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

Doctoral Committee:

Professor Ravishankar K. Iyer, Chair
Associate Professor Vikram S. Adve
Professor Wen-Mei W. Hwu
Associate Professor Grigore Rosu

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.

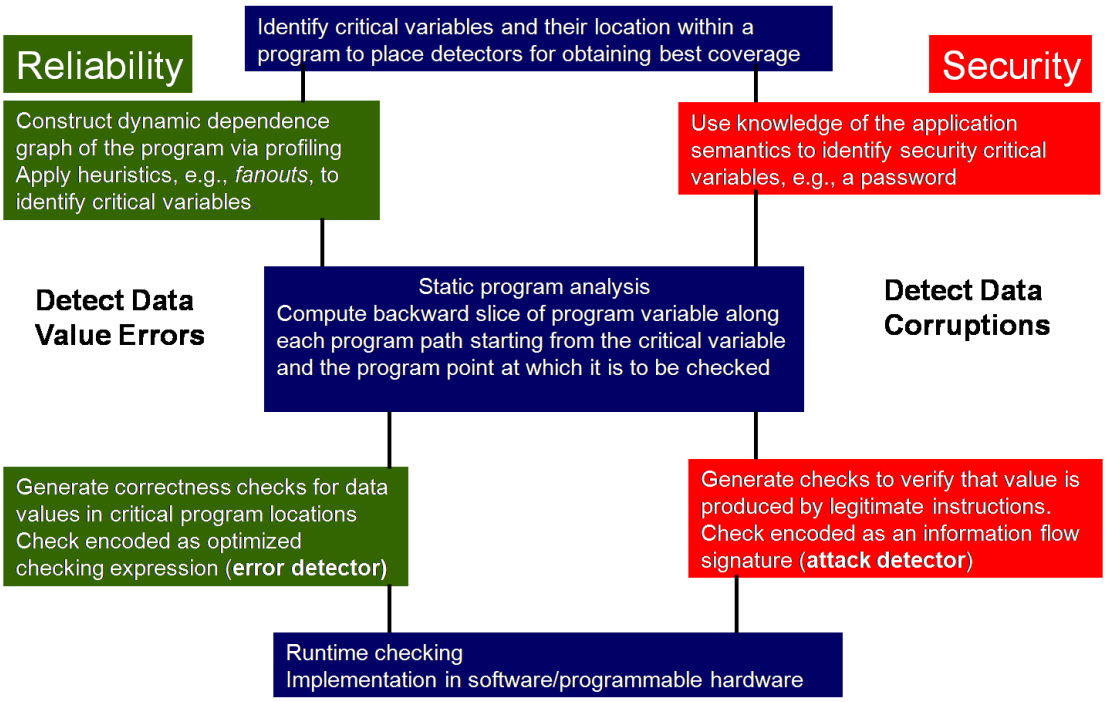


Figure 1: Conceptual unified framework for reliability and security

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.

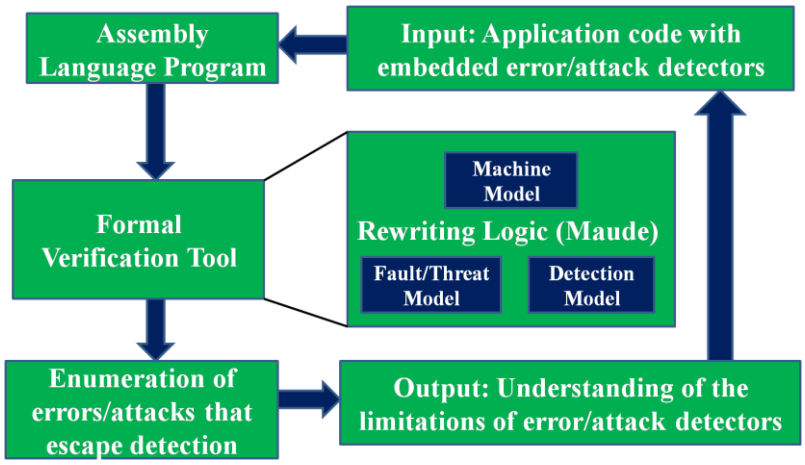


Figure 2: Unified formal framework for validation of detectors

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.

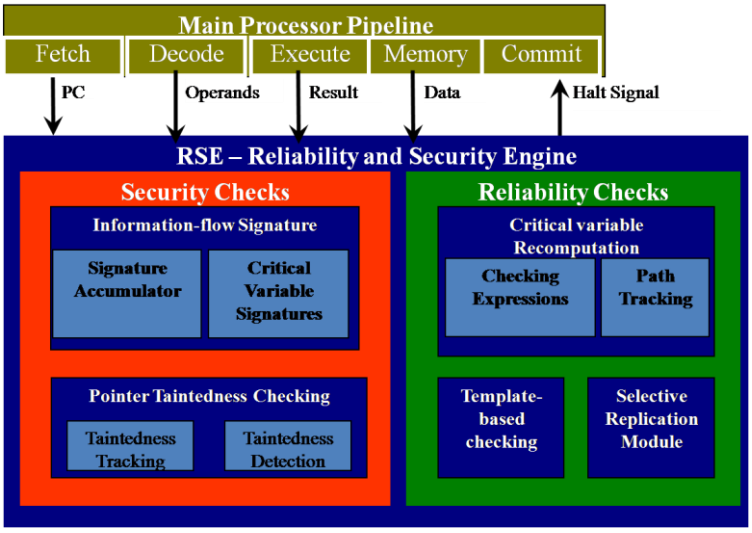


Figure 3: Hardware implementation of the detectors in the RSE Framework

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

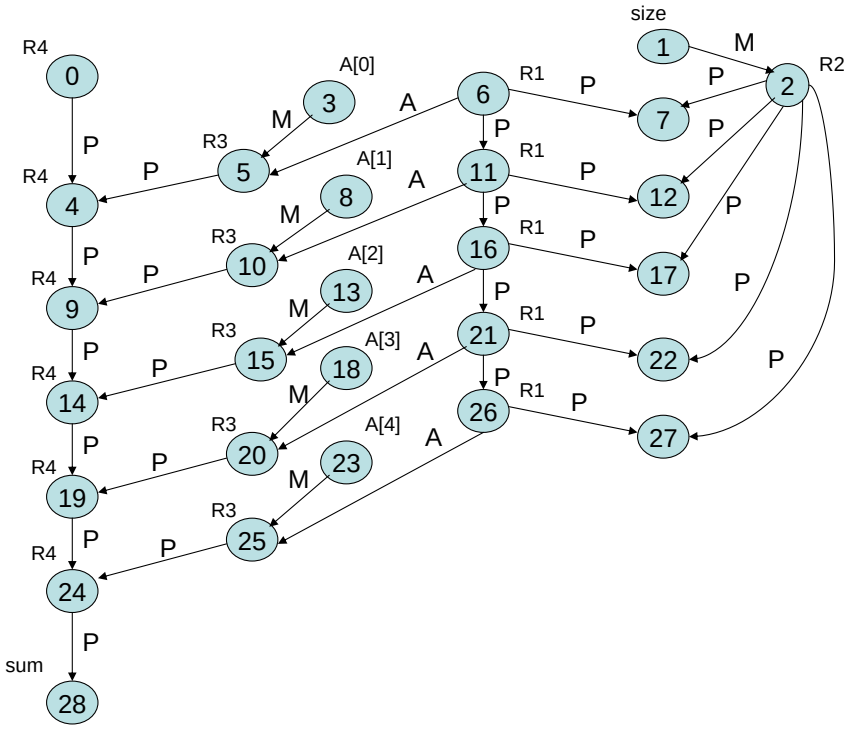


Figure 4: Example code fragment and its dynamic dependence graph (DDG)

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.

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

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

$$Propagation(N) = \{x | x \in S(N) \wedge dist(x,N) \leq CrashDist(N)\} \cup \{N\}$$
$$where \quad S(N) = \bigcup_{s \in Succ(N)} Propagation(s)$$

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.

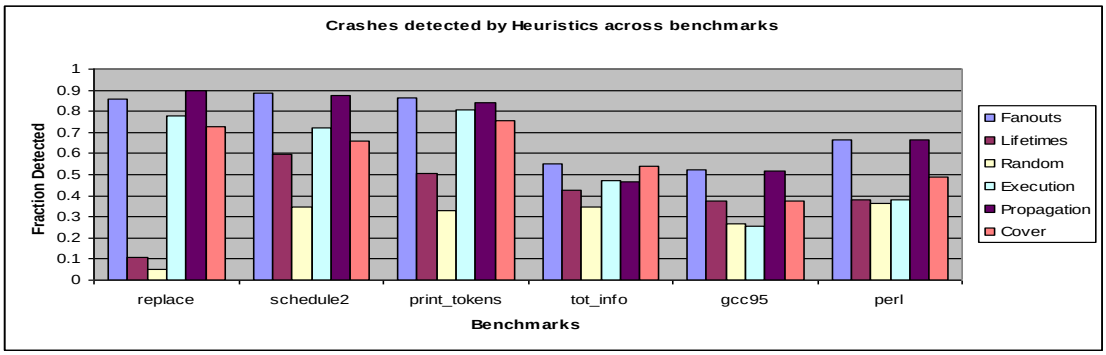


Figure 5: Crashes detected by metrics across benchmarks

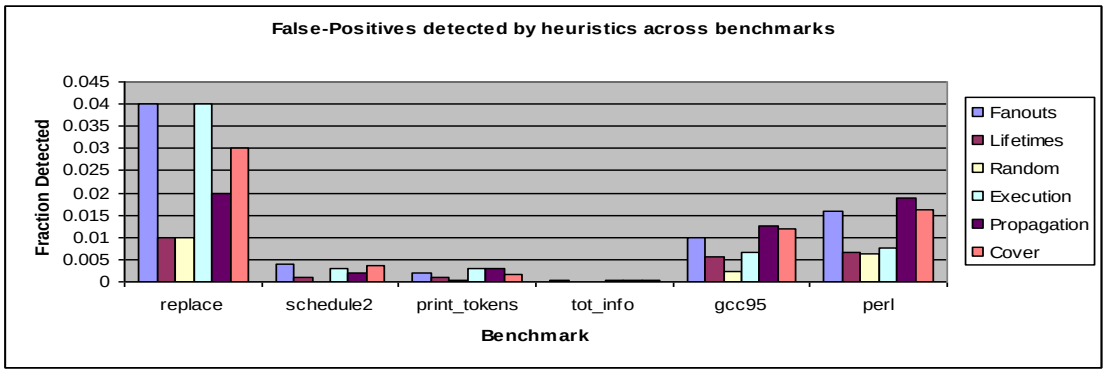


Figure 6: Benign errors detected by metrics across benchmarks

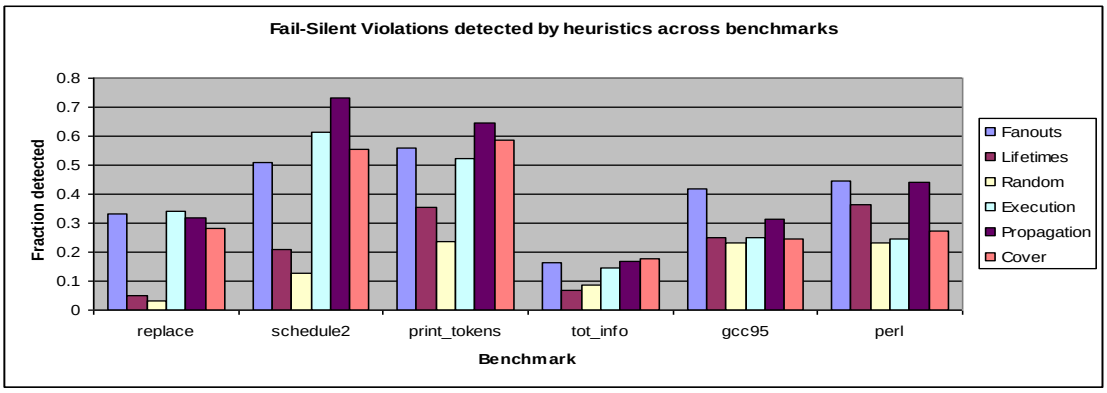


Figure 7: Fail-silent Violations detected by metrics across benchmarks

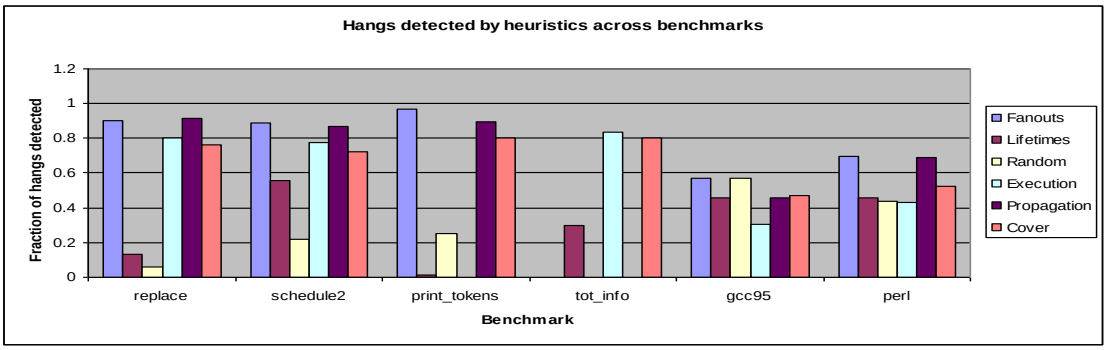


Figure 8: Hangs detected by metrics across benchmarks

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*.

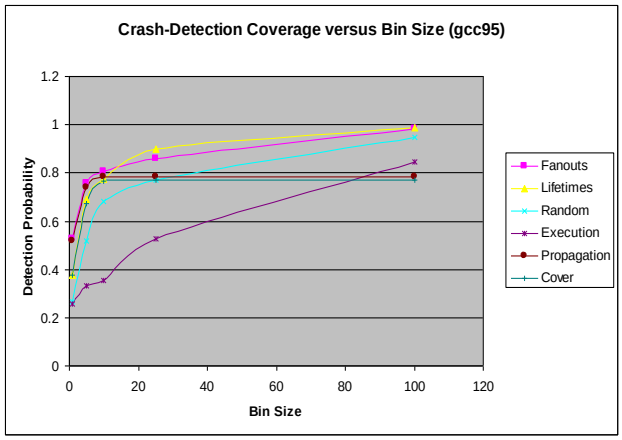


Figure 9: Effect of bin size on crash detection coverage for gcc

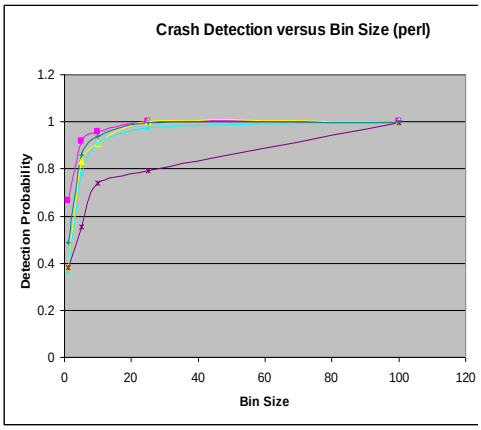


Figure 10: Effect of bin size on crash detection coverage for perl

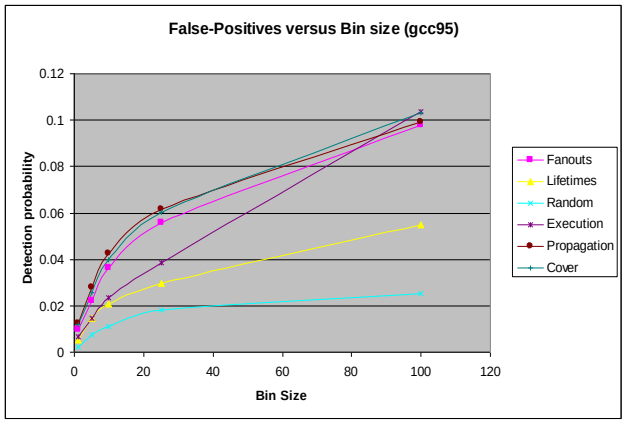


Figure 11: Effect of bin size on benign error detection rate for gcc

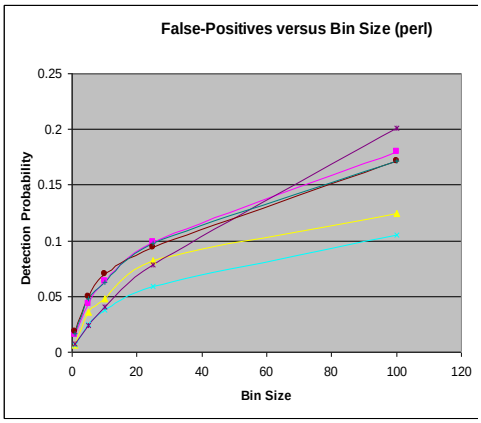


Figure 12: Effect of bin size on benign error detection rate for perl

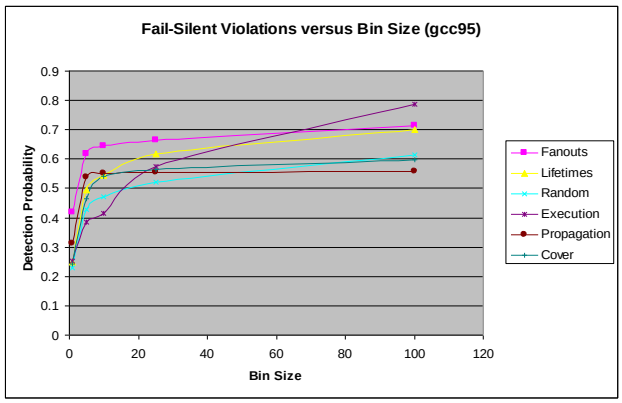


Figure 13: Effect of bin size on fail-silent violation coverage for gcc

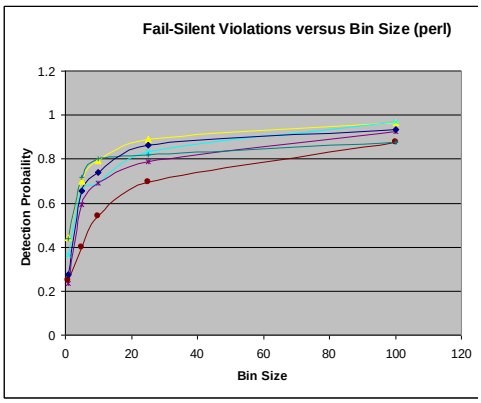


Figure 14: : Effect of bin size on fail-silent violation coverage for perl

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.

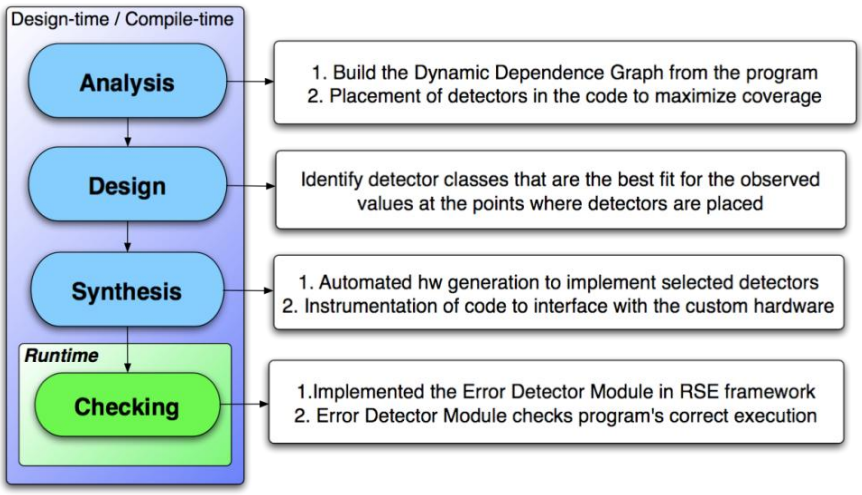


Figure 15: Steps in detector derivation and implementation process

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	$a_i \in \{ x, y, \ldots \}$	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:

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

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	Exception 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.

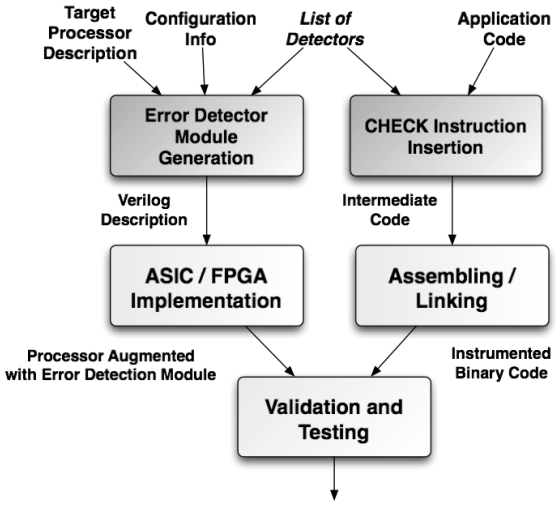


Figure 17: Design flow to instrument application and generate the EDM

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).

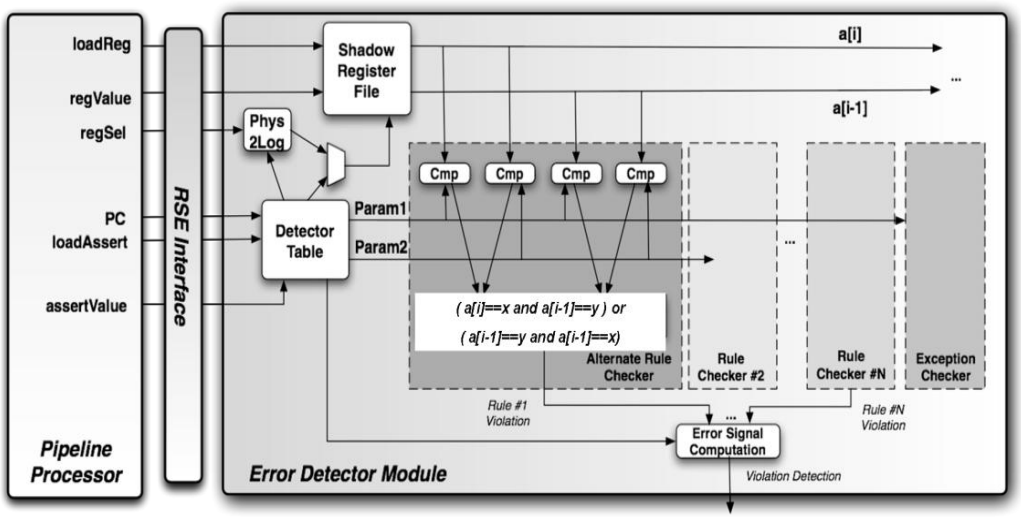


Figure 18: Architectural diagram of synthesized processor

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).

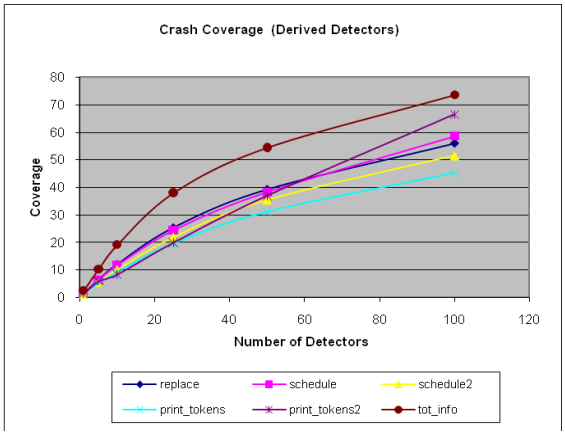


Figure 19: Crash coverage of derived detectors

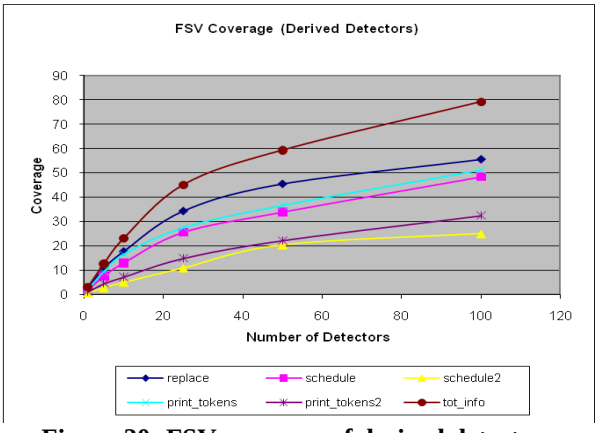


Figure 20: FSV coverage of derived detectors

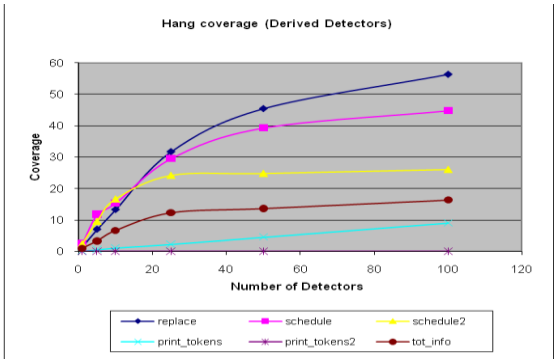


Figure 21: Hang coverage of derived detectors

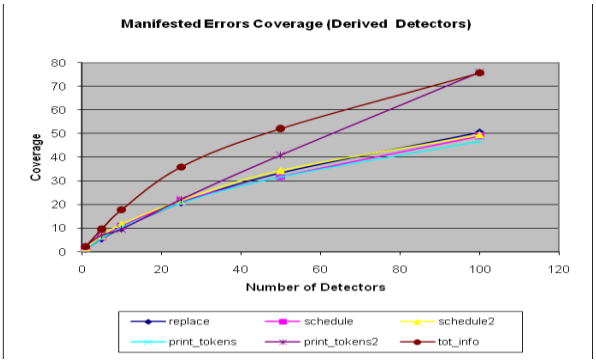


Figure 22: Total error coverage for derived detectors

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.

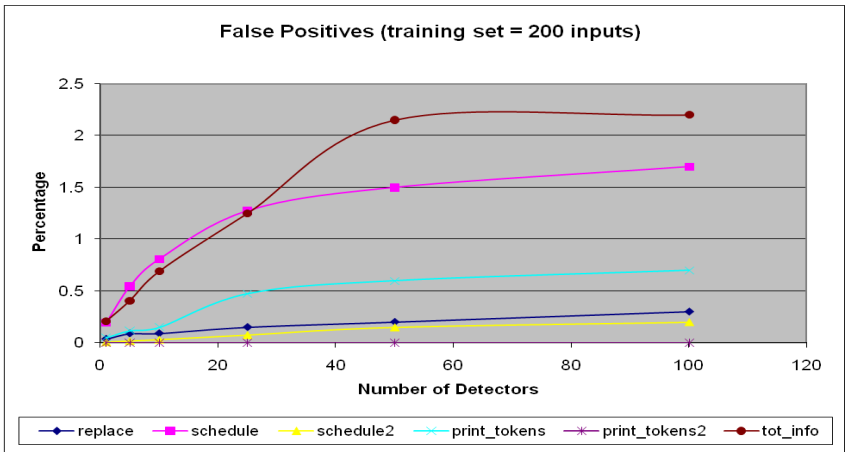


Figure 23: Percentage of false positives for 1000 inputs of each application

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.

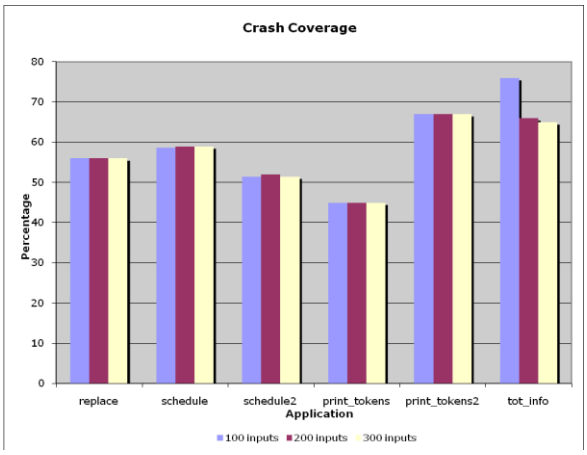


Figure 24: Crash coverage for different training set sizes

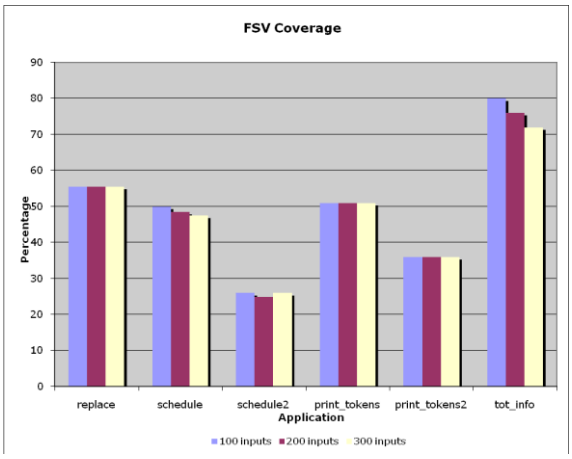


Figure 25: FSV coverage for different training set sizes

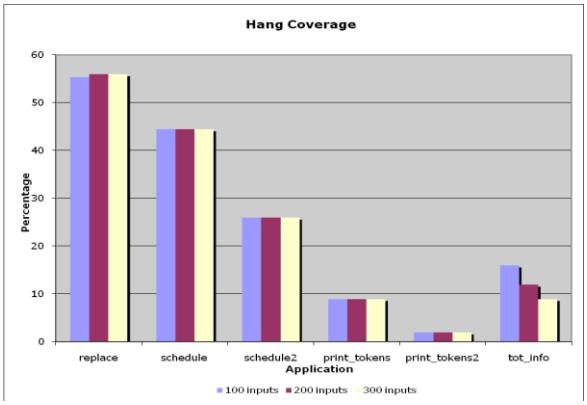


Figure 26: Hang coverage for different training set sizes

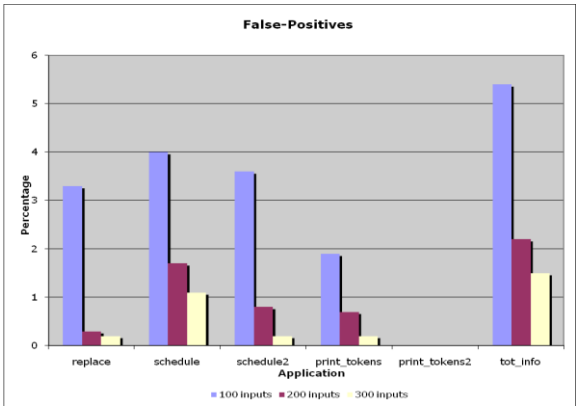


Figure 27: Benign errors for different training set sizes

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.

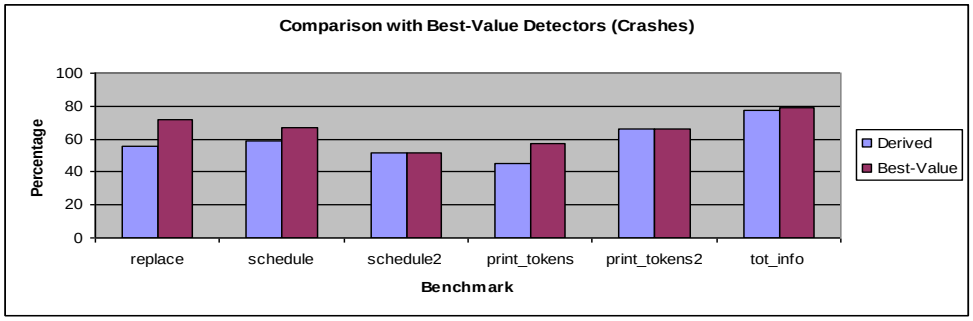


Figure 28: Comparison between best-value detectors and derived detectors for crashes

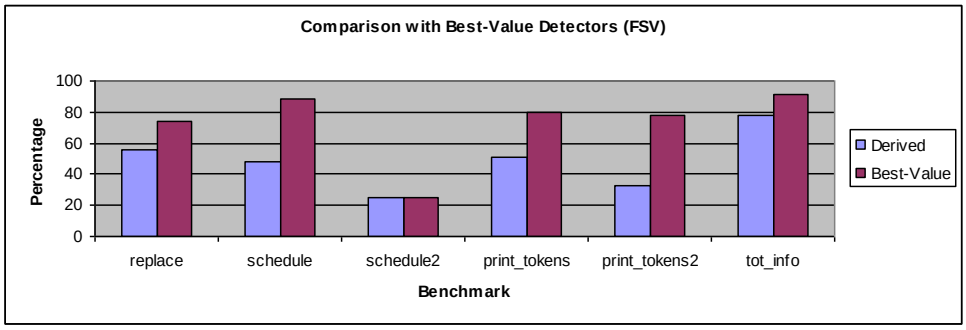


Figure 29: Comparison between best-value detectors and derived detectors for FSV

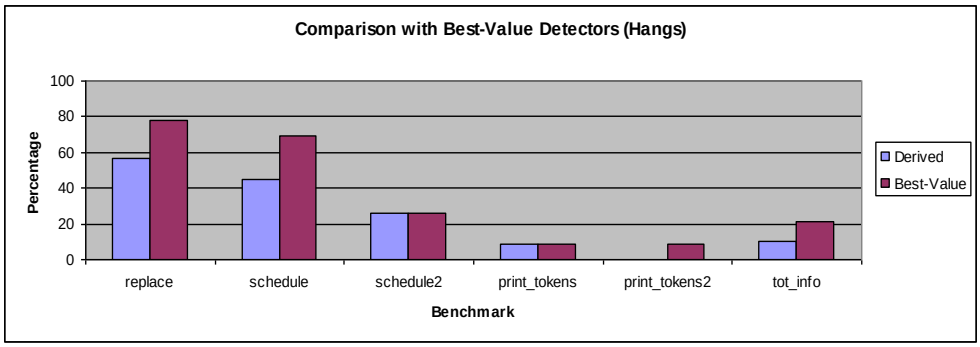


Figure 30: Comparison between best-value detectors and derived detectors for hangs

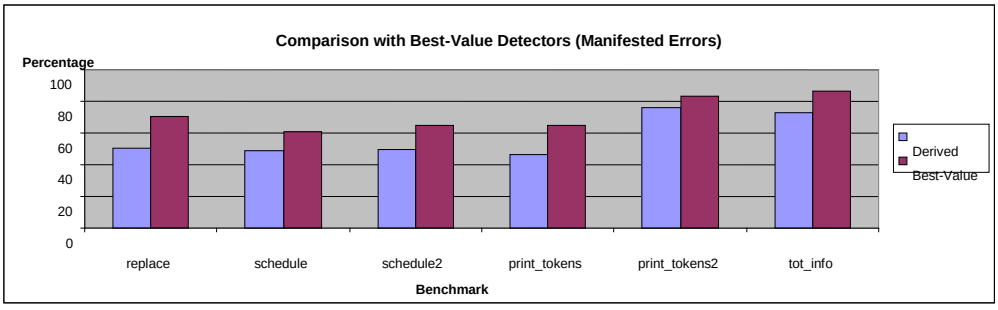


Figure 31: Comparison between best value detectors and derived detectors for manifested errors

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 $T_{with\ EDM}$ and $T_{without\ EDM}$ 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, $T_{CK\ with\ ED}$ and $T_{CK\ without\ ED}$ 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

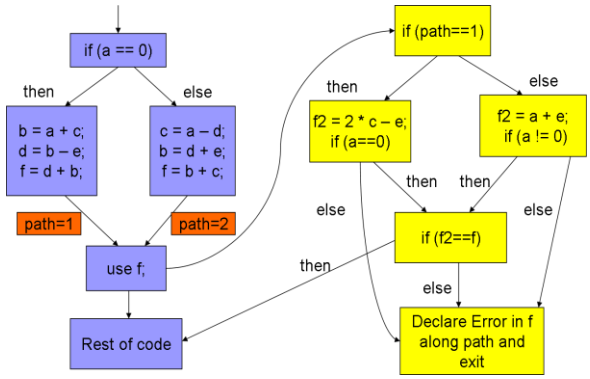


Figure 33: Example code fragment with detectors inserted

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

Erorr	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

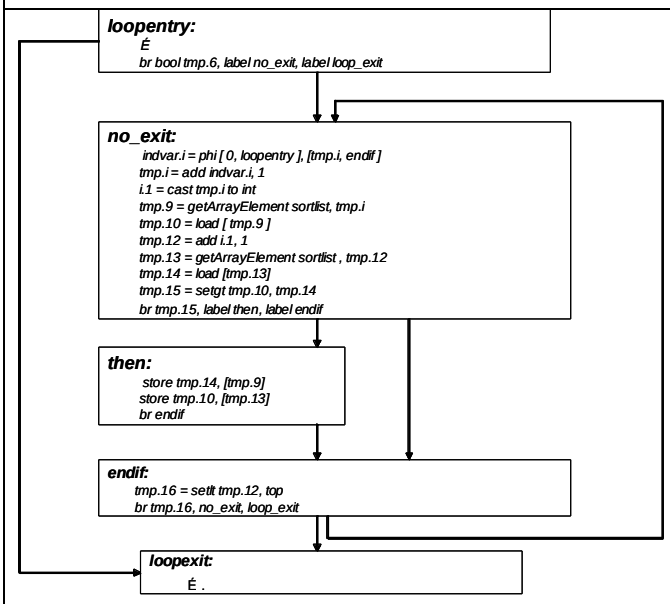
- **Errors in Execute and Memory Units:** An ALU instruction is executed incorrectly inside a functional unit, (OR) the wrong memory address is computed for a load/store instruction, resulting in data value corruption.
- **Errors in Cache/Memory/Register File Errors:** A value in the cache, memory, or 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)



(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.

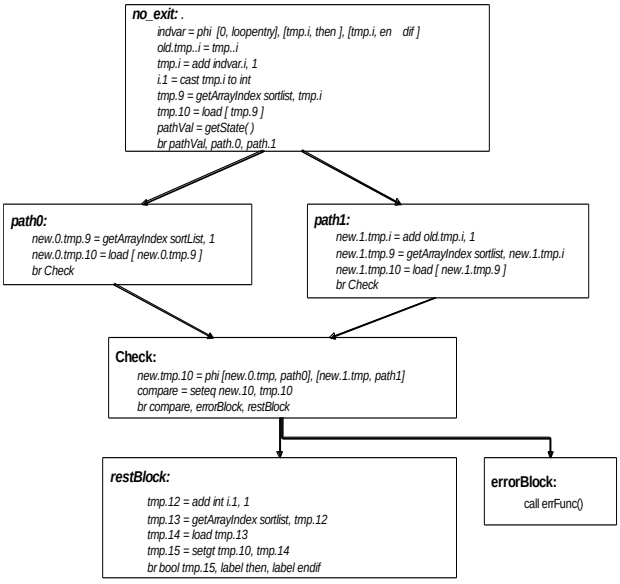


Figure 38: LLVM code with checks inserted by VRP

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.

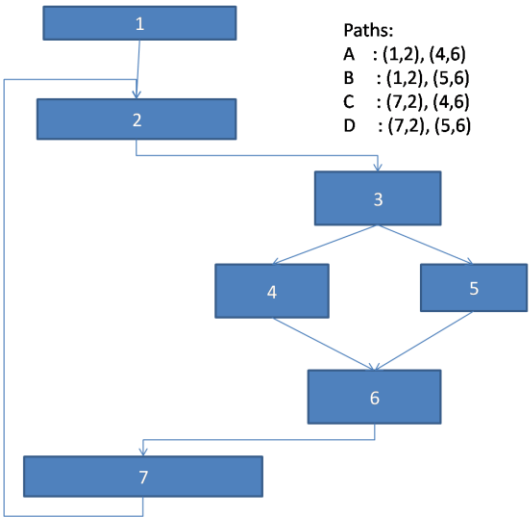


Figure 39: Example Control-flow graph and paths

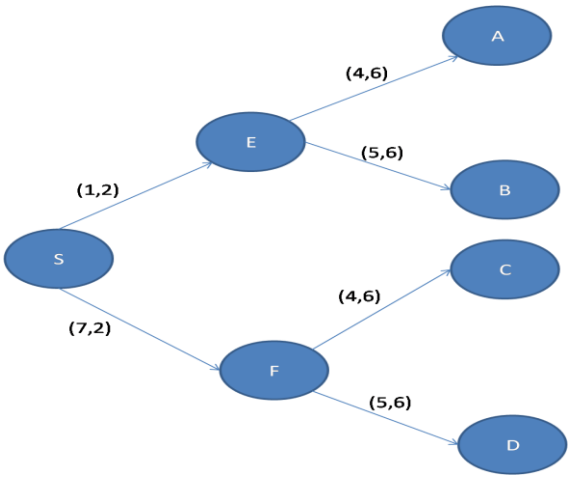


Figure 40: State machine corresponding to the Control Flow Graph

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.

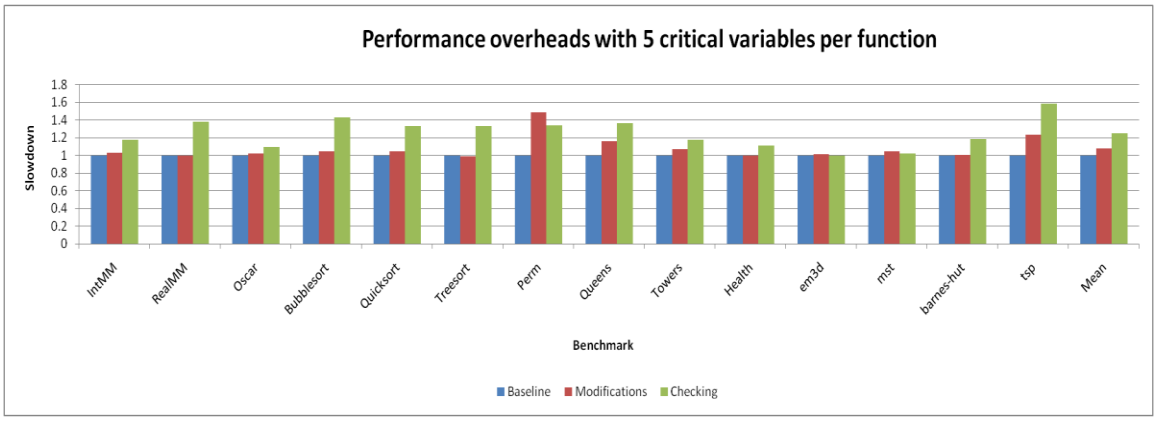


Figure 41: Performance overhead when 5 critical variables are chosen per function

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.

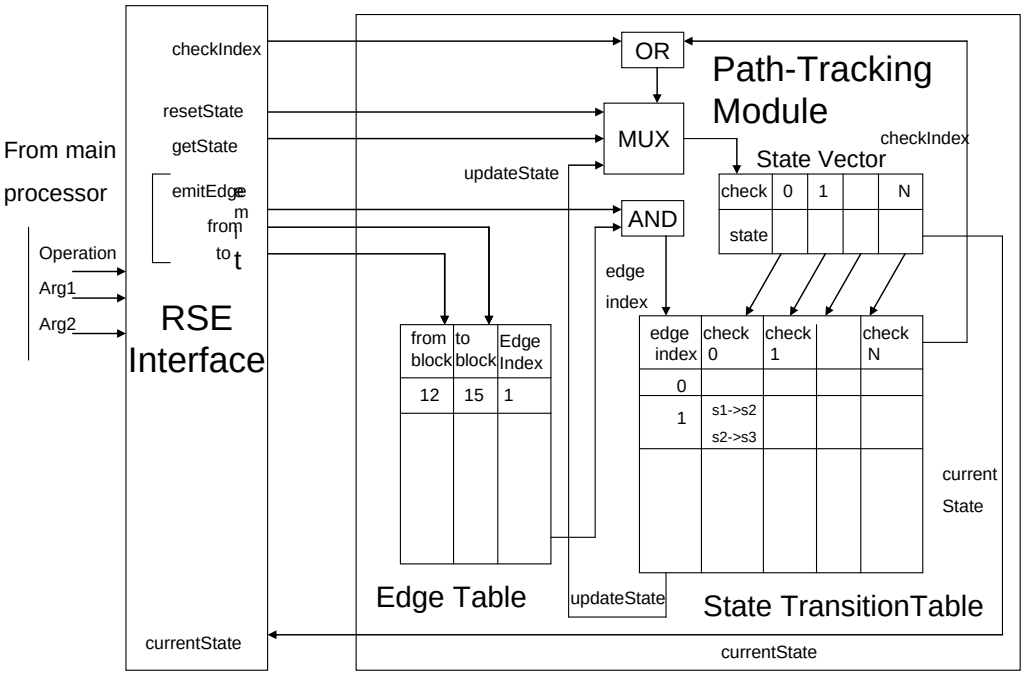


Figure 42: Hardware path-tracking module

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.