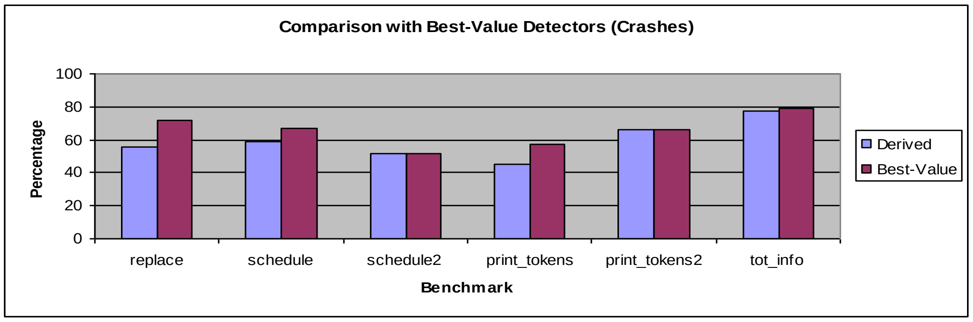


Figure 28: Comparison between best-value detectors and derived detectors for crashes

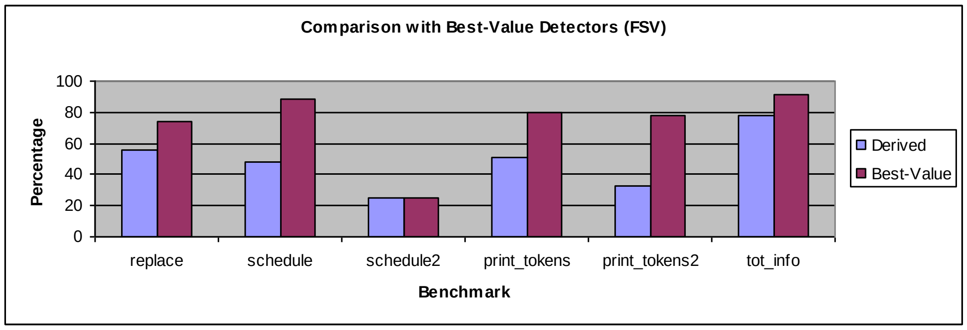


Figure 29: Comparison between best-value detectors and derived detectors for FSV

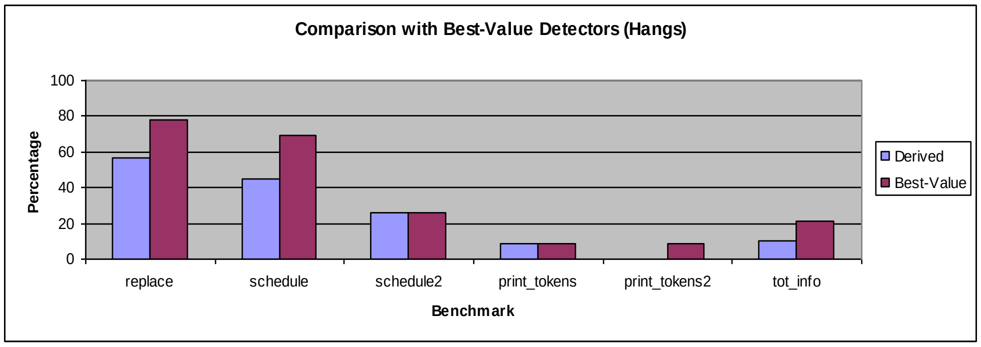


Figure 30: Comparison between best-value detectors and derived detectors for hangs

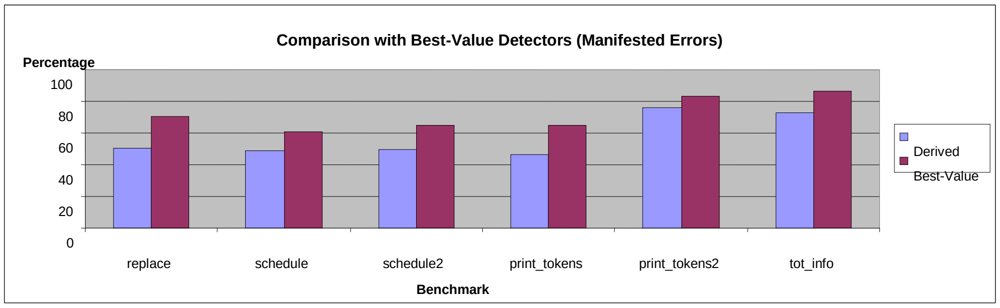


Figure 31: Comparison between best value detectors and derived detectors for manifested errors

FSV - the coverage of the derived detectors is between 40% (*print_tokens2*) and 85% (*tot_info*) of the coverage that can be achieved by the *best-value* detectors (Figure 29).

Hangs - the coverage of the derived detectors is between 50% (*tot_info)* and 100% (*schedule2, print_tokens2)* of the coverage of the *best-value* detectors. (Figure 30).

Manifested errors - the coverage of the derived detectors is between 70% (*replace)* and 90% (*print_tokens2)* of the coverage that can be achieved by the best data detectors (Figure 31)

We examine the reasons for the difference in coverage between the best-value and derived detectors as follows:

- The *best-value* detectors are tailored for each input (based on the golden run of the application for the input) and have 100% knowledge of the application execution for that input. The derived detectors must work across inputs, or they will have an increased false-positive rate. One way to address this problem is to design detectors that are functions of the input or are based on input characteristics;
- The *best-value* detectors store the entire history of values observed at the detector‘s location for that variable in the golden run and can check the value of the variable in the actual run against the value observed in the golden run. The derived detectors, store only the current and previous value of the variable, and use a generic rule and exception condition to check for an error. Thus, increasing the amount of historical information stored in the detector can increase its coverage.

- The derived detectors have much lower coverage compared to the best-value detectors, with respect to fail-silent violations. This is because the derived detectors are general across program inputs, whereas the best-value detectors are specialized for specific inputs. The coverage for crashes however, is not impacted by the generality of the detector, as typically crashes are caused due to corruptions of data values that are illegal or invalid across all inputs. However, the coverage for a fail-silent violation may be affected as a value that is illegal for one input may be valid for another input, but lead to the program printing the wrong output. As pointed out earlier, the coverage for FSVs can be improved by making the detectors a function of the program's inputs. This is a subject of future investigation.

3.8 HARDWARE IMPLEMENTATION RESULTS

The proposed design of the DLX processor, the RSE Interface and the Error Detector Modules for different applications were synthesized using Xilinx ISE 7.1 tools targeting a Xilinx Virtex-E FPGA. The Xilinx Virtex series of FPGAs consists mainly of several type of logic cells: (1) 4-input Look-Up Tables (*LUTs*) statically programmed during the bootstrap with the configuration bit-stream, (2) flip-flops (*FFs*), storage elements in the user visible system state, and (3) Block RAM (*BRAMs*), which are memory blocks that can store up to 4096 bits. Four LUTs and four FFs compose a logic unit called *Slice*.

Area and Clock Period Overhead - Table 11 reports the synthesis results in terms of area (i.e., FFs, LUTs, BRAM and total Slices) and minimum clock frequency, for the reference DLX processor and the complete RSE Interface.

Table 11: Area and timing results for the DLX processor and the RSE Framework

	FFs	LUTs	BRAMs	Slices	Clock Period [ns]
DLX processor	4873	16395	0	9526	58.8
Complete RSE Interface	2465	2329	0	1420	2.01

The synthesis results (in terms of area and minimum clock period for different configurations show that, for different workloads, the number of slices required for the implementation of the Error Detector modules ranges between 2685 and 2915, while the number of additional BRAMs is 9. The area overhead (with respect to the single superscalar DLX processor) of the single EDM is about 30%, while the area overhead of the complete (including the RSE Interface and the Error Detector module) is about 45%.

Performance Overhead - A measure of the performance overhead is given by the formula:

*Overhead = [Extra Clock Cycles * ($T_{CK,\ with\ ED}$ - $T_{CK,\ without\ EDM}$)] / (Total Clock Cycles * $T_{CK,\ without\ EDM}$)*

where $T_{\text{with EDM}}$ and $T_{\text{without EDM}}$ are the total execution times with and without Error Detector module respectively, *Extra clock cycles* is the number of additional clock cycles required to execute the code instrumented with the CHECK instructions, $T_{\text{CK with ED}}$ and $T_{\text{CK without ED}}$ are the minimum clock period of the overall system with and without the Error Detector module, respectively. In our implementation each CHECK instruction is assumed to load 32 bits and hence 6 CHECK instructions are used for loading a single detector. Due to space constraints, we do not report the results for all the workloads, but we report only the workload with the largest time overhead, i.e., *schedule2*. The number of extra clock cycle is 594, while the total number of clock cycles is nearly 1 million, $T_{CK\ with\ ED}$ is 58.82 ns and $T_{CK\ without\ ED}$ is 55.55 ns. Plugging these numbers in the time overhead formula, we found out that the total execution overhead for the detectors is about 5.6%.

3.9 RELATED WORK

Broadly, error detection techniques can be classified based on two criteria:

(1) How the detectors are derived (static or dynamic) and,

(2) How the checking is performed (static or dynamic)

These lead to 4 categories of detectors that span the spectrum of purely static techniques (e.g. Prefix [52], CCured [60], LCLint [53], Engler et al. [61] to purely dynamic techniques (e.g. DIDUCE [62], Maxion et al.[63]). This categorization also includes hybrid techniques in which the detectors are derived statically and checked dynamically (Voas et al.[57], Zenha-Rela et al. [64] and Hiller et al.[56]) and those in which the detectors are derived dynamically but checked statically (for example, DAIKON [43]). These techniques are described in Table 12.

Table 12: Descriptions of related techniques and tools

Technique	Description	Drawbacks
Prefix [52]	Uses symbolic execution through selected paths in a program to find known kinds of errors (e.g. NULL pointer dereferences)	1. Requires programmer to write annotations in the source code 2. High false-positive rate due to infeasible paths
C-Cured [60]	Verifies that points do not write outside their intended memory objects, thereby ensuring memory safety	1. Protects only against errors that violate memory safety – does not protect computation errors 2. Does not handle hardware errors or errors originating in unverified code.
LCLINT [53]	Checks if a program conforms to its specification and if it adheres to predefined programming rules	1. Requires programmer to provide specifications or write annotations in code 2. Only finds those errors that violate the predefined rules
Engler et al. [61]	Analyzes source files to find application-specific programming patterns and identifies violation of the discovered patterns as bugs	1. May incur false-positives i.e. the violation of the pattern may not necessarily be a bug. 2. Does not handle runtime errors or hardware faults – coverage limited to pattern violations
DAIKON [43]	Infers invariants from dynamic execution of program based on representative training inputs	1. Does not take placement of detectors into account - program may crash before the execution reaches the detector location. 2. Requires programmer intervention to filter out real bugs from false identifications
Voas et al. [57]	Considers a general methodology to embed detectors in programs to detect errors. Characterizes properties of good detectors.	1. Does not consider how to derive the detectors 2. Detector placement methodology relies heavily on programmer's knowledge of application.
Zenha-Rela et al. [64]	Evaluates the coverage provided by existing assertions in a program vis-à-vis control-flow error detection techniques and algorithm-based fault-tolerance	Does not consider deriving or embedding assertions in a program. Assume that assertions have already been inserted by programmer.
Hiller et al. [56]	Places error detectors in an embedded system to detect data errors. Consider different classes of detectors based on properties of the signals in an embedded system and the detectors are placed in the system to maximize the coverage	1. Programmer needs to specify class and parameters of each detector - detector derivation is not automated. 2. Detector placement based on extensive fault-injections, which are time-consuming
DIDUCE [62]	Uses software anomaly detection to locate corner cases and find bugs. Formulates strict hypothesis about program behavior in beginning and gradually relaxes them as program executes to learn new behavior.	1. Program may crash before reaching detector point, and the error will not be detected 2. Does not address errors that occur when invariants are being learned (at the beginning of program execution)
Maxion et al. [63]	Characterize the generic space of anomaly detectors for embedded applications.	Do not define specific types of error detectors or how they are derived from the application.

We published this work in the European Conference on Dependable Systems (EDCC) [27]. Since then three papers have been published based on the idea of using dynamically derived program invariants for runtime error detection. These papers use online or offline profiling of the program to build value-based invariants, and use special hardware to check the invariants at runtime. Racunas et al. [65] and Dimitrov and Zhou [66] consider detection of transient errors (similar to our technique), while Sahoo et al. [67] consider detection of permanent hardware errors. These techniques are considered in this section.

3.9.1 Perturbation-based Fault Screening

Perturbation-based fault screening detects deviations in the valid value spaces of static instructions in a program [65]. They define an instruction's valid space as "the set of result values that could be produced in the next dynamic instance of the instruction without being consistent with the current application state" [65]. A fault-screener is a mechanism to detect perturbations. This is similar to our notion of a detector, with the difference that we focus on selected critical variables (and the static instructions that compute them), whereas [65] considers all static instructions in the program. The fault-screeners considered in [65] are as follows:

1) Extended History Scanner: Keeps track of the set of values that a variable can assume. This is similar to the *Multi-Value* detector class in Table 6.

2) Dynamic Range Scanner: Checks if a value belongs to one or more range sets. This is a generalization of the *BoundedRange* class in Table 6.

3) Invariance Based Scanner: This checks if specific bits of a value are constant. This is a generalization of the *Constant* class in Table 6.

The other two fault-scanners considered in [65], namely *TLB-based* scanner and *Bloom filter* scanner have no corresponding representation in our technique.

The main difference between our technique and the one in [65] is that we employ detectors learned from multiple runs of the program over different inputs. The learning algorithm is performed offline and the invariants learned are inserted as detectors in the code. The technique in [28] on the other hand, learns the invariants *while* the program is executing and detects violations of the invariants as errors. This involves running the learning algorithm online, and extensive hardware support is required to keep the performance overheads low. Further, the fault-screeners are specific to a single execution of the application, and are discarded at the end of the execution. Our detectors on the other hand, are general across application inputs and are persistent across multiple executions. This allows them to detect errors even during the startup phase of the application, before the invariants are established. Finally, while a direct comparison of coverage between the two techniques is not possible (due to differences in the experimental techniques used), our technique detects between 50 to 75 % of manifested errors in an application, while the technique in [65] detects between 25 % and 60 % of manifested errors.

3.9.2 Limited Variance in Data Values (LVDV)

This technique uses hardware support to track program invariants at run-time, and uses the learned information to detect both hardware transient errors and selected software bugs [66]. The invariant considered in the paper is a value-based invariant known as “limited variance in data values (LVDV)”. This capitalizes on the observation that in a typical, error-free execution of the program, multiple instances of a static instruction differ only a small extent in the result bits [66]. Any large-scale deviation in the result bits is attributed to either a soft error (caused by radiation) or a software bug (introduced by the application developer).

The paper uses a hardware cache called an LVDV table to store the invariant bits of an instruction's result [66]. The structure is tagged with the instruction's address and is referenced during every cycle with the program counter (PC) of an instruction. The LVDV table is similar to the detector table in our technique, with the difference that the detector table is stored separately from the main processor, and is accessed using special CHECK instructions.

The LVDV technique operates in two modes – soft-error protection and software bug detection. For soft error protection, the invariants are learned on the fly during the initial phase of the program's execution and are used for detection in the subsequent phases. The main problem with this technique is that the program may experience errors in the initial phase or may exhibit substantially different behavior in later phases compared to the initial phase. The former may result in false-negatives and the latter may result in false-positives. In the software bug detection mode, the invariants learned during an execution of the program are reused during another execution. This identifies unusual or corner cases in programs, where bugs are likely to congregate. The goal of the LVDV technique is to present the violated invariants to the programmer, who can then make a judgment about whether the violation was due to a software error. However, this may result in both error-propagation (as the program is not stopped due to the error) as well as false-positives (as a large deviation in a value need not signify a software bug).

3.9.3 Software Anomaly Treatment (SWAT)

The SWAT technique detects permanent hardware errors by monitoring software for anomalies or symptoms [67]. Examples of symptoms include high activity in the operating system and fatal traps executed by the application. In addition, SWAT uses program-level invariants inserted by the compiler to detect residual errors that do not manifest as symptoms [30]. The invariants are derived by executing the program over multiple inputs and collecting dynamic traces. The traces are then analyzed offline to extract invariants on data values in the program. The only kinds of invariants considered in [67] are range-based, i.e. check if a value lies within a range.

Of the techniques considered in this section, the SWAT technique is closest to our work [67]. Both techniques use an offline process to derive error detectors based on dynamic execution traces of the application. The main difference between SWAT and our technique is that SWAT targets permanent hardware errors whereas we target transient hardware and software errors. Examples of permanent errors include stuck-at-faults in the decode unit or latch outputs of the integer ALU. These errors typically cause corruptions of values in multiple instructions and are consequently easier to detect than transient errors. However, false-positives present a much more severe problem as a permanent error will not disappear upon re-execution and SWAT uses diagnosis mechanisms to deal with false-positives. Table 13 summarizes the other differences between the techniques.

Table 13: Comparison of our technique with SWAT

Category	Our Technique	SWAT
Detector Locations	Focuses on critical locations where detection coverage is likely to be highest	Focuses on values stored to memory as these have high potential to catch faults
Detector Types	Considers six different classes of detectors and eight different exception classes (48 in all)	Considers only single detector type encompassing value ranges of variables
Detector Derivation	Based on a probability model to choose the detector and exception class	None required as only a single detector type is considered
Hardware/Compiler support	No compiler support required as we insert detectors into the program binary Hardware support in the form of reconfigurable monitor on the same die	Compiler support for inserting invariants in the program as checking code. Hardware support for error detection, diagnosis and recovery in firmware
Benchmarks and Experimental Methodology	Siemens suite (100 to 1000 lines of C)	SpecInt 2K (> 10000 lines of C code)
	Enhanced Simplescalar simulator for coverage evaluation and synthesis on FPGA hardware for performance evaluation	Virtutechs Simics full system simulator augmented with the Wisconsin GEMS timing models for both coverage and performance evaluation
Detection Coverage	50 to 75 % coverage for all manifested errors in the program	33 % coverage for errors that propagate to software and cause failures
Training Set/False-Positives	Train with 200 inputs, test with 1000 inputs False positive rate is about 2 %	Train with 12 inputs, unclear how many inputs used for testing False positive rate is less than 5 %

3.10 CONCLUSIONS

This chapter proposed a novel technique for preventing a wide range of data errors from corrupting the execution of a generic application. This technique consists of an automated methodology to derive fine-grained, application-specific error detectors by an algorithm based on dynamic traces of application execution. A set of error detector classes, parameters and locations, are derived in order to maximize the error detection coverage for a target application. The chapter also presents an automatic framework for synthesizing the detectors in hardware to enable low-overhead run-time checking of the application execution. The coverage of the derived detectors is evaluated using fault-injections and the hardware implementation of the detectors is synthesized to obtain area and performance overheads.

CHAPTER 4 STATIC DERIVATION OF ERROR DETECTORS

4.1 INTRODUCTION

This chapter presents a methodology to derive error detectors for an application based on compiler (static) analysis. The derived detectors protect the application from data errors. A data error is defined as a divergence in the data values used in the application from an error-free run of the program. Data errors can result from incorrect computation and would not be caught by generic techniques such as ECC in memory. They can also arise due to software defects (bugs).

In the past, static analysis [53]and dynamic analysis [43] approaches have been proposed to find bugs in programs. These approaches have proven effective in finding known kinds of errors prior to deployment of the application in an operational environment. However, studies have shown that the kinds of errors encountered by applications in operational settings are often subtle errors (such as in timing and synchronization)[6], which are not caught by static and dynamic methods.

Furthermore, programs upon encountering an error, may execute for billions of cycles before crashing (if they crash)[14], during which time the error may propagate to permanent state[38]. In order to detect runtime errors, we need mechanisms that can provide high-coverage, low-latency error detection to preempt uncontrolled system crash or hang and prevent error propagation that can lead to state corruption. This is the focus of this chapter.

Duplication has traditionally been used to provide high-coverage at runtime for software errors and hardware-errors [9]. However, in order to prevent error-propagation and preempt crashes, a comparison needs to be performed after every instruction, which in turn results in high performance overhead. Therefore, duplication techniques compare the results of replicated instructions at selected program points such as stores to memory [68, 69]. While this reduces the performance overhead of duplication, it sacrifices coverage as the program may crash before reaching the comparison point. Further, duplication-based techniques detect all errors that manifest in instructions and data. It has been found that less than 50% of these errors typically result in application failure (crash, hang or incorrect output) [70]. Therefore, more than 50% of the errors detected by duplication (benign errors) are wasteful.

The main contribution of this chapter is an approach to derive runtime error detectors based on application properties extracted using static analysis. The derived detectors preempt crashes and provide high-coverage in detecting errors that result in application failures. The coverage of the derived detectors is evaluated using fault-injection experiments. The key findings are as follows:

1. The derived detectors detect around 75% of errors that propagate and cause crashes. The percentage of benign errors detected is less than 3%.

2. The average performance overhead of the derived detectors across 14 benchmark applications is 33% (with hardware support for path-tracking).
3. The detectors can be implemented using a combination of software and programmable hardware.

4.2 RELATED WORK

This section considers related work on locating software bugs using static and dynamic analysis as well as on runtime detection of hardware and software errors.

4.2.1 Static Analysis Techniques

A multitude of techniques have been proposed to find bugs in programs based on static analysis of the application's source code [52, 53, 71, 72]. These techniques validate the program based on a well-understood fault model, usually specified based on common programming errors (e.g. NULL pointer dereferences). The techniques attempt to locate errors across all feasible paths in the program (a program path that corresponds to an actual execution of the program). Determining feasible paths is known to be an impossible problem in the general case. Therefore, these techniques make approximations that result in the creation of spurious paths, which are never executed. This in turn can result in the approach finding errors that will never occur in a real execution, leading to false detections.

Consider for example, the code fragment in Figure 32. In the code, the pointer *str* is initialized to NULL and the pointer *src* is initialized to a constant string. The length of the string *src* is computed in a *while* loop. If the computed length is greater than zero, a new buffer of that length is allocated on the heap and the stored in the pointer pointed to by *str*. Finally, the string pointed to by the pointer *src* is copied into the buffer pointed to by the pointer *str*.

```
int size = 0;
char* str = NULL;
char* src = “A String”;
while (src[size]!=’\0’)
        ++size;
if (size>0) {
     str = malloc(size+1);
}
strcpy(str,src size );
```

Figure 32: Example code fragment to illustrate feasible path problem faced by static analysis tools

Consider a static analysis tool that checks for NULL pointer dereferences. In the above program, the tool needs to resolve whether the value of *str* is NULL before the *strcpy* statement. For *str* to be NULL, the *then* branch of the *if* statement should not be executed, which in turn means that the predicate in the *if* statement, namely (*size>0)* should be false. The value of *size* is initialized to zero outside the *while* loop and incremented inside the loop. The tool needs to statically evaluate the *while* loop in order to conclude that the value of *size* cannot be zero after execution of the loop and before the *if* predicate[11]. Many static analysis tools would not perform such an evaluation in the interest of scalability. In fact, the evaluation of the loop may not even terminate in the general case (although in this example, it would terminate since the string is a constant string). Therefore the tool would report a potential NULL pointer dereference of *str* in the call to *strcpy.*

[11] In this example, it is enough to evaluate one iteration of the loop to arrive at the conclusion that *size* cannot be zero. But in the general case, it may be necessary to evaluate the entire loop.

The problem arises because the control path in which the *then* part of the *if* statement is not executed does not correspond to a real execution of the program. However, the static analysis tool does not have enough resolution to determine this information and consequently over-approximates the set of feasible paths in the program.

In the general case it is impossible for a static analysis tool to resolve all feasible paths in the program. In practice different static analysis tools provide varying degrees of approximations to handle the feasible path problem. We consider examples of four static analysis tools as follows:

LCLINT performs data-flow analysis to find common programming errors in C programs [53]. The analysis is coarse-grained and approximates branch predicates to be both true and false, effectively considering all paths as feasible. LCLINT may produce many spurious warnings and requires programmer annotations to suppress such warnings.

ESP also uses data-flow analysis to determine if the program satisfies a given temporal property [71]. However, the dataflow analysis is path-sensitive and takes into account specific execution paths in the program. In order to perform exact verification, any branch in the program that affects the property being verified must be modeled. The main approximation made by ESP is that it is sufficient to model those branches along which the property being verified differs on both sides of the branch. ESP is able to correctly identify feasible paths when two branches are controlled by the same predicate, or when one branch predicate implies another. However, for more complex branch predicates, ESP relies on programmer supplied annotations to resolve feasible paths in the program.

Prefix avoids the feasible path problem by performing symbolic simulation of the program as opposed to data-flow analysis [52]. The Prefix tool follows each path through a function and keeps track of the exact state of the program along that path. In order to keep the simulation tractable, only a fixed number of paths are explored in each function (typically 50). The main approximation made by Prefix is that the incremental benefit of finding more defects as the number of paths increases is small. It is unclear if the assumption holds for operational defects that may manifest along infrequently executed paths in the program.

SLAM is a model checking tool developed at Microsoft to verify properties of device drivers [72]. SLAM uses a technique known as predicate abstraction[73] to prune infeasible paths in the program. Given a C program, SLAM produces an equivalent boolean program in which all predicates are approximated as Boolean variables. In a Boolean program, there exist only a finite number of values that the predicates can assume, as opposed to potentially infinite values in the original program. Hence, it is easier to find feasible paths in the Boolean program than in the original program. The main problem is that a feasible path in the Boolean program need not correspond to a feasible path in the original program, and this can result in false-positives.

4.2.2 Dynamic Invariant Deduction

These techniques derive code-specific invariants based on dynamic characteristics of the application. An example of a system that uses this technique is **DAIKON** [43], which derives code invariants such as the constancy of variables, boundedness of a variable's range, linear relationships among sets of program variables and inequalities involving two or more program variables. DAIKON's primary purpose is to present the invariants to programmers, who can validate them based on their mental model of the application. The invariants are derived based on the execution of the application with a representative set of inputs, called the training set. Inputs that are not in this set may result in the invariants being violated even when there is no error in the application (false-positives). In order to avoid false-positives during application deployment in operational settings, the training set must well represent the application's execution in operational settings.

DAIKON derives invariants at entries and exits of procedures in the program. The assumption is that invariants represented as function pre-conditions and post-conditions are more useful to the programmer in finding bugs in the application. This limits the use of the generated invariants as assertions for error-detection, since the program may crash before reaching the assertions inserted by DAIKON.

A recent study uses DAIKON to infer data-structure invariants and repair data structures at runtime [74]. The idea is to infer constraints about commonly used data-structures in the program and monitor the data structure with respect to these constraints at runtime. If a constraint violation is detected, the data-structure is "repaired" to satisfy the constraint. The repaired data-structure may or may not be the same as the original data-structure, and hence the program may produce incorrect output after the repair (although it continues without crashing). In general, however, continuing to execute the program after an error has been detected can lead to harmful consequences. Further, the technique described in [74] considers only errors in the program data structure being monitored. It is intriguing to analyze how the technique can be extended to detect general faults in the application's data. To detect general faults, the fault must propagate to the data-structure's fields and violate one or more of the derived invariants for the data-structure. Our experience indicates that it is more likely that the application crashes due to a general error in its data, than for the error to propagate to specific locations in the program's data, unless the locations are chosen taking error propagation into consideration. This observation forms the basis for our detector placement technique in Chapter 2.

DIDUCE [62] is a dynamic invariant detection approach that uses invariants learned during an early phase of the program's execution (training phase) to detect errors in subsequent phases of the execution. The main assumption made by DIDUCE is that invariants learned during the training phase well represent the entire application's execution. It is unclear if this assumption holds in practice, especially for applications that exhibit phased behavior[12]. Further, when DIDUCE detects an invariant violation it does not stop the program but saves the program state for reporting back to the user, so that spurious invariant violations do not stop program execution[13]. This is useful from the point of view of debugging operational failures, but not from the point of view of providing online error-detection (and hence recovery) for applications.

[12] Application behavior varies in phases during program execution
[13].The DIDUCE paper does not present the percentage of spurious invariants found by the tool.

4.2.3 Rule-based Detectors

Rule-based detectors detect errors by checking whether the application satisfies predefined properties specified as rules. The checking can be done either statically at compile-time or dynamically at runtime.

Dynamic Rule-based detectors: Hiller et al. [56] provide rule-based templates to the programmer for specifying runtime error detectors for embedded applications. Examples of rules include a variable being constant, a variable belonging to a range and a monotonically increasing variable increasing by a bounded amount. However, the programmer needs to choose the right templates as well as the template parameters based on their understanding of the application semantics. In a companion paper, Hiller et al. [40] describe an automated methodology to place detectors in order to maximize error detection coverage. The method places detectors on executable paths in the application that have the highest probability of error propagation. Fault-injections into the application data are used to measure the error propagation probabilities along application paths. While the above technique is useful if the programmer has extensive knowledge of the applications and fault-injections can be performed, it is desirable to derive and place detectors without requiring such knowledge and without requiring fault-injections.

Static Rule-based detectors: Engler et al**.** [61] also use rule-based templates to find bugs in programs. The main differences are (1) The rules learned are based on commonly occurring patterns in the application source code rather than being specified by the programmer and (2) The rules are checked at compile-time rather than at runtime. Violations of the learned rules are considered as program bugs. The main assumption here is that programmers follow implicit rules in writing code that are not often documented, and a violation of such rules represents a program error. Static analysis of the application is used to extract the rules and statistical analysis is used to determine if a rule is significant from the point of view of error detection. The technique has been used to find errors and vulnerabilities in the Linux and BSD operating system kernels. Li et al. [75] extend the ideas presented in Engler et al.[61] to extract programming rules using a data-mining technique called *frequent item-set mining*. Their system, PR-Miner**,** extracts implicit programming rules based on static analysis of the application without requiring rule-based templates. The rules are extracted from localized code sections (such as functions) and applied to the entire code base. Violations of the rules are reported as bugs. The technique has been applied to large code-bases including Apache and MySQL, in addition to the Linux kernel.

Static rule-based techniques are useful for finding common programming errors such as copy-and-paste errors [75] or an error due to the programmer forgetting to perform an operation, such as releasing locks [61]. It is unclear if they can be used for detecting more subtle errors that occur in well-tested code, such as timing and synchronization errors, as these errors may not be easily localized to particular code sections[7]. Further, these techniques have large false-positive rates i.e. many errors do not correspond to real bugs. This leads to false detections and the programmer needs to filter out the real detections from the false ones.

4.2.4 Full Duplication Techniques

Duplication has traditionally been used to provide high-coverage at runtime for both software errors and hardware-errors [9]. Duplication based approaches are useful for protecting a system from transient hardware faults. However, they offer limited protection from software errors and permanent hardware faults. This is because both the original program and the duplicated program can suffer from common mode failures. Further, full duplication techniques result in the detection of many errors that have no impact on the application (benign errors)[70]. This constitutes a wasteful detection (and consequent recovery) from the application's viewpoint.

Duplication can be performed either in software or in hardware.

Software-based duplication approaches replicate the program at the source-level [45], instruction level [68] or at the compiler intermediate code level [69]. In order to prevent error-propagation and preempt crashes, software-based approaches must compare the duplicated programs after every instruction. However, such a comparison results in high performance overhead (2x-3x) [45]. Therefore, software duplication approaches perform the comparison only at certain instructions such as stores and branches[68, 69] in the program. This results in less than 100% coverage as the program may crash before reaching the comparison point. Even with this optimization, software-based duplication incurs relatively high performance overhead (60-90%).

Hardware-based duplication approaches such as those used in IBM G5 processors [10] execute redundant copies of each instructions transparent to the application and compare the results of the execution using special-purpose hardware. These techniques reduce the performance overhead of duplication, but have significant hardware design complexity and area overheads (30-35%)[10]. *Simultaneous redundant*-threading [76] is a hardware-based replication technique in which identical copies of the application are executed as independent threads in a Simultaneous Multithreaded (SMT) processor. *Slipstream processors*[77] explores a similar idea in the context of Chip Multiprocessor (CMP) systems. These techniques mask the performance overhead of replication by loose coupling among the redundant threads executing multiple copies of the same program, but lead to inefficient use of processor resources.

4.2.5 Diverse Execution Techniques

D*iverse execution techniques* can detect common mode failures that occur during duplication. Diversity can be implemented at multiple levels as considered by the following techniques:

N-version programming (NVP) is a design diversity technique [78] in which two or more versions of the same program are implemented by independent development teams. The versions are executed simultaneously and the results of their execution compared. The assumption made by NVP is that the versions produced by the independent teams suffer from different kinds of errors and hence an error in any one version of the software will be masked. However, Knight and Leveson [79] show that in practice, even independently produced versions of the software are likely to exhibit similar failures[14]. Further, NVP requires a tremendous cost in programmer time and resources in order to produce software versions that are truly independent. This limits the applicability of NVP to mission-critical systems rather than systems built with COTS (Commercial-Off-the-Shelf) components.

Data Diversity [80] is a variant of NVP in which a single version of the software is executed twice with minor changes in its inputs. The assumption is that software sometimes fails for certain values in its input space and by performing minor perturbations in the input values, it is possible to mask the failure while producing acceptable output. Data diversity can provide protection from both software errors as well as hardware errors (transient and permanent). The data diversity technique has been applied to certain classes of systems such as real-time control systems in which minor changes in the inputs produce acceptable outputs from the application semantics point of view. However in general-purpose applications, it may be unacceptable to perform minor perturbations in input values as these perturbations can result in totally different output values (or even in application failure). This may be unacceptable for the application.

ED4I [81] is a software-based diversity technique which transforms the original program into one in which each data operand is multiplied by a constant value *k*. The value of *k* is determined empirically to maximize the error-detection coverage based on the usage profiles of processor functional units during program execution. The original program

[14] Although the errors made by the teams may be different, the error manifestations are similar.

and the transformed program are both executed on the same processor and the results are compared. A mismatch indicates an error in the program. Since the transformed program operates on a different set of data operands than the original program, it is able to mask certain kinds of errors in processor functional units and memory (both transient and permanent). However, the technique cannot detect software errors that result in incorrect computation of data values in both the original program and the transformed program. This is because diversity is introduced in the data values but not in the instructions that compute the data values.

TRUMP [82] is a diversity technique that uses AN-codes [83] for error detection. Similar to ED4I, TRUMP multiplies each value used in the program by a constant to produce a transformed program. However, instead of comparing the value produced by the original program and the transformed program, TRUMP checks if the data value in the transformed program is divisible by the constant. If this is not the case, then TRUMP concludes that either the original program value or the transformed program value suffered an error. TRUMP also suffers from the same disadvantage of ED4I, namely, that it cannot detect software errors that result in common mode failures between the original program and the transformed program.

4.2.6 Runtime Error Detection Techniques

Runtime techniques have been proposed to detect errors during program execution. These techniques detect specific kinds of errors such as memory safety violations [22, 24, 84], race conditions [85], control-flow errors [86-88] and synchronization errors [89, 90]. None of these techniques however, can detect general errors in the program.

The runtime error detection techniques considered in the literature are as follows:

Memory Safety Checking techniques check every program store that is performed through a pointer (at runtime) to ensure that the write is within the allowed bounds of the pointer[22, 24, 84]. The techniques are effective for detecting common problems due to buffer overflows and dangling pointer errors. It is unclear whether they are effective in detecting random errors that arise due to incorrect computation unless such an error results in a pointer writing outside its allowed bounds. The techniques also requires checking every memory write, and this can result in prohibitive performance overheads (5x-6x)[22]. Smart compile time tricks can reduce the overhead [84], but rely on complex compiler transformations such as automatic pool-allocation [91] .

Race Detection techniques such as Eraser [85] check for race conditions in a multi-threaded program. A race condition occurs when a shared variable is accessed without explicit and appropriate synchronization. A race condition is only one instance of a fault-class broadly referred to as *timing errors*. Timing errors can result in corruption of data values used in the program and cause the program to produce incorrect outputs. The Eraser technique checks for races in lock-based programs by dynamically monitoring lock acquisitions and releases. The technique associates lock sets with each shared variable and dynamically learns these associations during the program’s execution. An error is flagged when the lockset is violated. It is unclear how representative are lock set violations of generic timing errors in the program.

Control-flow checking techniques ensure that a program's statically derived control-flow is preserved during its execution [86-88]. This is achieved by adding checks on the targets of jump instructions and at entries and exits of basic blocks. However, fault-injection experiments (at the hardware level) have shown that only 33% of the manifested errors result in violations of program control-flow [92] and can hence be detected by control-flow checking techniques.

Runtime-verification techniques attempt to bridge the gap between formal techniques such as model checking and runtime checking techniques. These techniques verify whether the program violates a programmer-specified safety property [89, 90] by constructing a model of the program and checking the model based on the actual program execution. The properties checked usually represent synchronization and timing errors in the program. However if there is a general error in the program, there is no guarantee that the program will reach the check before crashing. Therefore, it is unclear if the techniques provide useful runtime coverage for random hardware or software errors.

4.2.7 Executable Assertions

The only general way to detect runtime-errors is for the programmer to put assertions in the code, as demonstrated in [54, 93]. Rela et al. [64] evaluate the coverage provided by programmer-specified assertions in combination with control-flow checking and

Algorithm-Based Fault-Tolerance (ABFT)[94]. They find that assertions can significantly complement the coverage provided by ABFT and control-flow checking.

Leveson et al. [55] compare the error detection capabilities of self-checks (assertions) and diversity-based duplication techniques. They find that (1) Self-checks provide an order of magnitude higher error-detection coverage than diversity-based duplication, (2) For self-checks to be effective in detecting errors, they must be placed at appropriate locations in the application's code and (3) Self-checks derived from analysis of the application code (by the developer) are much more effective at detecting errors than those derived based on program specifications alone.

The detectors derived in this chapter can be considered as executable assertions that are derived automatically based on analysis of the application code (without programmer intervention) and placed at strategic locations to minimize error propagation. The detectors can be implemented both in hardware and in software.

4.2.8 Summary

The static techniques we have discussed are geared towards detecting errors at compile-time, while the dynamic analysis techniques are geared towards providing feedback to the programmer for bug finding. Both these types are *fault-avoidance* techniques (fault is removed before the program is operational) [95]. Despite the existence of these techniques and rigorous program testing, subtle but important errors such as timing errors persist in a program [6, 7].

Runtime-error detection techniques are geared towards addressing subtle software errors and also hardware errors. As we have already seen, full reication can detect many of these errors; but not only does it incur significant performance overheads, it also results in a large number of benign error detections that have no impact on the application[70]. Thus, there is a need for a technique that takes advantage of application characteristics and detects arbitrary errors at runtime without incurring the overheads of replication.

The question that we attempt to answer in this chapter is as follows: Is it possible to derive runtime error (attack) detectors based on application properties to minimize the detection latency and preempt application failures (compromise)? This is crucial for performing rapid recovery upon application failure as shown in [8].

4.3 APPROACH

This section presents an overview of the error detector derivation approach.

4.3.1 Terms and Definitions

Backward Program Slice of a variable at a program location is defined as the set of all program statements/instructions that can affect the value of the variable at that program location[96].

Critical variable: A program variable that exhibits high sensitivity to random data errors in the application is a critical variable. Placing checks on critical variables can achieve high detection coverage.

Checking expression: A checking expression is an optimized sequence of instructions that recompute the critical variable. *It is computed from the backward slice of the critical variable for a specific acyclic control path in the program.*

Detector: The set of all checking expressions for a critical variable, one for each acyclic, intra-procedural control path in the program.

4.3.2 Steps in Detector Derivation

The main steps in error detector derivation are as follows:

A. Identification of critical variables. The critical variables are identified based on an analysis of the dynamic execution of the program. The application is executed with representative inputs to obtain its dynamic execution profile, which is used to choose critical variables for detector placement. Critical variables are variables with the highest dynamic fanouts in the program, as errors in these variables are likely to propagate to many locations in the program and cause program failure. This approach was presented in [17], where it was shown to provide up to 85% coverage with 10 critical variables in the entire program. However, in this chapter, critical variables are chosen on a per-function basis in the program i.e. each function in the program is considered separately to identify critical variables in the function. This is because we consider intra-procedural slices for extracting backward slices (as explained below).

B. Computation of backward slice of critical variables. A backward traversal of the static dependence graph of the program is performed starting from the instruction that computes the value of the critical variable going back to the beginning of the function.

The slice is specialized for each acyclic control path that reaches the computation of the critical variable from the top of the function. The slicing algorithm used is a static slicing technique that considers all possible dependences between instructions in the program regardless of program inputs (based on source language semantics). Hence, the slice will be a superset of the actual dependencies during a valid execution of the program.

C. Check derivation, insertion, instrumentation.

- *Check derivation*: The specialized backward slice for each control path is optimized considering only the instructions on the corresponding path, to form the checking expression.

- *Check insertion:* The checking expression is inserted in the program immediately after the computation of the critical variable.

- *Instrumentation:* Program is instrumented to track control-paths followed at runtime in order to choose the checking expression for that specific control path.

D. Runtime checking in hardware and software. The control path followed is tracked (by the inserted instrumentation) in hardware at runtime. The path-specific inserted checks are executed at appropriate points in the execution depending on the control path followed at runtime. The checks recompute the value of the critical variable for the runtime control path. The recomputed value is compared with the original value computed by the main program. In case of a mismatch, the original program is stopped and recovery is initiated.

There are two main sources of runtime performance overhead for the detector:

(1) *Path Tracking:* The overhead of tracking paths is significant (4x) when done in software[15]. Therefore, a prototype implementation of path tracking is performed in hardware. This hardware is integrated with the Reliability and Security Engine (RSE)[1]. RSE is a hardware framework that provides a plug-and-play environment for including modules that can perform a variety of checking and monitoring tasks in the processor’s data-path. The path-tracking engine is implemented as a module in the RSE.

(2) *Checking:* In order to further reduce the performance overhead, the check execution itself can be moved to hardware. This would involve implementing the checking expressions directly in the RSE and compiling them to Field-Programmable Gate Arrays (FPGAs). This is an area of future investigation.

4.3.3 Example of Derived Detectors

The derived detectors are illustrated using a simplified example of an *if-then-else* statement in Figure 33. A more realistic example is presented in Section 4.4. In the figure, the original code is shown in the left and the checking code added is shown in the right. Assume that the detector placement analysis procedure has identified *f* as one of the critical variables that need to be checked before its use in the following basic block. For simplicity, only the instructions in the backward slice of variable *f* are shown in Figure 33.

[15] Based on a previous software-only evaluation of the technique

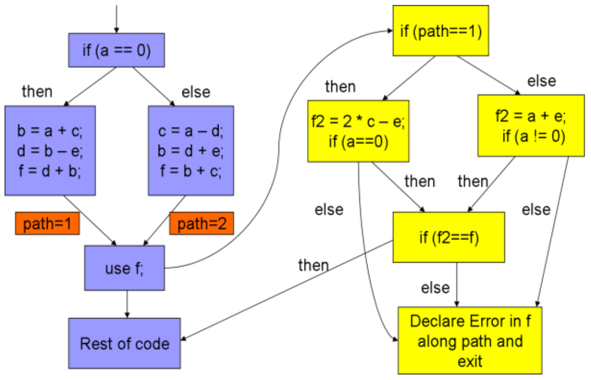


Figure 33: Example code fragment with detectors inserted

There are two paths in the program slice of *f*, corresponding to each of the two branches. The instructions on each path can be optimized to yield a concise expression that checks the value of *f* along that path (shown in yellow in Figure 33). In the case of the first path (*path=1*), the expression reduces to *(2 * c - e)* and this is assigned to the temporary variable *f2*. Similarly the expression for the second path (*path=2*) corresponding to the *else* branch statement reduces to *(a + e)* and is also assigned to *f2*. Instrumentation is added to keep track of paths at runtime.

At runtime, when control reaches the use of the variable *f*, the correct checking expression for *f* is chosen based on the value of the *path* variable and the value of *f2* is compared with the value of *f* computed by the original program. In case there is a mismatch, an error is declared and the program is stopped.

4.3.4 Software Errors Covered

Since the technique proposed in this chapter enforces the compiler-extracted source-code semantics of programs at runtime, it can detect any software error that violates the source program's semantics at runtime. This includes software errors caused by pointer corruptions in programs (memory corruption errors) as well as those caused by missing or incorrect synchronization in concurrent programs (timing errors). We consider how the proposed technique detects these errors:

Memory Corruption Errors: Languages such as C and C++ allow pointers to write anywhere in memory (to the stack and heap)[97]. Memory corruption errors are caused by pointers in the code writing outside their intended object[16] (according to source code semantics), therby corrupting other objects in memory. However, static analysis performed by compilers typically assumes that objects are infinitely far apart in memory and that a pointer can only write within its intended object[30]. As a result, the backward slice of critical variables extracted by the compiler includes only those dependences that arise due to explicit assignment of values to objects via pointers to the object. Therefore, the technique detects all memory errors that corrupt one or more variable in the backward slice of critical variables, as long as the shared state between the check and the main program is not affected (e.g. memory errors that affect function parameters will not be detected, as only intra-procedural slices are considered by the technique).

Figure 34 illustrates an example of a memory corruption error in an application and how the proposed technique detects the error. In the figure, function *foo* computes the running sum (stored in *sum)* of an array of integers (*buf)* and also the maximum integer (*max*) in the array. If the maximum exceeds a predetermined threshold, the function returns the

[16] We use the term object to refer to both program variables as well as heap- and stack- allocated objects.

accumulated sum corresponding to the index of the maximum element in the array (*maxIndex*).

```
int foo(int buf[]) {
1:      int sum[bufLen];
2:      int max = 0; int maxIndex = 0;
3:      sum[0] = 0;
4:      for (int i = 0; i < bufLen; ++i) {
5:           sum[i + 1] = sum[i] + buf[i];
6:           if (max < buf[i])  {
7:                      max = buf[i];
8:                      maxIndex = i;
9:           }
10:      }
11:     if (max > threshold)      return sum[maxIndex];
12:     return sum[bufLen];
}
```

Figure 34: Example of a memory corruption error

In Figure 34, the array *sum* is declared to be of size *bufLen*, which is the number of elements in the array *buf*. However, there is a write to *buf[i+1]* in line 5, where *i* can take values from 0 to *bufeLen.* As a result, a buffer overflow occurs in the last iteration of the loop, leading to the value of the variable *max* being overwritten by the write in line L5 (assuming that *max* is stored immediately after the array *buf).* The value of *max* would be subsequently overwritten with the value of the sum of all the elements in the array, which is something the programmer almost certainly did not expect (this results in a logical error).

In the above example, assume that the variable *max* has been identified as critical, and is being checked in line 9. Recall that the proposed technique will detect a memory corruption error *if and only if* the error causes corruption of the critical variable (which is the case in this example). In this case, the checking expression for *max* will depend on whether the branch corresponding to the *if* statement in line 6 is taken. If the branch is not taken, the value of *max* is the value of *max* from the previous iteration of the loop. If the branch is taken, then the value of *max* is computed to be the value *buf[i]*. These are the only possible values for the *max* variable, and are represented as such in the detector. The memory corruption error in line 5 will overwrite the variable *max* with the value *sum[bufLen],* thereby causing a mismatch in the detector's value. Hence, the error will be detected by the technique.

Note that the detector does not check the actual line of code or the variable where the memory error occurs. Therefore, it can detect any memory corruption error that affects the value of the critical variable, independent of where it occurs. As a result, it does not need to instrument all unsafe writes to memory as done by conventional memory-safety techniques (e.g.[24]).

Race Conditions and Synchronization errors: Race conditions occur in concurrent programs due to lack of synchronized accesses to shared variables[98]. Static analysis techniques typically do not take into account asynchronous modifications of variables when extracting dependences in programs. This also holds for the backward dependence graph of critical variables in the program. As a result, the backward slice only includes modifications to the shared variables made under proper synchronization. Hence, race conditions that result in unsynchronized writes to shared variables will be detected provided the write(s) are to the variables in the backward slice of critical variables that are not shared between the main program and the checking expressions. However, race conditions that result in unsynchronized reads may not be detected unless the result read by the read propagates to the backward slice of the critical variable. Note that the technique would not detect benign races (i.e. race conditions in which the final value of the variable is not affected by the order of the writes), as it checks the value of the variable being written to rather than whether the write is synchronized.

Figure 35 shows a hypothetical example of a race condition in a program. Function *foo* adds a constant value to each element of an array *a* which is passed into it as a formal parameter. It is also passed an array *a_lock*, which maintains fine-grained locks for each element of *A*. Before operating on an element of the array, the thread acquires the appropriate lock from the array *a_lock*. This ensures that no other thread is able to modify the contents of array *a[i], provided the other thread tries to acquire the lock before modifying a[i]*. Therefore, the locks by themselves do not protect the contents of *a[i]* unless all threads adhere to the locking discipline. The property of adherence to the locking discipline is hard to verify using static analysis alone because, (1) The thread modifying the contents of array *a* could be in a different module than the one being analyzed, and the source code of the other module may not be available at compile time, and (2) Precise pointer analysis is required to find the specific element of *a* being written to in the array (it may not even be possible to find this statically if the index is input dependent). Such precise analysis is often unscalable, and static analysis techniques perform approximations that may result in missed detections (or false-positives).

The proposed technique, on the other hand, would detect illegal modifications to the array *a* even by threads that do not follow the locking discipline. Assume that the variable *a[i]* in line 7 has been determined to be a critical variable. The proposed technique would place a check on *a[i]* to recompute it in line 8. Now assume that the variable *a[i]* was modified by an errant thread that does not follow the locking discipline.

This may cause the value of *a[i]* computed in line 7 to be different from what it should have been in a correct execution (which is its previous value added to the constant *c*). Therefore, the error is detected by the recomputation check in line 8.

```
1: void foo(int* a, mutex* alock, int n, int c) {
 2:       int i = 0;
 3:       int sum = 0;
 4:       for (i=0; i<n; i++) {
 5:             acquire_mutex( alock[i] );
 6:             old_a = a[i];
 7:              a[i] = a[i] + c;
 8:              check( a[i] == old_a + c)
 9:              release_mutex( alock[i] );
 10:      }
}
```

Figure 35: Example for race condition detection

The following can be noted in the example: (1) The source code of the errant thread is not needed to derive the check, (2) The check will fail only if the actual computed value is different and is therefore immune to benign races that have no manifestation on the computation of the critical variable, and (3) in this example, it is enough for the technique to analyze the code of the function *foo* to derive the check for detecting the race condition.

4.3.5 Hardware Errors Covered

Hardware transient errors that result in corruption of architectural state are considered in the fault-model. Table 14 shows a detailed characterization of the hardware errors covered by the technique. Examples of hardware errors covered include,

- **Errors in Instruction Fetch and Decode:** Either the wrong instruction is fetched, (OR) a correct instruction is decoded incorrectly resulting in data value corruption.

Table 14: Detailed characterization of hardware errors and their detection by the technique

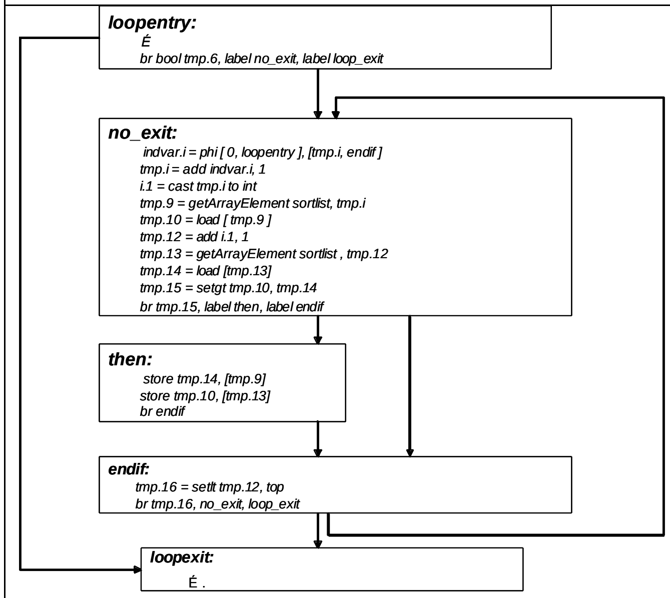
Errorr	Error Description	Detected under what condition ?
Instruction Fetch (IF)	Incorrect (but valid) instruction is fetched	If instruction affects critical value
	Incorrect (invalid) instruction is fetched	
	Invalid memory address is references	
	Extra instruction is inserted	If critical operand is influenced
	Instruction is skipped	If instruction is in backward slice of critical variable
	Same instruction is repeatedly fetched	
	No instruction is fetched	
Instruction Decode Stage	Decoded to invalid op-code	
	Decoded to valid but incorrect opcode	If incorrect op-code affects critical data operands
	Branch decoded to an invalid address	
	Branch decoded to valid but incorrect address	If the missed instruction is in the backward slice of critical variable (OR) if new instruction affects critical operand
	Wrong register operand(s) retrieved	If instruction is in the backward slice of the critical variable and reads from the wrong register (OR) a register holding a critical data operand is incorrectly written to
Errors in Instruction	Computation errors in integer operations	If instruction belongs to backward slice of critical variable and error is not logically masked in ALU
	Computation errors in FP operations	If error occurs in exponent or MSB of mantissa and is not logically masked in ALU
	Computation errors in load/store addresses	If address is valid and the instruction belongs to the backward slice of the critical variable
	Errors in resolving branch direction	If critical variable's value differs on both directions of the branch in question
	Errors in branch target address computation	If address is valid, and new target is not one of allowed targets and the check is reached
Memory Stage (MEM)	Invalid address is referenced in Load/Store	
	Data fetched from incorrect address for L/S	Data is used in critical value computation
	Data not fetched from memory for L/S	Data is used in critical value computation
	Data written to incorrect address for L/S	Data is used in critical operand computation (OR) critical operand is overwritten
	Data not written to memory for L/S	Data is used in critical operand computation
	Incorrect value is written to the PC on branch	If address is valid, and new target is not one of allowed targets and the check is reached
	Value is not written to the PC on branch	if critical variable's value differs on both directions of the branch
Write-back Stage (WB)	ALU instruction not written back	Instruction belongs to backward slice of critical variable
	ALU instruction written to wrong register	if register used in critical value computation is overwritten (OR) instruction belongs to backward slice
	Load instruction stalled indefinitely	
	Load instruction written to wrong register	if register used in critical value computation is overwritten (OR) instruction belongs to backward slice
	Exception occurs incorrectly during commit	
	Exception omitted during commit	Assuming critical value computation throws exception
storage elements	Errors in memory	Memory operand used in critical value computation but is not used in the checking expression
	Errors in cache	If the cached operand is used in original computation and not in checking expression
	Errors in registers	If original computation and checking expression use different registers and no value forwarding takes place
	Errors in register bus	If the same register is reread by the checking expression
	Errors in memory bus	If operand is reloaded by the checking expression

- **Errors in Execute and Memory Units:** An ALU instruction is executed incorrectly inside a functional unit, (OR) the wrong memory address is computed for a load/store instruction, resulting in data value corruption.
- **Errors in Cache/Memory/Register File Errors:** A value in the cache, memory, or register file experiences a soft error that causes it to be incorrectly interpreted in the program (if ECC is not used).

4.4 STATIC ANALYSIS

This section describes the static analysis technique to derive detectors and add instrumentation for path tracking to a program. The bubble-sort program shown in Figure 36(a) is used as a working example throughout this section. We use the LLVM compiler infrastructure [99] to derive error detectors for the program. A new compiler pass called the *Value Recomputation Pass (VRP)* was introduced into LLVM. The VRP performs the backward slicing starting from the instruction that computes the value of the critical variable to the beginning of the function. It also performs check derivation, insertion and instrumentation. The output of the VRP is provided as input to the optimization passes of LLVM in order to reduce the check to a minimal expression.

```
void Bubble(int srtElements, int* sortList) {
          int i, j,  top;
         bInitarr( sortList, srtElements );
         top=srtelements;
        while ( top>1 ) {//Outer-while-loop
               i=1;
              while ( i<top ) {// Inner while-loop
                       if ( sortlist[i] > sortlist[i+1] )
                       {
                                 j = sortlist[i];
                                sortlist[i] = sortlist[i+1];
                                sortlist[i+1] = j;
                       } // end-if
                       i=i+1;
              } // end-inner-while
             top=top-1;
       } // end-outer-while
}
```
(a)



```
loopentry:
      É
      br bool tmp.6, label no_exit, label loop_exit

no_exit:
       indvar.i = phi [ 0, loopentry ], [tmp.i, endif ]
      tmp.i = add indvar.i, 1
      i.1 = cast tmp.i to int
      tmp.9 = getArrayElement sortlist, tmp.i
      tmp.10 = load [ tmp.9 ]
      tmp.12 = add i.1, 1
      tmp.13 = getArrayElement sortlist , tmp.12
      tmp.14 = load [tmp.13]
      tmp.15 = setgt tmp.10, tmp.14
      br tmp.15, label then, label endif

then:
       store tmp.14, [tmp.9]
      store tmp.10, [tmp.13]
      br endif

endif:
     tmp.16 = setlt tmp.12, top
     br tmp.16, no_exit, loop_exit

loopexit:
         É .
```

(b)

Figure 36: (a) Example code fragment (b) Corresponding LLVM intermediate code

LLVM uses Static Single Assignment form (SSA) [100] as its intermediate code representation. In deriving the backward program slice, two well understood properties of SSA form are used as follows:

- In SSA form, each variable (value) is defined exactly once in the program, and the definition is assigned a unique name, which facilitates analyzing data dependences among instructions.
- SSA form uses a special static construct called the *phi* instruction that is used to keep track of the data dependences when there is a merging of data values from different control edges. The *phi* instruction includes the variable name for each control edge that is merged and the corresponding basic block. This instruction allows the specialization of the backward slice based on control-paths by the proposed technique.

A simplified version of the LLVM intermediate code corresponding to the inner-while loop in the bubble-sort program is shown in Figure 36b.

4.4.1 Value Recomputation Pass

The VRP takes LLVM intermediate code annotated with critical variables and extracts their path-specific backward slices. It computes the backward slice by traversing the static dependence graph of the program starting from the instruction that computes the value of the critical variable up until the beginning of the function. The VRP outputs instrumented LLVM intermediate code that tracks paths and invokes detectors. By extracting the path-specific backward slice and exposing it to other optimization passes in the compiler, the Value Recomputation Pass (VRP) enables aggressive compiler optimizations to be performed on the slice that would not be possible otherwise.

4.4.1.1 Overall Approach

The algorithm for performing path-specific slicing is shown in Table 15. To the best of our knowledge, this is the first path-specific static slicing algorithm developed to enable derivation of error detectors. The algorithm is explained as follows:

Table 15: Pseudocode of backward traversal algorithm

```
Function visit( seedInstruction, pathID, parent ):
      ActiveSet ={ seedInstruction }
       if parent==0:
              SliceList[ pathID ] = { }
        else:
              SliceList[ pathID ] = SliceList[ parent ]
       nextPathID = pathID
       while not empty( ActiveSet ):
                I = Remove instruction for ActiveSet
                Visited[ BasicBlock(I) ] = true
                 // Do not consider interprocedural slices
                if I is a function argument or constant:
                        terminal = true
                else if I is a non-phi instruction:
                         SliceList[ pathID] =  SliceList[PathID]
                                                    U { I }
                         ActiveSet = ActiveSet U operands( I )
                 else if I is a phi instruction:
                        for  each operand of the phi:
                            // Check if a loop is encountered
                           // or if  going back multiple iterations
                            if not ( Visited [ BasicBlock(operand) ]
                              and not CrossingInsn(I, operand) )
                                 nextPathID = pathID + 1
                                 result = Visit(operand,
                                                   nextPathID, pathID )
                                 terminal = terminal OR ~(result)
                            else:
                                SeedList = SeedList U { operand }
         // Add the path to the pathList if terminal path
         if (terminal)
                 PathList = PathList U { pathID }
        return terminal

Function computeSlices (criticalInstruction):
       SeedList = {  criticalInstruction }
       PathList = { }
        while not empty( SeedList ):
            seedInstruction=Remove instruction from SeedList
            call visit( seedInstruction, 0, 0 )
        return PathList, SliceList
```

The instruction that computes the critical variable in the program is called the critical instruction. In order to derive the backward program slice of a critical instruction, the algorithm performs backward traversal of the static data dependence graph. The traversal starts from the critical instruction and terminates when one or more of the following conditions are met:

- **The beginning of the current function is reached**. It is sufficient to consider intra-procedural slices in the backward traversal because each function is considered separately for the detector placement analysis. For example, in Figure 36a the array *sortList* is passed as an argument to the function *Bubble*. The slice does not include the computation of *sortList* in the calling function. If *sortList* is a critical variable in the calling function, say *foo*, then a detector will be derived for it when *foo* is analyzed.
- **A basic block is revisited in a loop**. During the backward traversal, if data dependence within a loop is encountered, the detector is broken into two detectors, one placed on the critical variable and one on the variable that affects the critical variable within the loop. This second detector ensures that the variable within the loop is computed correctly and hence the variable can be used without recomputing it in the first detector. Hence, only acyclic paths are considered by the algorithm.
- **A dependence across loop iterations is encountered**. Recomputing critical variables across multiple loop iterations can involve loop unrolling or buffering intermediate values that are rewritten in the loop. This in turn can complicate the design of the detector. Instead, the VRP splits the detector into two detectors, one for the dependence-generating variable and one for the critical variable.

- **A memory operand is encountered**. Memory dependences are not considered because LLVM promotes most memory objects to registers prior to running the VRP. Since there is an unbounded number of virtual registers for storing variables in SSA form, the analysis does not have to be constrained by the number of physical registers available on the target machine. However it may not always be possible to promote a memory objects to a register e.g. pointer references to dynamically allocated data. In such cases, the VRP duplicates the load of the memory object, provided the load address is not modified along the control path from the load instruction to the critical instruction.

4.4.1.2 VRP Algorithm Details

During the backward traversal, when a *phi*-instruction is encountered indicating a merge in control-flow paths, the slice is forked for each control path that is merged at the *phi*. The algorithm maintains the list of instructions in each path-specific slice in the array *SliceList*. The function *computeSlices* takes as input the critical instruction and outputs the *SliceList* array, which contains the instructions in the backwards slice for each acyclic path in the function.

The actual traversal of the dependence graph occurs in the function *visit*, which takes as input the starting instruction, an ID (number) corresponding to the control-flow path it traverses (index of the path in the *SliceList* array), and the index of the parent path. The *computeSlices* function calls the *visit* function for each critical instruction. The *visit* function visits each operand of an instruction in turn, adding it to the *SliceList* of the current path. When a *phi* instruction is encountered, a new path is spawned for each operand of the *phi* instruction (by calling the *visit* function recursively on the operand with a new path ID and the current path as the parent). The traversal is then continued along this new path. Only terminal paths are added to the final list of paths (*PathList)* returned by the *ComputeSlice* procedure. A terminal path is defined as one that terminates without spawning any new paths (as a result of forking).

Certain instructions cannot be recomputed in the checking expression, because performing recomputation of such instructions can alter the semantics of the program. Examples are *malloc*s, *frees*, function calls and function returns. Omitting *mallocs* and *frees* does not seem to impact coverage except for allocation intensive programs, as shown by our results in section 4.6.2. Omitting function calls and returns does not impact coverage for program functions because the detector placement analysis considers each function separately (section 4.3.2).

Assuming that the critical variable chosen for the example in Figure 36a is *sortlist[i]*, the intermediate code representation for this variable is the instruction *tmp.10* in Figure 36b. The VRP computes the backward slice of *tmp.10*, which consists of the two paths shown in Figure 37.

Path 0: no_exit → loopentry	Path 1: endif → loopentry
indvar.i = 0	indvar.i = tmp.i
tmp.i = add indvar.i, 1	tmp.i = add indvar.i, 1
tmp.9 =getArrayElement sortlist,tmp.i	tmp.9 = getArrayElement sortlist,tmp.i
tmp.10 = load[tmp.9]	tmp.10 = load [tmp.9]

Figure 37: Path-specific slices for example

4.4.1.3 VRP and Other Optimization Passes

After extracting the path-specific slices, the VRP performs the following operations on the slices:

- Places the instructions in the backward slice of the critical variable corresponding to each control path in its own basic block.
- Replaces the *phi* instructions in the slice with the incoming value corresponding to the control edges for the path. This allows subsequent compiler optimization passes to substitute the *phi* values directly in their uses through either constant propagation or copy propagation [101].
- Creates copies of variables used in the path-specific slices that are not live at the detector insertion point. For example, the value of *tmp.i* is overwritten in the loop before the detector can be reached and a copy *old.tmp.i* is created before the value is overwritten.
- Renames the operands in the slices to avoid conflicts with the main program and thereby ensure that SSA form is preserved by the slice.
- Instruments program branches with path identifiers considered by the backward slicing algorithm. This includes introduction of special instructions at branches pertaining to the paths in the slice, and also at function entry and exit points.

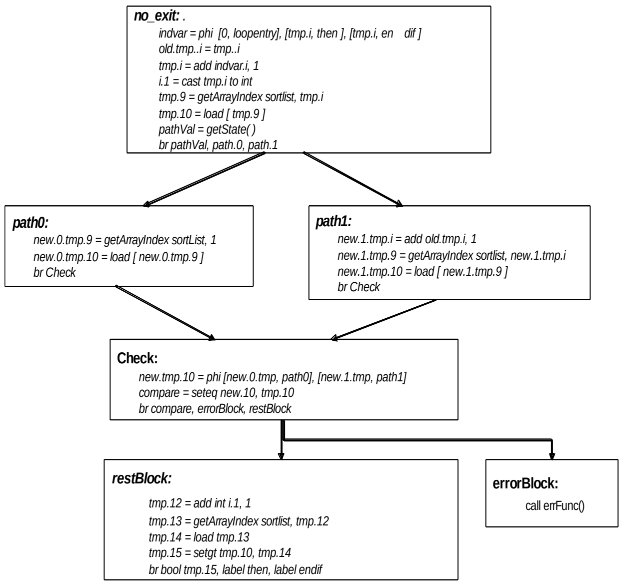
The standard LLVM optimization passes are invoked on the path-specific backward slices extracted by the VRP. The optimization passes yield reduced instruction sequences that compute the critical variables for the corresponding paths. Further, since there are no control-transfers within the sequence of instructions for each path, the compiler is able to optimize the instruction sequence for the path much more aggressively than it would have otherwise. This is because the compiler does not usually consider specific control paths when performing optimizations for reasons of space and time efficiency. *However, by selectively extracting the backward slices for critical variables and by specializing them for specific control paths, the VRP is able to keep the space and time overheads manageable (see Section 4.4.1.5)*

4.4.1.4 VRP Output

The LLVM intermediate code from Figure 36 with the checks inserted by the VRP is shown in Figure 38.

The VRP creates two different instruction sequences to compute the value of the critical variable corresponding to the control paths in the code. The first control path corresponds to the control transfer from the basic block *loopentry* to the basic block *no_exit* in Figure 38. The optimized set of instructions corresponding to the first control path is encoded as a checking expression in the block *path0* in Figure 38. The second control path corresponds to the control transfer from the basic block *endif* to the basic block *no_exit* in Figure 36. The optimized set of instructions corresponding to the second control path is encoded as a checking expression in the block *path1* in Figure 38.

The instructions in the basic blocks path0 and path1 recompute the value of the critical variable tmp.10. These instruction sequences constitute the checking expressions for the critical variable tmp.10 and comprise of 2 instructions and 3 instructions respectively.



```
no_exit: .
        indvar = phi  [0, loopentry], [tmp.i, then ], [tmp.i, en    dif ]
        old.tmp..i = tmp..i
        tmp.i = add indvar.i, 1
        i.1 = cast tmp.i to int
        tmp.9 = getArrayIndex sortlist, tmp.i
        tmp.10 = load [ tmp.9 ]
        pathVal = getState( )
        br pathVal, path.0, path.1

path0:
      new.0.tmp.9 = getArrayIndex sortList, 1
      new.0.tmp.10 = load [ new.0.tmp.9 ]
      br Check

path1:
      new.1.tmp.i = add old.tmp.i, 1
      new.1.tmp.9 = getArrayIndex sortlist, new.1.tmp.i
      new.1.tmp.10 = load [ new.1.tmp.9 ]
      br Check

Check:
      new.tmp.10 = phi [new.0.tmp, path0], [new.1.tmp, path1]
      compare = seteq new.10, tmp.10
      br compare, errorBlock, restBlock

restBlock:
          tmp.12 = add int i.1, 1
          tmp.13 = getArrayIndex sortlist, tmp.12
          tmp.14 = load tmp.13
          tmp.15 = setgt tmp.10, tmp.14
          br bool tmp.15, label then, label endif

errorBlock:
          call errFunc()
```

Figure 38: LLVM code with checks inserted by VRP

The basic block *Check* in Figure 38 compares the value computed by the checking expressions to the value computed in the original program. A mismatch signals an error and the appropriate error handler is invoked in the basic block *error.* Otherwise, control is transferred to the basic block *restBlock*, which contains the instructions following the computation of *tmp.10* in the original program.

4.4.1.5 Scalability

This section discusses factors that could potentially limit the scalability of the VRP algorithm and how these are addressed by the proposed technique.

- **Number of control paths:** This is addressed by considering only intra-procedural, acyclic paths in the program corresponding to the backward slices of critical variables in the program. At worst, this can be exponential in the number of branch instructions in the program. In practice however, the number of control paths is polynomial in the number of branch instructions (unless the program is performing decision tree like computations).

- **Size of checking expression:** The size of the checking expression depends on the number of levels in the dependence tree of the critical variable considered by the algorithm. Terminating the dependency tree at loop and function boundaries naturally limits the checking expression's size.
- **Number of detectors:** The number of critical variables per function is a tradeoff between the desired coverage and an acceptable performance overhead. Placing more detectors achieves higher coverage but may result in higher overheads. The algorithm may introduce additional detectors, for example, when splitting a detector into two detectors across loop iterations, but this reduces the size of each checking expression. Therefore, for a given number of critical variables, the number of detectors varies inversely as the size of each checking expression.

4.4.1.6 Coverage

The VRP operates on program variables at the compiler's intermediate representation (IR) level. In the LLVM infrastructure, the IR is close to the program's source code [99]and abstracts many of the low-level details of the underlying architecture. For example, the IR has an infinite number of virtual registers, uses Static Single Assignment (SSA), and has native support for memory allocation (*malloc* and *alloca*) and pointer arithmetic (*getElementPtr*[17] instruction). Moreover, the runtime mechanisms for stack manipulations and function calls are transparent to the IR. As a result, the VRP may not protect data that is not visible at the IR level. Therefore, the VRP is best suited for detecting errors that impact program state visible at the source level. Note that the generic approach presented in Section 4.3, however, is not tied to a specific level of compilation and can be implemented at any level.

The VRP operates on LLVM's intermediate code, which does not include common runtime mechanisms such as manipulation of the stack and base pointers. Moreover, the intermediate code assumes that the target machine has an infinite register file and does not take into account the physical limitations of the machine.

Data errors in a program can occur in three possible places (locations): (1) Source-level variables or memory objects, (2) Precompiled Libraries linked with the application, and (3) Code added by the compiler's target-specific code generator for common runtime operations such as stack manipulation and handling register-file spills. The technique presented in the chapter aims at detecting errors in the first category, and can be extended to detect errors in the second category provided the source code of the library is available or the library is compiled with the proposed technique. However, errors in the third category, namely those that occur in the code added by the compiler's code generator cannot be detected using the proposed technique unless the error affects one or more

[17] This is the general case of the *getArrayElement* instruction introduced previously

source-level variables or memory objects. This is because the code added by the compiler is transparent to the VRP and hence cannot be protected by the derived detectors.

The steps in compiling a program with LLVM are as follows: First, the application's source code along with the source (or intermediate) code of runtime libraries are converted to LLVM's generic intermediate code form. This intermediate form is in-turn compiled onto the target architecture's object code, which is then linked with pre-compiled libraries to form the final executable. The process is similar to conventional compilation, except that the application and the source libraries are first compiled to the intermediate code format (by a modified gcc front-end) before being converted to object code. Each level of compilation progressively adds more state (code and data) to the program. Table 16 shows the data elements of the program's state visible at each level of compilation.

It can be observed from the table that the intermediate code level does not include many data elements in the final executable as these are added by the compiler and linker. Since the VRP operates at the intermediate code level, it does not see the elements in the lower levels and the derived detectors may not detect errors in these levels. This can be addressed by implementing the technique at lower compilation levels.

Table 16: Information about the program that is available at different levels of compilation

Code Level	**Elements of program state that are visible**
Source Level	(1) local variables, (2) global variables and (3) dynamic data allocated on heap
Intermediate Code	(1) Branch addresses of *if* statements, loops , and case statements, (2) Temporary variables used in evaluation of complex expressions
Object Code	(1) Temporary variables to handle register file spills, (2) Stack manipulation mechanisms and (3) Temporary variables to convert out of SSA form

4.4.2 State Machine Generation

The VRP extracts a set of checking expressions for each detector in the program. Each checking expression in the set corresponds to an acyclic, intra-procedural control path leading up to the critical variable from the top of the function. The VRP also inserts instrumentation to notify the runtime system when the program takes a branch belonging to one of the paths in the set. This is done by inserting a special operation called *EmitEdge* that identifies the source and destination basic blocks of the branch with unique identifiers. The VRP then exports the basic block identifiers of the branches along each path in a separate text file for each detector in the program.

A post-processing analysis then parses these text files and builds a state-machine representation of the paths for each check. The state machines are constructed such that every instrumented branch in the program causes state transitions in one or more state machines. A complete sequence of branches corresponding to a control path for which a checking expression has been derived, will drive the state machine for the check to an accepting state corresponding to the checking expression.

- The algorithm used by the post-processing analysis to convert the control edge sequences to finite state machines is shown in Table 17. The algorithm processes the path files for each check, and adds states to the state machine corresponding to the check. The aim is to distinguish one path from another in the check, while at the same time introducing the least number of states to the state machine. This is because each state occupies a fixed number of bits in hardware, and our goal is to minimize the total number of bits that must be stored by the hardware module for path-tracking and consequently the area occupied by it.

- The algorithm in Table 17 works as follows: It starts in the starting state of the state machine and processes each edge in the list of edges for the path. It adds a new state for an edge if and only if there no transition exists for the edge from the current state in the state machine. If such a transition exists, it transitions to the state leading from the current state corresponding to the edge, and processes the next edge in the path. It continues until it has processed all the edges of the path, and marks the last state added as the accepting state for the path in the state machine. When the algorithm terminates, it outputs the transition table for the state machines, as well as the list of accepting states corresponding to each path of the check. The states are programmed into the hardware module for path-tracking (Section 4.8) at application load time.

Table 17: Algorithm to convert paths to state machines

```
for each critical variable V in the program:
    open the path-file corresponding to the variable
    for each path in the path-file:
        PathNumber ← Read path ID in path file
        Read an edge e = (src, sink) from the path file
        S ← Start_State
        Create an accepting state “A” for the path
        if this is the only edge for the path:
            if Transition[S, A] does not contain e
                Transition[S, A] <- Transition[S,A] U e
        else:
            current = S
            for each edge e in the path
                if there exists a state K such that
                  (Transition[current,K] contains e):
                    current ← K
                else:
                    Create a new state L
                    Transition[current, L] ← e
                    current ← L
            endfor
            Set current as the accepting state for path
    endfor
    close the path file for the critical variable
endfor
```

Figure 39 shows an example control-flow graph (CFG) of a program for which paths must be tracked. Each basic block in the CFG has been assigned a unique index by the VRP. Assume that the critical variable is computed in basic block with identifier 6.

The VRP has identified 4 acyclic paths in the backward slice of this critical variable labeled A to D. The paths consist of edge sequences that distinguish one path from another in the set of paths for a detector. Note that the edges in each path correspond to the control edges that result in the VRP forking a new path during the backward traversal shown in **Table 15**.

The state machine derived by the algorithm for the control-flow graph in Figure 39 is shown in Figure 40.The algorithm has introduced two new states *E* and *F* in addition to four accepting states *A, B, C* and *D* that constitute the accepting states for the four paths. Note that the transitions between states correspond to the edges identified by the VRP to distinguish one path from another. These correspond to the edges that merge paths in the SSA graph corresponding to the backward slice of critical variables.

The time-complexity of the algorithm in Table 17 is $O(|V| * |P| * |E|)$, where |V| is the number of critical variables in the program, |P| is the maximum number of control-paths in the backward slice of the variable and |E| is the maximum number of control-edges the control paths corresponding to each critical variable. The space complexity of the technique is $O(|V| * |\mathring{U} E| * H)$, where |H| is the maximum number of shared edges among control-paths corresponding to the critical variables, and Ů E represents the union of all the edges in the program's control paths.

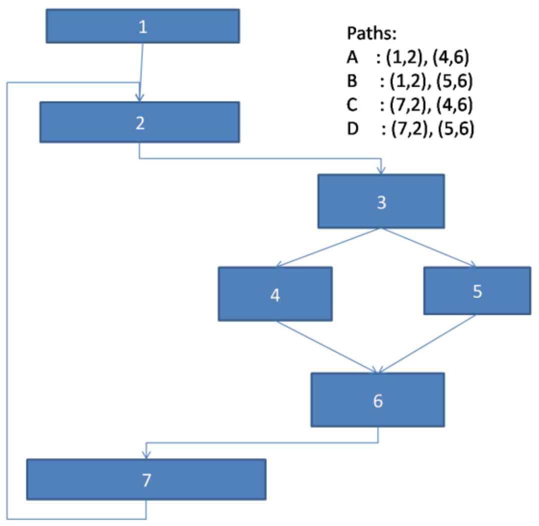


Figure 39: Example Control-flow graph and paths

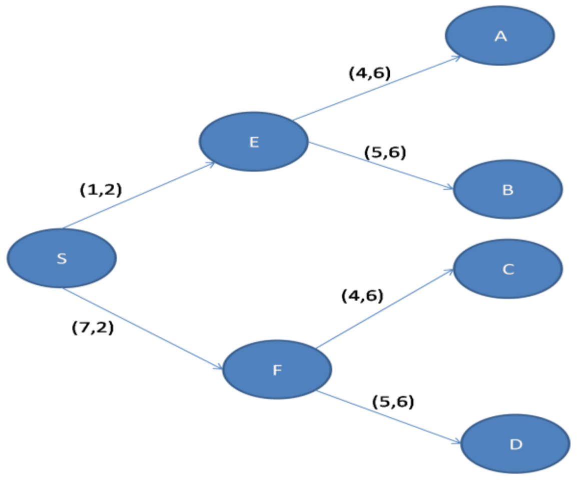


Figure 40: State machine corresponding to the Control Flow Graph

4.5 EXPERIMENTAL SETUP

This section describes the mechanisms for measurement of performance and coverage provided by the proposed technique. It also describes the benchmarks used for evaluating the technique.

4.5.1 Performance Measurement

All experiments are carried out on a single core Pentium 4 machine with 1GB RAM and 2.0 Ghz clock speed running the Linux operating system. The performance overheads of each component introduced by the proposed technique can be measured as follows:

Modification overhead: Performance overhead due to the extra code introduced by the *VRP* for instrumentation and checking. This code may cause cache misses and branch mispredictions and lead to performance overhead.

Checking overhead: Performance overhead of executing the instructions in each check to recompute the critical variable and compare the recomputed value with the original value.

T*he overhead of path-tracking is not considered in measuring performance overheads* because the path tracking is done in parallel with the execution of the main program using a specialized hardware module. The path-tracking module and can execute asynchronously and needs to be synchronized with the main processor only when the check is performed (see Section 4.8 for a detailed description).

We implemented the path-tracking module using software emulation and measured the performance overheads of the application with both path-tracking and checking enabled. We then measure the application overhead with only path-tracking enabled and subtract it from the earlier result in order to obtain the checking overheads. In order to obtain the code modification overheads, we executed the application with both path-tracking and checking disabled and measured the increase in execution time over the unmodified application.

4.5.2 Coverage Measurements

Fault Injections: In order to measure the coverage of the derived detectors, we inject faults into the data of the application protected with the derived detectors. A new LLVM pass inserts calls to a special *faultInject* function (invoked after the optimization phases) after the computation of each program variable in the original program. The variable to be injected is passed as an argument to the *faultInject* function. The uses of the program variable in the original program are substituted with the return value of the *faultInject* function.

At runtime, the call to the *faultInject* function corrupts the value of a single program variable by flipping a single bit in its value. The value into which the fault is injected is chosen at random from the entire set of dynamic values used in an error-free execution of the program (that are visible at the compiler's intermediate code level). In order to ensure controllability, only a single fault is injected in each execution of the application.

Error Detection: After a fault is injected, the following program outcomes are possible: (1) the program may terminate by taking an exception (crash), (2) the program may continue and produce correct output (success), (3) the program may continue and produce incorrect output (fail-silent violation) or (4) the program may timeout (hang). The injected fault may also cause one of the inserted detectors to detect the error and flag a violation.

When a violation is flagged, the program is allowed to continue (although in reality it would be stopped) in order that the final outcome of the program under the error can be observed. The coverage of the detector is classified based on the observed program outcome. For example, a detector is said to detect a crash if the detector upon encountering the error, flags a violation, after which the program crashes. Hence, when a detector detects a crash, it is in reality, preempting the crash of the program.

Error Propagation: Our goal is to measure the effectiveness of the detectors in detecting errors that propagate before causing the program to crash. For errors that do not propagate before the crash, the crash itself may be considered the detection mechanism (as the state can be recovered from a clean checkpoint). Hence, coverage provided by the derived detectors for non-propagated errors is not reported. In the experiments, error propagation is tracked by observing whether an instruction that uses the erroneous variable's value is executed after the fault has been injected. If the original value into which the error was injected is overwritten, the error propagation is no longer tracked. The program is instrumented to track error-propagation and the instrumentation is automatically inserted by a new LLVM pass that we introduced.

4.5.3 Benchmarks

Table 18 describes the programs used to evaluate the technique and their characteristics.

The first 9 programs in the table are from the Stanford benchmark suite[102] and the next 5 programs are from the Olden benchmark suite[103]. The former benchmark set consists of small programs performing a multitude of common tasks. The latter benchmark set consists of pointer-intensive programs commonly used to evaluate memory systems.

Table 18: Benchmark programs and characteristics

Benchmark	Lines of C	Description of program
IntMM	159	Matrix multiplication of integers
RealMM	161	Matrix multiplication of floating-points
FFT	270	Computes Fast-Fourier Transform
Quicksort	174	Sorts a list of numbers using quicksort
Bubblesort	171	Sorts a list of numbers using bubblesort
Treesort	187	Sorts a list of numbers using treesort
Perm	169	Computes all permutations of a string
Queens	188	Solves the N-Queens problem
Towers	218	Solves the Towers of Hanoi problem
Health	409	Discrete-event simulation using double linked lists
Em3d	639	Electro-magnetic wave propagation in 3D (using single linked lists)
Mst	389	Computes minimum spanning tree (graphs)
Barnes-Hut	1427	Solves N-body force computation problem using octrees
Tsp	572	Solves traveling salesman problem using binary trees

4.6 RESULTS

This section presents the performance (Section 4.6.1), and coverage results (Section 4.6.2) obtained from the experimental evaluation of the proposed technique. The results are reported for the case when 5 critical variables were chosen in each function by the placement analysis. We do not report results for other cases due to space constraints (these numbers are available on request).

4.6.1 Performance Overheads

The performance overhead of the derived detectors relative to the normal (uninstrumented) program's execution is shown in Figure 41. The results are summarized below:

- The average checking overhead introduced by the detectors is 25%, while the average code modification overhead is 8%. Therefore, the total performance overhead introduced by the detectors is 33%.

- The worst-case overheads are incurred in the case of the *tsp* application, which has a total overhead of nearly 80%. This is because *tsp* is a compute-intensive program involving tight loops. Placing checks within a loop introduces extra branch instructions and increases its execution time.

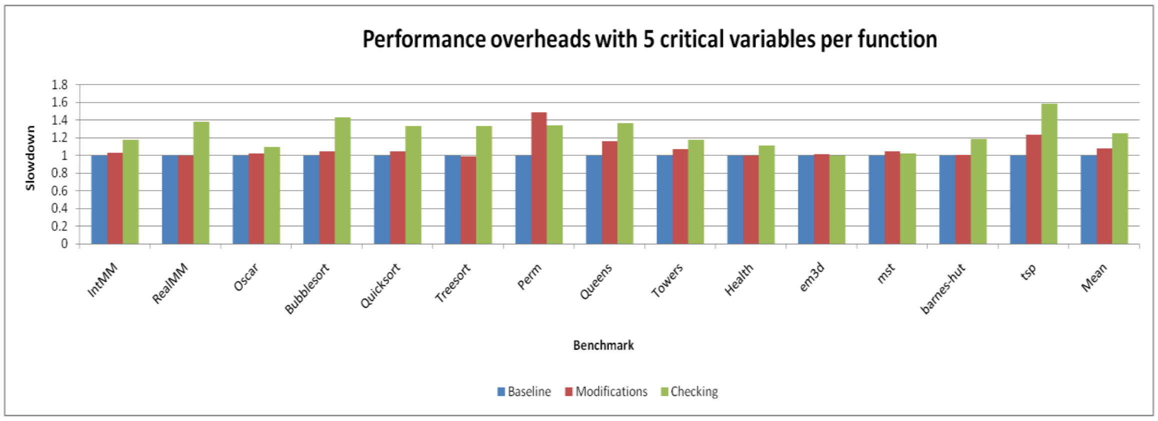


Figure 41: Performance overhead when 5 critical variables are chosen per function

4.6.2 Detection Coverage

For each application, 1000 faults are injected, one in each execution of the application. The error-detection coverage (when 5 critical variables are chosen in each function) for different classes of failure are reported in Table 19.

A blank entry in the table indicates that no faults of the type were manifested for the application. For example, no hangs were manifested for the *IntMM* application in the fault injection experiments. The second column of the table shows the number of errors that propagate and lead to the application crashing. The numbers within the braces in this column indicate the percentage of propagated, crash-causing errors that are detected before propagation.

Table 19: Coverage with 5 critical variables per function

Apps	**Prop. Crashes (%)**	**FSV (%)**	**Hang (%)**	**Success (%)**
IntMM	100 *(97)*	100		9
RealMM	100 *(98)*			0
FFT	57 *(34)*	7	60	0.5
Quicksort	90 (57*)*	44	100	4
Bubblesort	100 *(73)*	100	0	5
Treesort	75 *(68)*	50		3
Perm	100 *(55)*	16		0.9
Queens	79 *(61)*	20		3
Towers	79 *(78)*	39	100	2
Health	39 *(39)*	0	0	0
Em3d	79 *(79)*			1
Mst	83 *(53)*	79	0	5
Barnes-Hut	49 *(39)*		23	
Tsp	64 (64)		0	0
Average	77 (64)	41	35	2.5

The results in Table 19 are summarized as follows:

- The derived detectors detect 77% of errors that propagate and crash the program.

64% of crash-causing errors that propagate are detected before first propagation. These correspond to 83% of the propagated crash-causing errors that are detected by the derived detectors.

- The derived detectors detect 41% of errors that result in fail-silent violations

(incorrect outputs) and 35% of errors that result in hangs on average across applications.

- The number of benign errors detected is 2.5% on average across applications. Recall

that these errors have no effect on the execution of the application.

- The worst-case coverage for errors causing crashes (that exhibit error propagation) is

obtained in the case of the Olden program *health* (39%). The *health* program is allocation-intensive, and spends a substantial fraction (over 50%) of its time in *malloc* calls. Our technique does not protect the return value of *mallocs* as duplicating *malloc* calls changes the semantics of the program. Further, the technique does not place detectors within the body of the *malloc* function, as it does not have access to the source-code of library functions. This can be remedied by releasing versions of libraries compiled using the technique described in this chapter.

4.6.3 Discussion

The results indicate that our technique achieves 77% coverage for errors that propagate and cause the program to crash. Full-duplication approaches can provide 100% coverage if they perform comparisons after every instruction. In practice, this is very expensive and full-duplication approaches compare instructions only before store and branch instructions [68, 69]. With this optimization, the coverage provided by full-duplication is less than 100%. The papers that describe these techniques do not quantify the coverage in terms of error propagation, so a direct comparison with our technique is not possible.

The performance overhead of the technique is only 33 % (when 5 detectors are placed in each function), compared to full-duplication, which incurs an overhead of 60-100% when performed in software. Further, the proposed technique detects just 2.5 % of benign errors in an application compared to full-duplication, in which over 50% of the detected errors are benign [12].

4.7 COMPARISON WITH DDVF AND ARGUS

4.7.1 DDVF

DDVF [104] is an approach to detect errors in the processor by checking if the program's static dataflow graph (DFG) is followed at runtime – i.e. the runtime DFG corresponds to the static DFG. The static data-flow graph is constructed by analyzing the program binary and the runtime dataflow graph is tracked using processor modifications. Since computing the whole program data-flow graph is infeasible in practice, DDVF computes the DFG on a per-basic block basis and enforce the DFG for each basic block separately. In other words, it breaks down the problem of computing the static DFG for the program into the easier problem of computing the DFG for each basic block in the program. Thus, it can detect (hardware) errors that affect the intra-block DFG, but not those that affect the inter-block DFG. Further, it does not track memory dependences in the DFG - instead it approximates memory to be a single node in the DFG and consider memory loads and stores as in- and out- edges for the node. *In effect, the DDVF scheme tracks intra-block, register dependences among program instructions.* Table 20 compares the coverage of the DDVF technique with the Critical Variable Recomputation (CVR) technique. From Table 20, it can be observed that DDVF provides coverage for a much narrower range of errors and attacks compared to the CVR technique. On the other hand, the coverage provided by the DDVF technique is not limited to the backward slices of critical variables in the code. Further, DDVF requires no modifications to the compiler as the signatures are derived by direct analysis of the binary. This limits its coverage considerably as it does not consider memory dependences or inter-block control-flow.

Table 20: Comparison between the CVR and DDVF techniques in terms of coverage

Error Class	Explanation	DDVF detected ?	CVR detected ?
Code Errors	Corruption of program instructions	Yes, provided the number of bits in the signatures is large enough	Yes, If instruction belongs to the backward slice of critical variable (CV)
Control-flow Errors	Corruption of program's control-flow graph	Errors in Intra-block control-flow, but not in inter-block control-flow	Yes, If it bypasses an instruction used in CV computation or results in extra writes to the CV
Data Value Corruptions	Corruption of data values used in program	Errors in cache and registers, but not computation	Yes, If data value is in backward slice of critical variable
Software Errors	Memory corruption errors, race conditions in multi-threaded programs	No, because the program binary is used to derive the signatures	Yes, if the error violated the source-level properties of the critical variable (i.e. error leads to undefined source-level behavior)

4.7.2 Argus

In Argus [105], Meixner and Sorin deploy the DDVF scheme in a full-fledged implementation of a simple in-order processor on a FPGA. They present an enhanced version of the DDVF scheme called DCS (Dataflow and Control Signature). The main difference is that instead of embedding the signature of each basic block within itself, the signature of the (legal) successor blocks of a basic block are embedded within it. At runtime, the checker determines which of the legal successor's should be executed (based on the program's state) and compares the signature computed for the basic block with the signature stored in the chosen successor. In case of a mismatch, the program will be halted. A mismatch indicates that either the wrong successor to the basic block was chosen (control-flow error) or the signature computed for the basic block at runtime was incorrect (code error).

Argus is also equipped with standard fault-tolerance techniques such as watchdog timers, self-checking arithmetic and logical units (using modulo arithmetic) and parity bits on the address/data bus. The paper claims that taken together these techniques offer protection from 98.8 % of errors (both transient and permanent) for 12 % area overhead and 3.5 % performance overhead. These results are based on a model of a simple in-order core written in VHDL and synthesized using an FPGA and are likely to be higher in more complex processors.

Argus provides detection of errors in the code, control and data, but does not protect from errors where a legal but invalid (for that input) path is executed. Detecting legal but incorrect paths will require whole program analysis, rather than just basic-block level analysis as done by Argus. Further, our technique is able to provide protection from a much wider range of errors as we enforce “source-level invariants” as opposed to Argus, which only enforces “binary-level” invariants. Consequently, we can detect errors and attacks that break source-level invariants but not binary-level invariants e.g. memory corruption errors, race conditions and insider attacks.

4.8 HARDWARE IMPLEMENTATION

This section discusses the hardware module for tracking control paths in the program based on the finite state machines derived in section 4.4.2. The state machines are programmed into a reconfigurable hardware module at application load time. They keep track of the control path executed by the application for the derived detectors.

Related Work: Software-based path-profiling approaches [106] incur high overheads in space and time (up to 35 %) compared to hardware-based approaches[107, 108].

Vaswani et al. [107] propose a generic co-processor for profiling paths in hardware. The goal of this approach is to create statistical aggregates of application behavior, rather than track specific paths. Further, this approach requires a much higher degree of coupling with the pipeline, compared to our approach.

Zhang et al. [108] propose a hardware module that interfaces with the processor pipeline to track paths for detecting security attacks. However, their approach requires every branch in the program to be instrumented, which can lead to prohibitive overheads. Our approach is aimed at tracking specific control-paths in the program (for which checks are derived), and requires only selected control edges (branches) to be instrumented.

Implementation

As explained in Section 4.3.2, the path-tracking hardware is implemented as a module in the Reliability and Security Engine (RSE) and monitors the main processor's data path. It keeps track of the control path executed by the program, encoded as finite state machines.

Interface with the main processor: The main processor uses special instructions (called CHECK) to invoke the RSE modules. The path tracking module supports three primitive operations encoded as CHECK instructions. The operations are as follows:

emitEdge(from, to): Triggers transitions in the state machines corresponding to one or more detectors. Each basic block in the program is assigned a unique identifier assigned by the VRP. This operation indicates that control is transferred from the basic block with identifier *from* to the basic block with identifier *to.*

getState(checkID): Returns the current state of the state machine corresponding to the check, and is invoked just before the execution of the check in the program.

resetState(checkID): Resets the state-machine for the check given by *checkID.* This operation is invoked after the execution of the check in the program.

Module Components: The structure of the path-tracking module is shown in Figure 42.

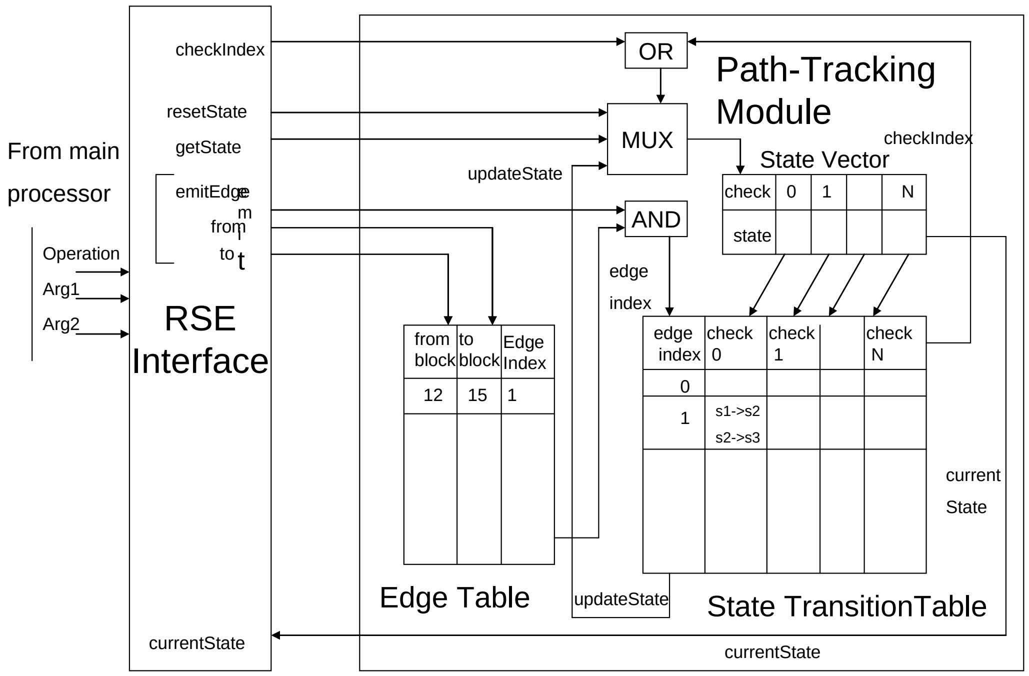


Figure 42: Hardware path-tracking module

The components of the path-tracking module are as follows:

1) *Edge Table*: Stores the mapping from control-flow edges to edge-identifiers for instrumented edges in the program. Each instrumented control-flow edge is assigned a unique index and is mapped to the identifiers assigned to the source and sink basic blocks for that edge (by the VRP).

2) *State Vector:* Holds the current state of the state machine corresponding to the detectors, with one entry for each detector inserted in the program.

3) *State Transition Table:* Contains the transitions corresponding to the state machines. The rows of the state transition table correspond to the edge indices, while the columns correspond to the checks. The cells of the table contain the transitions that are fired for each check when an instrumented branch is executed.

RSE Interface: This **c**onverts the CHECK instructions from the main processor into signals specific to the path-tracking module. This is done by tapping the Fetch_out signal from the main pipeline. The *Fetch_out* is one of the signals provided by the RSE framework [1]. Similarly, it converts signals from the path-tracking module into flags in the main processor. These are represented as special-purpose registers in the main processor

Module Operation: The operation of the path-tracking module for each of the primitive operations (executed in the main processor) is considered below:

*CHECK instruction with emitEdge operation is executed in the main processor***:**

- RSE interface asserts the *emitEdge* signal and sends the basic block identifiers that constitute the edge in the *from* and *to* lines.
- The *from* and *to* identifiers are looked up in the *edge table* and the edge index corresponding to the edge is sent to the *state transition table*.
- The row corresponding to the edge is looked up in the *state transition table*.
- For each non-empty table-entry in the column corresponding to the checks, the states in the LHS of the transitions stored in the table entry are compared to the current state of the check in the *state vector*.
- If the states match, then the transition is fired and the state vector entry corresponding to the check is updated with the state in the RHS of the transition that matched.

CHECK instruction with the getState operation is executed in the main processor:

- RSE interface asserts the *getState* signal and sends the identifier of the check on the *checkID* line to the path-tracking module.
- The path tracking module looks up the state in the *state vector* and sends it to the RSE interface through the *currentState* line. This in turn is sent to the main processor and is returned as the value of the CHECK instruction (through a special register in the RSE).

CHECK instruction with resetState operation is executed in the main processor: This is similar to the *getState* operation, but no value is returned to the RSE interface.

Function calls/returns: Since the technique tracks only intra-procedural paths, the *state vector* needs to be preserved across function calls and returns. This is done by pushing the *state vector* on a separate stack (different from the function call stack) along with the return address upon a function call and by popping the stack upon a return. The VRP generates code that uses special CHECK instructions to manipulate the stack on function calls/returns.

4.8.1 Area Overheads

The area overheads for the hardware module are dominated by the three main components of the module presented in Section 0. The other components are mainly glue combinational logic and occupy negligible area.

Table 21 presents the formulas used in estimating the size of the dominant hardware components. The size of each of these components depends on (1) the number of control-flow edges corresponding to state transitions (*m)*, (2) the number of checks that must be tracked for the application *(n)* and, (3) the maximum number of transitions in each entry of the transition table because the table must be big enough to hold the biggest entry (*k)*.

Table 21: Formulas for estimating hardware overheads

Hardware Component	**Size (bits)**	**Explanation**
Edge Table	m * 16 * 3	Each entry has 3 fields *from*, *to* and *edgeIndex*. Each fiels consists of 16 bits.
State Vector	n * 8	Each entry of the state vector consists of 8 bits, number of bits used for states
Transition Table	n * m * k * 16	Each state transition has two 8-bit fields to encode the starting and ending states

Table 22 presents the values of *m, n* and *k* for each application as well as the number of bits occupied by each structure. The sizes of the hardware structures (in bits) are calculated based on Table 21.

Table 22: Sizes of hardware structures (in bits)

App Name	**m**	**n**	**k**	**Size (bits)**
IntMM	10	21	3	1278
RealMM	10	21	3	1278
FFT	17	30	4	3096
Quicksort	19	29	5	3899
Bubblesort	5	11	1	383
Treesort	10	20	4	1440
Perm	16	27	1	1416
Queens	5	20	1	500
Towers	11	31	1	1117
Health	9	52	1	1316
Em3d	8	30	3	1344
Mst	17	33	10	6690
Barnes-Hut	43	118	6	33452
Tsp	9	48	2	1680
Average	14	37	4	4928

The average number of bits stored by the hardware module is estimated to be 4928. This corresponds to less than 1 KB of storage space in the hardware. The application exhibiting the worst-case overhead (Barnes-hut) occupies 33452 bits, corresponding to less than 4KB of memory. This easily fits into a standard FPGA BRAM cell which has 5096KB of memory.

4.8.2 Performance Overheads

The path-tracking module needs to be synchronized with the main processor only at the *getState* operation, and can execute asynchronously the rest of the time. Note that in our implementation of the path-tracking module the *getState* operation is simply a lookup in the state vector and takes constant time. Hence*, the getState operation takes constant time.* The *emitEdge, enterFunc* and *leaveFunc* operations can be buffered by the path-tracking module, while the application continues to execute on the main processor. These operations can then be performed asynchronously by the path-tracking module. We found that a buffer size of 1 sufficed to store the operations from the main processor.

4.9 CONCLUSION

This chapter presented a technique to derive error detectors for protecting an application from data errors. The error detectors were derived automatically using compiler-based static analysis from the backward program slice of critical variables in the program. The slice is optimized aggressively based on specific control-paths in the application, to form a checking expression. At runtime, the control path executed by the progrm is tracked using specialized hardware and the corresponding checking expressions are executed.

CHAPTER 5 FORMAL VERIFICATION OF ERROR DETECTORS

5.1 INTRODUCTION

Error detection mechanisms are vital for building highly reliable systems. However, generic detection mechanisms such as exception handlers can take millions of processor cycles to detect errors in programs [14]. In the intervening time, the program can execute with the activated error and perform harmful actions such as writing incorrect state to the file-system. There has been significant work on efficiently placing [17, 40] and deriving [56, 109, 110] error detectors for programs. An important challenge is to enumerate the set of errors the mechanism fails to detect, either from a known set or an unknown set. Typically, verification techniques target the defined set of errors the detector is supposed to detect. However, one cannot predict the kinds of errors that may occur in the field, and hence it is important to evaluate detectors under arbitrary conditions.

Fault-injection is a well-established to evaluate the coverage of error detection mechanisms [19]. However, there is a compelling need to develop a formal framework to reason about the efficiency of error detectors as a complement to traditional fault injection. This chapter shows how this can uncover possible “corner cases” which may be missed by conventional fault injection due to its inherent statistical nature. While there have been formal frameworks, each addresses a specific error detection mechanism (for example replication [111]), and cannot be easily extended to general detection mechanisms.

This chapter presents SymPLFIED, a framework for verifying error detectors in programs using symbolic execution and model-checking. The goal of the framework is to expose error cases that would potentially escape detection and cause program failure. The focus is on transient hardware errors. The framework makes the following unique contributions:

- Introduces a formal model to represent programs expressed in a generic assembly language, and reasons about the effects of errors originating in hardware and propagating to the application without assuming specific detection mechanisms,
- Specifies the semantics of general error detectors using the same formalism, which allows verification of their detection capabilities,
- Represents errors using a single symbol, thereby coalescing multiple error values into a single symbolic value in the program. This includes both single- and multi-bit errors in the register file, main memory, cache, as well as errors in computation.

To the best of our knowledge, this is the first framework that models the effect of arbitrary hardware errors on software, independent of the underlying detection mechanism. It uses model checking [112] to exhaustively enumerate the consequences of the symbolic errors on the program. The analysis is completely automated and does not miss errors that might occur in a real execution. However as a result of symbolically abstracting erroneous values, it may discover errors that may not manifest in the real execution of the program i.e. false-positives.

Previous work [113] has analyzed the effect of hardware errors on programs expressed in a high-level language (e.g. Java). Errors are modeled as bit flips in single data variable(s) in the program. While this is an important step, there are several limitations, namely (1) low-level hardware errors can affect multiple program variables as well as impact the program's control-flow, (2) errors in special-purpose registers such as the *stack pointer* are difficult to model in the high-level language, (3) Errors in the language runtime system (and libraries) cannot be modeled as they may be written in a different language.

This chapter considers programs represented at the assembly language level. The value of using assembly language is that any low-level hardware error that impacts the program can be represented at the assembly language level (as shown in section 5.3.3). Further, the entire application, including runtime libraries is amenable to analysis at the assembly language level.

It can be argued that in order to really analyze the impact of hardware errors, we need to model systems at even lower levels, *e.g* the register-transfer level (RTL). However, the consequent state space explosion when analyzing the entire program at such low levels can impact the practicality of the model. *An assembly language representation is a judicious tradeoff between the size of the model and the representativeness of hardware errors that can be considered in the model.*

In order to evaluate the framework, the effects of hardware transient errors are considered on a commercially deployed application, *tcas*. The framework identified errors that lead to a catastrophic outcome in the application, while a random fault injection experiment did not find any catastrophic scenario in a comparable amount of time. The framework is also demonstrated on a larger program, *replace*, to find instances of incorrect program outcomes due to hardware transient errors.

5.2 RELATED WORK

Prior literature related to this work is classified into the following categories:

Error Detection: Many error detection mechanisms have been proposed in the literature, along with formal proofs of their correctness [114, 115]. However, the verification methodology is usually tightly coupled with the mechanism under study. For example, Nicolescu et al. [116] proposes and verifies a control-flow checking technique by constructing a hypothetical program augmented with the technique and model-checks the program for missed detections. The program is carefully constructed to exercise all possible cases of the control-flow checking technique. It is non-trivial to construct such programs for other error-detection mechanisms.

Perry et al. [111] proposes the use of type-checking to verify the fault-tolerance provided by a specific error-detection mechanism namely, compiler-based instruction duplication. The paper proposes a detailed machine model for executing programs. The faults in the fault model (Single-Event Upsets) are represented as transitions in this machine model. The advantage of the technique is that it allows reasoning about the effect of low-level hardware faults on the whole program, rather than on individual instructions or data. However, the detection mechanism (duplication) is tightly coupled with the machine model, due to inherent assumptions that limit error propagation in the program and may not hold in non-duplicated programs.

Further, the type-checking technique in [111] either accepts or rejects a program based on whether the program has been duplicated correctly, but does not consider the consequences of the error on the program. As a result, the program may be rejected by the technique even though the error is benign and has no effect on the program's output.

Symbolic execution has been used for a wide variety of software testing and maintenance purposes [117]. The main idea in these techniques is to execute the program with symbolic values rather than concrete values and to abstract the program state as symbolic expressions. An example of a commercially deployed symbolic execution technique to find bugs in programs is Prefix [52]. However, Prefix assumes that the hardware does not experience errors during program execution.

A symbolic approach for injecting faults into programs was introduced by Larrson and Hahnle [113]. The goals of this approach are similar to ours, namely to verify properties of fault-tolerance mechanisms in the presence of hardware errors. The technique reasons on programs written in Java and considers the effect of bit-flips in program variables. However, a hardware error can have wide-ranging consequences on the program, including changing its control-flow and affecting the runtime support mechanisms for the language (such as the program stack and libraries). These errors are not considered by the technique.

Further, the technique presented in [113] uses theorem-proving to verify the error-resilience of programs. Theorem-proving has the intrinsic advantage that it is naturally symbolic and can reason about the non-determinism introduced by errors. However, as it stands today, theorem proving requires considerable programmer intervention and expertise, and cannot be completely automated for many important classes of programs.

Program verification techniques have been used to prove that a program's code satisfies a programmer-supplied specification [118].The specification precisely outlines the expected result of the program given certain initial conditions. Typically, program verification techniques are geared towards finding software defects and assume that the hardware and the program environment are error-free. In other words, they prove that the program satisfies the specification *provided* the hardware platform on which the program is executed does not experience errors.

Further, program verification techniques operate on an abstract representation of the program (such as a state machine) extracted from the program code [72, 119].The abstractions are derived based on the specific property being checked and cannot be used for evaluating the program under arbitrary hardware errors as such errors may not manifest in the abstraction.

Formal techniques have also been extensively applied to **microprocessor verification** [120]**.** The techniques attempt to prove that the implementation of the processor conforms to an architectural specification usually in the form of a processor reference manual.

Processor verification techniques focus on unmasking hardware design defects, as opposed to transient errors due to electrical disturbances or radiation.

Soft-errors in Hardware: Krautz et al. [121] and Seshia et al. [122] consider the effects of hardware transient errors (soft errors) on error-detection mechanisms implemented in hardware. While these techniques are useful for applications implemented as hardware circuits, it is not clear how the technique can be extended for reasoning about the effects of errors on programs. This is because programs are normally executed on general-purpose processors in which the manifestation of a low-level error is different from an error in an ASIC implementing the application.

Summary: The formal techniques considered in this section predominantly fall into the category of software-only techniques which do not consider hardware errors [118], or into the category of hardware-only techniques which do not consider the effects of errors on software [120]. Further, existing verification techniques are often coupled with the detection mechanism (e.g. duplication) being verified [111, 116].

Therefore, there exists no *generic* technique that allows reasoning about the effects of *arbitrary* hardware faults on software, and can be combined with an arbitrary fault model and detection technique(s). This is important for enumerating all hardware transient errors that would escape detection and cause programs to fail. Moreover, the technique must be *automated* in order to ensure wide adoption, and should not require programmer intervention.

This chapter attempts to answer the question: "*Is it possible to develop a framework to reason about the effects of arbitrary hardware errors on applications in an automated fashion, in order to understand where error detection mechanisms fail in detecting errors?*

5.3 APPROACH

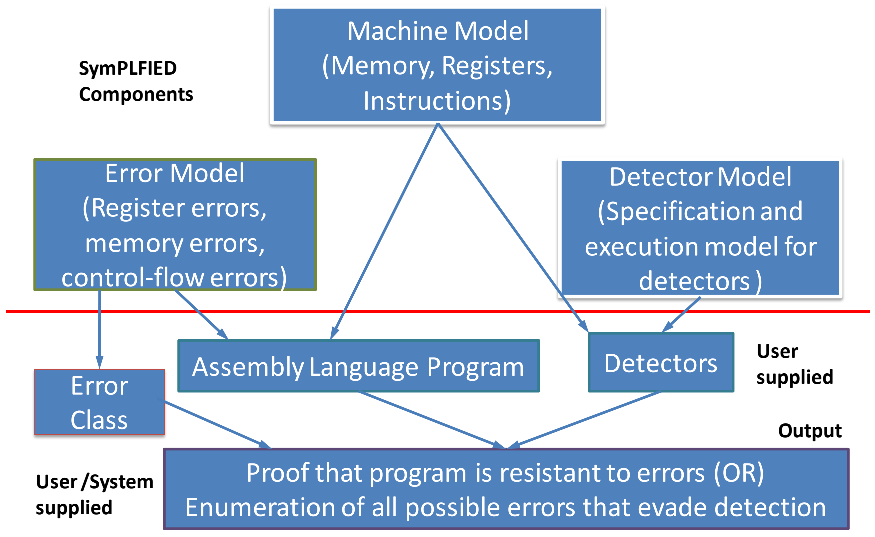
This section, introduces the conceptual model of the SymPLFIED framework and also the technique used by SymPLFIED to symbolically propagate errors in the program. The fault-model used by the technique is also discussed.

5.3.1 Framework

The SymPLFIED framework accepts a program protected with error detectors and enumerates all errors (in a particular class) that would not be detected by the detectors in the program. Figure 43 presents the conceptual design flow of the SymPLFIED framework.

Inputs: The inputs to the framework are (1) a program written in a target assembly language (e.g. MIPS), (2) error detectors embedded in the program code, and (3) a class of hardware errors to be considered (e.g. control-flow errors, register file errors).

Assembly Language: We define a generic assembly language in which programs are represented for formal analysis by the framework. Because the language defines a set of architectural abstractions found in many common RISC processor architectures, it is currentl portable across these architectures, with an architecture specific front-end. The assembly language has direct support for (1) Input/Output operations, so that programs can be analyzed independent of the Operating System (OS), and (2) Invocation of error detectors using special annotations, called CHECK, which allows detectors to be represented in line with the program.



SymPLFIED Components
Machine Model (Memory, Registers, Instructions)
Error Model (Register errors, memory errors, control-flow errors)
Detector Model (Specification and execution model for detectors)
Error Class
Assembly Language Program
Detectors
User supplied
Output
User /System supplied
Proof that program is resistant to errors (OR) Enumeration of all possible errors that evade detection

Figure 43: Conceptual design flow of SymPLFIED

Operation: The program behavior is abstracted using a generic assembly language described in Section 5.5. This is automatically translated into a formal mathematical model that can be represented in the Maude system [34]. Since the abstraction is close to the actual program in assembly language it is sufficient for the user to formulate generic specifications, such as an incorrect program outcome or an exception being thrown. Such a low-level abstraction of the program is useful to reason about hardware errors. The formal model can then be rigorously analyzed under error conditions against the above specifications using techniques such as model-checking and theorem-proving. *In this* *chapter, model-checking is used because it is completely automated and requires no programmer intervention*.

Outputs: The framework uses the technique described in section 5.3.2 and outputs either of the following:

1. Proof that the program with the embedded detectors is resilient to the error class considered, OR
2. A comprehensive set of all errors belonging to the error class that evade detection and potentially lead to program failure (crash, hang or incorrect output).

Components: The framework consists of the following formal models,

- **Machine Model:** Models the formal semantics of the machine on which the program is to be executed (e.g. registers, memory, instructions etc.).
- **Error Model:** Specifies error classes and error manifestations in the machine on which the program is executed e.g. errors in the class *register errors* can manifest in any register in the machine.
- **Detector Model:** Specifies the format of error detectors and their execution semantics. It also includes the action taken upon detecting the error e.g. halting the program.

By representing all three models in the same formal framework, we can reason about the effects of errors (in the error model) on both programs, represented in the machine model and on detectors, represented in the detector model, in a unified fashion.

Correctness: In order for the results of the formal analysis to be trustworthy, the model must be provably correct. There are two aspects to correctness, namely,

1. The model must satisfy certain desirable properties such as termination, coherence and sufficient completeness [34], AND
2. The model must be an accurate representation of the system being modeled.

The first requirement can be satisfied by formally analyzing the specification using automated checking tools for each desirable property listed above. This is obtained almost for free by expressing the model using Maude as formal checking tools are available to check the conformance of the model to the properties [123].

However, the second requirement is much harder to ensure as it cannot be checked by formal tools and is usually left to the model creator. We have attempted to validate the model by rigorously analyzing the behavior of errors in the model and comparing them with the behavior of the real system (Section **5.6.3**).

5.3.2 Symbolic Fault Propagation

The SymPLFIED approach represents the state of all erroneous values in the program using the abstract symbol *err*. The *err* symbol is propagated to different locations in the program during execution using simple error propagation rules (shown in section 5.5.2). The symbol also introduces non-determinism in the program when used in the context of comparison and branch instructions or as a pointer operand in memory operations. Because the same symbol is used to represent all erroneous values in the program, the approach distinguishes program states based on where errors occur rather than on the nature of the individual error(s). As a result, it avoids state explosion and can keep track of all possible places in the program the error may propagate to starting from its origin.

However, because errors in data values are not distinguished from each other, the set of error states corresponding to a fault is over-approximated. This can result in the technique finding erroneous program outcomes that may not occur in a real execution. For example, if an error propagates from a program variable *A* to another variable *B*, the variable *B*'s value is constrained by the value of the variable *A*. In other words, given a concrete value of *A* after it has been affected by the error, the value of *B* can be uniquely determined due to the error propagating from *A* to *B*.

The SymPLFIED technique on the other hand, would assign a symbolic value of *err* to both variables, and would not capture the constraint on *B* due to the variable *A*. As a result, it would not be able to determine that the value in register *B* even when given the value in register *A*. This may result in the technique discovering spurious program outcomes. Such spurious outcomes are termed *false-positives*.

While SymPLFIED may uncover false-positives, it will never miss an outcome that may occur in the program due to the error (in a real execution). This is because SymPLFIED systematically explores the space of all possible manifestations of the error on the program. Hence, the technique is *sound*, meaning it finds all error manifestations, but is not always *accurate*, meaning that it may find false-positives.

Soundness is more important than accuracy from the point of view of designing detection mechanisms, as we can always augment the set of error detectors to conservatively protect against a few false-positives (due to the inaccuracies introduced).

While a small number of false-positives can be tolerated, it must be ensured that the technique does not find too many false-positives as the cost of developing detectors to protect against the false-positives can overwhelm the benefits provided by detection. The SymPLFIED technique uses a custom constraint solver to remove false-positives in the search-space. The constraint solver also considerably limits state space explosion and quickly prunes infeasible paths. More details may be found in Section 5.5.2.

5.3.3 Fault Model

The fault-model considered by SymPLFIED includes transient errors in memory/registers and computation.

- Errors in memory/registers are modeled by replacing the contents of the memory location or register by the symbol *err. No distinction is made between single- and multi-bit errors.*
- Errors in computation are modeled based on where they occur in the processor pipelin*e and* how they affect the architectural state as shown in Table 23.
- Errors in processor control-logic (such as in the register renaming unit) are not considered by the fault-model.

The reason it is possible to represent such a broad class of errors in the model is because the program is represented in assembly language, which makes the elements of its state explicit to the analysis framework.

Table 23: Computation error categories and how they are modeled by SymPLFIED

Fault origin	**Error Symptom**	**Conditions under which Modeled**	**Modeling procedure**	
Instruction Decoder	One of the fields of an instruction is corrupted	One valid instruction is converted to another valid instruction	Instructions writing to a destination (e.g., *add*) - change the output target	*err* in the original and new targets (register or memory)
			Instructions with no target (e.g., *nop*) – replace with instructions with targets (*e.g. add*)	*err* in the new wrong target (register or memory)
			Instructions with a single destination (e.g.*add*) – replace with instruction with no target (e.g. *nop*)	*err* in the original target location (register or memory)
Address or Data Bus	Data read from memory, cache or register file is corrupted	Single and multiple bit errors in the bus during instruction execution	Errors in register data bus	*err* in source register(s) of the current instruction
			Error in cache bus	*err* in target registers of *load* instructions to the location
			Error in memory bus	*err* in target register of *load* instructions to the location
Processor Functional Unit	Functional unit output is corrupted	Single and multiple bit errors in registers/memory	Functional Unit output to register or memory	*err* in register or memory file being written to by the current instruction
Instruction Fetch Mechanism	Errors in the fetch unit	Single or multiple bit errors in PC or instruction	Fetch from an erroneous location due to error in *PC*	*PC* is changed to an arbitrary but valid code location
			Error in instruction while fetching	Modeled as Decode Errors

5.3.4 Scalability

As in most model-checking approaches, the exhaustive search performed by SymPLFIED can be exponential in the number of instructions executed by the program in the worst case. In spite of this limitation, model-checking techniques have been successfully scaled to large code-bases such as operating system kernels and web-servers [72, 119]. These approaches consider only parts of the system that are relevant to the property being verified. The relevant code portions are typically extracted by static analysis. However, static analysis is not very useful for dealing with runtime errors that may occur in hardware.

However, the error detection mechanisms in the program can be used to optimize the state space exploration process. For example, if a certain code component protected with detectors is proved to be resilient to all errors of a particular class, then such errors can be ignored when considering the space of errors that can occur in the system as a whole. This lends itself to a hierarchical or compositional approach, where first the detection mechanisms deployed in small components are proved to protect that component from errors of a particular class, and then inter-component interactions are considered. This is an area of future investigation.

5.4 EXAMPLES

This section illustrates the SymPLFIED approach in the context of an application that calculates the factorial of a number shown in Figure 2. The program is represented in the generic assembly language presented in Section 5.3.1.

5.4.1 Error Injection

We illustrate our approach with an example of an injected error in the program shown in Figure 43. Assume that a fault occurs in register *$3* (which holds the value of the loop counter variable) in line 8 of the program after the loop counter is decremented (*subi $3 $3 1).* The effect of the fault is to replace the contents of the register *$3* with *err*. The loop back-edge is then executed and the loop condition is evaluated by (*setgt $5 $3 $4).* Since *$3* has the value *err* in it, it cannot be determined if the loop condition evaluates to true or false. Therefore, the execution is forked so that the loop condition evaluates to true in one case and to false in the other case. The *true* case exits immediately and prints the value stored in *\$2*. Since the error can occur in any loop iteration, the value printed can be any of the following: *1!, 2!, 3!, 4!, 5!*. All these outcomes are found by SymPLFIED.

```
          ori  $2  $0  #1      --- initial product p = 1
           read $1              --- read i from input
           mov $3, $1
          ori $4 $0 #1        --- for comparison  purposes
 loop:    setgt $5 $3 $4      --- start of loop
           beq  $5 0 exit      ---- loop condition : $3 > $4
           mult $2 $2 $3            ---- p = p * i
          subi $3 $3 #1             ---- i = i - 1
           beq  $0 #0 loop          --- loop backedge
exit:    prints "Factorial = "
             print $2
             halt
```

Figure 44: Program to compute factorial in MIPS-like assembly language

The *false* case continues executing the loop and the *err* value is propagated from register *\$3* to register *\$2* due to the multiplication operation (*mul \$2 \$2 \$3*). The program then executes the loop back-edge and evaluates the branch condition. Again, the condition cannot be resolved as register \$3 is still *err*. The execution is forked again and the process is repeated ad-infinitum. In practical terms, the loop is terminated after a certain number of instructions and the value *err* is printed, or the program times out[18] and is stopped.

Complexity: Note that in order for a physical fault-injection approach to discover the same set of outcomes for the program as SymPLFIED, it would need to inject into all possible values (in the integer range) into the loop counter variable. This can correspond to 2^k cases in the worst case, where *k* is the number of bits used to represent an integer. In

[18] We assume that a watchdog mechanism is present in the program to monitor for infinite loops and hangs.

contrast, SymPLFIED considers at-most *(n+1)* possible cases, in this example, where *n* is the number of iterations of the loop. This is because each fork of the execution at the loop condition results in the *true* case exiting the loop and the program. In the general case though, SymPLFIED may need to consider 2^n possible cases. However, by upper-bounding the number of instructions executed in the program, the growth in the search-space can be controlled.

False-positives: In the example, not all errors in the loop counter variables will cause the loop to terminate early. For example, an error in the higher-order bits of the loop counter variable in register $3 may still cause the loop condition ($3 > $4) to be *false*. However, SymPLFIED would conservatively assume that both the *true* and *false* cases are possible, as it does not distinguish between errors in different bit-positions of variables. Note that in practice, false-positives were not a major concern, as shown in section **5.6.2**.

5.4.2 Error Detection

We now discuss how SymPLFIED supports error-detection mechanisms in the program. Figure 45 shows the same program in Figure 44, augmented with error detectors. Recall that detectors are invoked through special CHECK annotations as explained in Section 5.3.1. The error detectors together with their supporting instructions (*mov* instruction in line 8) are shown in bold.

```
1         ori  $2  $0  #1                 --- initial product p = 1
2         read $1                          --- read i from input
3         mov $3, $1
4         ori $4 $0 #1                     --- for comparison purposes
 loop:  setgt $5 $3 $4                    --- start of loop
6           beq  $5 0 exit
7          check ($4 < $3)
8           mov $6, $2
9          mult $2 $2 $3         ---- p = p * i
10           check ($2 >= $6 * $1)
11          subi $3 $3 #1         ---- i = i - 1
12           beq  $0 #0 loop       --- loop backedge
exit:    prints "Factorial = "
14           print $2
15           halt
```

Figure 45: Factorial program with error detectors inserted

The same error is injected as before in register *$3 (*the new line number is 11). As shown in Section 5.4.1, the loop back-edge is executed and the execution is forked at the loop condition *($3 > $4)*.

The *true* case exits immediately, while the *false* case continues executing the loop. The *false* case "remembers" that the loop condition *($3 < $4)* is false by adding this as a constraint to the search. The *false* case then encounters the first detector that checks if *($4 < $3)*. The check always evaluates to *true* because of the constraint and hence does not detect the error.

The program continues execution and the error propagates to *$2* in the *mul* instruction. However, the value of *$2* from the previous iteration does not have an error in it, and this value is copied to register *$6* by the *mov* instruction in line 8. Therefore, when the second detector is encountered within the loop (line 10), the LHS of the check evaluates to *err* and the RHS evaluates to *($6 * $1)*, which is an integer.

The execution is forked once again at the second detector into *true* and *false* cases. The *true* case continues execution and propagates the error in the program as before. The *false* case of the check throws an exception and the detector fails, thereby detecting the error.

The constraints for the *false* case, namely, *($6 * $3 >= $6 * $1)* are also remembered. Based on this constraint, as well as the earlier constraint *($3 > $4), the constraint-solver deduces that the second detector will detect the error if and only if the fault in register $3 causes it to have a value greater than the initial value read from the input (stored in register $1).*

The programmer can then formulate a detector to handle the case when the error causes the value of register $3 to be lesser than the original value in register $1. Therefore, the errors that evade detection are made explicit to the programmer (or to an automated mechanism) who can make an informed decision about handling the errors.

The error considered above is only one of many possible errors that may occur in the program. These errors are too numerous for manual inspection and analysis as done in this example. Moreover, not all these errors evade detection in the program and lead to program failure.

The main advantage of SymPLFIED is that it can quickly isolate the errors that would evade detection and cause program failure from the set of all possible transient errors that can occur in the program. It can also show an execution trace of how the error evaded detection and led to the failure. This is important in order to understand the weaknesses in existing detection mechanisms and improve them.

5.5 IMPLEMENTATION

We have implemented the SymPLFIED framework using the Maude rewriting logic system.

Rewriting logic is a general-purpose logical framework for specification of programming languages and systems [124].

Maude is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications [34]. *The main advantage of Maude is that it allows a wide variety of formal analysis techniques to be applied on the same specification*.

Supporting Tools: In order to make programs for existing architectures compatible with SymPLFIED, we provide a facility (through means of a *Perl* script) to translate programs written directly in the target architecture's assembly language into SymPLFIED's assembly language. In theory, any architecture can be supported but for now we support only the MIPS instruction set. We also built a query generator to explore the behavior of the program under common hardware error categories. Note that while the SymPLFIED framework can support arbitrary error classes, pre-defined error categories allow programmers to verify the resilience of their programs without having to write complex specifications (or any specifications).

In this section, we describe the details of the machine, detector and error models and show how the resilience of programs to hardware errors can be verified through exhaustive search i.e. bounded model-checking.

5.5.1 Machine Model

This section describes the machine model for executing assembly language programs using Maude.

Equations and Rules: As far as possible, we have used equations instead of rewrite rules for specifying the models. The main advantage of using equations is that Maude performs rewriting using equations much faster than using rewrite rules. However, equations must be deterministic and cannot accommodate ambiguity. The machine model is completely deterministic because for a given instruction sequence, the final state can be uniquely determined in the absence of errors. Therefore the machine model can be represented entirely using equations. However, the error model is non-deterministic and hence requires rewrite rules.

Assumptions: The following assumptions are made by the machine model when executing a program.

1. An attempt to fetch an instruction from an invalid code address results in an "illegal instruction" exception being thrown. The set of valid addresses is defined at program load time by the loader.
2. Memory locations are defined when they are first written to (by store instructions). An attempt to read from undefined memory location results in an "illegal address" exception being thrown. It is assumed that the program loader initializes all locations prior to their first use in the program.
3. Program instructions are assumed to be immutable and hence cannot be overwritten during execution.
4. Arithmetic operations are supported only on integers and not on floating point numbers.

Machine State: The central abstraction used in the machine model is the notion of *machine state,* which consists of the mutable components of the processor's structures. The machine state is carried from instruction to instruction in program execution order, with each instruction optionally looking up and/or updating the state's contents. The machine state is obtained by concatenating one or more of the machine elements in a single 'soup' of entities. For example, the soup, *PC(pc) regs(R) mem(M) input(In) output(out)* represents a machine state in which the (1) current program counter is denoted by *pc,* (2) register file is denoted by *R,* (3) memory is denoted by *M* and (4) input and output streams are *in* and *out* respectively.

Execute Sub-Model: We consider example instructions from each instruction class and illustrate the equations used to model them. These equations are defined in the *execute* sub-model and use primitives defined in other sub-models (e.g. the *fetch* primitive).

1. **Arithmetic Instruction:** Consider the execution of the addi instruction, which adds the value[19] v to the register given by rs and stores the results in register rd. In the equation given below, the <_,_> operator represents the machine state obtained by executing an instruction (given by the first argument) on a machine state (given by the second argument). C represents the code of the program and is written outside the state to enable faster rewriting by Maude (as it is assumed to be immutable). The {_,_} groups together the code and the machine-state. The elements of the machine

[19] The term *value* is used to refer to both integers and the *err* symbol

state in the equations are composable, and hence can be matched with a generic symbol *S* representing the “rest of the state”. This allows new machine-state elements can be added without modifying existing equations.

```
eq { C,  < addi rd rs v , PC(pc) regs(R) S > } = { C, < fetch( C, pc ), PC(next(pc)) regs(R[rd] <- R[rs] + v) S > } .
```

2. **Branch Instructions:** Consider the example of the *beq rs, v, l* instruction, which branches to the code label *l* if and only if the register *rs* contains the constant value *v*. The equation for *beq* is similar to the equation for the *addi* operation except that it uses the in-built if-then-else operator of Maude. Note the use of the *isEqual* primitive rather than a direct == to compare the values of the register *rs* and the constant value *v*. This is because the register *rs* may contain the symbolic constant *err* and hence needs to be resolved accordingly (by the error model).

```
eq { C , < beq rs v l , pc(PC) regs(R) S > =    if isEqual(R[rs], v)   then { C, < fetch(C, pc), PC(next(pc)) regs(R) S >}  else { C, <
fetch(C, l), PC(l) regs(R) S >} fi .
```

3. **Load/Store Instructions:** Consider the example of the *ldi rt, rs, a* which loads the value in the memory location at the address given by adding the offset *a* to the value in the register *rs.* However, the load address needs to be checked for validity before loading the value. This is done by the *isValid* primitive (defined in the *Memory Submodel).*

```
eq { C , < ldi rt rs a , PC(pc) regs(R) mem(M) S > =  if ( isValid(R[rd] + a, M)  )  then { C, < fetch(C, pc), C(next(pc)) mem(M)
regs(R[rt] <- M[a + R[rs]]) > } else { C, < throw “Illegal addr”, PC(next(pc)) mem(M) regs(R) > } fi .
```

4. **Input/Output Operations: I**nput and output operations are supported natively on the machine since the operating system is not modeled. An example is the print instruction whose equation is as follows:

```
eq { C, < print rs, PC(pc) regs(R) output(O) S > } = { C, < fetch(C, pc), PC(next(pc)) regs(R) output(O << R[rd]) S > } .
```

5. **Special Instructions:** These instructions are responsible for starting and stopping the program. e.g. *halt* and *throw* instructions to terminate the program. The halt instruction transforms the super-state prior to their execution into a machine state in order to facilitate the search for final solutions by the model-checker (section 5.5.4). Its equation is given by:

```
eq { C, < halt , PC(pc) S > } = PC(done) S .
```

5.5.2 Error Model

The overall approach to error injection and propagation was discussed in Section 5.3.2, but in this section we discuss the implementation of the approach using rewriting logic in Maude. The implementation of the error model is divided into five sub-models as follows:

Error Injection Sub-Model: The error-injection sub-model is responsible for introducing symbolic errors into the program during its execution. The injector can be used to inject the *err* symbol into registers, memory locations or the program counter when the program reaches a specific location in the code. This is implemented by adding a breakpoint mechanism to the machine model described in Section 5.5.1 The choice of which register or memory location to inject into is made non-deterministically by the injection sub-model using rewrite rules.

Error Propagation Sub-Model: Once an error has been injected, it is allowed to propagate through the equations for executing the program in the machine model. The rules for error propagation are also described by equations as shown below. In the equations that follow, *I* represents an integer.

eq err + err = err .   eq err + I = err .   eq I + err = err .

eq err – err = err .   eq err – I = err .   eq I – err = err .

*eq err * I = if (I==0) then 0 else err fi .*

*eq I * err = if (I==0) then 0 else err fi .*

eq err / I = if (I==0) then throw "div--zero" else err fi .

eq I / err = if isEqual(err, 0) then throw "div- zero" else err fi

*eq err * err = if isEqual(err, 0) then 0 else err fi .*

eq err / err = if isEqual(err, 0) then throw "div-zero" else err fi

In other words, any arithmetic operation involving the *err* value also evaluates to *err (*unless it is multiplied by 0). Note also how the divide-by-zero case is handled.

Comparison Handling Sub-Model: The rules for comparisons involving one or more *err* values are expressed as rewrite-rules as they are non-deterministic in nature. For example, the rewrite rules for the *isEqual* operator used in section 5.5.1 are as follows:

rl isEqual(I, err) => true . rl isEqual(I, err) => false .

rl isEqual(err, err) => true . rl isEqual(err, err) => false .

The comparison operators involving err operands evaluate to either true or false non-deterministically. This is equivalent to forking the program's execution into the true and false cases. However, once the execution has been forked, the outcome of the comparison is deterministic and subsequent comparisons involving the same unmodified locations must return the same outcome (otherwise false-positives will result). This can be accomplished by updating the state (after forking the execution) with the results of the comparison. In the *true* case of the *isEqual* primitive, the location being compared can be updated with the value it is being compared to. However, the *false* case is not as simple, as it needs to "remember" that the location involved in the comparison is not equal to the value it is being compared with. The same issue arises in the case of non-equality comparisons, such as *isGreaterThan*, *isLesserThan, isNotGreaterThan* and *isNotLesserThan.*

The *constraint tracking and solving* sub-model remembers these constraints and determines if a set of constraints is satisfiable, and if not, truncates the state-space exploration for the case corresponding to the constraint. This helps avoid reporting false-positives.

Constraint Tracking and Solving Sub-Model: A new structure called the *ConstraintMap* is added to the machine state in Section 5.5.1. The *ConstraintMap* structure maps each register or memory location containing *err* to a set of constraints that are satisfied by the value in the location. An example of a set of constraints for a location is the following: *notGreaterThan(5) notEqualTo(2) greaterThan(0).* This indicates that the location can take any integer value between 0 and 5 excluding 0 and 2 but including

5. The constraints for a location are updated whenever a comparison is made based on the location if and only if it contains the value *err*. Constraints are also updated by arithmetic and logic operations in the program.

For a given location, it may not be possible to find a value that satisfies all its constraints simultaneously. Such constraints are deemed un-satisfiable and the model-checker can terminate the search when it comes to a state with an un-satisfiable set of constraints (such a state represents a false-positive). The constraint solver determines whether a set of constraints is un-satisfiable and eliminates redundancies in the constraint-set.

Memory- and Control Handling Sub-Model: Memory and Control errors are also handled non-deterministically using rewrite rules as follows:

Errors in jump or branch targets: The program either jumps to an arbitrary (but valid) code location or throws an “illegal instruction” exception.

Errors in pointer values of loads: The program either retrieves the contents of an arbitrary memory location or throws an “illegal-address” exception.

Errors in pointer values of stores: The program either overwrites the contents of an arbitrary memory location, or creates a new value in memory.

5.5.3 Detector Model

Error detectors are defined as executable checks in the program that test whether a given memory location or register satisfies an arithmetic or logical expression. For example, a detector can check if the value of register *$(5)* equals the sum of the values in the register

$(3) and memory location *(1000)* at a given program counter location. If the values do not match, an exception is thrown and the program is halted.

In our implementation, each detector is assigned a unique identifier and the CHECK instructions encode the identifier of the detector they want to invoke in their operand fields. The detectors themselves are written outside the program, and the same detector can be invoked at multiple places within the program's code.

We assume that the execution of a detector does not fail i.e. the detectors themselves are free of errors.

A detector is written in the following format:

det (ID, Register Name or Memory Location to Check, Comparison Operation, Arithmetic Expression)

1. The arguments of the detector are as follows:
2. The first argument of the detector is its identifier.
3. The second argument is the register or memory location checked by the detector.
4. The third argument is the comparison operation, which can be any of ==, =/=, >, <, <= or >=.
5. The final argument is the arithmetic expression that is used to check the detector's register or memory location and is expressed in the following format:

Expr :: = Expr + Expr | Expr – Expr | Expr * Expr | Expr / Expr | (c) | (Reg Name) | *(memory address)

Using the above notation, the detector introduced earlier would be written as:

*det(4, $(5), == , ($3) + *(1000)).*

The equations for the detector's execution are independent of the equations in the machine model, and hence are not affected by errors introduced in the machine other than those that are present in the registers or memory locations used in the detector's expression. Execution of a detector also updates the constraints for the location being checked in the *ConstraintMap* structure described in section 5.5.2.

5.5.4 Model-checking

The exhaustive *search* feature of Maude is used to model-check programs [123]. The aim of the search command is to expose interesting "outcomes" of the program caused by errors in a particular category. The "outcome" is a user-defined function on the machine state described in Section 5.5.1 and must be specified in the *search* command. For example, the following search command obtains the set of executions of the program that will print a value of *err* under all single errors in registers (one per execution).

```
search regErrors( start(program, first, detectors) ) =>! (S:MachineState) such that ( output(S) contains err ) .
```

The *search* command systematically explores the search space in a breadth-first manner starting from the initial state and obtaining all final states that satisfy the user-defined predicate, which can be any formula in first-order logic. The programmer can query how specific final states were obtained or print out the search graph, which will contain the entire set of states that have been explored by the model checking. This can help the programmer understand how the injected error(s) lead to the outcome(s) printed by the search.

Termination: In the absence of errors, most programs can be modeled as finite-space systems provided (1) they terminate after a finite amount of time or (2) they perform repetitive actions without terminating but revisit states. However, errors can cause the state space to become infinitely large, as the program may loop infinitely due to the error, never revisiting earlier states. In practice, this is impossible, since the program data is physically represented as bits and there are only a finite number of bits available in a machine. However, the state space would be so large that it is practically impossible to explore in full.

In order to ensure that the model-checking terminates, the number of instructions that is allowed to be executed by the program must be bounded. This bound is referred to as the *timeout* and must be conservatively chosen to encompass the number of instructions executed by the program during all possible correct executions (in the absence of errors). After the specified number of instructions is exceeded, a “timed out” exception is thrown and the program is halted. We assume that the processor has a watchdog mechanism.

5.6 CASE STUDY

We have implemented SymPLFIED using Maude version 2.1. Our implementation consists of about 2000 lines of uncommented Maude code split into 35 modules. It has 54 rewrite rules and 384 equations.

This section reports our experience in using SymPLFIED on the *tcas* application [125], which is widely used as an advisory tool in air traffic control for ensuring minimum vertical separation between two aircrafts and hence avoid collisions. The application consists of about 140 lines of C code, which is compiled to 913 lines of MIPS assembly code, which in turn is translated to 800 lines of SymPLFIED's assembly code (by our custom translator). In the later part of this section, we describe how we apply SymPLFIED on the *replace* program of the Siemens program suite [51] to understand the effects of scaling to larger programs.

tcas takes as input a set of 12 parameters indicating the positions of the two aircrafts and prints a single number as its output. The output can be one of the following values: 0, 1 or 2, where 0 indicates that the condition is unresolved, 1 indicates an upward advisory and 2 indicates a downward advisory. Based on these advisories, the aircraft operator can choose to increase or decrease the aircraft's altitude.

5.6.1 Experiment Setup

Our goal is to find whether a transient error in the register file during the execution of *tcas* can lead to the program producing an incorrect output (in this case, an advisory). We chose an input for *tcas* in which the *upward advisory* (value of 1) would be produced under error-free execution.

We directed SymPLFIED to search for runs in which the program did not throw an exception and produced a value other than 1 under the assumption of a single register error in each execution. The search command is identical to the one shown in section 5.5.4.

This constitutes about (800 * 32) possible injections, since there are 32 registers in the machine, and each instruction in the program is chosen as a breakpoint. In order to reduce the search space, at each breakpoint, only the register(s) used by the instruction was injected. This ensures that the fault is activated in the program.

In order to ensure quick turn-around time for the injections, they were started on a cluster of 150 dual-processor AMD Opteron machines. The search command is split into multiple smaller searches, each of which sweeps a particular section of the program code looking for errors that satisfy the search conditions. The smaller searches can performed independently by each node in the cluster, and the results pooled together to find the overall set of errors. The maximum number of errors found by each search task was capped at 10, and a maximum time of 30 minutes was allotted for task completion (after which the task was killed).

In order to validate the results from SymPLFIED, we augmented the Simplescalar simulator [50] with the capability to inject errors into the source and destination registers of all instructions, one at a time. For each register we injected three extreme values in the integer range as well as three random values, so that a representative sample of the errors in each value can be considered.

5.6.2 SymPLFIED Results

For the *tcas* application, we found only one case where an output of 1 is converted to an output of 2 by the fault injections. This can potentially be catastrophic as it is hard to distinguish from the correct outcome of *tcas*. None of the other injections found any other such case. We also found cases where (1) *tcas* printed an output of 0 (unresolved) in place of 1, (2) the output was outside the range of the allowed values printed by *tcas* and

(3) numerous cases where the program crashed. We do not report these cases as *tcas* is only an advisory tool and the operator can ignore the advisory if he or she determines that the output produced by *tcas* is incorrect.

We also found violations in which the value is computed correctly but printed incorrectly. We do not consider these cases as the output method may be different in the commercial implementation of *tcas*.

Running Time: Of the 150 search tasks started on the cluster, 85 tasks completed within the allotted time of 30 minutes. The remaining 65 tasks did not complete in the allotted time (as the timeout chosen was too large). We report results only from the tasks that completed. Of the 85 tasks that completed, 70 tasks did not find any errors that satisfy the conditions in the search command (as either the error was benign or the program crashed due to the error). These 70 tasks completed within 1 minute overall.

The time taken by the 15 completed tasks that found errors satisfying the search condition, (including the catastrophic outcome) is less than 4 minutes, and the average time for task completion is 64 seconds. Even without considering the incomplete tasks we were able to find the catastrophic outcome for *tcas*, shown below.

Initially, we were surprised by the unusually low number of catastrophic failures reported in *tcas*. However, closer inspection revealed that the code has been extremely well-engineered to prevent precisely these kinds of error from resulting in catastrophic failures. The *tcas* application has been extensively verified and checked for safety violations by multiple studies [126-128]. Nevertheless, the fact that SymPLFIED found this failure at all is testimony to its comprehensive evaluation capabilities. Further, this failure was not exposed by the injections performed using Simplescalar. In order to understand better the error that lead to *tcas* printing the incorrect value of 2, we show an excerpt from the *tcas* code in Figure 46.

```
int alt_sep_test()
{
   enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);
   tcas_equipped = Other_Capability == TCAS_TA;
   intent_not_known = Two_of_Three_Reports_Valid &&          (Other_RAC == NO_INTENT);
   alt_sep = UNRESOLVED;
   if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped)) {
            need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
            need_downward_RA = Non_Crossing_Biased_Descend() && Own_Above_Threat();
            if (need_upward_RA && need_downward_RA)
               alt_sep = UNRESOLVED;
            else if (need_upward_RA)
               alt_sep = UPWARD_RA;
            else if (need_downward_RA)
               alt_sep = DOWNWARD_RA;
            else
               alt_sep = UNRESOLVED;
   }
      return alt_sep;
}
```

Figure 46: Portion of *tcas* code corresponding to error

Optimizations: In order to reduce the number of states explored by the model-checker, we inject errors only into the registers used in each instruction of the program. Further, we inject the error just before the instruction that uses the register, in order to ensure fault activation. The effect of the injection is equivalent to injecting the register at an arbitrary code location so that the error is activated at the instruction.

The code shown in Figure 46 corresponds to the function *alt_sep_test*, which tests the minimum vertical separation between two aircrafts and returns an advisory. This function in turn calls the function *Non_Crossing_Biased_Climb()* and the *Own_Above_Threat()* function to decide if an upward advisory is needed for the aircraft. It then checks if a downward advisory is needed by calling the function *Non_Crossing_Biased_Descend()*

and the function *Own_Below_Threat()*. If neither or both advisories are needed, it returns the value *0* (unresolved). Otherwise, it returns the advisory computed in the function.

The error under consideration occurs in the body of the called function *Non_Crossing_Biased_Climb()* and corrupts the value of register *$31*which holds the function return address. Therefore, instead of control being transferred to the instruction following the call to the function *Non_Crossing_Biased_Climb()* in *alt_sep_test()*, the control gets transferred to the statement *alt_sep* = *DOWNWARD_RA* in the function. This causes the function to return the value *2* instead of the value *1*, which is printed by the program. We have verified that the error exposed above corresponds to a real error and is not a false-positive by injecting these faults into the augmented Simplescalar simulator.

Note that the above error occurs in the stack, which is part of the runtime support added by the compiler. Hence, in order to discover this error, we need a technique like SymPFLIED that can reason at the assembly language (or lower) level.

5.6.3 SimpleScalar Results

We performed over 6000 fault-injection runs on the *tcas* application using the modified Simplescalar simulator to see if we can find the catastrophic outcome outlined above. We ensured that both SymPLFIED and Simple-scalar were run for the same time to find these outcomes. The SymPLFIED injections were run with 150 tasks, and each completed task took a maximum time of 4 minutes. This constitutes 10 hours in total. We were able to perform 6000 automated fault-injection experiments with Simplescalar in that time. The results are summarized in column 2 of Table 24.

Table 24: SimpleScalar fault-injection results

Program Outcome	Percentage	
	# faults = 6253	*# faults = 41082*
0	1.86% (117)	2.33% (960)
1	53.7% (3364)	56.33% (23143)
2	0% (0)	0% (0)
Other	0.5% (29)	1.0% (404)
Crash	43.4% (2718)	40.43% (16208)
Hang	0.4% (25)	0.8% (327)

Table 24 shows that even though we injected exhaustively into registers of all instructions in the program, Simplescalar was unable to uncover even a single scenario with the catastrophic outcome of '2', whereas the symbolic error injection performed by SymPLFIED was able to uncover these scenarios with relative ease. This is because in order to find an error scenario using random fault injections, not only must the error be injected at the right place in the program (for example, register *$31* in the *Non_Crossing_Biased_Climb* function), but also the right value must be chosen during the injection (for example, the address of the assignment statement must be chosen in the *alt_sep_test* function in Figure 46. Otherwise the program may crash due to the error or the error may be benign in the program.

We also extended the SimpleScalar based fault injection campaign to inject 41000 register faults to check if such an injection discovers errors causing the catatrophic outcome. The injection campaign completed in 35 hours but was still unable to find such an error. The results of this extended set of injections in shown in column 3 of Table 24.

5.6.4 Application to Larger Programs

In order to evaluate the effectiveness of the formal analysis as we scale to larger applications, we analyzed the *replace* program using SymPLFIED. *replace* is the largest of the Siemens benchmarks[51], used extensively in software testing. The *replace* program matches a given string pattern in the input string and replaces it with another given string. The code translates to about 1550 lines of assembly code spanning 22 functions. Table 25 lists some key functions.

Table 25: Important functions in *replace*

makepat	Constructs pattern to be matched from input reg exp
getccl	Called by makepat when scanning a '[' character
dodash	Called by getccl for any character ranges in pattern
amatch	Returns the position where pattern matched
locate	Called by amatch to find whether the pattern appears at a string index

Using the same experimental setup as described in Section 5.6.1, we ran SymPLFIED on the *replace* program to find all single register errors (one per execution) that lead to an incorrect outcome of the program. The overall search was decomposed into 312 search tasks.

Results: Of these 202 completed execution within the allotted time of 30 minutes. In 148 of the completed search tasks, either the error was benign or the program crashed due to the error, while 54 of the search tasks found error(s) leading to incorrect outcome. We consider the execution trace of an example error.

Example Scenario: An input parameter to the `dodash` function that holds the delimiter (']') for a character range was injected. An erroneous pattern is constructed, which leads to a failure in the pattern match. As a result, the program returns the original string without the substitution. The analysis completed in an average of 4 minutes where no erroneous solutions where found. For the injection runs that found an erroneous outcome the analysis took an average of 10 minutes.

5.7 CONCLUSION

This chapter presented SymPLFIED a modular, flexible framework for performing symbolic fault-injection and evaluating error-detectors in programs. We have implemented the SymPLFIED framework for a MIPS-like processor using the Maude rewriting logic engine. We demonstrate the SymPLFIED framework on a widely-deployed application *tcas,* and use it to find a non-trivial case of a hardware transient error that can lead to catastrophic consequences for the *tcas* system. We also demonstrate the technique on the *replace* program to illustrate its scalability.

CHAPTER 6 FORMAL VERIFICATION OF ATTACK DETECTORS

6.1 INTRODUCTION

Insider threats have gained prominence as an emerging and important class of security threats [129-131]. An insider is a person who is part of the organization and either steals secrets or subverts the working of the organization by exploiting hidden system flaws for malicious purposes. This chapter considers *application-level* insider attacks. For example, an insider may load a malicious plugin into a web browser that overwrites the address bar with the address of a phishing website. Or a disgruntled programmer may plant a logical flaw in a banking application that allows an external user to fraudulently withdraw money. Both are examples of how a trusted insider can compromise an application and subvert it for malicious purposes.

We define an application-level insider attack as one in which a malicious insider attempts to overwrite one or more data items in the application, in order to achieve a specific attack goal. The overwriting may be carried out by exploiting existing vulnerabilities in the application (e.g. buffer overflows), by introducing logical flaws in the application code or through malicious third-party libraries. It is also possible (though not required) to launch insider attacks from a malicious operating system or higher-privileged process. Application-level insider attacks are particularly insidious because, (1) by attacking the application an insider can evade detection by mimicking its normal behavior (from the point of view of the system), and (2) to attack the application, it is enough for the insider to have the same privilege as that of the application, whereas attacking the network or operating system may require super-user privileges.

Before defending against insider attacks, we need a model for reasoning about insiders. Previous work has modeled insider attacks at the network and operating system (OS) levels using higher-level formalisms such as attack graphs [132-134] and process calculi [135]. However, modeling application-level insider attacks requires analysis of the application's code as an insider has access to the application and can hence launch attacks on the application's implementation. Higher-level models are too coarse grained to enable reasoning about attacks that can be launched at the application code level. Further, higher-level models typically require application vulnerabilities (if present) to be identified up-front in order to reason about insider attacks on the system.

This chapter introduces a technique to formally model application-level insider attacks on the application code expressed in assembly language. The advantage of modeling at the assembly code-level is that the assembly code includes the program, its libraries, and any state added by the compiler (e.g. stack pointer, return addresses). Therefore, all *software-based* insider attacks on the application can be modeled at the assembly-code level.

The proposed technique uses a combination of symbolic execution and model checking to systematically enumerate *all* possible insider attacks in a given application corresponding to an attack goal. The technique can be automatically deployed on the application's code and no formal specifications need to be provided other than generic specifications about the attacker's end goal(s) (with regard to the application's state or final output).

The value of the analysis performed by the proposed technique is that it can expose non-intuitive cases of insider attacks that may be missed by manual code inspection. This is because the technique exhaustively considers corruptions of data items used in the application (under a given input), and enumerates all corruptions that lead to a successful attack (based on the specified attack goal). Thus, it is able to identify *all vulnerable data items* in the application corresponding to the attack goal. The results of the analysis can be used to guide the development of defense mechanisms (eg. assertions) to protect the application.

We have implemented the proposed technique as a tool, *SymPLAID*, which directly analyzes MIPS-based assembly code. The tool identifies for each attack, (1) The program point at which the attack must be launched, (2) The data item that must be overwritten by the attacker, and (3) The value that must be used for overwriting the data item in order to carry out the attack.

SymPLAID is built as an enhancement of our earlier tool, SymPLFIED [136], used to evaluate the effect of transient errors on the application. SymPLFIED also builds a formal model of the application at the assembly code level. However, SymPLFIED groups individual errors into a single abstract class (*err)*, and considers the effect of the entire class of errors on the program. This is because in the case of randomly occurring errors, we are more interested in the propagation of the error rather than the precise set of circumstances that caused the error.

In contrast, security attacks are launched by an intelligent adversary and hence it is important to know precisely what values are corrupted by the attacker (and how the corruption is carried out) in order to design efficient defense mechanisms against the attack(s). Therefore, SymPLAID was built from the ground up to emphasize precision in terms of identifying the specific conditions for an attack. Thus, rather than abstracting the attacker's behavior into a single class, the effect of each value corruption is considered individually, and its propagation tracked in the program. While this may appear to sacrifice scalability, the gains due to the extra precision in terms of evaluating fewer program forks outweigh the losses (see Section 6.4.4 for details).

The chapter makes the following key contributions:

- Introduces a formal model for reasoning about application-level insider attacks at the assembly-code level,
- Shows how application-level insiders may be able to subvert the execution of the application for malicious purposes,
- Describes a technique to automatically discover *all* possible insider attacks in an application using symbolic execution and model checking on the application code,
- Demonstrates the proposed techniques using a case-study drawn from the OpenSSH application[137], and finds all possible insider attacks, including several non-intuitive attacks that may be missed by simple, manual inspection.

6.2 INSIDER ATTACK MODEL

This section describes the attack model for insider attacks and an example scenario for an insider attack. The example scenario is considered in more detail in Section 6.3.

6.2.1 Characterization of Insider

Capabilities: The insider is a part of the application and has unfettered access to the program's address space. This includes the ability to both read and write the program's memory and registers. However, we assume that the insider cannot modify the program's code, which is reasonable since in most programs the code segment is marked read-only.

An attacker may get into the application and become an insider in the following ways:

- By a logical loophole in the application planted by a disgruntled or malicious programmer,
- Through a malicious (or buggy) third-party library loaded into the address space of the application,
- By exploiting known security loopholes such as buffer overflow attacks and planting the attack code,
- By overwriting the process's registers or memory from another process (with higher privilege) or debugger,
- Through a security vulnerability in the operating system or virtual machine (if present)

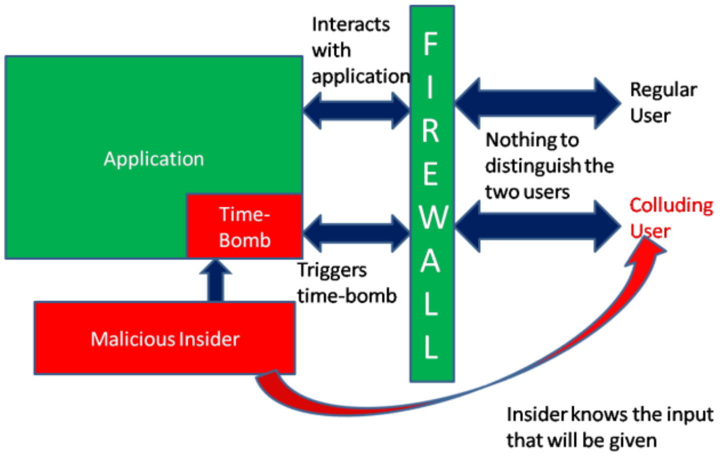
In each of the above scenarios, the insider can corrupt the values of either memory locations or registers while the application is executing. The first three scenarios only require the insider to have the same privileges as the applications, while the last two require higher privileges.

Goal: The attacker's goal is to subvert the application to perform malicious functions on behalf of the attacker. However, the attacker wants to elude detection or culpability (as far as possible), so the attacker's code may not directly carry out the attack, but may instead overwrite elements of the program's data or control in order to achieve the attacker's aims. From an external perspective, it will appear as though the attack originated due to an application malfunction, and hence the attack code will not be blamed. Full execution replay may be able to find the attack [138], but it incurs considerable time and resource overheads. Therefore, the attacker can execute code to overwrite crucial elements of the program's data or control elements.

It is assumed that the attacker does not want to crash the application, but wants to subvert its execution for some malicious purpose. The attack is typically launched only under a specific set of inputs to the program (known to the attacker), and the input sequence that launches the attack is indistinguishable from a legitimate input for the program. Even if the insider is unable to launch the attack by himself/herself, he/she may have a colluding user who supplies the required inputs to launch the attack. Note that the colluding user does not need to have the same privileges as the insider in order to launch the attack.

6.2.2 Attack Scenario

Figure 47 shows an example attack scenario where the insider has planted a “logic bomb” in the application which is triggered under a specific set of inputs. The bomb could have been planted by the insider through the first, second or third scenario considered in section 6.2.1. .Normal users are unlikely to accidentally supply the trigger sequence and will be able to use the application without any problems. However, a colluding user knows about the bomb and supplies the trigger sequence as input. Perimeter based protection techniques such as firewalls will not notice anything amiss as the trigger sequence is indistinguishable from a regular input for all practical purposes. However, the input will trigger the bomb in the application thereby launching the security attack on behalf of the insider.



Interacts with application
Application
Time-Bomb
Triggers time-bomb
Malicious Insider
FIREWALL
Regular User
Nothing to distinguish the two users
Colluding User
Insider knows the input that will be given

Figure 47: Attack scenario of an insider attack

6.2.3 Problem Definition

The problem of attack generation from the insider’s point of view may be summed up as follows: “If the input sequence to trigger the attack is known (AND) the attacker’s code is executed at specific points in the program, what data items in the program should be corrupted and in what way to achieve the attack goal?"

This chapter develops a technique to automatically discover conditions for insider attacks in an application given (i) the inputs to trigger the attack (e.g. a specific user-name as input), (ii) the attacker's objective stated in terms of the final state of the application (e.g. to allow a particular user to log in with the wrong password) and (iii) the attacker's capabilities in terms of the points from which the attack can be launched (e.g. within a specific function). The analysis identifies both the target data to be corrupted and what value it should be replaced with to achieve the attacker's goal.

To facilitate the analysis, the following assumptions are made about the attacker by the technique.

1. Only one value can be corrupted, but the corrupted value can be any valid value. This assumption ensures that the footprint of the attack is kept small and is hence easier to evade detection (from a defense technique)

2. The corruption is only allowed at fixed program points. This assumption reflects the fact that an insider may be able launch their attacks only at fixed program points.

We are working on relaxing these assumptions to consider attackers with higher capabilities or privileges. However, as we show in Section 6.3.1, even under these assumptions, a malicious insider can mount a significant number of attacks.

6.3 *EXAMPLE CODE AND ATTACKS*

This section considers an example code fragment to illustrate the attack scenario in Section 6.2.2. This is motivated by the OpenSSH program [137]. We consider the real OpenSSH application in Section 6.5. The example is also used in Section 6.5 to demonstrate the operation of the SymPLAID tool.

Figure 48 shows an example code fragment containing the *authenticate* function. The authenticate function copies the value of the system password into the *src* buffer and the value entered by the user into the *dest* buffer (in both cases it reads them into the *tmp* buffer first to validate the values). It then compares the values in the *src* and *dest* buffers and if they match, it returns the value 1 (authenticated). Otherwise it returns the value 0 (unauthenticated) to the calling function.

```
int authenticate(void* src, void* dest, void* temp, int len){
    1: readInput(temp);
    2: strncpy(src, temp, len)
    3: readInput(temp);
    4: strncpy(dest, temp, len);
    5: if (! strncmp(dest, src, len) ) return 1;
    return 0;
}
```

Figure 48: Code of authenticate function

6.3.1 Insider Attacks

We first take the attacker's perspective in coming up with insider attacks on the code in Figure 48. The attacker's goal is to allow a colluding user[20] to be validated even if he/she has entered the wrong password. The following assumptions are made in this example, for simplicity of explanation:

[20] The colluding user may be the same person as the attacker (who wants to evade detection), but we distinguish between these two roles in this chapter.

- The attack can be invoked only within the body of the *authenticate* function.
- The attacker can overwrite the value of any register or local variable, but not global variables and heap buffers (due to practical limitations such as not knowing the exact address of globals and dynamic memory).
- The attack points are at function calls within the *authenticate* function, i.e., the arguments to any of the functions called by the *authenticate* function may be overwritten prior to the function call.

Table 26 shows the set of all possible attacks the attacker could launch in the above function. The first column shows the program point at which the attack is launched, the second column shows the variable to overwrite and the third column shows the value that should be written to the variable. The fourth column explains the attack in more detail.

A particularly interesting attack found is presented in row 6 of Table 26, where the *dest* argument of the *strncpy* function was set to overlay the *src* string in memory. This replaces the first character of the *src* string with '\0', effectively converting it to a NULL string. The *dest* string also becomes NULL as the *dest* buffer is not filled by the *strncpy* function. As a result, the two strings will match when compared and the *authenticate* function will return '1' (authenticated).

As Table 1 shows, discovering all possible insider attacks manually (by inspection) is cumbersome and non-trivial even for the modestly sized piece of code that is considered in Figure 48. Therefore, we have developed a tool to generate the attacks automatically -

SymPLAID. Although the tool works on assembly language programs, we have shown the program as C-language code in Figure 48 for simplicity.

Table 26: Insider attacks on the authenticate function

Program Point	Variable to be corrupted	Corrupted value of variable	Comments/Explanation
strncmp point (line 5)	dest	src buf	The src buffer is compared with itself
	src	dest buf	The dest buffer is compared with itself
	src	tmp buf	The dest buffer is compared with the tmp buffer which contains the same string
	len	<= 0	The strncmp function terminates early and returns 0 (the strings are identical)
strncpy point (line 4)	temp	src buf	This copies the string in the source buffer to the destination buffer, thereby ensuring that the strings match
	dest	srcBuf – strlen(buf)	This writes a '\0' character in the src buffer, effectively converting it to a empty string. The dst buffer is also empty as it is not initialized, and hence the strings match.
readInput point (line 3)	temp	dest buf	The value in the tmp buffer is left unchanged, and is copied to the dst buffer. This value is also stored in the src buffer and hence the strings match.
	temp	Any unused location in memory	

We have validated the attacks shown in Table 26 using the GNU debugger (gdb) to corrupt the values of chosen variables in the application on an AMD machine running the Linux operating system. All the attacks shown in Table 26 were found to be successful i.e. they led to the user being authenticated in spite of providing the wrong password.

6.3.2 Defense Techniques

We now take the defender's perspective in designing protection mechanisms for insider attacks.

The attacks in Table 26 consist of both "obvious attacks" as well as surprising corner cases. It can be argued that finding obvious attacks is not very useful as they are likely to be revealed by manual inspection of the code. However, the power of the proposed technique is that it can reveal *all* such attacks on the code, whereas a human operator may miss one or more attacks. This is especially important from the developer's perspective, as *all* the security holes in the application need to be plugged before it can be claimed that the application is secure (as all the attacker needs to exploit is a single vulnerability). Moreover, the ability to discover corner-case attacks is the real benefit of using an automated approach.

Below we discuss some examples of detection mechanisms for the example presented in Figure 48. The mechanisms are designed based on the attacks discovered in Table 26.

- We insert a check before the call to the *strncmp* function. In particular, we check that the *src* and *dest* buffers of the *strncmp* function do not overlap with each other or with the *temp* buffer. We also check whether the length argument is greater than 0. This prevents attacks in rows 1 to 4 of Table 26. Note that the check is stronger than necessary.

- We insert a check after the call to the *readInput* function in line 3 to ensure that the *temp* buffer is non-empty. This prevents attacks in the rows 7 to 8 of Table 26.

- We insert a check before the call to the *strncpy* function to ensure that neither the *temp* buffer nor the *dest* buffer overlap with the *src* buffer.

If any of the above conditions is violated, the application is aborted and an attack is detected. The insider cannot corrupt the values in both the checks and the program as only one value in the application is allowed to be corrupted (as per our assumptions). Figure 49 shows the code in Figure 48 with the above checks inserted. The checks are represented as *assert* statements.

```
int authenticate(void* src, void* dest, void* temp, int len){
    1: readInput(temp);
    2: strncpy(src, temp, len)
    3: readInput(temp);
 assert( isNotEmpty(temp) );
 assert( noOverlap(temp, src) and noOverlap(temp, dest) )
    4: strncpy(dest, temp, len);
 assert( noOverlap(src, dest) and noOverlap(src, temp) );
 assert( len > 0 );
    5: if (! strncmp(dest, src, len) ) return 1;
    return 0;
}
```

Figure 49: Code of authenticate function with assertions

6.4 TECHNIQUE AND TOOL

As mentioned in the previous section, enumerating insider attacks by hand is cumbersome and non-trivial. Therefore, automating the discovery of insider attacks is essential. This section describes the key techniques used in the automation and the design of a tool to perform the discovery.

6.4.1 Symbolic Execution Technique

We represent an insider attack as a corruption of data values at specific points in the program's execution i.e. attack points. The attack points are chosen by the program developer based on knowledge of where an insider can attack the application. For example, all the places where the application calls an untrusted third-party library are attack points as an insider can launch an attack from these points. In the worst-case, every instruction in the application can be an attack point.

The program is executed with a known (concrete) input, and when one of the specified execution points is reached, a single variable[21] is chosen from the set of all variables in

[21] We use the term variable to refer to both registers and memory locations in the program. This includes stack, heap and static data.

the program and assigned to a symbolic value (i.e. not a concrete value). The program execution is continued with the symbolic value. The above procedure is repeated exhaustively for each data value in the program at each of the specified attack points. This allows enumeration of all insider attacks on a given program.

The key technique used to comprehensively enumerate insider attacks is *symbolic execution-based model checking*. This means that the program is executed with a combination of concrete values and symbolic values, and model-checking is used to "fill-in" the symbolic values as and when needed. Symbolic values are treated similar to concrete values in arithmetic and logical computations performed in the system. The main difference is in how branches and memory accesses based on expressions involving symbolic values are handled as follows:

When a branch decision involving a symbolic expression is reached, the program is forked – one fork executes the branch assuming that the branch decision is true, and the other fork executes the branch assuming that the branch decision is false. The branch decision is added as a constraint to the program state, and the symbolic expression is evaluated based on the constraint. In case the solution converges to a single value, all symbolic expressions in the state are replaced with their concrete values. Otherwise, the constraint is added to a pool of constraints and the program's execution is continued. The global pool of constraints is maintained in parallel with the concrete program state, and updated on each branch executed by the program.

When a memory read (write) involving a symbolic expression in the address operand is encountered, the entire memory space of the system is scanned[22] and each location in memory is considered to be a potential target for the memory read (write). For each potential target, the execution of the program is forked and the symbolic expression is assigned to be equal to one of the addresses in memory. The value read (written) is assigned to the value stored in the corresponding address. As a result, the symbolic expression will evaluate to a unique value, which is then substituted in all symbolic expressions in the state. Thus, a memory access with a symbolic expression as the pointer operand converts the state into one in which all values are concrete.

Symbolic expressions are also used to represent indirect control transfers in the program (through a function pointer, for example). Indirect control transfers are treated similar to memory accesses through symbolic pointer expressions. In other words, each code location is treated as a potential target for the indirect branch, and the execution is forked with the constraint for the expression added to the fork.

For each program fork encountered above, the model checker checks whether (1) The fork is a viable one, based on the past constraints of the symbolic expressions, and (2) whether the fork leads to a desired outcome (of the attacker). If these two conditions are satisfied, the model checker will print the state of the program corresponding to the fork i.e. attack state.

[22] The exhaustive search may incur significant overheads. Section 6.6.5 presents ways to make this search less performance intensive.

6.4.2 SymPLAID Tool

The symbolic execution technique described in the previous section has been implemented in an automated tool – SymPLAID (*Symbolic Program Level Attack Injection and Detection*). This is based on our earlier tool, SymPLFIED, used to study the effect of transient errors on programs [136]. The differences between SymPLAID and SymPLFIED are explained in Section 6.4.3.

SymPLAID accepts the following inputs: (1) an assembly language program along with libraries (if any), (2) a set of pre-defined inputs for the program, (3) a specification of the desired goal of the attacker (expressed as a formula in first-order logic) and (4) a set of attack points in the application. It generates a comprehensive set of insider attacks that lead to the goal state.

For each attack, SymPLAID generates both the location (memory or register) to be corrupted as well as the value that must be written to the location by the attacker. Figure 50 shows the conceptual view of SymPLAID from a user's perspective.

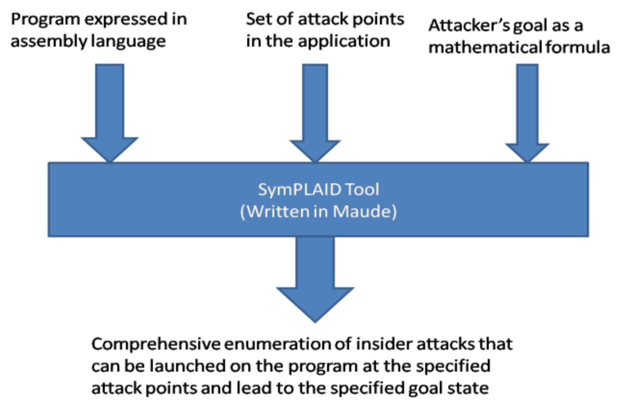

Program expressed in assembly language
Set of attack points in the application
Attacker's goal as a mathematical formula
SymPLAID Tool (Written in Maude)
Comprehensive enumeration of insider attacks that can be launched on the program at the specified attack points and lead to the specified goal state

Figure 50: Conceptual view of SymPLAID's usage model

SymPLAID directly parses and interprets assembly language programs for a MIPS processor. The current implementation supports the entire range of MIPS instructions, including (1) arithmetic/logical instructions, (2) memory accesses (both aligned and unaligned) and (3) branches (both direct and indirect). However, it does not support system calls. The lack of system call support is compensated for by the provision of native support for input/output operations. Floating point operations are also not considered by SymPLAID. This is reasonable as floating-point operations are not typically used by security-critical code in the majority of applications.

SymPLAID is implemented using Maude, a high-performance language and system that supports specification and programming in rewriting logic [34]. SymPLAID models the execution semantics of an assembly language program using both equations and rewriting rules. Equations are used to model the concrete semantics of the machine, while rewriting rules are used for introducing non-determinism due to symbolic evaluation.

6.4.3 Differences with SymPLFIED

This section discusses the differences between the analysis performed by the SymPLFIED and SymPLAID tools. The first column of Table 27 shows an example code fragment (in a MIPS-like assembly language). The state of each register in the program as determined by SymPLFIED and SymPLAID (after executing the instruction in the row) is shown in the second and third columns of Table 27 respectively.

Assume that the value in register $2 has been corrupted (either by a transient error or by an insider attack) in instruction 2. Both SymPLAID and SymPLFIED represent the value in register $2 using the abstract symbol *err*. Until this point, the state of the program in both SymPLFIED and SymPLAID is the same. Then instruction 3 is executed, which subtracts 5 from the value in register $2. At this point, the states diverge: SymPLFIED represents the value in register $5 also with the symbol *err*, thereby approximating the dependencies among the value. On the other hand, SymPLAID represents the value in register $5 by the symbolic expression *(err – 5)*, which is more precise.

Table 27: Example code illustrate SymPLFIED and SymPLAID

Code	**SymPLFIED**	**SymPLAID**
[1 | movi $3, #(10)]	$3 = 10	
[2 | addi $2, 0, #(err)]	$2 = err	
[3 | subi $5, $2, #(5)]	$5 = err	$5 = err – 5
[4 | muli $4, $(5), #(2)]	$4 = 2 * err	$4 = 2 * (err – 5)
[5 | bne $4, $3, 7]	$4 == 1 or 0 ?	$4 = 1 or 0 ?
[6 | print $5]	$2 = err $3 = 10, $4 = 10 $5 = err output: err	$2 = 10 $3 = 10 $4 = 10 $5 = 5 output: 5

The execution then continues on to instruction 4, which multiplies the value in register $5 with a constant 2. SymPLFIED once again approximates the value in register $5 with the symbol *err*, whereas SymPLAID stores the symbolic expression *2 * (err – 5)* in register $5.

Finally, execution reaches instruction 5, which checks if the value in register $2 is not equal to the constant 10. If so, the program branches to location 7 and bypasses instruction 6. Otherwise, it executes instruction 6 which prints the value stored in register $5.

In the case of both SymPLFIED and SymPLAID, the execution is forked at instruction 5. This is because the value in register $4 contains a symbolic expression involving *err* in the case of SymPLAID, and the symbol *err* in the case of SymPLFIED. In both cases, the value cannot be uniquely evaluated. Hence, the tool must consider both the case where the equality holds and the case where it does not by forking the program's execution.

Let us consider the fork where the equality holds and the branch is not taken. In this case, the control reaches statement 6 and the value in register $6 is printed by the program. SymPLFIED and SymPLAID would print a different value at this instruction.

In the case of SymPLFIED, when the branch in instruction 5 is not taken, the value in register $4 is set to 10. No other changes are made to the registers. Hence, when instruction 6 is reached, register $5 contains the value *err*, which is printed.

In the case of SymPLAID, when the branch is not taken, the value of register $4 is set to 10 as done in the case of SymPLFIED. However, the assignment to register $4 triggers a wave of updates in the system. This is because the symbolic expression in register $4, which is *(err – 5) * 2* is set equal to 10. The value of *err* is then uniquely determined to be 5 by solving the above equation. The values of registers $2, $4 and $5 are updated based on the solved value to be 10, 10 and 5 respectively. This effectively converts all symbolic expressions in the state to concrete ones and is an example of condition 1 in section 6.4.1. Hence, when instruction 6 is reached, register $5 contains the value *5*, which is printed.

The above example illustrates how SymPLAID is able to achieve higher precision than SymPLFIED by tracking dependencies among the corrupted values in registers and memory. *As a result, it is able to isolate the value(s) that must be injected by an attacker* *to achieve their attack goal.* In this example, if the attacker wants to make the program print the value 5, he/she must overwrite the value in register $2 at instruction 2 with the value 5. The attack is automatically discovered by SymPLAID but not by SymPLFIED.

The other differences with SymPLFIED are that SymPLAID supports a wider range of instructions (e.g. unaligned loads and stores) and has a more precise constraint solver.

6.4.4 Precision and Scalability

SymPLAID maintains precise dependencies both in terms of arithmetic and logical constraints and solves them using a custom constraint solver. However, calls to the constraint solver can be expensive, and are minimized as follows:

- SymPLAID keeps track of symbolic expressions as part of the application state, and does not solve them until a decision point is reached, viz., branches, memory accesses and indirect control transfers.
- SymPLAID maintains simple linear constraints in a separate constraint map, and can identify infeasible states (such as the same erroneous location being assigned to multiple values) without invoking the constraint solver.
- Finally, SymPLAID replaces symbolic states with concrete states at the earliest opportunities, thereby reducing calls to the constraint solver and also minimizing the number of forks in the program.

There are two cases where SymPLAID performs approximations to conserve space. These are as follows: First, SymPLAID can only solve linear constraints. In practice, few security critical branches or memory access operations involve non-linear constraints and hence this does not result in false-positives.The second approximation occurs when an unaligned memory access is performed in SymPLAID. In order to conserve space, SymPLAID stores values in memory as integral values over the entire word, and hence cannot model corruptions of individual bytes in a word. This can result in loss of precision leading to false-positives.

6.5 DETAILED ANALYSIS

This section illustrates how SymPLAID identifies conditions for successful insider attacks in the context of the example considered in Section 6.3. The code shown in Figure 51 is a modified version of the MIPS assembly code for the example in Figure 49.

In Figure 51, instructions and labels are expressed within '[', ']', and a | character separates the instruction from its label. Comments are prefixed with a --- and can follow the instruction. For ease of analysis, we generated simplified versions of the standard library functions, but with the same functionality as the original functions.

Figure 51 shows the assembly code of the *strncmp* function and an excerpt from the *authenticate* function that calls the *strncmp* function. In this section, we will consider how SymPLAID analyzes the effect of data value corruptions introduced in the *authenticate* function to generate the set of attacks in the first three rows of Table 26.

Consider the attacks that can be launched at the call site of the *strncmp* function in instruction 44 of the *authenticate* function. This is the *attack point* in this example. We assume that the insider can only corrupt values in registers. The following discussion considers only the cases that lead to successful attacks.

```
Strncmp: ------- String comparison routine
        [ 0 | ldo $(16) #(4) $(esp)  ]    --- load the src
        [ 1 | ldo $(17) #(8) $(esp)  ]    --- load the dest
        [ 2 | ldo $(18) #(12) $(esp) ]    --- load the length
        [ 3 | movi $(1) #(1)         ]     --- result to return
        [ 4 | bltzi $(18) #(15)       ]   --- while length > 0
        [ 5 | lbu $(19) #(0) $(16)   ]   --- load of src
        [ 6 | lbu $(20) #(0) $(17)   ]   --- load of dest
        [ 7 | seq $(21) $(19) $(20)  ]   --- is *src==*dest ?
        [ 8 | beqii $(21) #(0) #(15) ]   --- if not equal, break
        [ 9 | beqii $(19) #(0) #(16) ]  ---if end of string, break
        [ 10 | addi $(16) $(16) #(1) ]          --- increment src
        [ 11 | addi $(17) $(17) #(1) ]          --- increment dest
        [ 12 | subi $(18) $(18) #(1) ]   --- decrement length
        [ 13 | beqii $(0) #(0) #(4)  ]   --- loop backedge
        [ 14 | movi $(1) #(0)       ]   --- store in register 1
        [ 15 | return              ]   --- return

Authenticate Function Excerpt: …
[ 21 | ldo $(1) #(tmpAddr) $(0)    ]  --- Retrieve temp
[ 22 | ldo $(2) #(srcAddr) $(0)    ] --- Retrieve the src
[ 23 | ldo $(3) #(lengthAddr) $(0) ] --- Retrieve the length
[ 24 | sto $(1) #(4) $(esp)       ] --- Push parameters on stack
[ 25 | sto $(2) #(8) $(esp)       ]
[ 26 | sto $(3) #(12) $(esp)     ]
[ 27 | call #(strncpyLoc)   ] --- call the string copy function
[ 28 | ldo $(1) #(tmpAddr) $(0)    ] --- load the temp buf
[ 29 | sto $(1) #(4) $(esp)       ] --- push buffer onto stack
[ 30 | call #(readInputLoc)  ] --- call the readInput function
[ 31 | ldo $(1) #(tmpAddr) $(0)    ] --- Retrieve src address
[ 32 | ldo $(2) #(destAddr) $(0)   ] --- Retrieve dest address
[ 33 | ldo $(3) #(lengthAddr) $(0) ] --- Retrieve the length
[ 34 | sto $(1) #(4) $(esp)       ] --- Push parameters on stack
[ 35 | sto $(2) #(8) $(esp)       ]
[ 36 | sto $(3) #(12) $(esp)      ]
[ 37 | call #(strncpyLoc)   ] --- call the string copy function
[ 38 | ldo $(1) #(srcAddr) $(0)     ] --- load the source
[ 39 | ldo $(2) #(destAddr) $(0)    ] --- load the dest address
[ 40 | ldo $(3) #(lengthAddr) $(0)  ] --- load the length
[ 41 | sto $(1) #(4) $(esp)        ] --- push the parameters
[ 42 | sto $(2) #(8) $(esp)        ]
[ 43 | sto $(3) #(12) $(esp)       ]
[ 44 | call #(strncmpLoc)          ] --- call strncmp function
[ 45 | beqii $(1) #(0) #(48) ] --- if equal to 0, goto 48
[ 46 | movi $(1) #(0)          ] ---unequal
[ 47 | beqii $(0) #(0) #(49)    ] --- go to the end
[ 48 | movi $(1) #(1)           ] --- equal
[ 49 | return                          ]
```

Figure 51: Assembly code corresponding to Figure 2

Case 1: Assume that the insider has corrupted the value of register \$3, which contains the length of the string. To model this attack, SymPLAID replaces the value in register \$3 with the symbol (*err)* just prior to calling the *strncmp* function. Consequently, the value *err* is pushed onto the stack as an argument to the function. The *strncmp* function loads the value *err* into register \$(18) (instruction 2 of *strncmp*). It then initializes the result of the comparison in register \$1 to 1, and evaluates if the value in register \$18 is lesser than 0 (instruction 4 of *strncmp*). If so, it branches to instruction 15 of the *strncmp* function, which in turn returns the value in register \$1 to the *authenticate* function.

When SymPLAID encounters the comparison instruction 4 (in *strncmp*), it cannot uniquely resolve the comparison as it has no information on the value in register \$18 (which is *err)*. Consequently, it forks the execution into two – one fork sets the value of register \$1 to be lesser than 0, and the other sets the value to be greater than or equal to 0. The former case takes the branch to instruction 15, and exits the function, thereby returning the value 1 to the *authenticate* function. This causes the *authenticate* function to return the value 1 to its caller – which is the desired goal of the attacker. The latter fork in which the value of register \$(18) is set to be greater than 0 does not however result in the outcome desired by the attacker. Therefore, in the above example, SymPLAID discovers that by setting the value of the length parameter of the *strncmp* function to a value lesser than 0, the attacker successfully achieves the goal of authenticating the user.

Case 2: Assume that the attacker overwrites the value in register \$2, which holds the address of the *dest* string. This is passed as an argument to the *strncmp* function, and copied into register \$17 by instruction 2 of the function. SymPLAID represents the value in register \$2 by the symbol *err* and tracks its propagation to register \$17. This value is used as a pointer argument to by the load-byte-unsigned (instruction 6) in *strncmp*. At this point, symPLAID cannot uniquely determine the memory address that the register refers to, and hence it forks the execution so that each fork evaluates the symbolic expression to a different address.

The memory state of the program is generated by executing the code of the *authenticate* function until the fork point. SymPLAID assigns the value in register $17 to each of the memory addresses in succession. Simultaneously, it assigns the value loaded into register $6 to the value stored in the corresponding address. Of all the possibilities, only one of them leads to a state that satisfies the attacker's goal (which is that the *authenticate* function returns 1). This occurs when the address of the *src* buffer is passed to the function, as it will match the other argument which is also *src*.

Case 3: A similar case arises when the value in register $1 is corrupted prior to the call to *strncmp*. The value in register $1 holds the value of the string *src,* which is passed as the first argument to the *strncmp* function. The value is stored in register $16 by instruction 1 and is used as as a pointer in instruction 9 of the strncmp function. As in the previous case, SymPLAID assigns a value of *err* to the register, and forks the execution at instruction 9, assigning each memory address in succession to the pointer value in register $16. Of all the values considered by SymPLAID, only two values (*dest* and *tmp*) satisfy the attacker's goal, and are hence reported by SymPLAID as attacks.

Summary: Note that the output from SymPLAID is in the form of raw data and may consist of repetitive or redundant states. The output needs to be post-processed in order to identify more generic classes of potential attacks to drive the development of defense mechanisms that can protect the application against a range of discovered attacks. The post-processing is currently done manually.

6.6 CASE STUDY

To evaluate the SymPLAID tool on a real application, we considered a reduced version of the OpenSSH application [137] involving only the user-authentication part. This is because SymPLAID does not support all the features used in the complete SSH application, e.g. system calls. We retain the core functions in the authentication part of OpenSSH with little or no modifications, and replace the more complex ones with stub versions – i.e. simplified functions that approximate the behavior of their original versions. We also replace the system calls with stubs. The reduced version is called the authentication module.

The authentication module emulates the behavior of the SSH application starting from the point after the user enters his/her username and password to the point that he/she is authenticated (or denied authentication) by the system. The authentication module consists of about 250 lines of C code (excluding standard libraries). The functions in the module are shown in Table 26.

Table 28: Functions in the OpenSSH authentication module

Function Name	LOC(C)	Functionality
fakepw	15	Fills a structure with a default (fake) password and returns it
shadow_pw	7	Stub version of a system call to retrieve the hash of the password
getpwnam	19	Stub version of a system call to retrieve password for a username
pwcopy	22	Makes a field-by-field copy of the password structure
sys_auth_passwd	29	Checks if the user supplied password matches system password
allowed_user	6	Stub version of a complex function to check if a user is in the list of allowed users
xcrypt	7	Stub version of a system call to encrypt the password using a salt value (based on username)
getpwnam-allow	43	Checks if a user is allowed to login and if so retrieves their password record makes a copy using pwcopy
auth_password	14	Checks if the username is allowed AND the user password is correct
main	47	Reads in the username and password and calls the above functions in the expected order

We ran SymPLAID on the authentication module after compiling it to MIPS assembly using the *gcc* compiler. As before, the goal is to find insider attacks that will allow the user to be authenticated. It is assumed that the insider can overwrite the value of a register in any instruction within the authentication module. The input to the authentication module is the username and password. The username may or may not be a valid username in the system, and the password may or may not be correct. These lead to four possible categories, of which one is legitimate and three are attacks. SymPLAID discovered attacks corresponding to the categories where an invalid username is supplied with a valid password (for the application) and where a valid user-name is supplied with an incorrect password. In this section, we consider both categories of attacks.

6.6.1 Category 1: Invalid User-name

The authentication part of SSH works as follows[23]: when the user enters his/her name, the program first checks the user-name against a list of users who are allowed to log into the system. If the user is allowed to log into the system, the user record is assigned to a data-structure called an *authctxt* and the user details are stored into the *authctxt* structure. If the name is not found on the list, the record is assigned to a special data-structure in memory called as *fake. fake* is also an *authctxt* structure, except that it holds a dummy username and password. This ensures that there is no observable difference in the time it

[23] We consider only the case where authentication is done using the keyboard.

takes to process legitimate and illegitimate users (which may enable attackers to learn if a username is valid by repeated attempts to login).

In order to prevent potential attackers from logging on by providing this dummy password, the *authctxt* structure has an additional field called *valid.* This field is set to *true* only for legitimate *authctxt* records i.e. those for which the username is in the list of valid users for the system. The *fake* structure has the *valid* field set to *false* by default. In order for the authentication to succeed, the encrypted value of the user password must match the (encrypted) system password, *and* the valid flag of the *authctxt* record must be set to the value 1.

Figure 53 shows the *auth_password* function that performs the above checks. The function first calls the *sys_auth_passwd* to check if the passwords match, and then checks if the *valid* flag is set in the *authctxt* record. Only if both conditions are true will the function return 1 (authenticated) to its caller.

```
int sys_auth_passwd(Authctxt *authctxt, const char *password) {
        struct passwd *pw = authctxt->pw;
        char *encrypted_password;
        char *pw_password = authctxt->valid ?
                shadow_pw(pw) : pw->pw_passwd;
        if (strcmp(pw_password, "") == 0 &&
            strcmp(password, "") == 0)
                return (1);
        encrypted_password = xcrypt(password,
                (pw_password[0] && pw_password[1]) ?
                 pw_password : "xx");
    return (strcmp(encrypted_password, pw_password) == 0);
}
int auth_password(Authctxt *authctxt, const char *password) {
         int permit_empty_passwd = 0;
        struct passwd * pw = authctxt->pw;
        int result, ok = authctxt->valid;
        if (*password == '\0' && permit_empty_passwd == 0)
                return 0;
        result = sys_auth_passwd(authctxt, password);
        if (authctxt->force_pwchange)
                disable_forwarding();
        return (result && ok);
}
```

Figure 52: SSH code fragment corresponding to the attack

An insider can launch an attack by setting the *valid* flag to true for the *fake* authctxt structure. This will authenticate a user who enters an invalid user name, but enters the password stored in the *fake* structure. The password in the *fake* structure is a string that is hardcoded into the program.

To mimic this attack, we supply an invalid user-name and a password that matches the *fake* (dummy) password. We expected SymPLAID to find the attack where the insider overwrites the valid flag of the *fake* structure. SymPLAID found this attack, but it also found a host of other attacks that overwrite the frame pointer of the function. We describe a particularly interesting attack found by SymPLAID here.

The attack occurs in the *sys_auth_password* function, at line 11 before the call to the *strcmp* function (in Figure 52) .At this point, the insider corrupts the value of the stack pointer (stored in register $30 in the MIPS architecture) to point within the stack frame of the caller function, namely *auth_password*. When the *strcmp* function is called, it pushes the current frame pointer onto the stack, increments the stack pointer and sets its frame pointer to be equal to the value of the stack pointer (corrupted by the attacker). Figure 53 shows the stack layout when the function is called (only the variables relevant to the attack are shown).

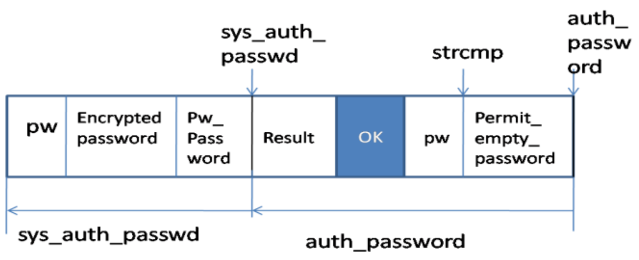


Figure 53: Stack layout when strcmp is called

The top-row of Figure 53 shows the frame-pointers of the functions on the stack due to the attack. Observe that the attack causes the stack frame of the *strcmp* function to overlap with that of the *auth_password* function. The *strcmp* function is invoked with the addresses of the *encrypted_pasword* and the *pw_password* buffers in registers[24] *$3* and *$4*. The function copies the contents of these registers to locations within its stack frame at offsets of 4 and 8 respectively from its frame pointer. This overwrites the value of the local variable *ok* in the *auth_password* function with a non-zero value (since both buffers are at non-zero addresses). When the strcmp returns, the value of $30 is restored to the frame pointer of *sys_auth_passwd*, which in turn returns to the *auth_password* function. The *auth_password* function checks if the result returned from *sys_auth_password* is non-zero and if the *ok* flag is non-zero. Both conditions are satisfied, so it returns the value 1 to its caller, and the user is successfully authenticated by the system.

6.6.2 Category 2: Incorrect Password

The second category corresponds to the case when the application is executed with a valid username but with the wrong password. We ran SymPLAID on the application and asked it to find attacks where the user is successfully authenticated. We consider a particularly interesting example attack found by SymPLAID. The attack is described in this section.

[24] In the MIPS architecture, function arguments are passed in registers

The attack occurs in the function *sys_auth_password* shown in Figure 52. As can be seen from the Figure, the function *sys_auth_password* returns 1 to its caller (*auth_password*) if either the encrypted version of the user password matches with the encrypted version of the system password, OR if both passwords are empty strings. In a normal execution of the SSH application, the user password is checked by the *auth_password* function, and if empty, a special flag *permit_empty_password* is checked. This flag indicates if the user is allowed to have an empty password (at account creation time, for example). If the flag is not set, the application is aborted. Therefore, under normal circumstances, the user password cannot be empty. However in the case where it is empty, the *auth_password* function returns a value ‘1’ provided the corresponding system password is also empty.

A naïve attacker may try overwriting the value of *permit_empty_passwd* and entering an empty string for the password. However, this would require that the system password is also empty. Since we assume that only one corruption is allowed per execution, the attacker will not be able to corrupt both the system password and the user password simultaneously to make both of them point to empty strings, and the attack will not succeed. A better option for the attacker may be to overwrite the value of the system password (*pw_password*) after it is returned from the *shadow_pw()* function. This would not work either as the attacker would need to overwrite the user password (*authctxt->pw*) in order for the attack to succeed, which is not possible given the single value restriction.

To craft a successful attack, observe that the system password is returned by the function *shadow_pw (*since the username is valid, *authctxt->valid* is set to 1). Therefore, the attacker can try to make *shadow_pw* return an empty string and in the process, also overwrite the contents of the *password* variable. The difficulty with this approach is that *shadow_pw* is a system call and its value is determined based on the value of the system password. Nonetheless, it is possible to make *shadow_pw* return an empty string by passing it a NULL string as argument. This can be done by shifting the frame pointer of the *sys_auth_passwd* function to a memory location where the value stored in the address corresponding to the *pw* variable is 0, AND the value corresponding to the *password* variable points to an empty string. The attack is carried out after the check on *authctxt->valid* but before the call to *shadow_pw* at line 4 in Figure 52.

6.6.3 Summary: Attacks Found

Table 29 summarizes the attacks discovered by SymPLAID for the two categories presented in sections 6.6.1and 6.6.2. The attacks shown in Table 29 are confined to the two functions shown in Figure 52. We do not consider attacks that originate in the stubs for the system calls or those that originate in the main function as these are artifacts of the authentication module, rather than the application. We have validated the attacks shown in Table 29 on the OpenSSH application compiled for the MIPS architecture. We used the SimpleScalar emulator for carrying out the insider attacks.

Table 29: Summary of attacks found by SymPLAID for the module

FUNC	CAT	Attack found by SymPLAID for the function and category shown in the first two columns
auth_password	1	After initializing the value of local variable *ok* with *authctxt->valid*, overwrite it with a non-zero value.
	1	After setting up the stack with the local variables and return address, but before calling *sys_auth_passwd*, overwrite the stack pointer with an address such that *sys_auth_passwd* overwrites the value of the variable *ok* when storing on the stack
	2	After the call to *sys_auth_password*, overwrite its return value (register $2) with a non-zero value
	2	After the call to *sys_auth_password*, shift the frame pointer of the function such that before the function returns, it will read non-zero values in the relative addresses of the local variables *result* and *ok*
	1, 2	Overwrite the value of register $2 with a non-zero value when the function is about to return, so that it always returns *true*
sys_auth_password	1	Before the call to *strcmp*, inject the stack pointer with an address in the previous stack frame so that the value of *ok* is overwritten with a non-zero value. This makes *auth_password* return 1.
	1	Before the call to the strcmp function, overwrite the frame pointer with an address in the previous stack frame, so that when *strcmp* returns a non-zero value (since the strings are different), the return address is the address of the instruction that called *auth_password (*from *main).*
	2	After the call to the *xcrypt* function, set the value returned by it in register $2 to the address of the buffer where the encrypted system password is stored. This sets the user password pointer to the encrypted value of the system password.
	2	Before calling the *strcmp* function, change either its first argument (register $4) to the address of buffer with the encrypted system password, or change the second argument (register $5) to the address of the buffer containing the encrypted user password
	2	After the call to the shadow_pw address, change its return value to the address of the buffer that contains the user password
	2	Change the return value of the function (register $2) to a non-zero value

6.6.4 Spurious Attacks

The approximations in the authentication module may introduce spurious attacks. These are attacks that work on the authentication module but do not work on the real OpenSSH application. This is because the stub functions in the module introduced approximations that did not mimic the real system's behavior in all cases. Since the model-checker explores all possible behaviors of the system, it flagged the non-conforming cases as attacks. A similar phenomenon was observed by Musuvati and Engler [139].

Most of the spurious attacks discovered were easy to filter out as they were launched from stub functions that were system calls in the real program. However, there was one subtle attack that at first seemed like a real attack, but turned out to be spurious. This is described here in this section.

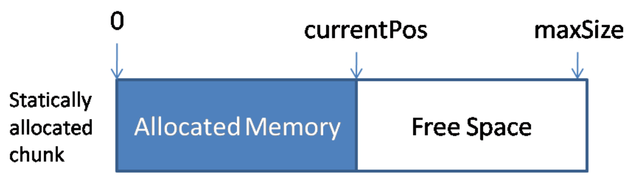


Figure 54: Schematic diagram of chunk allocator

The OpenSSH program uses its own custom memory allocator (*xmalloc*) to store the buffers containing the user and system passwords (not shown in Figure 52). *xmalloc* allocates memory in chunks, and if it runs out of space in the chunk, it calls the system malloc to allocate a new chunk. Our authentication module simplifies this behavior by allocating a single static chunk of memory when the application is initialized, and then satisfying all application malloc requests from this chunk. The initial chunk is chosen to be large enough to accommodate both the password buffers and other dynamic memory used in the program. In order to ensure that we do not exceed the size of the initial chunk, every allocation request in the program is checked to ensure that it is within the bounds of the space remaining in the chunk.. Figure 54 shows a schematic of the allocator.

The simplified memory allocator has a runtime check of the form:

if (currentPos + sizeRequested > maxSize) return NULL;

where *currentPos* is the location of the next free location in the initial chunk, and *maxSize* is the size of the initial chunk.

The SymPLAID tool finds an attack that effectively overwrites the location of the *currentPos* variable in memory with a value that is greater than the value of *maxSize*. This causes all subsequent *malloc* requests from the application to be declined and the value NULL to be returned. In order to prevent a NULL pointer violation, the calling function makes the pointers point to a special sentinel value in memory. If this attack is carried out at the beginning of the authentication module (in the *getpwnamallow* function, say), this will cause both the password strings to point to the same sentinel value, and hence they will match with each other. Therefore, the *sys_auth_password* function will return 1, and the malicious user will be authenticated, thereby leading to a successful attack.

In reality, this attack cannot be achieved easily as the custom allocator in *ssh* will not merely return NULL if it exceeds the bounds of the current chunk, but will get a new chunk from the operating system (using *brk* in linux), and maintain a linked list of the allocated and free chunks. It is conceivable that a more sophisticated version of the attack can be mounted by overwriting the head of the free list with NULL to simulate the conditions leading to memory exhaustion. However, we have not tested the more sophisticated attack.

The above situation could have been avoided had we modeled a more accurate version of the memory allocator used by OpenSSH. However, a similar situation could have arisen in any of the other stub functions. Therefore, any approximation of system behavior is likely to lead to spurious outcomes. The only way to avoid this situation is to analyze the entire system (application, libraries and operating system) using the model-checker as done in [139]. However, this can lead to state space explosion in the model-checker and is an area of ongoing work.

6.6.5 Performance Results

This section reports the performance overheads incurred by SymPLAID in finding the attacks on the OpenSSH application. We executed SymPLAID on a parallel cluster consisting of dual-processor AMD Opteron nodes, each of which has 2 GB RAM. This is because the search task is highly parallelizable and can be broken into independent sub-tasks, with each sub-task considering attacks in a different code region of the application. The authentication module consists of about 500 assembly language instructions, and the task was broken up into 50 parallel sub-tasks, each of which considers a code region of 10 instructions. The maximum time allowed for completion of a sub-task is approximately 2 days (after which the task is forcibly terminated) and its execution time recorded as 48 hours.

Table 30: Time taken by SymPLAID for each function

Function Name	**LOC (assembly)**	**Number of States**	**Total Time (sec)**	**Attacks found ?**
getpwnamAllow	37	6391	325861	No
sys-auth-passwd	54	36896	366108	Yes
fakepw	29	11	115	No
xcrypt	37	26921	429683	Yes
shadow-pw	26	10342	272236	Yes
allowed-user	20	11	115	Yes
auth-password	40	26921	429683	Yes
getpwnam	37	27724	534601	Yes
pwCopy	52	23547	471185	Yes
main	114	3137	297526	Yes

Table 30 shows the time and space requirements of the sub tasks categorized by the function which they were analyzing for attacks. The space requirements are reported in terms of the number of unique “states” visited by the model-checker. The time taken is reported in seconds. The results are aggregated across multiple sub-tasks for the function and the cumulative time and space requirements are reported. Note that this is not equivalent to running the sub-tasks for the function as a single aggregate task as the sub-tasks may have significant state sharing across them. Hence the time and space taken by a single aggregate task is likely to be smaller than the aggregated results in Table 30. Based on the results in Table 30, the total time taken to execute all sub-tasks is at most 3127113 seconds or 36.2 days. However, we are able to finish the task in 2 days due to the highly parallel nature of the search task. The total number of states explored by the sub-tasks is 161091.

While the running time seems high, it is not a concern as the goal is to discover all potential attacks (in a reasonable time frame) and to find protection mechanisms against them. The analysis can be easily parallelized and executed on multiple nodes as independent sub-tasks (as we did). Therefore, as we move towards multi-core and large-scale parallel computers, the analysis time is bound to decrease. Finally, model-checking is a very active area of research and new techniques are being invented to make model-checking faster. We can take advantage of such approaches to reduce the running time of SymPLAID.

The reasons for high running time are: (1) SymPLAID performs an exhaustive search of memory locations whenever it encounters a load/store through a register containing a symbolic expression., (2) When SymPLAID encounters an indirect jump instruction with a register containing a symbolic expression, it needs to scan the entire code-base as potential targets for the jump, and (3) SymPLAID distinguishes between states with minor differences in their representation (not relevant to attack).

We are currently investigating the following optimizations to reduce the running time.

- Instead of performing an exhaustive search of memory locations, consider multiple locations in the form of abstract memory regions. Further, if memory safety-checking techniques are deployed, it is enough to consider memory accesses within the write-set of a pointer location.
- Restrict the scope of indirect jumps to be within a function or module. Control-flow checking techniques to reduce the scope of valid jump targets in a program can be deployed.

6.7 RELATED WORK

We classify related work into three broad categories as follows: (1) Identification and generation of insider attacks, (2) Symbolic execution techniques to find security vulnerabilities and, (3) Fault Injection techniques to perturb application state.

6.7.1 Insider Attacks

Insider attacks have been a significant source of security threats, and efforts have been made to model insider attacks at the network level. Philips and Swiler[134] introduced the attack graph model to represent the set of all possible attacks that can be launched in a network. The nodes of an attack graph represent the state of the network, and each path in the graph represents a possible attack.. Ritchey and Amman [133] introduce a model-checking based technique to automatically find attacks starting from a known goal state of the attacker. Sheyner et. al. generalize this technique to generate all possible attack paths, thereby generating the entire attack graph [132] . Chinchani et al. present a variant of attack graphs called key-challenge graphs that are specifically tuned to represent insider attacks [140].

Insider attacks were also modeled at the operating system level by Probst et al.[135]. In this model, applications are represented as sets of processes that can access sets of resources in the system. An insider is modeled as a malicious process in the system that may access resources in violation of the system's security policy. Their technique builds a process interaction graph for the system and performs reachability analysis to discover insider attacks.

Attack-graphs and process graphs are too coarse grained for representing application-level attacks, and hence we directly analyze the application's code. Further, we do not require the developer to provide a formal description of the system being analyzed, which can require significant effort. Since we analyze the application's code directly, we can model attacks both in the design and implementation of the application. This is important as an insider typically has access to the application's code, and can launch low-level attacks on its implementation.

6.7.2 Symbolic Execution

Symbolic execution is a well-explored technique to find program errors [117]. Recently, it has also been used to find security vulnerabilities in applications [141-144].

Kruegel et al. present a technique to automatically generate mimicry attacks against system-call based attack detection techniques [143]. By symbolically executing the program, their technique can find attack inputs that can execute a malicious system call, while replicating its context, thereby remaining undetected (by the monitor).

EXE is a symbolic execution technique to generate security attacks against applications [144]. Their approach directly executes the application code on symbolic inputs, and progressively constrains them when conditional branches or assertions are encountered.

Molnar and Wagner generate attacks to exploit integer conversion errors in programs [142]. Their technique starts with a valid (non-attack) input and attempts to mutate it into an input that exploits a given integer conversion vulnerability.

Bouncer generates filters (program assertions) to block exploits of known memory corruption vulnerabilities [141]. The technique starts with an attack that exploits a given vulnerability, and symbolically executes the program to generate a set of constraints under which the vulnerability can be exploited. It then uses the generated constraints to block all inputs that may exploit the vulnerability.

The above techniques are concerned with generating attack inputs for the applications to exploit known or unknown vulnerabilities. In contrast, our technique attempts to generate attacks for a given input, assuming that the attacker is already present in the system. Further, the attacks found using our technique do not require the application to have an exploitable vulnerability (e.g. buffer overflows), but can be launched by a malicious insider in the system.

6.7.3 Fault-Injection Techniques

Fault-injection is an experimental technique to assess the vulnerability of computer systems to random events or faults [19]. An artificial perturbation signifying a random fault is introduced in the system and the behavior of the system is studied under the perturbation. Traditional fault-injection is statistical in nature, and is hence not guaranteed to expose all corner scenarios in the application. Consequently, it is not well-suited for modeling security attacks, as attackers typically exploit corner-case or unexpected behaviors. In spite of these limitations, researchers have used fault-injection to find security violations in systems. We consider some examples of such techniques.

Boneh, DeMillo and Lipton pioneered a study in which they found that transient hardware errors could adversely affect the security guarantees provided by public-key cryptosystems [145]. Subsequent studies have shown that many commonly used cryptographic systems can be broken by hardware errors in their implementation [146]. The main difference between these studies and ours is that our technique can be applied for any general security-critical system rather than only crypto-systems. Further, we allow the attacker to inject any value into the processor’s registers or memory which are more illustrative of insider attacks.

Xu et al. consider the effect of transient errors (Single-bit flips) in control-flow instructions on application security [147]. They use a technique known as “selective-exhaustive” injection to inject all possible errors in code segments that are known to be critical to the integrity of the systems from a security point of view. Our technique may be viewed as a form of selective-exhaustive injection, but into program data rather than instructions. The other difference is that we consider the effect of all possible data value corruptions rather than just single bit flips.

Govindavajhala and Appel [148] explore the use of transient errors to attack a virtual machine when the attacker has physical access to the machine. They show that transient errors can break the protections of the virtual machine up to 70% of the time, depending on the platform and the attacker's ability to execute a specially crafted program. The main difference between this work and ours is that we do not require attackers to have physical access to the machine, nor launch specially crafted applications.

6.8 CONCLUSION

This chapter presented a novel approach to discover insider attacks in applications. An automated technique to find all possible insider attacks on application code is presented. The technique uses a combination of symbolic execution and model-checking to systematically enumerate insider attacks for a given goal of the attacker. We have implemented the technique in the SymPLAID tool, and demonstrate it using the code segments corresponding to the authentication part of the OpenSSH application. We find several instances of potential insider attacks which may be missed by simple, manual inspection of the code.

CHAPTER 7 INSIDER ATTACK DETECTION BY INFORMATION-FLOW SIGNATURE (IFS) ENFORCEMENT

7.1 INTRODUCTION

The growing complexity of applications has necessitated a shift towards outsourcing of application components to third-party vendors often spanning geographic boundaries. The ubiquity of the internet has allowed software libraries to be freely distributed in source/binary forms and reused among applications. In this environment, a malicious developer may plant a logical loophole or backdoor in a library or module used by a security-critical application. The developer could leak details about the loophole to an attacker, who could then exploit the application through the loophole or backdoor when it is deployed in the field. Such backdoors and loopholes are extremely hard to detect unless detailed code auditing is performed (if the source code of the module is available). However, due competitive pressures in bringing a product to market, organizations often do not perform these tasks for code that is not developed in-house. Even for modules developed in-house (i.e. within the organization), the original developer(s) may have long left the company and the expertise to understand the code may be lost.

This problem is partially alleviated by open-source software, due to the “many eyeballs” scanning the source code for potential loopholes[149]. However, even in open-source software, it is possible for a malicious developer to plant a backdoor or loophole in an unused (or rarely used) part of the code. While backdoors have been ferreted out in popular open-source packages such as the Linux Kernel [150], they may not be as easy to detect in open-source packages that are not as widely-deployed. For example, a study of open-source software packages distributed using the popular SourceForge repository found that about 50 % of the packages had fewer than 70 downloads during the month-long duration of the study [151]. Further according to [152], a recent study of 100 open- and closed-source software packages found that about 23 of them had unwanted code, and about 79 packages had dead code of some form or another, although they were not necessarily malicious.

It is also possible for a malicious system administrator or IT manager to replace a system library or in-house software package with a modified version, in order to subvert existing security checks or induce malicious behavior. A recent CERT report on insider attacks [153] shows that while such attacks are relatively rare (only 15 out of 200 cases studied in the report belonged to this category), the attacker can cause extensive damage through these means. For example, the report illustrates how an IT manager in a state agency was able to carry out an extensive fraud-scheme undetected for two years by commenting out a single-line of source code in an in-house software program, and compiling and releasing the modified version within the organization.

Malware is defined as any program that attempts to perform malicious activities in the system [154]. Malware takes many forms, including viruses, worms and Trojans. Viruses and worms attach themselves to executable files and are activated when the program is executed. Trojans are stand-alone programs that masquerade as programs with legitimate functionality in order to trick users into executing the Trojan program.

Both static and dynamic approaches have been proposed for malware detection. Static malware detectors (including commercial virus scanners) look for known patterns of instructions representing a virus or Trojan in executables. However, such static checkers can be bypassed by malware that performs simple program obfuscations on itself [155]. Christodorescu et. al. [156] takes into account instruction semantics to determine if a given instruction sequence is a semantic variant of a known malware sequence. This approach requires the original sequence template of instructions in the malware to be specified, and variants are automatically detected by the technique. Dynamic approaches for malware detection check whether a program (i.e. potential malware) performs system-level malicious activities such as overwriting operating system files or registry entries. The check can be performed by monitoring the program using software [157] or hardware [158]. These approaches attempt to emulate conditions in the field in order to trick the malware into revealing its malicious behavior.

In contrast to malware, an untrusted module in an application does not attempt to infiltrate the system by performing system-level malicious activities. Rather the module may overwrite key elements of the application's control and data-space to achieve the attacker's goals. Hence, dynamic malware detection approaches will not be effective at detecting the attack. Further the instruction sequences corresponding to the malicious activities carried out by the module are specific to the application being attacked. Hence, the sequences may not be detected by a generic pattern-matching or semantics-based approach for static malware detection.

The attack model assumed in this chapter is that an untrusted module (whose source code may or may not be available) has been linked together with the application prior to the application's deployment. The untrusted module has a malicious snippet of code hidden in it (possibly masquerading as legitimate code) which performs some malicious activity in the context of the application e.g. overwrite security-critical data or bypass security checks. We refer to the untrusted module as an "application-level insider[25]". For example, an insider in a login program may overwrite the program's internal state to allow an attacker to log into the system even if he/she does not provide the correct password.

Analyzing an application's source code for application-level insiders is a hard problem. This is because the malicious code can masquerade as code with some legitimate functionality and pass manual audits[152]. Automated code analysis techniques can detect the malicious code segment, but these techniques require a specification of the program to be provided. In order to detect insider attacks, every line of an application's source code must be checked against its specification (as it is a potential hiding place for malicious code). Formulating a specification for each line of code is cumbersome and few developers choose to write such detailed specifications. The problem is exacerbated at the binary level, as binary code is often stripped of source symbols and obfuscated precisely to inhibit its understanding by a tool or human. Further, the malicious code

[25] An insider attack is defined as one in which a privileged entity i.e. an insider, abuses its privileges to exploit system loopholes or perform malicious activities. In this case, the privilege afforded to the untrusted module is that it is allowed to execute in the same address space as the application.

may not even be revealed by testing techniques as it may be activated only under a specific combination of inputs or environment variables. While it is possible to exhaustively test the program under every legal combination of inputs and examine its behavior, such exhaustive testing is not done due to practicality reasons of time and resource overheads.

The goal of this chapter is to develop a technique to detect application-level insider attacks. We focus on protecting a subset of the application's data that is critical to the security of the application from updates by untrusted modules (i.e. whose source code is neither available nor inspected). This *critical data* has to be identified based on the application's semantics either manually or automatically using specifications about the program.

The technique presented in the chapter uses the idea of Information-Flow Signatures (IFS) to detect application-level insider attacks that illegitimately overwrite critical data in the application. The programmer identifies the security-critical data through annotations in the source code. The IFS encodes the sequence of instructions that can legitimately write (directly or indirectly) to the critical data through the normal (attack-free) control-flow of the program. The IFS is derived using static analysis of the program's code (by enhancing the compiler), and requires the source-code of only those modules that can legitimately write to the critical data, i.e. the trusted modules. The program is instrumented to ensure that the statically-derived IFS is followed at runtime – any deviation represents an attack and the program is halted. Note that only attacks that violate the integrity of critical data are detected by the IFS. We do not consider attacks on the confidentiality of the critical data or Denial-of-Service (DoS) attacks on the application.

In earlier work [159], we introduced the idea of Information-flow signatures (IFS). This chapter builds significantly on the initial idea and shows how to derive and enforce the IFS technique for real applications. The derivation of the IFS is implemented by enhancing the LLVM optimizing compiler [99], while the runtime enforcement is done via custom software libraries. The technique is demonstrated in the context of three commonly deployed server applications (OpenSSH, WuFTP and NullHTTP) to protect security-critical modules. *We find that the performance overheads of the proposed technique range from 7.5 % to 100 % when measured with respect to the execution time of the module(s) protected, but are negligible (less than 1 %) when evaluated in the context of the entire application as the protected module constitutes only a small part of the application's execution time.*

The security guarantees provided by the IFS technique depend on the choice of critical data in an application. We demonstrate in [160] an automatic technique to choose critical data by systematically enumerating all possible attacks that may be launched by an insider. In order to deploy the technique in [160], the user has to provide generic specifications about the attacker's goal (for example, to log in with the wrong password) and the system will automatically identify the critical variables in the application to be protected in order to foil the attacker's goals). However, for the applications considered in this chapter, the critical data was chosen manually based on our understanding of the application's source code[26]. We show experimentally that the technique can detect both insider attacks and external attacks that impact the critical data in each application.

The chapter makes the following contributions:

- Proposes a novel technique to protect critical data in applications from insider attacks through the concept of Information-Flow Signatures (IFS)
- Leverages and enhances existing static analysis techniques to extract the backward slices of critical variables and instrument the instructions in the slice to derive the IFS of the application (Section 4).
- Analyzes the efficacy of the proposed technique against both generic and targeted attacks (Section 5). Also, discusses the effect of approximations (made in the prototype) on the technique's effectiveness.
- Evaluates the performance overheads of the IFS technique for the benchmark applications and the resilience of the applications protected with the IFS technique to attacks (Section 7).
- Proves the efficacy of the IFS technique in detecting insider attacks, and shows that the IFS technique detects all external attacks impacting critical data that are also detected by existing techniques (see Section 7.10).

[26] This is because the formal technique could not be scaled to the applications presented in this chapter. This has to do with the limitations of the model-checker used in the earlier work, and is orthogonal to the technique for identifying critical data.

7.2 *RELATED WORK*

Insider attacks are attacks in which a privileged entitiy ("insider") abuses its privilege to attack the system. Insider attacks can be mounted at the hardware, virtual machine, operating-system, and network levels. Table 31 provides a brief overview of techniques deployed at each level to foil insider attacks. Each technique ensures 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. This is because an application-level insider does not need to change the hardware, operating system, virtual machine or network in order to launch its attack[27].

Table 31: Insider attacks at different layers of the system stack

Layer	Techniques	Comments
Hardware	King et al. [161] Alkabani et. al. [162]	These techniques consider malicious backdoors in hardware either at the design stage or at the synthesis stage. At the design stage [161], the problem is similar to the application-level insider attack problem (and there is no known solution). At the synthesis stage [162], the problem is similar to tampering with the application's executable file and can be alleviated by embedding hidden keys in the synthesized netlist for comparison with the original netlist.
Virtual Machine	King et al [163], Rutkowska [164]	The insider controls the boot-process prior to the loading of the Operating System (OS) and Virtual Machine (VM) and loads their own malicious VM in order to host the OS/VM [163]. From the malicious VM, the insider can wreak considerable damage totally transparent to the operating system or application. Later work has shown that it is possible to launch the attack even after the OS has finished loading (through hardware virtualization hooks) [164]. The malicious VM can be detected through timing measurements made from the guest VM/OS [165]
Network	Upadhyaya et al. [166], Sheyner et al., Philip and Swiler [132, 134]	The attacker is assumed to control one or more nodes in the network through which it is assumed that the attacker tries to launch attacks on other nodes. A structure called the attack graph is used to represent the possible malicious actions of the attacker [132, 134]. These attacks can be detected by enhancing Intrusion Detection Systems (IDS) to look for anomalous patterns of traffic within the network [166] that are representative of a network-level insider.
Operating System	Rootkit detection [167, 168]	Rootkit detection looks for anomalous data-structures or memory access patterns of the OS to determine if a malicious module has hooked into the OS (through an existing vulnerability). These techniques are based on knowledge of the operating systems' state (or some approximation of it) prior to the infection by the root-kit.
	Micro-kernel based OS [169, 170]	Structured the operating system into multiple, non-overlapping services. Each service runs in its own privileged compartment, and communicates with other services through a thin layer called the micro-kernel. So even if one service is compromised, the other services are not.

[27] This is analogous to how reliability techniques at lower-layers of the system stack does not obviate the need for application-level protection. This is because errors that occur in the application are not detected by lower-level techniques such as ECC in memory.

The rest of this section discusses security techniques deployed at the application level, which 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), and (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 from corruption due to errors and security attacks such as *Samurai* and *redundant data diversity*.

External security techniques such as memory safety checking [23, 171] are designed to protect applications from memory-corruption attacks (e.g. buffer-overflow, format string). This class of techniques is not effective for thwarting insider attacks as insiders do not need to exploit memory-corruption vulnerabilities in the application. Taintedness detection techniques [25-27] prevent application input from influencing high-integrity data in applications such as pointers. However, insiders do not need to use inputs to influence security-critical data as they are 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.

Security techniques such as randomization [28, 172] attempt to obscure a program's layout or instruction set from attackers. However, randomization can be bypassed by an insider who is itself subject to the same randomization as the application. For example, a malicious module in an application that is subject to address-space randomization (ASR) can calculate the absolute address of a stack variable in a different function by examining the addresses of its local variables, and adding a fixed offset to them. Unlike in the case of an external attacker who requires multiple attempts to bypass the randomization [31, 173], an insider can bypass it in a single attempt.

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 techniques [174, 175]. These techniques assume that the attacker seizes control of the application by executing unwanted or malicious system calls such as *exec,* or by skipping existing system calls, e.g. *seteuid*. This is because system-calls provide a conduit to attack other applications executing on the same system as well as the Operating System (OS) itself. However, an application-level insider's goal is to subvert the execution of the attacked application, and not necessarily attack other applications or the OS. Hence, system-call based detection techniques will not protect against insiders who overwrite the attacked application's data or control without launching system calls.

Techniques such as oblivious hashing [176, 177] and code attestation [178] detect malicious modifications of the application's 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 its distribution (as considered in this chapter). A technique that offers limited protection from insider attacks is remote audit [179], which ensures that the application's code is not skipped at runtime. However, remote audit does not protect against malicious modifications of the application's data by an application-level insider.

The only known technique that can effectively thwart application-level insider attacks is privilege separation [180]. 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 a trusted module's address space, unless the trusted module explicitly shared the data with the untrusted module (through an IPC call). However, privilege separation incurs overheads of up to 50 % when deployed in real applications [181] (measured as a fraction of the entire application's execution time, not just the protected module's 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 [182] and redundant data diversity [183] also protect critical data in an application from accidental and malicious corruption respectively. However, they both require the programmer to manually identify read/write operations on the critical data, which can be cumbersome. Further, Samurai only protects against corruption of critical data on the heap, and not for critical data on the stack or registers. Redundant data diversity requires replicating the entire process and executing it in lock-step, even though the critical data may constitute only a small portion of the application's 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 high performance overheads. *The question we ask in the chapter is "Can we protect the integrity of critical data from insider attacks with low performance overheads and minimal intervention from the programmer?"*

7.3 ATTACK MODEL

While the focus of this chapter is on insider attacks, the attack model also considers external attacks on the critical data of the application. This is because in addition to overwriting the critical data by itself, an insider can also plant a memory corruption vulnerability in the application which will be exploited by an external attacker to overwrite the security critical data. Hence, it is important to consider both external attacks and insider attacks on the critical data, as insider attacks are a super-set of external memory-corruption attacks on the application.

In the case of external attacks, we assume that the attacker exploits a memory corruption vulnerability (e.g., buffer overflow, format string) to overwrite critical data either directly or indirectly. A direct over-write means that the compromised instruction overwrites the security critical data through a pointer. An indirect over-write means that the compromised instruction overwrites a data element that influences the critical data through a data- or control- dependence. In both cases, the net effect is to influence the value of the critical data in such a way so as to benefit the attacker. The attacker may also launch system calls on the program's behalf to impact the critical data[28]. However, the attacker is prevented from launching new processes through the system call interface, as such attacks would be detected by OS-level checking mechanisms.

In the case of insider attacks, we assume that an attacker can corrupt any program data or change the program's control-flow during calls to untrusted code that is controlled by the attacker. From within the untrusted code, the attacker can modify the value of any location in the stack, heap or processor registers in order to influence the critical variable. The attacker can also modify the return address or a function frame pointer on the application stack to force the program to return to a different address than the intended one. Note that an insider attack may be possible even if the application does not have any memory corruption vulnerabilities.

We do not assume that the source code of un-trusted third-party functions is available for analysis – however, it is assumed that the source code of all trusted modules is available. We also assume that the attacker cannot modify the application's code once it is loaded into memory. This is reasonable as in many systems the code segment is marked read-only after the program is loaded (unless the program is self-modifying).

7.4 APPROACH AND ALGORITHM

This chapter focuses on protecting the integrity of critical data from application-level insider attacks. Critical data is defined as any variable or memory object which, if

[28] We assume that the attacker cannot infiltrate the Operating System (OS) by executing the system call and exploiting an OS vulnerability.

corrupted by an attacker can lead to security compromise of the application. In the proposed approach, the portion of the program that manipulates the critical data (i.e. the trusted module) is statically analyzed and instrumented with code to ensure that runtime modifications of the critical data follow the language-level semantics of the application. This corresponds to statically extracting the backward slice of the critical data, and ensuring 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 violates the established dependencies in the slice and hence, can be detected. The approach ensures that information-flow to the critical data is in accordance with the program's source code, hence the name *Information-Flow Signatures (IFS)*.

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

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 at runtime.
2. The instructions in the backward slices of the critical data are executed in the order of their occurrence along a specific set of acyclic paths in the program.
3. Either all the instructions in the backward slice along a specific path are executed at runtime, or no instruction along the path is executed.

Overall Algorithm: The algorithm for deriving and checking the IFS is split into four phases as follows:

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) as well as 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[29]. In the case of memory objects, the annotations are placed on the allocation sites in the program (i.e. calls to *malloc).*

In this chapter, we use the term critical variable to refer to both critical variables and memory objects.

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 the 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.

[29] We consider programs translated to Static Single Assignment (SSA) form, so each variable has a unique definition in the program.

3. Replace all uses of the instruction with the value returned by the encode operation within the function.

4. 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.

5. Generate the sequences of instructions for each function for each acyclic control-flow path in the function.

6. Add instrumentation functions at the beginning and end of function calls in order to push and pop the current state of the state machine on to a stack (see below).

Phase 2: Code Generation: (Carried out by custom programs)

1. 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.

2. Generate the encoding and decoding operations to check the data-values of the program as it executes (see below). Also, check the validity of the program's control-flow using the state machines derived above in step 1.

Phase 3: Runtime: (Carried out by the generated code)

1. Track the runtime path based on the state machines generated in step 1. If the path does not correspond to a valid path, raise an alarm and stop the program (see explanation below).

2. 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[30] [184].

Encoding/Decoding Operations: The encoding/decoding operations protect the data in the backward slice after it is produced (enforce invariant 1). The encoding operation used in this chapter is duplication, where the operand is stored in a special, protected memory location. During decoding, the original value is compared with the stored value of the operand. A mismatch indicates that the operand has been tampered with, i.e., an attack. We assume that the attacker cannot modify the values stored by the encoding operations (as they are stored in protected memory).

It is possible to incorporate more advanced encoding functions such as checksums or even encryption to provide stronger protection, constrained by the incurred performance overheads. The above algorithm is orthogonal to the mechanism for protecting encoded operands.

[30] The backward slice was used to *recompute* the value of selected program variables to check if they have been corrupted by an error.

State Machines: The state machines track the sequence of calls to the encoding functions and check if the program follows valid control-flow. 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's 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).

7.5 EXAMPLE CODE AND ATTACKS

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.

Consider the *sys_auth_password* function shown in Figure 55(a). The function accepts an *authctxt* data-structure and a password variable, and checks if the user password matches the password stored in the *authctxt* structure. If the passwords match, the function returns the value 1 and the user is authenticated by the system (not shown). The function also encrypts the user password prior to comparing it with the system password.

Critical Data: 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 invalid user-name or password being given. Therefore, we designate the value returned by the function as the critical variable (i.e., the *authenticated* variable defined in line 6). The value returned in line 3 is not considered critical as it is a compile-time constant.

```
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

Figure 55: (a) Example code fragment from SSH program and (b) Functions called from within the code fragment and their roles

Library Functions: The *sys_auth_passwd* function in Figure 55(a) calls four other functions, namely *shadow_pw, xcrypt, strcmp and log_user_action.* The functionality provided by each of these functions is outlined in Figure 55(b). Of the four functions, the first three are trusted (secure), because they manipulate the critical data and hence, are part of the backward slice. We assume that the source code of the trusted functions is available for analysis.

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

(1) **External attack:** We assume that the *log_user_action* function contains a format-string vulnerability, i.e., it invokes *printf*() using the user-name directly as the first

argument without specifying a format-string argument. An external attacker exploits this vulnerability to overwrite any memory location in the program.

(2) **Internal attack:** We assume that the *log_user_action* function is supplied by a malicious attacker as part of an external library whose source code is not available for the analysis. The attacker can overwrite any location in memory or registers and change the program's control-flow from within the function. This is so *even if the application contains no memory-corruption vulnerabilities in and of itself*.

7.6 IFS IMPLEMENTATION EXAMPLE

This section illustrates the operation of the IFS algorithm given in Section 7.4. Figure 56 shows the code in Figure 55(a) instrumented with the encoding and decoding functions. Recall that the critical variable chosen is *authenticated.* The backward slice of the *authenticated* variable corresponds to the instructions in the program that can potentially influence the variable's value. Since we consider only intra-procedural slices, we are limited to the instructions in the *sys_auth_passwd* function. These are shown in green (or light gray) in Figure 56.

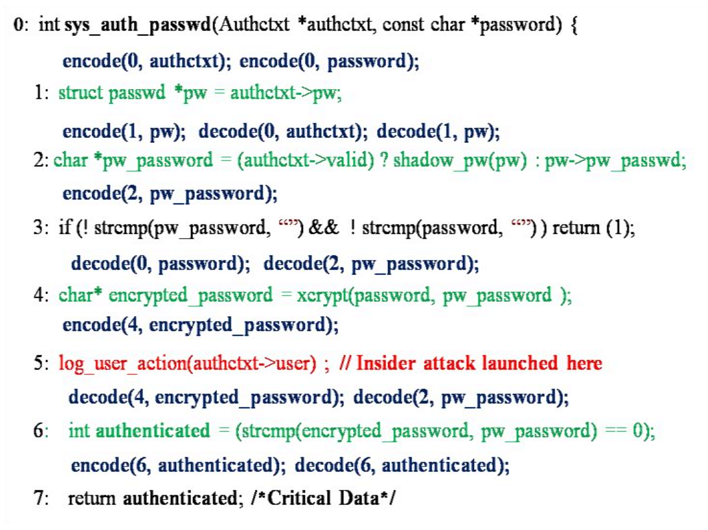
The encoding and decoding operations are represented as functions. In Figure 56, the encoding functions (*encode)* are inserted immediately after the statement that produces a value within the backward slice, while the decoding functions (*decode*) are inserted immediately before the statement within the slice that uses the original value of the variable. Each encoding or decoding function is passed as arguments: (1) the number of the program statement in the slice and (2) the value produced by the program statement.

The functions are marked as *volatile* to prevent the compiler from reordering them during its optimization passes.

The state machines derived for the function in Figure 56 are shown in Figure 57. The accepting states of the state machine are states 2 and 6 (shown in green or as light colored ovals). In the figure, the states in the state machines correspond one-to-one to the encoding calls in the program and are labeled with the encode call arguments.

From Figure 56 and Figure 57, it should be apparent that under normal (attack-free) operation of the program, every value that is encoded has a corresponding decode function before its use in the function (and vice-versa). Further, the state machine reaches an accepting state before the function exits. Therefore, in normal operation, the instrumentation functions do not raise an alarm or perform a false-detection.

We now consider the operation of the program in Figure 56 under attacks. Recall that the goal of the attacker is to overwrite the value of the *authenticated* variable in the program. We first consider generic attacks, where the attacker is unaware that the IFS scheme is deployed, and then targeted attacks where the attacker is aware of the deployment of the IFS technique and actively tries to evade detection by the inserted checks.



```
0:  int sys_auth_passwd(Authctxt *authctxt, const char *password) {
      encode(0, authctxt); encode(0, password);
   1:  struct passwd *pw = authctxt->pw;
      encode(1, pw);  decode(0, authctxt);  decode(1, pw);
   2: char *pw_password = (authctxt->valid) ? shadow_pw(pw) : pw->pw_passwd;
      encode(2, pw_password);
   3:  if (! strcmp(pw_password, "") && ! strcmp(password, "") ) return (1);
       decode(0, password);  decode(2, pw_password);
   4:  char* encrypted_password = xcrypt(password, pw_password );
      encode(4, encrypted_password);
   5:  log_user_action(authctxt->user) ;  // Insider attack launched here
       decode(4, encrypted_password); decode(2, pw_password);
   6:   int authenticated = (strcmp(encrypted_password, pw_password) == 0);
       encode(6, authenticated); decode(6, authenticated);
   7:   return authenticated; /*Critical Data*/
```

Figure 56: OpenSSH example with instrumentation added by IFS technique

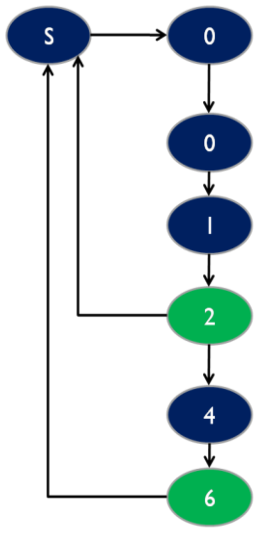


Figure 57: State machines derived by IFS

7.6.1 Generic Attacks

Generic attacks are those in which the attacker is unaware that the IFS scheme is being deployed. Section 7.10 provides a generic proof of the efficacy of the IFS technique for each class of attacks considered in this section.

External attacks: An external attacker attempts to exploit the format string vulnerability in the *log_user_action* function by crafting an appropriate input to the program. We consider three kinds of external attacks as follows:

***(E1) 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 (which does not belong to the backward slice of the critical variable). However, once the system call is executed, any attempt by the attacker to overwrite the critical data from within the system call will be detected. For example, if the attacker tries to execute a file read call with the address of the critical variable (*authenticated)* as an argument, the IFS technique detects this as an attack and halts the program.

(E2) Attacker overwrites function-pointers/return address to impact the program's critical data: Let us assume that the attacker overwrites the return address on the stack from within the *log_user_action* function. The goal of the attacker here is to make the function return directly to line 7, in effect bypassing the initialization of the authenticated variable in line 6. Let us further assume that the *authenticated* variable is assigned to a non-zero, value[31] 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 skipping of line 6 results in a control-flow pattern that does not correspond to any valid path within the backward slice of the critical variable *authenticated*. Hence, the state machine in Figure 57 is not in an accepting state when the end of the function is reached (as the corresponding *encode* operation is also skipped[32]). Consequently, any attempt to use the returned value in the called function results in a failure of the *decode* operation and the attack is detected.

[31] This is fairly common for local variables in C, which are assigned to arbitrary values prior to their initialization.
[32] Section 7.9.2 presents an in-depth analysis of this particular attack, and explains how the attack is detected by the IFS technique.

(E3) Attacker overwrites the critical variable authenticated or any variable in the backward slice- Assume that the attacker chooses a format string 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 1, which is the attacker's goal. The IFS technique detects this attack because all variables within the backward slice are in an encoded form prior to the call to the *log_user_action* function. Overwriting the variable results in an error during the decode operation prior to its use.

Insider Attacks: In the case of insider attacks, we assume that the function *log_user_action* is under the control of the attacker. In this case, the function is considered to be a black box in the sense that it can overwrite any set of variables in the program and jump anywhere in the program (recall that its source code may not be available). We consider three cases, depending on the arguments to the *log_user_action* function.

(I1) The log_user_action() function modifies the contents of encrypted_password*:* This is not allowed according to the semantics of the *sys_auth_passwd* since the *log_user_action* is passed the *authctxt->user* pointer which can never point to any variable in the backward slice of *authenticated* (as determined by the compiler's pointer analysis[33]). The attack is detected by the encoding and decoding operations inserted by the IFS technique.

The next two attacks are not possible for the example in Figure 55(a). Nonetheless, we consider them to illustrate possible insider attacks that may be launched on code that is slightly different from the one in Figure 55(a).

(I2) The log_user_action function modifies the contents of encrypted_password, but it is only allowed to modify the contents of the password variable: In addition to passing the user-name to the *log_user_action* function, assume that the *sys_auth_passwd* function also passes a pointer to the user password (not the system password). Since the password variable has already been used in the backward slice before the call to the *log_user_action* function, the programmer may think that the call is harmless. However, it is possible for the *log_user_action* function to maliciously overwrite the contents of the *encrypted_passwd* or *pw_passwd* strings to make the strings match. This will also be detected by the IFS technique, as only the arguments to the *log_user_action* function, namely, the *authctxt->user* and *password* variables, are decoded prior to the function call. Overwrites of any other variable in the backward slice is detected by the IFS technique.

(I3) The log_user_action function is allowed to modify encrypted_passwd by virtue of being passed a pointer to the encrypted_password variable: The attack is not detected

[33] It is assumed that the function cannot access global variables unless they are marked as extern, in which case they are treated as function arguments.

by the IFS technique, as the malicious function belongs to the backward slice of the critical variable. Hence, a malicious update is indistinguishable from a legitimate update through the pointer argument of the function. However, a warning can be raised at compile time whenever a pointer to a variable in the backward slice of the critical variable is passed to an untrusted function (i.e., any function whose source-code is not available). This is outside the scope of the current IFS technique.

7.6.2 Targeted Attacks

Targeted attacks are those in which the attacker is aware of the IFS scheme and actively tries to defeat the protection. We assume that the attacker tries to avoid detection as much as possible.

***Attack 1:* Attacker corrupts a data value produced in the backward slice and calls the corresponding encoding function with the corrupted value.**

The attack is detected if the corruption occurs after the value has been encoded. This is because attempting to call an encoding function that has just been called does not correspond to a valid state transition, unless the function repeats in the state machine. Even in the case that the function repeats, the attacker would need to skip the subsequent calls of the encoding functions in the code. In the example in Figure 56, assume that the attacker corrupts the value of *pw_password* after the call to the encoding function *encode_2* and calls the *encode_2* function one more time to re-encode the corrupted value. The attack is detected because the state machine in Figure 57 does not have a transition from state 2 for the *encode_2* call. In order to evade detection, the attacker has to call all the *encode* functions in the state machine following state 2 in Figure 57 or reset the state machine and make it transition through all subsequent states until state 2 is reached. At that point, the call to *encode_2* becomes valid again and the attacker escapes detection. However, this increases the chances of the attack being detected by other means e.g., timing-based techniques.

***Attack 2:* Attacker corrupts a data value and manages to bypass the call to the decoding function prior to using the decoded operand within the backward slice of a critical variable**

The attack is detected at the next use of the operand in the program by the call to the decoding function. To get away without being detected, the attacker needs to bypass **all** calls to the decoding functions before the variable is used in the backward slice. He/she also needs to ensure that in bypassing the calls to the decoding functions, the encoding functions are not bypassed, as this is detected by the state machine transitioning to an invalid state or not being in an accepting state prior to the function’s return.

In the example in Figure 56, assume that the attacker corrupts the value of *authenticated* just before line 7 and bypasses all program statements that subsequently use this operand, including calls to the decode function. The attacker would be able to get away undetected because no checks are performed on the corrupted value. However, in carrying out the attack, the attacker cannot execute any of the code that subsequently uses the value of the *authenticated* variable. This may result in large-scale deviations from the control-flow of the original program and can be detected by alternate techniques, e.g. control-flow checking [185].

Consider an attack where the attacker changed the value of *pw_password* after it was encoded and managed to bypass the call to *decode_2* in line 3. The attack is detected in line 5 when the call to *decode_2* is encountered again. Bypassing this second call affects the control-flow of the program and results in the call *encode_6* being skipped in line 6. This attack is detected because the corresponding state machine is not in an accepting state when the function exits i.e., the path is an invalid program path.

Attack 3: **The attacker does a replay attack i.e. he/she executes the program, observes a sequence of valid transitions and replaces a run of the state machine with the observed sequence of calls to *encode*.**

This attack is detected by a duplication-based encoding scheme (assumed in this chapter), but may bypass other static encoding schemes. In order to detect the attack, the encoding function must be based on a random seed that is chosen at application load time, i.e., each invocation of the program produces a different encoding based on the chosen seed. The randomization ensures that the attacker is not be able to replace a sequence of encoding calls with one from a different instance of the program's execution (unless the seed is the same).

7.7 DISCUSSION

Any static analysis technique, including the IFS, must necessarily approximate the behavior of the program in order to be practically realizable. Typically, the analysis technique over-approximates the behavior of the program, which in turn leads to false-negatives i.e., missed attacks. This section analyzes the effects of approximations made by the IFS technique on its security guarantees. Unlike the previous section, we do not consider specific attacks mounted by the attacker, but frame the discussion in a more general context.

7.7.1.1 Effect of 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. However, there are two ways of mitigating the impact of intra-procedural slicing as follows:

- **Function inlining:** This involves inlining the body of the called function into the caller, so that the caller and the callee are treated as one function. This has practical limitations in terms of handling large functions and recursive calls in the code, but does not require programmer intervention beyond specifying the critical variables in the program.
- **Choosing critical variables in each function:** The user can choose variables in each function such that the entire backward slice is covered. This requires understanding of the dependencies across functions and specifying the critical variables in each function.

The above problems can be solved by considering inter-procedural slices, i.e. context-sensitive dependence analysis. The issue of context-sensitivity is an important one that needs to be addressed by static analysis. However, the issue of context-sensitivity is orthogonal to the IFS technique and is not considered in this chapter.

7.7.1.2 Effect of Acyclic Paths

The IFS technique extracts acyclic control paths in the application and converts the sequences of calls to the encoding functions on the paths into a 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. This may be exploited by an attacker who may make the loop execute for fewer or greater number of iterations than allowed by the source program (the attacker's intent may be to bypass security checks performed in loop iterations, or to introduce a semantic violation). Detecting such attacks requires timing/semantic information about program loops or including the loop-counter in the backward slice.

7.7.1.3 Pointer Approximations

When an instruction accesses memory through a pointer variable, the compiler needs to compute the set of locations read/written by the instruction. This set is typically an over-approximation of the set of locations read/written to by the instruction at runtime, and is known as the points-to set of the instruction [101]. The proposed technique relies on the compiler's inferred points-to set for extracting the backward slices. An attacker can replace the memory address used in an instruction (belonging to the backward slice) with another address in the instruction's points-to-set. The attack will be detected only if either the replacement is carried out from an instruction that is not in the backward slice or the instruction performing the replacement causes the state machine to follow invalid paths or perform illegal transitions.

7.7.1.4 Window of Vulnerability between Encoding Function Calls and Instructions

Any software-based checking scheme will have a window of vulnerability which is the time between carrying out the security check and actually carrying out the privileged operation. If an attack is mounted during this window, it may escape undetected (also known as TOCTTOU vulnerabilities). In the implementation of the IFS scheme, the encoding functions are introduced immediately after the instructions producing the operand, and the decoding functions are introduced immediately before the instructions using the operand. This ensures that the window of vulnerability of the operand is as narrow as possible. In reality, the introduction of encode and decode functions is done at the compiler's intermediate code level, and the code generator may introduce multiple instructions between the encoding/decoding calls and the operands they protect. Furthermore, the compiler may reorder the instructions around the calls to the encoding/decoding functions, leading to dilation of the vulnerability window. Marking the functions as volatile as done by the technique prevents their reordering, but does not alleviate the code-generation problem.

7.7.1.5 Legal but Invalid Control-flow Paths in the Program

Finally, the technique only checks if a sequence of calls to the encoding function is a legal one in the program's control-flow graph. It is possible for an attacker to replace a sequence of encoding calls with a legal but invalid sequence in the program. However, to avoid detection the sequence must be replaced in its entirety, which may be much harder to achieve for the attacker.

Summary: The approximations made by the compiler can have a non-negligible impact on the security guarantees provide by the IFS technique. This is so for any security technique that relies on static analysis, for example the WIT technique [171]. However, the IFS technique differs from these other techniques in two significant ways. First, only the approximations made by the compiler that pertain to the backward slice(s) of the critical variable(s) affect the security guarantees provide by the technique. Secondly, the technique does not need to analyze the code of modules that are not allowed to modify the critical data in the program. This simplifies the code base that must be analyzed statically and hence, the resulting code can be analyzed with higher accuracy.

7.8 EXPERIMENTAL SETUP

Implementation: The IFS technique has been implemented as a new pass in the LLVM compiler [99] called the IFS pass. The IFS pass is executed after the lexing, parsing and intermediate representation phases of the compiler, but prior to the register-allocation and code-generation phases. The pass extracts the backward slices of critical data and assigns a unique identifier to each slice instruction. The identifiers of the instructions along different paths in the function are written to a text file. The text file is parsed by Python scripts that generate custom C code to implement the state-machines[34]. The generated C

[34] LLVM's intermediate representation is strongly-typed, hence the program types of the instructions are taken into account when generating custom code.

code is linked with runtime libraries for the *encode* and *decode* operations. The IFS pass consists of about 1500 lines of C++ code and the scripts constitute about 500 lines of Python code. The runtime libraries constitute less than 100 lines of uncommented C code.

Benchmarks: We demonstrate the IFS technique on server applications. This is because server applications are (1) typically executed with super-user privileges, which makes them extremely attractive targets for attackers, (2) 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) 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 [137]

(2) **WuFTP:** Implementation of the File Transfer Protocol (FTP), consisting of over 25000 lines of C code [186].

(3) **NullHTTP:** a small and efficient multithreaded HTTP server . Consists of about 2500 lines of C code [187].

Modules: In the case of the OpenSSH and WuFTP applications, we focus on the security-critical modules of the application, whereas for NullHTTP, we protect the entire application. For OpenSSH, we protect the authentication module while for WuFTP, we protect the user login module (includes authentication and permission checking). In order to facilitate the analysis by the IFS technique, we extract these modules as standalone programs called stubs. Each stub can be executed independently of the main application, and completely encapsulates the security-critical functionality of the module. The stub for OpenSSH consists of about 250 lines of code, while for WuFTP, the stub consists of about 500 lines of code. The overheads for these two applications are reported in terms of the execution time of the stubs. *Note that the stubs have a much smaller execution time compared to the entire application, and hence better represent the performance overhead of the IFS technique.* (The overheads of the IFS technique were too low to measure in the context of the entire OpenSSH and WuFTP programs). For NullHTTP, we report overheads relative to the application considered as a whole (due to its relatively small size).

Critical Variables: In each of the target applications, we choose 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's identity in order to access the user's files/directories and perform malicious activities so that the user is blamed.
- **NullHTTP:** Attacks that either modify the client request or the response in order to send malicious or unintended content to the user.

Table 32 shows the critical variables in each application that are chosen (manually) in each application.

Table 32: Critical variables in the applications and the rationale for choosing the variables as critical

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

For the OpenSSH and WuFTP stubs, the LLVM compiler [99] is able to aggressively inline the functions into a single function. Hence, the backward slice of the critical variable encompasses all instructions in the stub. In the NullHTTP application, the LLVM compiler inlines all the functions related to processing a client's request into a single function (*htloop*). The entire backward slices of the critical data are contained entirely within this function. Table 33shows the static characteristics of the inlined function containing the critical variables in each application.

Table 33: Static characteristics of the instrumentation in each application

Application	**Total number of assembly instructions in the function**	**Number of encode calls in the function**	**Number of decode calls in the function**	**Number of acyclic control paths**
OpenSSH	2430	252	296	103
WuFTP	800	7	6	2
NullHTTP	5594	404	543	198

Performance Measurements: In order to measure the performance overhead of the IFS technique, we executed both the original, non-instrumented program and the instrumented version of the program. The measurements are conducted using the *gettimeofday()* system call on a 2.0 Ghz Pentium 4 Linux system (2 GB RAM).

Resilience Measurements: In order to evaluate the resilience of the technique to attacks, we mounted both insider attacks and external attacks on the applications protected with the IFS technique. The insider attacks consisted of replacing a specific function (which does not belong to the backward slice of the critical data) with a malicious surrogate function. The external attacks consisted of planting memory corruption vulnerabilities in the program and exploiting them through specially constructed inputs to impact the critical data in the application.

7.9 EXPERIMENTAL RESULTS

This section presents the results of the experiments evaluating the performance and resilience of the IFS technique. It also examines the reasons for the overheads.

7.9.1 Performance Overheads

OpenSSH Authentication Module: In order to evaluate the performance overheads introduced by the IFS technique, the authentication stub is executed with three inputs, consisting of (1) wrong user-name, (2) correct user-name, correct password and, (3) correct user-name, wrong password. In each case, the execution time of the instrumented program is compared to the execution time of the original, non-instrumented version for a given input. The results are shown in Table 34. As observed in the table, the performance overhead ranges from about 47% to 95%, across the inputs, with a mean value of 79 %.

Table 34: Execution times of SSH authentication stub

Input	Original (uS)	Instrumented (uS)	Overheads(%)
1	155	228	47
2	159	306	95
3	159	310	95
Mean			79

WuFTP Login Module: We consider three inputs for measuring the performance overhead of the instrumented stub, (1) attempt to log in with the wrong username (username must be ftp) (2) log in with correct username and wrong password, and (3) log in with correct username and correct password. Table 35 shows the execution time overheads for each of the inputs. As seen from the table, the performance overhead of the instrumented application ranges from about 4 % to 11 %, with a mean value of 7.5 %.

Table 35: Execution times of FTP login stub

Input	Original (uS)	Instrumented (uS)	Overhead(%)
1	45	50	7
2	90	90	4
3	48	53	11
Mean			7.5

NullHTTP Application: In order to evaluate the performance overhead introduced by the NullHtp server application, we developed a multi-threaded client program (in C) to request web-pages from the server using the HTTP POST command. The NullHTTP server is inherently multi-threaded and hence throughput is a more meaningful measure of performance overhead than latency. The client program spawns multiple threads, each of which sends an HTTP request to the server requesting a given webpage. The NullHTTP server in turn spawns a new thread to handle each incoming connection from the client. We measure the total time at the client to successfully complete execution of all spawned threads. The client executes on the same machine as the server in order to eliminate the effect of network latency in the measurements (as far as possible).

Table 36 summarizes the results for the NullHTTP program. For a single-threaded client, the total time taken for satisfying the request is approximately twice as much for the instrumented version compared to the original, non-instrumented version. As the number of threads in the client increase, so do the times taken for processing the requests at the server due to the increased workload. However, the overheads steadily decrease from 86 % to 0 % as the number of threads increase from 1 to 25. In the single-threaded case, the performance overhead is dominated by latency, while in the multi-threaded case, the overhead is dominated by the throughput. Thus, the IFS technique has a substantial impact on the latency of the NullHTTP server, but small impact on the throughput.

Table 36: Execution times of NullHTTP program

Threads	Original (uS)	Instrumented (uS)	Overhead (%)
1	3052	5674	86
5	20436	21067	16
10	39057	42324	10
25	100177	101916	0

Discussion: From the results above, it can be concluded that the performance overheads are highly dependent on the nature of the application and the choice of critical data. For example, in the SSH stub application, the IFS technique introduces an overhead of nearly 100%, while for FTP, the overhead is less than 10%. The reason for this difference is that in SSH, the backward slice of the critical variable comprises of about 10% of the instructions in the program, while in FTP it comprises less than 1 % of the instructions (see Table 33). Consequently, SSH has a higher number of *encode* and *decode* calls in the program compared to FTP. Further, the number of control-paths that must be tracked by the state machines is much higher in SSH than FTP (100 versus 2). As a result, the state machine for the SSH application has many more states compared to that of the FTP application, and hence incurs higher overhead for tracking state transitions.

The NullHTTP application's characteristics mirror those of the SSH application in terms of the number of instructions in its backward slice (also 10%). Consequently, the performance overheads incurred by the IFS technique for the NullHTTP application (in the single-threaded case) is close to 90 %. However, the overhead can be masked in concurrent requests due to the stateless nature of the HTTP protocol. This in turn allows the threads to be executed in parallel with each other with little or no data sharing among them.

A cursory glance at the results may lead the unsuspecting reader to think that the entire application's execution time is slowed down by a factor of two for the SSH application. However, this is not the case as the above overheads are reported as a fraction of the execution time of the authentication module, which encompasses only a very small fraction of the execution overhead of the application (about 1% or less in a typical user session). Therefore, when evaluated in the context of the entire application, the IFS technique incurs negligible performance overhead for the SSH application. The same reasoning applies for the FTP application. In the case of the NullHTTP application, while a single request may be slowed down by a factor of two, the overall throughput of the server is minimally impacted. Further, the request processing time at the server is only a small fraction of the overall latency experienced by a typical HTTP request which is often routed through multiple network hops.

Analysis of the sources of performance overhead of the instrumentation: In order to understand how to reduce the performance overheads of the IFS technique, it is important to understand the contribution of each instrumentation component added by the IFS technique. The contribution of a component is measured by commenting out the component from the code generated by the IFS technique, and measuring the execution time with and without the component. The difference in the execution times yields the performance overhead due to the component. The following components are considered in the study:

- **Encode:** Execution of the encoding functions to encode variables within the backward slice
- **Decode:** Execution of the decoding functions to decode encoded variables prior to their use
- **Transitions:** Performing transitions in the state machines depending on the encoding function executed.
- **Checks:** Checking if the state machine is in an accepting state before returns of instrumented functions
- **Memory:** Encoding and decoding of memory objects **(**immediately after stores and immediately before loads)
- **Other:** Additional instrumentation added by the compiler for support (e.g., stack handling, error reporting)

We consider the OpenSSH application as it incurred the highest performance overheads among the target applications. The performance overhead of each component of the instrumentation is as follows.

The calls to the decoding functions have the maximum contribution to the overhead (28%), as these are called 802 times in the program (not to be confused with the static counts in Table 33). This is followed by encoding and decoding of memory operands (21%), as these may necessitate an extra load or store instruction. The state transitions constitute about 19% of the overhead, while the calls to the encode function brings another 15 % (this does not include the overhead of state-machine transitions). This is because the program calls the encoding function 620 times, and each call causes one or more state-machine transitions. The operation to check if a state machine is in an accepting state constitutes only 1% of the overhead, as it is called only at function exits (and there is only one function in the OpenSSH stub after inlining). The category, *other*, constitutes 12 % of the overhead due to the runtime support code added by the IFS pass. An additional 4 % overhead is unaccounted for due to measurement errors.

Techniques to reduce the performance overhead: Based on the above results, we see that the encoding and decoding operations constitute the highest overhead among the instrumentation components. The overhead can be alleviated by implementing the encoding and decoding operations in hardware. This will require provision of a high-speed hardware cache to store and retrieve the encoded values on demand. This functionality can also be retrofitted onto existing TPM modules in processors [188]. If implemented as a cache, it will incur nearly zero overheads, and the performance overhead of the technique can be reduced by nearly 65%. Further reduction in the performance overheads (by about 20%) can be achieved by using hardware to implement the state machines for tracking encoding calls in the program. However, this requires the hardware to be reconfigurable, as each application will have a unique set of state machines that must be configured into the hardware at application load time. The hardware module can be implemented as a part of the Reliability and Security Engine (RSE) [1], which is a hardware framework for executing application-level checks. This is a direction for future investigation.

7.9.2 Resilience Measurements

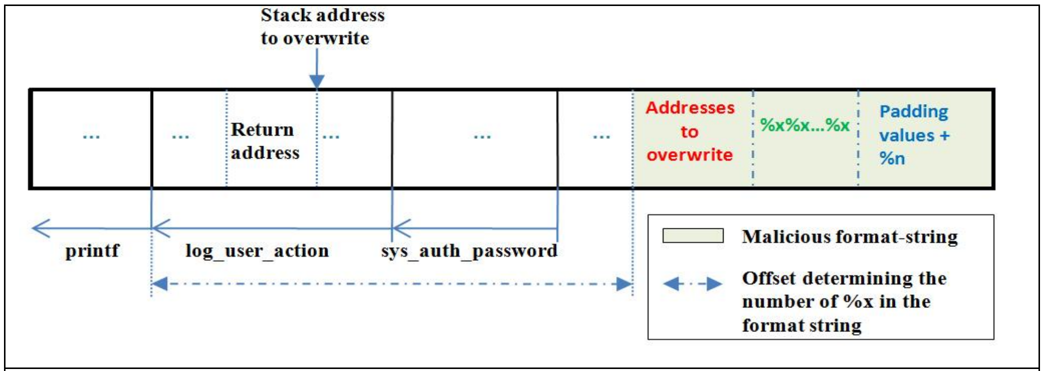
This section discusses the results of experimentally testing the resilience of the SSH stub application (the other applications are not discussed due to lack of space). We first consider the external attacks and then the insider attacks from Section 7.6.1. Recall that in both cases, the attacker's goal is to get the *sys_auth_passwd* function to return 1 even if the supplied username and password are not valid.

External Attack: As mentioned in Section 7.6.1, we assume that the external attacker exploits the format string vulnerability in the *log_user_action* function. The attacker has to supply a well crafted username containing malicious format strings along with an arbitrary password in order to overwrite either the pointer to the *encrypted_password* (E3) or the return address of *log_user_action* (E2). The methodology used to craft the format string is similar in both attacks. Due to the lack of space, only the control-flow attack (E2) is discussed below.

Figure 58(a) shows the stack configuration after *printf* is called within the vulnerable *log_user_action* function. The format string corresponding to the attack is shown in Figure 58(b). A typical malicious format string can be divided into three components.

The beginning of the string contains the addresses the attacker wants to overwrite. The second section of the string is generally composed with '%x' parameters in order to increment the internal stack pointer of the format function until it points to the beginning of our format string. The final part of the string contains the paddings and the '%n' parameters that allow us to write desired value[35] into chosen addresses. In order to mount the attack, the attacker needs to determine the following by running the program offline[36].

1. The offset from the bottom of *printf*' functions stack frame and the stored format string on the stack.
2. The address on the stack of the *log_user_action* function's return address.
3. The address of the instruction the attacker wants to jump to, i.e. the return statement of *sys_auth_passwd.*



```
\x6c\xe5\xff\xbf JUNK \x6d\xe5\xff\xbf JUNK \x6e\xe5\xff\xbf JUNK
\x6f\xe5\xff\xbf%x%x%x%x%x%x%x%x%x%x%x%272x%n%130x%n%47x%n%258x%n
```

Figure 58: (a) Stack layout after the call to printf during the attack and (b) Attacker-supplied format string

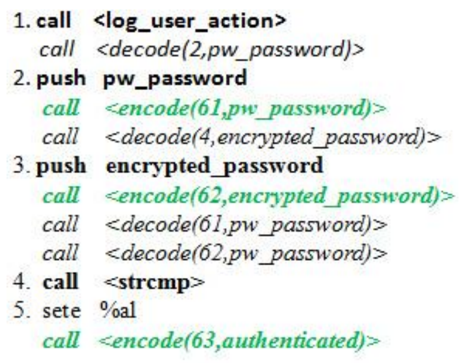
---

[35] We are able control the last significant byte of the targeted memory addresses. Therefore, in order to control the value of all four bytes of the targeted memory address the attacker needs to overwrite 4 consecutive addresses shifted by one byte each time.

[36] In order to ensure repeatability of the inputs, we assume that address-space randomization is disabled while carrying out the attacks. In practice, an attacker may achieve similar results by repeatedly attacking the application with different addresses or through information-leaks in the program.

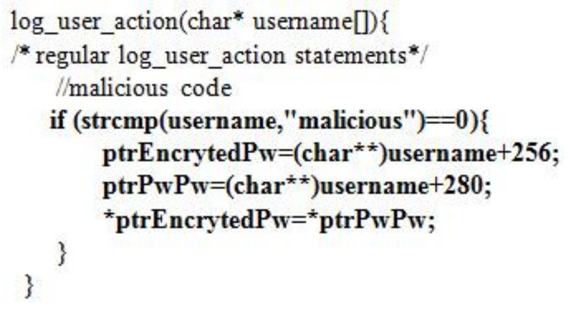
Detection: As explained in Section 7.6.1, the attacker bypasses the *strcmp* function in line 6 (and its corresponding *encode operation*) and jumps directly to the return statement in line 7 (Figure 56). This will cause the state machine to be in a non accepting state when *sys_auth_passwd* returns, and the attack will be detected.

Examining the code in Figure 56, it may be thought that the attacker can achieve her goal undetected if she jumps to the *encode(6,authenticated)* statement instead of jumping directly to the return statement. However, this is not the case because the instrumentation is done at the assembly level and an encoding function is inserted after each instruction in the backward slice. The earlier explanation in Section 7.6.1 had coalesced multiple encoding calls at the instruction-level into a single call for simplicity of explanation. Figure 59 shows the relevant assembly code of the instrumented *sys_auth_passwd* function to illustrate the above attack. The assembly code corresponds to statement 5 in Figure 56. The instrumentation is such that it is impossible to bypass the *strcmp* function without bypassing the encodings of its arguments *encode(61,pw_password)* and *encode (62,pw_password).*



```
1. call   <log_user_action>
   call   <decode(2,pw_password)>
2. push   pw_password
   call   <encode(61,pw_password)>
   call   <decode(4,encrypted_password)>
3. push   encrypted_password
   call   <encode(62,encrypted_password)>
   call   <decode(61,pw_password)>
   call   <decode(62,pw_password)>
4. call   <strcmp>
5. sete   %al
   call   <encode(63,authenticated)>
```

Figure 59 : Assembly code of the instrumented *sys_auth_passwd* function



```
log_user_action(char* username[]){
/* regular log_user_action statements*/
    //malicious code
    if (strcmp(username,"malicious")==0){
        ptrEncrytedPw=(char**)username+256;
        ptrPwPw=(char**)username+280;
        *ptrEncrytedPw=*ptrPwPw;
    }
 }
```

Figure 60: Source code of the malicious *log_user_action* function

Insider Attacks: Section 7.6.1 considers three kinds of insider attacks on the application. Due to the lack of space, we will only discuss attack I1 in detail. For illustration purposes, Figure 60 shows the source code of the malicious *log_user_action* function (remember that this code is not available to the IFS technique as *log_user_action* is an untrusted function). Since the pointers *encrypted_password* and *pw_password* are stored on the stack, it is possible for the malicious library function to corrupt their values by using a fixed offset from *log_user_action*'s argument: *username*. Supplying "malicious" as username and an arbitrary (but fixed length) password enables the insider to be authenticated as a legitimate user, thereby achieving the attacker's goal. In practice, this attack is likely to be more subtle as the attacker would try to hide their tracks more cleverly.

Detection: The attack is detected by the *decode(4,encrypted_password)* instruction (Figure 59) in the *sys_auth_passwd* function. The value of *encrypted_password* is checked prior to its use in the *strcmp* function in line 4 (the check is done by the *decode* operation). Any modifications of this value by the *log_user_action* will result in a deviation from the variable's value when it was encoded prior to calling the *log_user_action* function and the attack will be detected. An analogous argument can be made for corruptions of the *pw_password variable*.

7.10 PROOF OF EFFICACY OF THE IFS TECHNIQUE

This section provides a semi-formal proof of the efficacy of the IFS technique against both insider attacks as well as external memory corruption attacks. We show that the IFS technique can detect any attempt by an untrusted third-party module to overwrite critical data (independent of whether the module's source code is available) in violation of its expected behavior in the application. We also show that the IFS technique detects memory corruption attacks that result in execution of unwanted system-calls, violations of the application's control flow, or overwriting of security critical data in the program. For each external attack category, we show that the IFS technique detects all attacks that would be detected by state of the art security techniques.

We first discuss insider attacks and then discuss memory corruption attacks launched by an external attacker.

1. Attacks launched by an insider: In this case, we assume that an untrusted third-party module is loaded into the same address space as the application. The module may modify the application's data, change its control-flow or execute system calls on behalf of the application. We assume that (1) the module may not modify the application's code, (2) it cannot hook into any system calls made by the application, and (3) the IFS checks themselves cannot be bypassed en-masse (though the individual encode/decode operations may be bypassed).

A function call is legitimately allowed to modify a variable if and only if (1) The variable is declared as a global variable in the program, or (2) The function is passed a pointer that may potentially alias the variable, or (3) The variable is dynamically allocated on the heap and can be reached through a pointer passed to the function. An assignment of the variable to a function's return value is treated as a definition of the variable in the calling function rather than as a modification in the called function.

We consider three cases when an untrusted third-party function is invoked by the application. The cases correspond to whether the source code of the third-party function is available for analysis and whether the function is legitimately allowed to influence the critical data's value according to the C language semantics.

Case 1: The source code of the third-party function is NOT available AND the function is NOT allowed to legitimately modify any variable in the backward slice of the critical variable.

Proof: Since the function is not legitimately allowed to modify variables in the backward slice of critical data, the IFS technique will not insert calls to the encode operation after the function call to re-encode the modified data in the slice. If the function does attempt to modify any of the encoded data, it will violate the encoding, which will be detected when the modified data is decoded prior to its use (within the function). It is also possible for the function to modify the control-flow of its calling function by overwriting its return address. These will be detected by the state-machine transitions of the IFS if the modified control-flow impacts the critical data in any way.

Case 2: The source code of the third party function is NOT available AND the function IS allowed to legitimately modify one or more variables in the backward slice of the critical variable.

Proof: The function is legitimately allowed to modify one or more variables in the backward slice (as determined by the static analysis) and hence the variables will be decoded prior to the function call. After the function returns, the variables are re-encoded by the IFS technique. Any modification made by the function to the variables is reflected in the values used in the slice. Note that this is a potential security hazard as the function could perform unknown operations on the variables in the backward slice, thereby influencing the critical variable. However, only modifications to the variables that the function is legitimately allowed to modify are reflected in the program – all other modifications are detected by the IFS technique prior to their use in the program. Further, a compile-time warning is emitted if the backward slice includes a function for which the source code is not available, and the programmer must explicitly override the warning (after presumably vetting that the function is indeed doing the right thing for the values it is allowed to modify in the slice). Also, any modifications of the program's control-flow by the untrusted function are treated similar to violations of control-flow by external attackers and are hence detected by the IFS (provided the modifications impact the critical data either directly or indirectly).

In the above case, since the source code of the function is not available, the function as a whole is assumed to modify the backward slice variables that it is legitimately allowed to modify. In other words, the individual instructions that actually modify the variables are not bracketed with encode and decode operations. Hence, an attacker may be able to modify the data through a different instruction than the one that was not allowed to do so (within the function). However, the modification needs to be in the set of variables that the function is legitimately allowed to modify.

Case 3: The source code of the third-party function IS available, in which case it is straightforward to check if the function can legitimately modify one or more variables in the backward slice.

Proof: It is assumed that the function is trusted, since its source code is available, and hence it can be analyzed (statically) to determine the set of instructions in the function that are legitimately allowed to modify the data in the backward slice of the critical variable. In this case, there is no ambiguity about the data that the function is allowed to modify (subject to the usual sources of imprecision inherent in static analysis). However, the precision in the determination of the instructions that perform the modifications depends on whether the slicing is intra-procedural or inter-procedural. In the case of intra-procedural slicing, the entire function is assumed to modify the variables in the backward slice (that it is legitimately allowed to modify) and this is similar to case 2. In case the slicing is inter-procedural or if the function can be inlined into the calling function, then it is possible to identify the individual instructions in the function that write to the variables in the backward slice of the critical variable. The instructions can be bracketed by calls to the encode and decode operations to ensure that the window of vulnerability of the data in the slice is minimized. Thus, performing intra-procedural analysis weakens the security guarantees with respect to the instructions within the function, but does NOT weaken the security guarantees with respect to the effect of the function on the rest of the application.

2. **Memory corruption attacks:** These include attacks that do one or more of the following: (a) launch a system call on behalf of the application to execute another process, (b) change the control-flow of the program by executing the attacker's code as a result of overwriting function pointers/return addresses, or (c) overwrite security-critical data in the application either directly or indirectly. The IFS technique will detect all three cases provided they modify the critical data. We consider each case as follows.

(a) Attacks that launch system-calls on behalf of the application to overwrite the critical data:

The goal is to show that the IFS scheme will detect an anomalous sequence of system calls that ultimately influence the critical data (i.e. by writing to the data directly or indirectly) provided the attack is also detected by existing system-call based detection techniques[174] . In order to keep this discussion at a generic level, we consider an idealized system-call detection technique, which accepts a system-call sequence if and only if the sequence of system-calls corresponds to a valid path in the program (as determined through static analysis).

Proof Sketch: Let the anomalous sequence of system calls be (S_1 S_2 …. S_k). We know that this system call sequence is not admitted by the language of valid system call sequences in the system, say S. i.e. there exists NO valid program path which corresponds to the system call sequence S = (S_1 S_2 …. S_k).

We also know that the sequence of system calls impacts the critical variable[37] either directly or indirectly. Consider a system call S_i in the sequence that impacts the critical variable (there must be at least one such S_i in the sequence). There are two possible ways in which S_i can impact the critical variable (illegitimately).

(1) S_i can write to the memory address containing a variable in the backward slice of the critical variable.

If the write by the system call S_i is illegitimate, it will be detected by the IFS as the values in the backward slice are in an encoded form prior to the system call. Any overwriting of the values will violate the encoding and will hence be detected when they are used within the program (during the decode operation).

(2) S_i can modify the control-flow of the program illegitimately to influence the computation of the critical variable. Let us assume that the runtime sequence of encode operations resulting from these actions is 'I'. We know that the system-call sequence S is not a valid one in the program. We need to show that the sequence I does not correspond to a valid IFS in the program.

The proof proceeds by contradiction – we start by assuming that the signature I is a valid one in the program i.e. there exists a program path along which I belongs to the IFS. This implies there exists a valid program path along which the sequence of instructions transcribed by I also belong to the backward slice of the critical variable (CV). Now,

[37] We assume that the application has only one critical variable. The extension to multiple critical variables is straightforward.

consider only the system calls in the backward slice of the CV. By definition, this system call sequence must correspond to a valid program path. Further, the calls themselves are a proper sub-sequence of the system-call based signature of the application. Since they correspond to a valid path, the reason that the system-call sequence S is rejected must be because of a sub-sequence S’ that does not involve any legitimate writes to the critical variable (either direct/indirect). Hence, the sequence S’ can be removed from the signature S to form a new system-call based signature S’’ that impacts the critical variable. However, this corresponds to a valid system-call sequence, and is hence not rejected by the system-call based technique. This is a contradiction of our initial assumption that the system-call based detection technique rejects *all* anomalous system-call sequences that impact the critical variable (recall that we considered idealized system-call sequences).

(b) Attacks that violate the control-flow integrity of the application to influence critical data:

The goal is to show that the IFS scheme will detect attacks that violate the application’s control-flow and influence the critical data, provided the attacks are also detected by control-flow checking schemes. Control-flow checking [185]detects attacks that violate the application’s control flow in violation of the program semantics. The violation can occur through the injection of new code by the attacker, or by the attacker overwriting either function-pointers or return addresses on the stack.

Proof Sketch: Let the valid set of basic-blocks in the program (function) belong to the set B = { B_1 ,B_2 , … B_n}. The Control-Flow Signature (CFS) is a regular expression consisting of some combination of B_is. A control-flow checking technique essentially compares the runtime control-flow to the CFS and checks for validity. A valid CFS is defined as a CFS that corresponds to a valid path in the program (function).

Let the set of basic blocks in which the critical variable[38] (CV) is defined be denoted by the set V. Let BS(V) denote the set of basic-blocks which comprise instructions in the backward slice of the CV. Note that this is different from the IFS as the IFS consists of all instructions in the backward slice of the CV, whereas BS(V) only considers the basic blocks containing the instructions. Let's call this the BIFS.

Our goal is to show that no control-flow sequence that is rejected by the CFS but accepted by the BIFS contains instructions that modify the critical variable either directly or indirectly.

As in the previous case, the proof proceeds by contradiction. Let us assume that there exists a control-flow sequence C that is rejected by CFS but accepted by BIFS and containing at least one instruction I that modifies the value of the critical variable (otherwise, the proof is done). There are two cases for the instruction I as follows:

1. *The instruction I was part of the original program*: In this case, 'I' will be a part of the backward slice of the program, and hence its parent basic block B will be a part of the

[38] As before, we assume a single critical variable in the program without loss of generality.

BIFS. Further, the paths considered by the IFS technique in constructing state machines (at compile-time) are derived from the program's control-flow graph (CFG), and are hence a subset of the paths present in the CFS. Therefore any sequence involving the block B that registers as a deviation from the CFS will also be registered as a deviation by the BIFS. This implies that the sequence will also be rejected by the BIFS – which is a contradiction of our initial assumption.

2. *The instruction I is introduced by the attacker through a code-injection attack***:** In this case, 'I' will not be able to modify the value of the critical variable, or any other variable in the backward slice of the critical variable. This is because all the variables will have been encoded when the instruction I is executed. Moreover, any attempt by instruction I to create a value and call an encoding function will result in a violation of the BIFS (because the basic block containing I will not belong to B, the domain set of BIFS). Finally, if I tries to perform a jump to the middle of an existing control-flow sequence in the BIFS, or to truncate the BIFS, it will be treated analogous to case 1 (i.e. any such jump that is detected by the CFS is also detected by the BIFS). Therefore, either I cannot modify the CV or it will result in a sequence that violates the BIFS, which is a contradiction of our assumption.

Hence, the BIFS detects all control-flow attacks that influence the critical variable and are detected by a control-flow checking technique.

(c) Attacks that overwrite the critical variable either directly or indirectly – The goal is to show that the IFS technique is at least as effective as any other memory-safety checking technique (e.g. the WIT technique [171]) in detecting memory corruption attacks that attempt to overwrite the critical data. Let us assume that the instruction that performs the overwriting is I and that I belongs to the original program (otherwise, it is equivalent to the attacker mounting a code injection attack and this was covered in case b). Assume that the set of variables in the target set of instruction I is given by T. The WIT technique will detect any attempt by I to write to a variable outside the set T[39]. Since we assume that the WIT technique detects this attack, it must follow that the critical variable V is outside the set T. We consider three cases for the instruction I as follows:

i. *I does not belong to the backward slice of the critical variable:* The attack will be detected as all the variables in the backward slice of I will be encoded and any overwriting of them will result in an incorrect value when decoded. In case 'I' attempts to call an encoding function, it will necessarily violate the sequence of state transitions derived by the IFS technique, and the attack is detected.

ii. *I belongs to the backward slice of the critical variable, but is not valid for the current execution path:* Since the backward slice is conservative by definition, it has to include all instructions that could potentially write to the critical variable, even if they are not valid for the current execution (input). This would be detected by the IFS if (and only if) the execution of the instruction results in an invalid state transition in the state machines i.e. there is at least one other instruction in the path that makes the path invalid. This need

[39] In reality, WIT offers a much weaker guarantee, namely that 'I' does not write to objects of a different color than itself. Since merging of colors can occur, I can write to an object outside its target set of the same color. Nonethess, we consider a stronger version of the WIT technique's guarantees.

not always be true – for example, if I is the only instruction along the *then* branch of an *if* statement, when the valid execution consists of the *else* statement. However, such cases would not be detected by the WIT technique, as WIT has to conservatively assume that all paths are valid in the program and hence assign the same color to all instructions that can potentially write to the same set of objects (even if they are on different paths). In this case, the guarantees provided by the IFS technique are *stronger* than that of WIT – provided the attack substitutes instructions corresponding to paths consisting of at least two backward-slice instructions.

iii. *I belongs to the backward slice of the critical variable, and is valid for the current execution path* – This attack cannot be detected by the WIT technique as it does not take into account the order of instructions in the program in determining the validity of a write. However, the IFS technique can provide limited protection against this class of attacks, provided the overwriting instruction is executed in an order that is inconsistent with the backward slice i.e. there exists no program path in which the execution of the instruction would result in an order consistent with the control-flow graph of the program. This is because the execution of the instruction has to either trigger an invalid state transition in the state machine or the values written to by the instruction are dynamically dead at the time of the overwriting (and hence the overwriting is benign). However, if this constraint is not satisfied, the IFS technique cannot detect the attack (and neither can the WIT technique).

Thus we see that in the case of memory corruption attacks on the critical data, the IFS technique can detect all attacks that would also be detected by a memory-safety checking technique such as WIT.

Summary: Thus we show semi-formally that the IFS technique detects all cases of insider attacks that attempt to modify critical data independent of whether the untrusted module's source code is available for analysis. In the case of external memory-corruption attacks, the IFS technique detects any attempt to impact the security-critical data provided the attack is also detected by existing, state-of-art, security techniques.

7.11 CONCLUSION

This chapter introduced an approach to protect security critical data from insider attacks. The approach leverages existing static analysis techniques to extract the backward slices of critical variables and to convert the slice into a Information-flow Signature (IFS). The IFS is tracked and checked at runtime using automatically generated code. A deviation of the runtime signature from the derived IFS indicates a security attack. We have deployed the technique on three widely-used open-source applications to protect security-critical data and have shown that the technique detects both insider attacks and external memory corruption attacks with low performance overheads.

Future work will involve (1) implementing the encoding and decoding functions in hardware to reduce the performance overheads of the IFS technique and (2) incorporating context-sensitivity in the slicing algorithm to increase the coverage of the technique.

CHAPTER 8   CONCLUSIONS AND FUTURE WORK

8.1   CONCLUSIONS

This dissertation has demonstrated a unified approach for providing reliability and security to applications in an automated fashion. The approach presented in this dissertation enhances the compiler and the runtime system to derive application-aware error and attack detectors that continuously monitor the application for errors and attacks. The key characteristic of the approach is that it protects critical data in the application from a wide-range of errors and attacks. Critical data is defined as any data in the application, which if corrupted can cause failures with long downtimes or hard-to-detect security compromises.

The errors considered by the approach include hardware transient errors in memory and computation as well as software defects that cause transient data corruptions. Examples of the latter are soft-errors, errors in the processor's control-logic and variation-induced errors. Examples of the latter are memory corruption errors and race conditions. The attacks considered by the approach include external attacks that exploit memory corruption vulnerabilities in the application as well as internal attacks launched by a trusted insider in the same address space as the application.

The dissertation also presents a unified approach to formally model the effects of low-level errors and security attacks on the application. The formal approach exhaustively enumerates the effects of errors (attacks) according to a given fault (threat) model and helps in validating the efficacy of the derived detectors.

Finally, the detectors derived using the approach proposed in this dissertation have been implemented using reconfigurable hardware in the context of the Reliability and Security Engine (RSE) [33], which is a hardware framework for executing application-aware checks. The detectors have been prototyped as part of the Trusted Illiac project at UIUC.

In summary, application-aware dependability is a viable approach for building highly reliable and secure applications with lower performance overheads compared to traditional dependability techniques such as duplication or type-safety.

8.2   FUTURE WORK

This section provides a roadmap of future work in the directions explored by this dissertation.

Compilers and program analysis techniques: Compilers typically focus on optimizing program performance by removing redundancies in the program's source code. This is because redundancies in the program code can result in wasteful computation and consequently loss of performance. However, redundancies also offer advantages in terms of increasing program resilience to hardware and software errors, provided the redundancies can be turned into appropriate runtime checks.

Currently, even if redundancies are present in an application's source code, they offer little benefit to the application as they are not represented as runtime checks. In the future, compilers could convert redundancies in the program into runtime checks for error detection. Compilers can also introduce redundancies in a controlled manner into the original code or avoid removing certain redundancies that were originally present.

However, a judicious trade-off must be achieved between introducing too much redundancy, which would hurt performance, and not introducing any redundancy, which may impact the application's reliability and security.

The technique proposed in Chapter 4 is one way of introducing redundancies in the computation of critical variables in the form of runtime checks. Further, the check is not just a straightforward duplication of the critical variable's computation, but a selectively optimized version and is hence different from the original version. However, the technique did not attempt to explicitly diversify the representation of the check with respect to the original computation. Diversification can lead to detection of permanent hardware errors and software bugs that would not be detected by duplication. This is an area of future investigation.

Program verification: Program verification techniques such as theorem proving and model-checking analyze the program's code in order to prove properties about the program with respect to a formal specification. The formal specifications are provided by the programmer and the verification tool attempts to statically establish whether the program satisfies the specification along each program path. As discussed in Section 4.2.1, these techniques are vulnerable to the *feasible path problem*, which lead to exploration of program paths that will never occur during a concrete execution of the program. Recent work has considered the use of dynamic analysis to drive the verification along concrete program execution paths [189, 190]. The main idea in these systems is to piggyback the symbolic exploration of paths by the verification tool onto the concrete execution of the program and use the information from the concrete execution to prune infeasible paths.

The SymPLFIED technique described in Chapter 5 is a symbolic technique for formally validating fault-tolerance properties of an application. SymPLFIED also faces the feasible path problem as it statically explores the program's error outcomes. This leads to state-space exploration when model-checking large programs. One way of enhancing the analysis is to use runtime information gathered during the dynamic execution of the program to explore only those paths that are likely to be executed during a concrete execution. However, this is different from the techniques proposed in [189, 190], as these techniques do not need to consider the effects of random errors in the state-space exploration. The main challenge introduced by random errors is that the program may follow paths that do not occur in any dynamic error-free execution. Hence, new methods of integrating dynamic execution profiles with symbolic execution are needed.

Runtime systems for program monitoring: Runtime monitoring systems provide a flexible method to observe programs and perform adaptations on the fly depending on changes in the requirement or the environment. The techniques proposed in this dissertation also fall under the broad umbrella of runtime monitoring techniques. However, existing runtime monitoring are geared towards detecting specific kinds of errors [90] and security attacks [191]. The technique proposed in this dissertation on the other hand, can detect a broad class of errors and attacks that impact critical variables in the application. The technique can be integrated into a program monitoring framework such as Monitoring Oriented Programming (MOP) [192] to detect generic runtime errors and security attacks. This offers the advantage that the checks can be expressed in a formal fashion in order to facilitate reasoning about their detection capabilities. Further, the checks can be adaptively enabled or disabled at runtime depending on the prevalence of errors or attacks in the program's environment as well as the maximum performance overhead that the end-user is willing to incur when executing the application.

Micro-architecture design: The technique developed in this dissertation uses a combination of software and reconfigurable FPGA (Field-Programmable Gate Array) hardware to execute the derived detectors. In the future, it is possible that the processor itself directly executes the detectors using specialized functional units with no intervention from the software. Each processor could have dedicated functional units to execute specific detectors. The functional units would be configured by the processor manufacturer to include checks used by a standard set of workloads (benchmarks). This is similar to the approach in [193] for accelerating program operations using hardware.

An interesting challenge in this approach is to specify a set of common detector patterns across a range of applications to be configured into hardware. The processor's front-end can automatically identify the specified patterns in the application's binary and transparently schedule the checks onto dedicated functional units with no involvement from the compiler. This approach will completely move the complexity of runtime adaptation to errors and attacks from the software to the hardware. It will also obviate the distribution of multiple versions of an application with different checks for different environments, and instead allow the hardware to transparently control both the degree and nature of the runtime checks that are executed for an application.

REFERENCES

[1] Nakka, N., 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: IEEE Computer Society.

[2] Ross, J.W. and G. Westerman, Preparing for utility computing: The role of IT architecture and relationship management. IBM Systems Journal, 2004. 43(1): p. 5-19.

[3] Rappa, M.A., The utility business model and the future of computing services. IBM Systems Journal, 2004. 43(1): p. 32-42.

[4] Hayes, B., Cloud computing. 2008.

[5] Armbrust, M., A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, Above the clouds: A Berkeley view of cloud computing. University of California, Berkeley, Tech. Rep, 2009.

[6] Gray, J. Why do computers stop and what can be done about it. in Symposium on Reliable Distributed Systems. 1986: IEEE.

[7] Sullivan, M. and R. Chillarege. Software defects and their impact on system availability-a study of field failures in operating systems. in Twenty-First Symposium on Fault-Tolerant Computing. 1991.

[8] Lee, I. and R.K. Iyer, Software dependability in the Tandem GUARDIAN system. IEEE Transactions on Software Engineering, 1995. 21(5): p. 455-467.

[9] Spainhower, L. and W. Bartlett, Commercial Fault Tolerance: A Tale of Two Systems. IEEE Transactions on Dependable and Secure Systems, 2004. 1(1): p. 87-96.

[10] Slegel, T.J., I. Robert M. Averill, M.A. Check, B.C. Giamei, B.W. Krumm, C.A. Krygowski, W.H. Li, J.S. Liptay, J.D. MacDougall, T.J. McPherson, J.A. Navarro, E.M. Schwarz, K. Shum, and C.F. Webb, IBM's S/390 G5 Microprocessor Design. IEEE Micro, 1999. 19(2): p. 12-23.

[11] Saggese, G.P. and A. Vetteth, Microprocessor Sensitivity to Failures: Control vs Execution and Combinational vs Sequential Logic, in Proceedings of the 2005 International Conference on Dependable Systems and Networks. 2005, IEEE Computer Society.

[12] Nakka, N., K. Pattabiraman, and R. Iyer. Processor-level Selective Replication. in International Conference on Dependable Systems and Networks (DSN). 2007: IEEE.

[13] Rinard, M., C. Cadar, D. Dumitran, D.M. Roy, T. Leu, and W.S.B. Jr., Enhancing server availability and security through failure-oblivious computing, in Proceedings of the 6th conference on Symposium on Opearting Systems Design \& Implementation - Volume 6. 2004, USENIX Association: San Francisco, CA.

[14] Gu, W., Z. Kalbarczyk, R. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors. in International Conference on Dependable Systems and Networks. 2003: IEEE Computer Society.

[15] Chillarege, R. and R.K. Iyer, Measurement-based analysis of error latency. IEEE Transactions on Computers, 1987. 36(5): p. 529-537.

[16] Goswami, K.K., R.K. Iyer, and L. Young, DEPEND: A Simulation-Based Environment for System Level Dependability Analysis. IEEE Transactions on Computers, 1997. 46(1): p. 60-74.

[17] Pattabiraman, K., Z. Kalbarczyk, and R.K. Iyer. Application-based metrics for strategic placement of detectors. in Pacific Rim Dependable Computing. 2005. Changsha, China: IEEE CS Press.

[18] Weiser, M. Program Slicing. in Fifth International Conference on Software Engineering. 1981: IEEE Computer Society Press.

[19] Hsueh, M.-C., T.K. Tsai, and R.K. Iyer, Fault Injection Techniques and Tools. Computer, 1997. 30(4): p. 75-82.

[20] Cowan, C., C. Pu, D. Maier, J. Walpole, P. Bakke, A.G. SteveBeattie, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. in Seventh Usenix Technical Symposium. 1998: Usenix.

[21] Forrest, S., S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff, A Sense of Self for Unix Processes, in Proceedings of the 1996 IEEE Symposium on Security and Privacy. 1996, IEEE Computer Society.

[22] Ruwase, O. and M.S. Lam. A practical dynamic buffer overflow detector. in 11th Annual Network and Distributed System Security. 2004.

[23] Dhurjati, D., S. Kowshik, and V. Adve. SAFECode: enforcing alias analysis for weakly typed languages. in ACM SIGPLAN conference on Programming language design and implementation. 2006. Ottawa, Ontario, Canada: ACM Press.

[24] Jones, R.W.M. and P.H.J. Kelly. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. in Automated and Algorithmic Debugging. 1997.

[25] Suh, G.E., J.W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. in 11th international conference on Architectural support for programming languages and operating systems. 2004. Boston, MA, USA: ACM Press.

[26] Chen, S., N. Nakka, J. Xu, Z. Kalbarczyk, and R. Iyer. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. in International Conference on Dependable Systems and Networks. 2005: IEEE Computer Society.

[27] Dalton, M., H. Kannan, and C. Kozyrakis, Raksha: a flexible information flow architecture for software security, in Proceedings of the 34th annual international symposium on Computer architecture. 2007, ACM: San Diego, California, USA.

[28] Xu, J., Z. Kalbarczyk., and R. Iyer. Transparent runtime randomization for security. in 22nd International Symposium on Reliable Distributed Systems. 2003.

[29] Bhatkar, S., D.C. DuVarney, and R. Sekar, Address obfuscation: an efficient approach to combat a board range of memory error exploits, in Proceedings of the 12th conference on USENIX Security Symposium - Volume 12. 2003, USENIX Association: Washington, DC.

[30] Berger, E.D. and B.G. Zorn. DieHard: probabilistic memory safety for unsafe languages. in ACM SIGPLAN conference on Programming language design and implementation. 2006. Ottawa, Ontario, Canada: ACM Press.

[31] Shacham, H., M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, On the effectiveness of address-space randomization, in Proceedings of the 11th ACM conference on Computer and communications security. 2004, ACM: Washington DC, USA.

[32] Sezer, E.C., P. Ning, C. Kil, and J. Xu, Memsherlock: an automated debugger for unknown memory corruption vulnerabilities, in Proceedings of the 14th ACM conference on Computer and communications security. 2007, ACM: Alexandria, Virginia, USA.

[33] Nakka, N., Reliability and Security Engine: A processor-level framework for application-aware detection, in Electrical and Computer Engineering. 2006, UIUC: Urbana-Champaign.

[34] M.Clavel, F.Duran, S.Eker, P.Lincoln, N.Marti-Oliet, J.meseguer, and J.Quesada, Maude: Specification and Programming in Rewriting Logic, in Maude System Documentation. 1999, SRI.

[35] LEON3 Implementation of the Sparc V8. [cited; Available from: http://www.gaisler.com.

[36] Iyer, R.K., D.J. Rossetti, and M.C. Hsueh, Measurement and modeling of computer reliability as affected by system activity. ACM Trans. Comput. Syst., 1986. 4(3): p. 214-237.

[37] Basile, C., W. Long, Z. Kalbarczyk, and R. Iyer. Group communication protocols under errors. in 22nd International Symposium on Reliable Distributed Systems. 2003.

[38] Chandra, S. and P.M. Chen. How Fail-Stop are Faulty Programs? in Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing. 1998: IEEE Computer Society.

[39] Bartlett, J.F., A NonStop kernel, in Proceedings of the eighth ACM symposium on Operating systems principles. 1981, ACM: Pacific Grove, California, United States.

[40] Hiller, M., A. Jhumka, and N. Suri. On the Placement of Software Mechanisms for Detection of Data Errors. in International Conference on Dependable Systems and Networks. 2002: IEEE Computer Society.

[41] Jeffrey, M.V. and W.M. Keith, The Avalanche Paradigm: An Experimental Software Programming Technique for Improving Fault-tolerance, in Proceedings of the IEEE Symposium and Workshop on Engineering of Computer Based Systems. 1996, IEEE Computer Society.

[42] Goradia, T., Dynamic impact analysis: a cost-effective technique to enforce error-propagation. SIGSOFT Softw. Eng. Notes, 1993. 18(3): p. 171-181.

[43] Ernst, M.D., J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. in 21st international conference on Software engineering. 1999. Los Angeles, California, United States: IEEE Computer Society Press.

[44] Narayanan, S.H.K., S.W. Son, M. Kandemir, and F. Li, Using loop invariants to fight soft errors in data caches, in Proceedings of the 2005 conference on Asia South Pacific design automation. 2005, ACM: Shanghai, China.

[45] Benso, A., S. Chiusano, P. Prinetto, and L. Tagliaferri. A C/C++ Source-to-Source Compiler for Dependable Applications. in International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8). 2000: IEEE Computer Society.

[46] Nethercote, N. and A. Mycroft, Redux: A Dynamic Dataflow Tracer. Electronic Notes on Theoretical Computer Science, 2003. 89(2).

[47] Zhang, X., R. Gupta, and Y. Zhang, Cost and precision tradeoffs of dynamic data slicing algorithms. ACM Trans. Program. Lang. Syst., 2005. 27(4): p. 631-661.

[48] Chillarege, R., W.-L. Kao, and R.G. Condit, Defect type and its impact on the growth curve, in Proceedings of the 13th international conference on Software engineering. 1991, IEEE Computer Society Press: Austin, Texas, United States.

[49] Kao, W.-l., R.K. Iyer, and D. Tang, FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior Under Faults. IEEE Trans. Softw. Eng., 1993. 19(11): p. 1105-1118.

[50] Austin, T., E. Larson, and D. Ernst, SimpleScalar: An Infrastructure for Computer System Modeling. Computer, 2002. 35(2): p. 59-67.

[51] Hutchins, M., H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. in 16th international conference on Software engineering. 1994. Sorrento, Italy: IEEE Computer Society Press.

[52] Bush, W.R., J.D. Pincus, and D.J. Sielaff, A static analyzer for finding dynamic programming errors. Software Practice and Experience, 2000. 30(7): p. 775-802.

[53] Evans, D., J. Guttag, J. Horning, and Y.-M. Tan. LCLint: a tool for using specifications to check code. in 2nd ACM SIGSOFT symposium on Foundations of software engineering. 1994. New Orleans, Louisiana, United States: ACM Press.

[54] Andrews, D. Using executable assertions for testing and fault tolerance,. in 9th Faul-tolerance Computing Symposium. 1979. Madison, WI: IEEE.

[55] Leveson, N.G., S.S. Cha, J.C. Knight, and T.J. Shimeall, The use of self checks and voting in software error detection: an empirical study. IEEE Transactions on Software Engineering, 1990. 16(4): p. 432-443.

[56] Hiller, M. Executable Assertions for Detecting Data Errors in Embedded Control Systems. in International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8). 2000: IEEE Computer Society.

[57] Voas, J., Software testability measurement for intelligent assertion placement. Software Quality Control, 1997. 6(4): p. 327-336.

[58] Patterson, D.A. and J.L. Hennessy, Computer architecture: a quantitative approach. 1990: Morgan Kaufmann Publishers Inc. 594.

[59] Wang, N.J. and S.J. Patel, ReStore: Symptom-Based Soft Error Detection in Microprocessors. IEEE Trans. Dependable Secur. Comput., 2006. 3(3): p. 188-201.

[60] Necula, G.C., S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. in ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2002. Portland, Oregon: ACM Press.

[61] Engler, D., D.Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. in Eighteenth ACM Symposium on Operating systems principles. 2001. Banff, Alberta, Canada: ACM Press.

[62] Hangal, S. and M.S. Lam. Tracking down software bugs using automatic anomaly detection. in 24th International Conference on Software Engineering. 2002. Orlando, Florida: ACM Press.

[63] Maxion, R.A. and K.M.C. Tan, Anomaly Detection in Embedded Systems. IEEE Trans. Comput., 2002. 51(2): p. 108-120.

[64] Rela, M.Z., H. Madeira, and J.G. Silva. Experimental evaluation of the fail-silent behaviour in programs with consistency checks. in Annual Symposium on Fault-tolerant Computing. 1996. Sendai.

[65] Racunas, P., K. Constantinides, S. Manne, and S.S. Mukherjee, Perturbation-based Fault Screening, in Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture. 2007, IEEE Computer Society.

[66] Dimitrov, M. and H. Zhou, Unified Architectural Support for Soft-Error Protection or Software Bug Detection, in Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques. 2007, IEEE Computer Society.

[67] Sahoo, S., M.-l. Li, P. Ramachandran, V.S. Adve, S.V. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. in Proceedings of International Conference on Dependable Systems and Networks (DSN). 2008. Anchorage, AK: IEEE.

[68] Oh, N., P.P. Shirvani, and E.J. McCluskey, Error detection by duplicated instructions in super-scalar processors. IEEE Transactions on Relibility, 2002. 51(1): p. 63-75.

[69] Reis, G.A., J. Chang, N. Vachharajani, R. Rangan, and D.I. August. SWIFT: Software Implemented Fault Tolerance. in International symposium on Code generation and optimization. 2005: IEEE Computer Society.

[70] Iyer, R.K., N.M. Nakka, Z.T. Kalbarczyk, and S. Mitra, Recent advances and new avenues in hardware-level reliability support. Micro, IEEE, 2005. 25(6): p. 18-29.

[71] Das, M., S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. in Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation. 2002. Berlin, Germany: ACM Press.

[72] Ball, T. and S. Rajamani. The SLAM Toolkit. in 13th International Conference on Computer Aided Verification. 2001: Springer-Verlag.

[73] Flanagan, C. and S. Qadeer. Predicate abstraction for software verification. in 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2002. Portland, Oregon: ACM Press.

[74] Demsky, B., M.D. Ernst, P.J. Guo, S. McCamant, J.H. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. in International symposium on Software testing and analysis. 2006. Portland, Maine, USA: ACM Press.

[75] Li, Z. and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. in 13th ACM SIGSOFT international symposium on Foundations of software engineering. 2005. Lisbon, Portugal: ACM Press.

[76] Mukherjee, S.S., M. Kontz, and S.K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. in 29th annual international symposium on Computer architecture. 2002. Anchorage, Alaska: IEEE Computer Society.

[77] Sundaramoorthy, K., Z. Purser, and E. Rotenburg. Slipstream processors: improving both performance and fault tolerance. in Ninth international conference on Architectural support for programming languages and operating systems. 2000. Cambridge, Massachusetts, United States: ACM Press.

[78] Avizenies, A., The Methodology of N-Version Programming, in Software Fault Tolerance, M.R. Lyu, Editor. 1995, Wiley. p. 23-46.

[79] Knight, J.C. and N.G. Leveson, An experimental evaluation of the assumption of independence in multiversion programming. IEEE Transactions on Software Engineering, 1986. 12(1): p. 96-109.

[80] Ammann, P.E. and J.C. Knight, Data Diversity: An Approach to Software Fault Tolerance. IEEE Transactons on Computers, 1988. 37(4): p. 418-425.

[81] Oh, N., S. Mitra, and E.J. McCluskey, ED<sup>4</sup>I: error detection by diverse data and duplicated instructions. IEEE Transactions on Computers, 2002. 51(2): p. 180-199.

[82] Jonathan, C., A.R. George, and I.A. David. Automatic Instruction-Level Software-Only Recovery. in International Conference on Dependable Systems and Networks (DSN'06). 2006: IEEE Computer Society.

[83] Proudler, I.K., Idempotent AN codes, in IEE Colloquium on Signal Processing Applications of Finite Field Mathematics. 1989. p. 8/1-8/5.

[84] Dhurjati, D. and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. in 28th international conference on Software engineering. 2006. Shanghai, China: ACM Press.

[85] Savage, S., M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, Eraser: a dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems, 1997. 15(4): p. 391-411.

[86] Alkhalifa, Z., V.S.S. Nair, N. Krishnamurthy, and J.A. Abraham, Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection. IEEE Transactions on Parallel and Distributed Systems, 1999. 10(6): p. 627-641.

[87] Bagchi, S., Y. Liu, K. Whisnant, Z. Kalbarczyk, R. Iyer, Y. Levendel, and L. Votta. A framework for database audit and control flow checking for a wireless telephone network controller. in Dependable Systems and Networks. 2001. Goteborg: IEEE CS Press.

[88] Oh, N., P.P. Shirvani, and E.J. McCluskey, Control-flow checking by software signatures. IEEE Transactions on Reliability, 2002. 51(1): p. 111-122.

[89] Havelund, K. and G. Rosu, An Overview of the Runtime Verification Tool Java PathExplorer. Formal Methods in System Design, 2004. 24(2): p. 189-215.

[90] Kim, M., M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky, Java-MaC: A Run-Time Assurance Approach for Java Programs. Formal Methods in System Design, 2004. 24(2): p. 129-155.

[91] Lattner, C. and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. in ACM SIGPLAN conference on Programming language design and implementation. 2005. Chicago, IL, USA: ACM Press.

[92] Ohlsson, J., M. Rimen, and U. Gunneflo. A study of the effects of transient fault injection into a 32-bit RISC with built-in watchdog. in Twenty-Second International Symposium on Fault-tolerant Computing. 1992: IEEE.

[93] Saib, S.H. Executable Assertions - An Aid To Reliable Software. in 11th Asilomar Conference Circuits Systems and Computers. 1978.

[94] Kuang-Hua, H. and J.A. Abraham, Algorithm-Based Fault Tolerance for Matrix Operations. IEEE Transactions on Computers, 1984. C-33(6): p. 518-528.

[95] Avizienis, A., J.C. Laprie, B. Randell, and C. Landwehr, Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing, 2004. 1(1): p. 11-33.

[96] Tip, F., A survey of program slicing techniques. Journal of Programming Languages, 1995. 3(3): p. 121-189.

[97] Kernighan, B.W. and D.M. Ritchie, The C Programming Language. 1989: Prentice Hall Press.

[98] Engler, D. and K. Ashcraft, RacerX: effective, static detection of race conditions and deadlocks. SIGOPS Oper. Syst. Rev., 2003. 37(5): p. 237-252.

[99] Lattner, C. and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis \& Transformation. in international symposium on Code generation and optimization. 2004. Palo Alto, California: IEEE Computer Society.

[100] Cytron, R., J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck, Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems, 1991. 13(4): p. 451-490.

[101] Muchnick, S.S., Advanced compiler design and implementation. 1997: Morgan Kaufmann Publishers Inc. 856.

[102] Weicker, R.P., An Overview of Common Benchmarks, in Computer. 1990. p. 65-75.

[103] Carlisle, M.C. and A. Rogers. Software caching and computation migration in Olden. in Fifth ACM SIGPLAN symposium on Principles and practice of parallel programming. 1995. Santa Barbara, California, United States: ACM Press.

[104] Meixner, A. and D.J. Sorin, Error Detection Using Dynamic Dataflow Verification, in Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques. 2007, IEEE Computer Society.

[105] Meixner, A., M.E. Bauer, and D.J. Sorin, Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. IEEE Micro, 2008. 28(1): p. 52-59.

[106] Ball, T. and J.R. Larus, Efficient path profiling, in Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture. 1996, IEEE Computer Society: Paris, France.

[107] Vaswani, K., M.J. Thazhuthaveetil, and Y.N. Srikant, A Programmable Hardware Path Profiler, in Proceedings of the international symposium on Code generation and optimization. 2005, IEEE Computer Society.

[108] Zhang, T., X. Zhuang, S. Pande, and W. Lee, Anomalous path detection with hardware support, in Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems. 2005, ACM: San Francisco, California, USA.

[109] Pattabiraman, K., G.P. Saggese, D. Chen, Z. Kalbarczyk, and R.K. Iyer. Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware. in Sixth European Dependable Computing Conference. 2006. Coimbra, Portugal: IEEE CS Press.

[110] Pattabiraman, K. and R. Iyer, Automated Derivation of Application-Aware Error Detectors using Static Analysis. 2006, University of Illinois (Urbana-Champaign).

[111] Perry, F., L. Mackey, G.A. Reis, J. Ligatti, D.I. August, and D. Walker, Fault-tolerant typed assembly language, in Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. 2007, ACM: San Diego, California, USA.

[112] Clarke, E.M. and B.-H. Schlingloff, Model checking, in Handbook of automated reasoning. 2001, Elsevier Science Publishers B. V. p. 1635-1790.

[113] Larsson, D. and R. Hahnle, Symbolic Fault Injection. International Verification Workshop (VERIFY),International Conference on Automated Deduction (CADE), 2007. 259: p. 85-103.

[114] Arora, A. and S.S. Kulkarni, Detectors and Correctors: A Theory of Fault-Tolerance Components, in Proceedings of the The 18th International Conference on Distributed Computing Systems. 1998, IEEE Computer Society.

[115] Jhumka, A., M. Hiller, V. Claesson, and N. Suri, On systematic design of globally consistent executable assertions in embedded software, in Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems. 2002, ACM: Berlin, Germany.

[116] Nicolescu, B., N. Gorse, Y. Savaria, E.M. Aboulhamid, and R. Velazco, On the use of model checking for the verification of a dynamic signature monitoring approach. IEEE Transactions on Nuclear Science, 2005. 52(5(2)): p. 1555-1561.

[117] King, J.C., Symbolic execution and program testing. Commun. ACM, 1976. 19(7): p. 385-394.

[118] Boyer, R.S. and J.S. Moore, Program verification. J. Autom. Reason., 1985. 1(1): p. 17-23.

[119] Chen, H., D. Dean, and D. Wagner, Model-checking one million lines of C code. Network and Distributed System Security Symposium, 2004: p. 171-185.

[120] Srivas, M. and M. Bickford, Formal Verification of a Pipelined Microprocessor. IEEE Softw., 1990. 7(5): p. 52-64.

[121] Krautz, U., M. Pflanz, C. Jacobi, H.W. Tast, K. Weber, and H.T. Vierhaus, Evaluating coverage of error detection logic for soft errors using formal methods, in Proceedings of the conference on Design, automation and test in Europe: Proceedings. 2006, European Design and Automation Association: Munich, Germany.

[122] Seshia, S.A., W. Li, and S. Mitra, Verification-guided soft error resilience, in Proceedings of the conference on Design, automation and test in Europe. 2007, EDA Consortium: Nice, France.

[123] Clavel, M., F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. in Rewriting Technologies and Applications. 2001: Springer.

[124] Serbanuta, T.F., G. Rosu, and J. Meseguer, A Rewriting Logic Approach to Operational Semantics (Extended Abstract). Electron. Notes Theor. Comput. Sci., 2007. 192(1): p. 125-141.

[125] Administration, F.A., TCAS II Collision Avoidance System (CAS) System Requirements Specification. 1993.

[126] Lygeros, J. and N. Lynch, On the formal verification of the TCAS conflict resolution algorithms. In Proceedings of the 36th IEEE Conference on Decision and Control, 1997: p. 1829--1834.

[127] Coen-Porisini, A., G. Denaro, C. Ghezzi, and M. Pezz, Using symbolic execution for verifying safety-critical systems. SIGSOFT Softw. Eng. Notes, 2001. 26(5): p. 142-151.

[128] Anderson, R.J., P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J.D. Reese, Model checking large software specifications. SIGSOFT Softw. Eng. Notes, 1996. 21(6): p. 156-166.

[129] Randazzo, M.R., M.M. Keeney, E.F. Kowalski, D.M. Cappelli, and A.P. Moore, Insider Threat Study: Illicit Cyber Activity in the Banking and Finance Sector. 2004, U.S. Secret Service and CERT Coordination Center/Software Engineering Institute: Philadelphia, PA. p. 25.

[130] Keeney, M.M. and E.F. Kowalski, Insider Threat Study: Computer System Sabotage in Critical Infrastructure Sectors. 2005, CERT/CC: Philadelphia, PA.

[131] Polk, W. and L. Bassham, Threat assessment of malicious code and human threats. 1994, NIST, Computer Security Division.

[132] Sheyner, O., J. Haines, S. Jha, R. Lippmann, and J.M. Wing, Automated Generation and Analysis of Attack Graphs, in Proceedings of the 2002 IEEE Symposium on Security and Privacy. 2002, IEEE Computer Society.

[133] Ammann, P., D. Wijesekera, and S. Kaushik, Scalable, graph-based network vulnerability analysis, in Proceedings of the 9th ACM conference on Computer and communications security. 2002, ACM: Washington, DC, USA.

[134] Phillips, C. and L.P. Swiler, A graph-based system for network-vulnerability analysis, in Proceedings of the 1998 workshop on New security paradigms. 1998, ACM: Charlottesville, Virginia, United States.

[135] Probst, C.W., R.R. Hansen, and F. Nielson, Where can an Insider Attack ?, in Formal Aspects in Security and Trust. 2007, Springer Berlin / Heidelberg. p. 127-142.

[136] Pattabiraman, K., N. Nakka, and Z. Kalbarczyk. SymPLFIED: Symbolic Program Level Fault-Injection and Error-Detection Framework. in International Conference on Dependable Systems and Networks (DSN). 2008.

[137] OpenSSH Development Team., OpenSSH 4.21. 2004.

[138] King, S.T. and P.M. Chen, Backtracking intrusions. SIGOPS Oper. Syst. Rev., 2003. 37(5): p. 223-236.

[139] Musuvathi, M. and D.R. Engler, Model checking large network protocol implementations, in Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1. 2004, USENIX Association: San Francisco, California.

[140] Chinchani, R., A. Iyer, H.Q. Ngo, and S. Upadhyaya, Towards a Theory of Insider Threat Assessment, in Proceedings of the 2005 International Conference on Dependable Systems and Networks. 2005, IEEE Computer Society.

[141] Costa, M., M. Castro, L. Zhou, L. Zhang, and M. Peinado, Bouncer: securing software by blocking bad input, in Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. 2007, ACM: Stevenson, Washington, USA.

[142] Molnar, D.A. and D. Wagner, Catchconv: Symbolic execution and run-time type inference for integer conversion errors. 2007, EECS Department, University of California, Berkeley: Berkeley, CA.

[143] Kruegel, C., E. Kirda, D. Mutz, W. Robertson, and G. Vigna, Automating mimicry attacks using static binary analysis, in Proceedings of the 14th conference on USENIX Security Symposium - Volume 14. 2005, USENIX Association: Baltimore, MD.

[144] Cadar, C., V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler, EXE: automatically generating inputs of death, in Proceedings of the 13th ACM conference on Computer and communications security. 2006, ACM: Alexandria, Virginia, USA.

[145] Boneh, D., R. DeMillo, and R.J. Lipton. On the importance of checking crypto-graphic protocols for faults. in Advances in Cryptgraphy (EuroCrypt). 1997: Springer.

[146] Voyiatzis, A.G. and D.N. Serpanos, Active Hardware Attacks and Proactive Countermeasures, in Proceedings of the Seventh International Symposium on Computers and Communications (ISCC'02). 2002, IEEE Computer Society.

[147] Xu, J., S. Chen, Z. Kalbarczyk, and R.K. Iyer, An Experimental Study of Security Vulnerabilities Caused by Errors, in Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS). 2001, IEEE Computer Society.

[148] Govindavajhala, S. and A.W. Appel, Using Memory Errors to Attack a Virtual Machine, in Proceedings of the 2003 IEEE Symposium on Security and Privacy. 2003, IEEE Computer Society.

[149] Hissam, S., D. Plakosh, and C. Weinstock. Trust and Vulnerability in Open Source Software. in Proceedings - Software Engineering. 2002: IEE.

[150] Jeremey. Linux Kernel Backdoor attempt. 2003 [cited; Available from: http://kerneltrap.org/node/1584.

[151] Hunt, F. and P. Johnson. On the Pareto-distribution of Sourceforge projects. in Open Source Software Development Workshop. 2002.

[152] Veracode, Protecting Your Applications from Backdoors: How Static Binary Analysis Helps Build High-Assurance Applications, in White Paper. 2008.

[153] Cappelli, D.M., T. Caron, R.F. Trzeciak, and A.P. Moore, Programming Techniques Used as an Insider Attack Tool, in Spotlight On, CERT, Editor. 2008, Software Engineering Institute (SEI): Pittsburgh, PA.

[154] Skoudis, E. and L. Zeltser, Malware: Fighting Malicious Code. 2003, Upper Saddle River, NJ, USA: Prentice Hall PTR.

[155] Christodorescu, M. and S. Jha, Testing malware detectors. SIGSOFT Softw. Eng. Notes, 2004. 29(4): p. 34-44.

[156] Christodorescu, M., S. Jha, S.A. Seshia, D. Song, and R.E. Bryant. Semantics-aware malware detection. 2005.

[157] Bayer, U., C. Kruegel, and E. Kirda. TTAnalyze: A tool for analyzing malware. 2006.

[158] Dinaburg, A., P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. 2008: ACM New York, NY, USA.

[159] Iyer, R.K., Z. Kalbarczyk, K. Pattabiraman, W. Healey, W.-M.W. Hwu, P. Klemperer, and R. Farivar, Toward Application-Aware Security and Reliability. IEEE Security and Privacy, 2007. 5(1): p. 57-62.

[160] Pattabiraman, K., N. Nakka, Z. Kalbarczyk, and R. Iyer, Discovering Application-level Insider Attacks using Symbolic Execution, in Submission to 24th IFIP International Security Conference (SEC). 2008, IFIP: Cyprus, Greece.

[161] King, S.T., J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and implementing malicious hardware. 2008: USENIX Association Berkeley, CA, USA.

[162] Alkabani, Y.M. and F. Koushanfar, Active hardware metering for intellectual property protection and security.

[163] King, S.T., P.M. Chen, Y.-M. Wang, C. Verbowski, H. Wang, and J. Lorch. SubVirt: Implementing malware with virtual machines. in IEEE Symposium on Security and Privacy (Oakland). 2006.

[164] Rutkowska, J., Introducing Blue Pill. The official blog of the invisiblethings. org. June, 2006. 22.

[165] Garfinkel, T., K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: VMM detection myths and realities.

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

[167] Rhee, J., R. Riley, D. Xu, and X. Jiang, Defeating Dynamic Data Kernel Rootkit Attacks via VMM-based Guest-Transparent Monitoring.

[168] Baliga, A., V. Ganapathy, and L. Iftode. Automatic Inference and Enforcement of Kernel Data Structure Invariants. 2008: IEEE Computer Society Washington, DC, USA.

[169] Tanenbaum, A.S., J.N. Herder, and H. Bos, Can we make operating systems reliable and secure? Computer, 2006. 39(5): p. 44-51.

[170] Hieb, J. and J. Graham, Designing Security-Hardened Microkernels For Field Devices. Critical Infrastructure Protection II, 2008: p. 129.

[171] Akritidis, P., C. Cadar, C. Raiciu, M. Costa, and M. Castro, Preventing Memory Error Exploits with WIT, in Proceedings of the 2008 IEEE Symposium on Security and Privacy (sp 2008) - Volume 00. 2008, IEEE Computer Society.

[172] Barrantes, E.G., D.H. Ackley, S. Forrest, and D. Stefanovi, Randomized instruction set emulation. ACM Trans. Inf. Syst. Secur., 2005. 8(1): p. 3-40.

[173] Sovarel, A.N., D. Evans, and N. Paul, Where's the FEEB? the effectiveness of instruction set randomization, in Proceedings of the 14th conference on USENIX Security Symposium - Volume 14. 2005, USENIX Association: Baltimore, MD.

[174] Dean, D.W.a.R. Intrusion detection via static analysis. in International Conference on Security and Privacy (Oakland). 2001: IEEE.

[175] Giffin, J., Model-based intrusion detection system and evaluation, in Department of Computer Science. 2006, University of Wisconsin: Madison, WI.

[176] Chen, Y., R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M.H. Jakubowski, Oblivious Hashing: A Stealthy Software Integrity Verification Primitive, in Revised Papers from the 5th International Workshop on Information Hiding. 2003, Springer-Verlag.

[177] Jacob, M., M.H. Jakubowski, and R. Venkatesan, Towards integral binary execution: implementing oblivious hashing using overlapped instruction encodings, in Proceedings of the 9th workshop on Multimedia \&amp; security. 2007, ACM: Dallas, Texas, USA.

[178] Seshadri, A., M. Luk, E. Shi, A. Perrig, L.v. Doorn, and P. Khosla, 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.

[179] Monrose, F., P. Wyckoff, and A.D. Rubin. Distributed Execution with Remote Audit. in ISOC Network and Distributed System Security Symposium. 1999.

[180] Provos, N., M. Friedl, and P. Honeyman, Preventing privilege escalation, in Proceedings of the 12th conference on USENIX Security Symposium - Volume 12. 2003, USENIX Association: Washington, DC.

[181] Brumley, D. and D. Song, Privtrans: Automatically partitioning programs for privilege separation.

[182] Pattabiraman, K., V. Grover, and B.G. Zorn, Samurai: protecting critical data in unsafe languages, in Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008. 2008, ACM: Glasgow, Scotland UK.

[183] Nguyen-Tuong, A., D. Evans, J.C. Knight, B. Cox, and J.W. Davidson. Security through Redundant Data Diversity. in IEEE International Conference on Dependable Systems and Networks (DSN). 2008. Anchorage, AK: IEEE.

[184] Pattabiraman, K., Z. Kalbarczyk, and R. Iyer. Automated Derivation of Application-Aware Error Detectors using Static Analysis. in Internation Online Testing Symposium (IOLTS). 2007: IEEE.

[185] Abadi, M., M. Budiu, U.l. Erlingsson, and J. Ligatti. Control-flow integrity. in 12th ACM conference on Computer and communications security. 2005. Alexandria, VA, USA: ACM Press.

[186] Wu-ftp development group., WuFTP.

[187] NullLogic, NullHttpd web server. 2001.

[188] IBM. Linux TPM Device Driver. 2002 [cited; Available from: http://tpmdd.sourceforge.net/.

[189] Sen, K., D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. in Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering. 2005. Lisbon, Portugal: ACM Press.

[190] Godefroid, P., N. Klarlund, and K. Sen. DART: directed automated random testing. in ACM SIGPLAN conference on Programming language design and implementation. 2005. Chicago, IL, USA: ACM Press.

[191] Bauer, L., J. Ligatti, and D. Walker. Composing security policies with polymer. in ACM SIGPLAN conference on Programming language design and implementation. 2005. Chicago, IL, USA: ACM Press.

[192] Rosu, G. and F. Chen. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. in Third International Workshop on Runtime Verification. 2003. Boulder, Colorado: Elveiser.

[193] Gatzka, S. and C. Hochberger. On the Scope of Hardware Acceleration of Reconfigurable Processors in Mobile Devices. in 38th Annual Hawaii International Conference on System Sciences. 2005.

APPENDIX A: LIST OF PUBLICATIONS

The following papers, technical reports and journal articles have been published based on the work done in this dissertation. The chapters corresponding to the papers are indicated.

- **Towards Application-aware Security and Reliability**, Ravishankar Iyer, Zbigniew Kalbarczyk, Karthik Pattabiraman, William Healey, Peter Klemperer, Reza Farivar and Wen-Mei Hwu, *IEEE Security and Privacy Magazine,* January 2007. Material from this paper appears in Chapter 1.
- **Application-based Metrics for Strategic Placement of Error Detectors**, Karthik Pattabiraman, Zbigniew Kalbarczyk and Ravishankar Iyer, Proceedings of the International Symposium on *Pacific Rim Dependable Computing (PRDC),* December 2005. Material from this paper appears in Chapter 2.
- **Dynamic Derivation of Application-aware Error Detectors and their Hardware Implementation**, Karthik Pattabiraman, Giacinto Paolo Sagesse, Daniel Chen, Zbigniew Kalbarczyk and Ravishankar Iyer, *Proceedings of European Dependable Computing Conference (EDCC)*, October 2006. Journal version submitted to the IEEE Transactions on Dependable and Secure Computing (TDSC). Material from this paper appears in Chapter 3.
- **Automated Derivation of Application-aware Error Detectors using Static Analysis**, Karthik Pattabiraman, Zbigniew Kalbarczyk and Ravishankar Iyer, Proceedings of *International Online Test Symposium (IOLTS)*, February 2007 - Journal version to appear in the IEEE Transactions on Dependable and Secure Computing (TDSC). Material from this paper appears in Chapter 4.
- **SymPLFIED: Symbolic Program Level Fault-Injection and Error Detection Framework**, Karthik Pattabiraman, Nithin Nakka, Zbigniew Kalbarczyk and Ravishankar Iyer, Proceedings of *International Conference on Dependable Systems and Networks (DSN), June 2008.* Material from this paper appears in Chapter 5.
- **Discovering Application-level Insider Attacks using Symbolic Execution, Karthik Pattabiraman**, Nithin Nakka, Zbigniew Kalbarczyk and Ravishankar Iyer *To appear at the IFIP International Information Security Conference (SEC), May 2009*. Material from this paper appears in Chapter 6.
- **Insider Attack Detection by Information-Flow Signature Enforcement**, with William Healey, Flore Yuan, Zbigniew Kalbarczyk and Ravishankar Iyer, Under submission. Material from this paper appears in Chapter 7.

AUTHOR'S BIOGRAPHY

Karthik Pattabiraman received the Bachelor's degree in Information Technology from the University of Madras (India) in August 2001 and the Master's degree in Computer Science from the University of Illinois (Urbana-Champaign) in December 2004. His research focuses on the design of reliable and secure applications using innovations in compilers, formal methods and computer architecture. Based on his dissertation research, he was awarded the 2008 William C. Carter award by the IEEE Technical Committee on Fault-Tolerant Computing (TC-FTC) and the IFIP Working group on Dependability (WG 10.4) "*to recognize an individual who has made a significant contribution to the field of dependable computing through his or her graduate dissertation research*". Karthik's dissertation laid the foundation of the Trusted Illiac project at the University of Illinois, and Karthik was the lead student involved in building the Trusted Illiac prototype.

Karthik has done internships at Microsoft Research, IBM Research and Los Alamos National Labs, and has consulted for Microsoft Research. He has also been actively involved in the dependability community and co-organized the first and second workshops on Compiler and Architectural Techniques for Application Reliability and Security (CATARS) at the IEEE International Conference on Dependable Systems and Networks (DSN), 2008 and 2009. He is now a post-doctoral researcher at Microsoft Research and will soon join the University of British Columbia (UBC), Vancouver as an assistant professor of Electrical and Computer Engineering (ECE).