SymPLFIED: Symbolic Program Level Fault Injection and Error Detection Framework

Karthik Pattabiraman, *Member IEEE*, Nithin M. Nakka, Zbigniew Kalbarczyk, *Member IEEE*, and Ravishankar K. Iyer,*Fellow IEEE*

Abstract— *This paper introduces SymPLFIED, a program-level framework that allows specification of arbitrary error detectors and the verification of their efficacy against hardware errors. SymPLFIED comprehensively enumerates all transient hardware errors in registers, memory, and computation (expressed symbolically as value errors) that potentially evade detection and cause program failure. The framework uses symbolic execution to abstract the state of erroneous values in the program and model checking to comprehensively find all errors that evade detection. We demonstrate the use of SymPLFIED on a widely deployed aircraft collision avoidance application, tcas. Our results show that the SymPLFIED framework can be used to uncover hard-to-detect catastrophic cases caused by transient errors in programs that may not be exposed by random fault injection based validation. Further, the errors exposed by the framework help us formulate a set of error detectors for the application to avoid the catastrophic case and other incorrect outcomes.*

Index Terms— Fault injection, Model checking, Error detection

1 Introduction

Error detection mechanisms are vital for building highly reliable systems. However, generic detection mechanisms such as exception handlers can take millions of processor cycles to detect errors in programs [3]. In the intervening time, the program can execute with the activated error and perform harmful actions such as writing incorrect state to the file system. There has been significant work on efficiently placing and deriving error detectors for programs [1], [2]. An important challenge is to enumerate the set of errors the mechanism fails to detect, either from a known set or an unknown set. Typically, verification techniques target the set of errors that the detector is defined to detect. While this is valuable, one cannot predict the kinds of errors that may occur in the field, and hence it is important to evaluate detectors under arbitrary conditions in order to emulate those in the field.

Fault injection is a well-established technique to evaluate the coverage of error detection mechanisms [4], [5]. However, due to its inherent statistical nature, fault injection may miss "corner cases" that escape detection and cause the program to fail. Thus, there is a compelling need to develop a formal framework to reason about the efficiency of error detectors as a complement to traditional fault injection. While formal frameworks have been developed before, each addresses a specific error detection mechanism (for example, replication in [12]), and cannot be easily extended to general detection mechanisms.

This paper presents SymPLFIED, a framework for verifying error detectors in programs using symbolic execution and model checking. The goal of the framework is to expose error cases that would potentially escape detection and cause program failure. *To the best of our knowledge, SymPLFIED is the first framework that models the effect of arbitrary hardware errors on software, independent of the underlying detection mechanism.* It uses model checking [18] to exhaustively enumerate the consequences of the symbolic errors on the program[1]. The analysis is completely automated and does not miss errors that might occur in a real execution. However, as a result of abstracting erroneous values, it may discover errors that do not manifest in the real execution of the program[2], i.e., false-positives.

The paper makes the following contributions:

1. Introduces a formal model to represent programs expressed in a generic assembly language, and reasons about the effects of errors originating in hardware and propagating to the software application without assuming specific error detection mechanisms.
2. Specifies the semantics of general error detectors using the same formalism, which allows verification of their detection capabilities.
3. Represents errors using a single symbol, thereby coalescing multiple error values into a single symbolic value in the program. This includes both single- and multi-bit errors in the register file, main memory, cache, as well as errors in computation and control-logic.
4. Evaluates the framework on a real application (*tcas*) and discovers non-trivial cases of errors that escape detection.

Previous work [16] has analyzed the effect of hardware errors on programs expressed in a high-level language (e.g. Java). Errors are modeled as bit flips in single data variable(s) in the program. While this is an important step, it suffers from several limitations, namely (1) low-level hardware errors can affect multiple program variables and impact the program's control-flow (while modeling control-flow errors is possible in high-level languages, it is less fine-grained), (2) errors in special-purpose registers such as the *stack pointer* are difficult to model in the high-level language, and (3) errors in the language's library functions cannot be modeled as the libraries may be written in a different language than the program (while this limitation may be overcome by writing a contract for the library function, such contracts require manual specification and are time and effort intensive).

This paper considers programs represented at the assembly language level. The value of using assembly language is that many low-level hardware errors that impact the program can be represented at this level. Further, the entire application, including runtime libraries, is amenable to analysis at the assembly language level. It can be argued that in order to really analyze the impact of hardware errors, we need to model systems at even lower levels, *e.g.* the register-transfer level (RTL). However, the consequent state space explosion when analyzing the program at such low levels can render the approach impractical. Therefore, we believe that the assembly language level constitutes a judicious tradeoff between

- *Karthik Pattabiraman is with the Department of Electrical and Computer Engineering, University of British Columbia. E-mail: karthikp@ece.ubc.ca*
- *Nithin Nakka is with Nexstest Systems, a Teradyne, Inc. company. Email: Nithin.Nakka@Nextest.com.*
- *Zbigniew Kalbarczyk is with the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign. E-mail: kalbarcz@illinois.edu*
- *Ravishankar Iyer is with the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign. E-mail: rkiyer@illinois.edu*

Manuscript received Aug 16, 2012. Please note that all acknowledgments should be placed at the end of the paper, before the bibliography.

[1] In this paper, we use the term model checking to refer to exhaustive state space enumeration. This is also known as explicit state model checking.

[2] While SymPLFIED symbolically abstracts error values, it requires concrete inputs for the program in order to perform its analysis.

scalability and representativeness.

In order to evaluate the framework, the effects of hardware transient errors are considered on a commercially deployed application, *tcas*. The framework identified errors that lead to a catastrophic outcome in the application in a reasonable amount of time (less than five minutes when run on a cluster). However, a random fault injection experiment did not find any catastrophic scenario in a comparable amount of time, or when run for more than three times as much time as SymPLFIED. Further, the results from SymPLFIED were used to design error detectors for the *tcas* program. The detectors were found to be effective in avoiding the catastrophic scenario, although they suffer from some limitations that SymPLFIED identifies. Finally, SymPLFIED is also demonstrated on a larger program, *replace* to demonstrate its scalability.

2 Related Work

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

Error Detection: Many error detection mechanisms have been proposed in the literature, along with formal proofs of their correctness [10], [11]. However, the verification methodology is usually tightly coupled with the mechanism under study. For example, [11] proposes and verifies a control-flow checking technique by constructing a hypothetical program augmented with the technique and model-checks the program for missed detections. The program is carefully constructed to exercise all possible cases of the control-flow checking technique. However, it is non-trivial to construct such representative programs for other error detection mechanisms.

Perry et al. [12] propose the use of type-checking to verify the fault-tolerance provided by a specific error detection mechanism, namely compiler-based instruction duplication. The paper proposes a detailed machine model for executing programs. The faults in the fault model (Single-Event Upsets) are represented as transitions in the machine model. The advantage of the technique is that it allows reasoning about the effect of low-level hardware faults on the whole program, rather than on individual instructions or data. However, the detection mechanism (duplication) is tightly coupled with the machine model, due to inherent assumptions that limit error propagation in the program and may not hold in programs protected with other mechanisms than duplication.

Other recent work proposes a formal logic to verify programs under a wide range of fault models and detection techniques [13]. The technique presented in [13] either accepts or rejects a program based on whether the detectors successfully detect an error. However, it does not consider the consequences of the error on the program. As a result, the program may be rejected by the technique even though the error is benign and has no effect on the program.

Symbolic execution has been used for a wide variety of software testing and maintenance purposes [14]. The main idea in these techniques is to execute the program with symbolic values rather than concrete values and to abstract the program state as symbolic expressions. An example of a commercially deployed symbolic execution technique to find bugs in programs is Prefix [15]. However, Prefix assumes that the hardware does not experience errors during program execution.

A symbolic approach for injecting faults into programs was introduced in [16]. The goals of this approach are similar to ours, namely to verify properties of fault-tolerance mechanisms in the presence of hardware errors. The technique reasons about the effect of single bit-flips on programs written in the Java language. However, as pointed out earlier, a hardware error can have wide-ranging consequences on the program, which cannot be easily modeled at the high level.

Further, the technique presented in [16] uses theorem proving to verify the error resilience of programs. Theorem proving has the intrinsic advantage that it is naturally symbolic and can reason about the non-determinism introduced by errors. However, theorem proving requires considerable programmer intervention and expertise, and cannot be completely automated for many important classes of programs.

Program verification techniques have been used to prove that a program's code satisfies a programmer-supplied specification [7]. The specification precisely outlines the expected result of the program given certain initial conditions. Typically, program verification techniques are geared towards finding software defects and assume that the hardware and the program environment are error free. In other words, they prove that the program satisfies the specification *provided* the hardware platform on which the program is executed does not experience errors. Further, program verification techniques operate on an abstract representation of the program extracted from the program code. The abstractions are derived based on the specific property being checked and cannot be used for evaluating the program under arbitrary hardware errors as such errors may not manifest in the abstraction.

Formal techniques have also been extensively applied to **microprocessor verification** [6]. The techniques attempt to prove that the implementation of the processor conforms to an architectural specification usually in the form of a processor reference manual. Processor verification techniques typically focus on unmasking hardware design defects, as opposed to transient errors due to electrical disturbances or radiation.

Soft-errors in hardware: The techniques presented in [8] and [9] consider the effects of hardware transient errors (soft errors) on error detection mechanisms implemented in hardware. While these techniques are useful for applications implemented as hardware circuits, it is not clear how the technique can be extended for reasoning about the effects of errors on programs. This is because programs are normally executed on general-purpose processors in which the manifestation of a low-level error is different from an error in a hardware implementation of the application.

Summary: The formal techniques considered in this section predominantly fall into the category of software-only techniques which do not consider hardware errors [7], or into the category of hardware-only techniques which do not consider the effects of errors on software [6]. Further, existing verification techniques are often coupled with the detection mechanism (e.g., duplication) being verified [11], [12].

Therefore, there exists no *generic* technique that allows reasoning about the effects of *arbitrary* hardware faults on software, and can be combined with an arbitrary fault model and detection technique(s). This is important for enumerating all hardware transient errors that would escape detection and cause programs to fail. Moreover, the technique must be *automated*. This paper answers the question: "*Is it possible to develop a framework to reason about the effects of arbitrary hardware errors on applications in an automated fashion, to understand where error detection mechanisms fail in detecting errors?*".

3 APPROACH

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

3.1 Framework

The SymPLFIED framework accepts a program with or without error detectors, and enumerates all errors (in a particular class) that would not be detected by the detectors (if any) and lead to program failure. Figure 1 presents the conceptual design flow of the SymPLFIED framework.

Inputs: The inputs to the framework are (1) a program written in a target assembly language (e.g., MIPS) along with its inputs, (2) error detectors embedded in the program code (optionally), and (3) a class of hardware errors to be considered (e.g., control-flow errors, register file errors) by the system. Note that the error detectors are not a necessity. In Section 6.2, we deploy the framework on the *tcas* program which has no embedded error detectors.

Assembly Language: We define a generic assembly language in which programs are represented for formal analysis by the framework. Because the language defines a set of architectural abstractions found in many common Reduced Instruction Set Computer (RISC) architectures, it is portable across these architectures [26]. The assembly language has direct support for (1) input/output operations, so that programs can be analyzed independent of the Operating System (OS), (2) invocation of error detectors using special annotations, called *CHECK*, which allows detectors to be represented in line with the program's text, and (3) exception handling and reporting of errors without the need for an Operating System (OS).

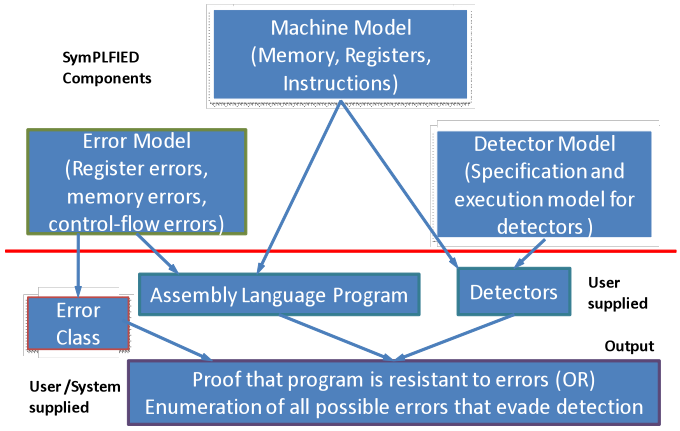

Figure 1: Conceptual representation of the SymPLFIED Framework

Operation: The program is expressed using a generic assembly language described in Section 5. This language is automatically translated into a formal mathematical model that can be represented in the Maude system [17]. Since the abstraction is close to the actual program in assembly language it is sufficient for the user to formulate generic specifications, such as an incorrect program outcome or an exception being thrown. Such a low-level abstraction of the program is useful to reason about hardware errors.

The formal model can be rigorously analyzed under error conditions against the above specifications using techniques such as model checking and theorem-proving. *In this paper, model checking is used because it is completely automated and requires no programmer intervention*[3]. However, the SymPLFIED framework supports the use of theorem-proving and other formal tools provided in the Maude system if desired [19].

Outputs: The framework uses the technique described in section 3.2 and outputs either of the following:

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

Components: The framework consists of the following models:

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

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

Extensibility: The models described above are decoupled from each other and can be composed together in a plug-and-play fashion. For example, the machine model can be replaced with a model corresponding to a different architecture without changing either the error model or the detector model. Similarly, the detector and error models can be modified independent of the other models. This is because each model exposes well-defined interfaces to the other models, and as long as the interfaces are obeyed, the implementation of the models can be modified independently.

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

[1] The model must satisfy certain properties such as termination, coherence, and sufficient completeness [17].,

[2] The model must be an accurate representation of the system being modeled.

The first requirement can be satisfied by formally analyzing the specification using automated checking tools for each desirable property listed above. This is obtained for free by expressing the model using Maude's language, as Maude has formal tools to check the conformance of the model to the properties [19]. We do not discuss this part of the analysis.

However, the second requirement is much harder to ensure as it cannot be checked by formal tools and is usually left to the model creator. We validate the model (Section 6.3) by injecting thousands of faults into a processor simulator, SimpleScalar [21] and comparing the results with those from SymPLFIED.

3.2 Symbolic Fault Propagation

The SymPLFIED approach represents the state of all erroneous values in the program using the abstract symbol *err*. The *err* symbol is propagated to different locations in the program during execution using simple error propagation rules (shown in section 5.2). The symbol also introduces non-determinism in the program when used in the context of comparison and branch instructions or as a pointer operand in memory operations. Because the same symbol is used to represent all erroneous values in the program, the approach

[3] Of course, the results from model checking still need human interpretation.

distinguishes program states based on where errors occur rather than on the nature of the individual error(s). As a result, it avoids state explosion and can keep track of all possible places in the program the error may propagate to starting from its origin.

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

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

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

Completeness is more important than accuracy from the point of view of designing detection mechanisms, as we can always augment the set of error detectors to conservatively protect against a few false-positives. While a small number of false-positives can be tolerated, it must be ensured that the technique does not find too many false-positives as the cost of developing detectors to protect against the false-positives can overwhelm the benefits of detection. SymPLFIED uses a custom constraint solver to remove false-positives as much as possible (Section 5.2).

3.3 Categories of Errors Considered

SymPLFIED considers transient errors in memory/registers, computation and control-logic, which manifest in the architectural state of the processor. The reason it is possible to represent such a broad class of errors in the model is because the program is represented in assembly language, which exposes its low-level state to the framework.

Errors in memory/registers are modeled by replacing the contents of the memory location or register by the symbol *err. No distinction is made between single- and multi-bit errors.*

Errors in computation are modeled based on where they occur in the processor pipeline *and* how they affect the architectural state. The manifestation of these errors is shown in Table 1.

Errors in the processor's control logic (such as in the register renaming unit) are modeled based on their manifestation in the other parts of the processor. These are also shown in Table 1.

3.4 Scalability and Guarantees

As in most model checking approaches, the exhaustive search performed by SymPLFIED can be exponential in the number of instructions executed by the program in the worst case. However, the error detection mechanisms in the program can be used to optimize the state space exploration process. For example, if a certain code component protected with detectors is proved to be resilient to all errors of a particular class, then such errors can be ignored when considering the space of errors that can occur in the system as a whole. This suggests a hierarchical approach, where first the detection mechanisms deployed in each component are proved effective, and then inter-component interactions are considered.

Because SymPLFIED exhaustively explores every possible consequence of an error in the program, it guarantees the completeness of the failure outcomes produced due to the error. However, there are two barriers to achieving this guarantee in practice. First, the model checker may terminate before exploring the entire space if it runs out of memory or resources. While we specify a timeout for each model checking task to limit the total number of states explored by the model-checker, some tasks run out of memory even before this timeout is reached, thereby voiding the guarantee The second barrier to achieving the guarantee is that SymPLFIED does not consider the interactions of the program with its environment. For example, a real-time program may miss its deadlines due to delays introduced by the error; or the program may invoke a system-call with the wrong arguments due to the error, leading to its termination.

4 Examples

This section illustrates the SymPLFIED approach in the context of an application that calculates the factorial of a number shown in Figure 2. The program is represented in the generic assembly language presented in Section 3.1. The details of the language are presented in Table 4.

4.1 Error Injection

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

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

Complexity: Note that in order for a physical fault injection approach to discover the same set of outcomes for the program as SymPLFIED, it would need to inject all possible values (in the integer range) into the loop counter variable. This can correspond to 2^k cases in the worst case, where k is the number of bits used to represent an integer. In contrast, SymPLFIED considers at most $(n+1)$ possible cases, in this example, where n is the number of iterations of the loop. This is because each fork of the execution at the loop condition results in the *true* case exiting the loop and the program. In the general case though, SymPLFIED needs to consider 2^n cases.

Table 1: Computation error categories and how they are modeled by SymPLFIED

Fault origin	Error symptom	Conditions under which modeled	Modeling procedure	
Instruction Decoder	One of the fields of an instruction is corrupted	One valid instruction is converted to another valid instruction	Instructions writing to a destination (e.g., *add*) - change the output target	*err* in both the original and faulty targets (register or memory)
			Instructions with no target (e.g., *nop*) – replace with instructions with targets (*e.g., add)*	*err* in the new wrong target (register or memory)
			Instructions with a single destination (,e.g. *add)*– replace with instruction with no target (e.g., *nop*)	*err* in the original target location (register or memory)
Address or Data Bus	Data read from memory, cache or register file is corrupted	Single and multiple bit errors in the bus during instruction execution	Errors in register data bus	*err* in source register(s) of the current instruction
			Error in cache bus	*err* in target registers of *load* instructions to the location
			Error in memory bus	*err* in target register of *load* instructions to the location
Processor Functional Unit	Functional unit output is corrupted	Single and multiple bit errors in registers/memory	Functional unit output to register or memory	*err* in register or memory file being written to by the current instruction
Instruction Fetch Mechanism	Errors in the fetch unit	Single or multiple bit errors in PC or instruction	Fetch from an erroneous location due to error in *PC*	*PC* is changed to an arbitrary but valid code location
			Error in instruction while fetching it from the instruction cache	Modeled as errors with their origin in the instruction decoder (see row 1 of table)
Control Logic	Register rename error	An architectural register mapped to an incorrect physical register	Instruction reads from or writes to erroneous register instead of the correct register according to the instruction.	Destination register has an *err* in both the original and faulty registers.
				Source register has *err* in source operand
	Forwarding unit error	Instruction can read sources from one of many functional units	An incorrect functional unit's output is provided as the source for the instruction	*err* in source operand
	Scheduler error	Scheduler checks for operands that are about to become ready	Scheduler errors sets operand ready bit high even though operands are not yet ready	*err* in the source operand (reads stale value)

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

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

Figure 2: Program to compute factorial with (A) no error detectors, and (B) embedded error detectors.

4.2 Error Detection

We now discuss how SymPLFIED supports error detection mechanisms in the program. Figure 2B shows the program in Figure 2A augmented with error detectors. Recall that detectors are invoked through special CHECK annotations as explained in Section 3.1. The error detectors together with their supporting instructions (*mov* instruction in line 8) are shown in bold. The same error is injected as before in register *$3* (the new line number is 11). As shown in Section 4.1, the loop back-edge is executed and the execution is forked at the loop condition *($3 > $4)*.

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

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

The execution is forked once again at the second detector into *true* and *false* cases. The *true* case continues execution and propagates the error in the program as before. The *false* case of the check throws an exception and the detector fails, thereby detecting the error. The constraints for the *false* case, namely *($6 * $3 >= $6 * $1)* are also remembered. Based on this constraint, as well as the earlier constraint *($3 > $4), the constraint-solver deduces that the second detector will detect the error if and only if the fault in register $3 causes it to have a value greater than the initial value read from the input (stored in register $1).*

The programmer can then formulate a detector to handle the case when the error causes the value of register $3 to be lesser than the original value in register $1. Therefore, the errors that evade detection are made explicit to the

programmer (or to an automated mechanism) who can make an informed decision about handling the errors.
The error considered above is only one of many possible errors that may occur in the program. These errors are too numerous for manual inspection and analysis as done in this example. Moreover, not all these errors evade detection in the program and lead to program failure.
The main advantage of SymPLFIED is that it can quickly isolate the errors that would evade detection and cause program failure from the set of all possible transient errors that can occur in the program. It can also show the programmer an execution trace of how the error evaded detection and led to the failure. This is important in order to understand the weaknesses in existing detection mechanisms and improve them.

5 IMPLEMENTATION

We have implemented the SymPLFIED framework using the Maude rewriting logic system.
Rewriting logic is a general-purpose logical framework for specification of programming languages and systems. **Maude** is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications [17]. *The main advantage of Maude is that it allows a wide variety of formal analysis techniques to be applied on the same specification*. [19] provides a primer on Maude.
Custom Translator: In order to make programs for existing architectures compatible with SymPLFIED, we provide a facility to translate programs written in the target architecture's assembly language into SymPLFIED's assembly language. While we support only the MIPS instruction set, it is possible to also support other RISC architectures (see Appendix C for details).
In this section, we describe the details of the machine, detector and error models, and model checking.

5.1 Machine Model

This section describes the machine model for executing assembly language programs using Maude.
Equations and Rules: As far as possible, we have used equations instead of rewrite rules for specifying the models. The main advantage of using equations is that Maude executes equations much faster than rewrite rules. However, equations must be deterministic and cannot accommodate ambiguity. The machine model is completely deterministic because for a given instruction and input sequence, the final state can be uniquely determined in the absence of errors. Therefore the machine model can be represented entirely using equations. The detection model is also deterministic and uses only equations. However, the error model is non-deterministic and hence requires rewrite rules.
Assumptions: The following assumptions are made by the machine model when executing a program. Section 5.5 discusses the implications of these assumptions.

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

Machine State: The central abstraction used in the machine model is the notion of *machine state*, which consists of the mutable components of the processor's structures. The machine state is carried from instruction to instruction in program execution order, with each instruction optionally looking up and/or updating the state's contents. The machine state is obtained by concatenating one or more of the machine elements in a single 'soup' of entities. For example, the soup, *PC(pc) regs(R) mem(M) input(In) output(out)*, represents a machine state in which the (1) current program counter is denoted by *pc*, (2) register file is denoted by *R*, (3) memory is denoted by *M*, and (4) input and output streams are *in* and *out* respectively. Note that the program's code is not considered part of the machine state as it is assumed to be immutable.
Logical Organization: The machine model is divided into six sub-models, each of which represents a specific aspect of the machine being modeled. The sub-models and their functionality are described in Table 2.

Table 2: Sub-models of the machine model

Sub-model	Functionality
Fetch and Decode	Retrieves instructions from code memory and converts them to a form suitable for execution
Register	Implements register file lookups and update operations
Memory	Implements memory lookups and update operations
Exception	Handles error conditions and exceptional cases encountered by the program at runtime
Stream	Represents input, output, and error streams of a program. Needed to implement console and file I/O
Execute	Executes an instruction by updating the state of the machine

We consider each of the sub-models in Table 2 as follows:
Fetch/Decode sub-model: This sub-model defines the operations to retrieve an instruction from code-memory and interpret it. As mentioned before, code is stored in a separate memory and is not part of the machine state. In the equations below, C represents the code memory, L represents the address of the fetched instruction, and I represents the instruction stored at location L.

$$ceq\ fetch(\ [L \mid I]\ C, L\ ) = I \quad if\ notTerminal(L)\ .$$
$$ceq\ fetch(\ C, L\ ) = throw(\ instException\ L\ )\ if\ notTerminal(L)\ .$$

In the above equations, note that the fetch process requires that the program has not terminated (i.e., the halt or throw instructions are executed). Further, Maude follows a "match-first" strategy for equations and hence the first equation is used for matching the labels of instructions in the code memory (as it is written first in the module). Only if the first equation does not return any match is the second equation triggered, which throws an *instException*.
Currently, there is no separate model for decode as instructions are directly stored in their decoded form in code memory. This is because the interpretation of an instruction does not depend on previous instructions in a RISC processor (which is what we model).
Register sub-model: The register file is modeled as an array of 32 general purpose integer registers. We do not

currently model floating-point registers. The program counter is not part of the register file, but is stored separately as part of the machine state. The equations for reading and writing to registers are as follows. In the equations, *R* refers to the register file, *r* refers to a specific register, and *v*, *v1*, *v2* are values[4].

ceq R(r = v) [r] = (v) if (r =/= $(0)) .
ceq R[r] = (0) if (r == $(0)).
ceq ((r = v1) R) [r <- v2] = ((r = v2) R) if (r =/= $(0)) .
ceq (R [r <- v2]) = R if (r == $(0)) .

The first two equations correspond to reading of registers in the register file and the next two correspond to writing of registers. Note that updates to register $0 are ignored as the register $0 is hardwired to 0 in the MIPS processor.

Memory sub-model: Memory is modeled similar to the register file, but with three important differences: First, new locations can be added to the memory when they are first written to in the program. Second, memory reads and writes need to be aligned to the word size of the machine or else an alignment exception is thrown (addresses in unaligned loads and stores are aligned before being issued). Finally, reads of an uninitialized memory location will throw an exception.

The equations below represent the modeling of the above-described behavior. *M* represents the memory state, *a* represents an address, and *v*, *v1*, *v2* represent values.

eq (a = v) M [a] = v .
ceq M [a] = memException(a) if isAligned(a) .
ceq M [a] = alignException(a) .
eq (((a = v2) M) [a <- v1]) = (a = v1) M .
ceq (M [a <- v1]) = ((a = v1) M) if isAligned(a) .
ceq (M [a <- v1]) = exMem(alignException(a), M) .

Exception sub-model: The exception sub-model includes the exceptions that are thrown in the machine sub-model. The default action on an exception is to terminate execution and print the exception to the output stream. Table 3 shows the list of supported exceptions. The equation to throw an exception is presented later.

Table 3: Exceptions supported in the machine model

Exception Type	Explanation
memException	Address not found in memory
instException	Address does not contain valid instruction
divException	Attempt to divide by 0 in ALU instruction
checkException	Check failed (corresponds to detector model)
IOException	Input stream empty or output stream full
alignException	Address not aligned to word size of MIPS
timeoutException	Program timed out after N instructions

Stream sub-model: The stream sub-model provides an abstraction of the input-output interface for the program. This is required since we do not model the operating system. Each program is assumed to have an input stream and an output stream by default. The program may request to open other streams to model file operations. The only operations allowed on a stream are reading the next value or appending a value to the stream. These are expressed by the operators << and >> respectively; however, the equations are not presented.

Execute Sub-model: The execute sub-model is used to execute instructions in the machine and is responsible for updating the machine state. It defines the initial state of the machine and starts executing the program.

The command to start a program takes as argument the program and its inputs. Its equation is as follows:

eq start(pgm, input) = { C, < fetch(C, 0), initState(input) > } .

In the above equation, the <_,_> operator represents the machine state obtained by executing an instruction (given by the first argument) on a machine state (given by the second argument). *C* represents the code of the program and is written outside the state to enable faster rewriting by Maude (as it is assumed to be immutable). The {_,_} operator groups together the code and the machine state into what is known as a *super-state*. Super-states represent intermediate stages of the program's execution.

The initial state of the machine (initState) is as follows:

- The program counter is initialized to the first instruction of the program (instruction 0).
- The register file is initialized to all zeroes.
- The input stream is initialized to the program's input and the output stream is cleared.
- The memory contents of the program are cleared and the address 0 is initialized to the value 0.

Instruction Classes: Table 4 presents a comprehensive view of the instructions supported by SymPLFIED.

Table 4: Assembly language instructions supported

Instruction	Semantics
Arithmetic Instructions	
movi rd, imm	R[rd] ← imm
mov rd, rs	R[rd] ← R[rs]
addi rd, rs, imm	R[rd] ← R[rs] + imm
add rd, rs, rt	R[rd] ← R[rs] + R[rt]
subi rd, rs, imm	R[rd] ← R[rs] – imm
sub rd, rs, rt	R[rd] ← R[rs] - R[rt]
mult rd, rs, rt	R[rd] ← R[rs] * R[rt]
div rd, rs, rt	R[rd] ← R[rs] / R[rt]
Branch Instructions	
beq rs, v, addr	if (R[rs] == v) jump to addr
beqi rs, v, rd	if (R[rs] == v) jump to R[rd]
balr rs, addr	R[rs] ← PC; jump to addr
balri rs, rd	R[rs] ← PC; jump to R[rd]
Logical Instructions	
shl rd, rs, rt	R[rd] ← R[rs] << R[rt]
shr rd, rs, rt	R[rd] ← R[rs] >> R[rt]
and rd, rs, rt	R[rd] ← R[rs] & R[rt]
or rd, rs, rt	R[rd] ← R[rs] | R[rt]
xor rd, rs, rt	R[rd] ← R[rs] ^ R[rt]
not rd, rs	R[rd] ← ~R[rs]
shli rd, rs, imm	R[rd] ← R[rs] << imm
shri rd, rs, imm	R[rd] ← R[rs] >> imm
andi rd, rs, imm	R[rd] ← R[rs] & imm
ori rd, rs, imm	R[rd] ← R[rs] | imm
xori rd, rs, imm	R[rd] ← R[rs] ^ imm
noti rd, imm	R[rd] ← ~imm
Load/Store Instructions	
ldo rt, rs, a	R[rt] ← Mem[R[rs] + a]
ld rd, rt, rs	R[rd] ← Mem[R[rs] + R[rt]]
sto rt, rs, a	Mem[R[rs] + a] ← R[rt]
st rd, rt, rs	Mem[R[rd] + R[rt]] ← R[rd]
Special Instructions	
read rs	R[rs] ← Input stream
print rt	R[rt] → Output stream
throw Ex	Cause exception Ex to be thrown
halt	halt the machine
prints Str	Str → Output stream
nop	Do nothing
check Expr	if (! Expr) throw CheckException
Comparison Instructions	
setlt rd, rs, rt	R[rd] ← (R[rs] < R[rt])
setgt rd, rs, rt	R[rd] ← (R[rs] > R[rt])
seteq rd, rs, rt	R[rd] ← (R[rs] == R[rt])

[4] A value can be an integer or the *err* symbol (denotes an error).

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

1. **Arithmetic Instruction:** Consider the execution of the *addi* instruction, which adds the value *v* to the register given by *rs* and stores the results in register *rd*.

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

The elements of the machine state in the *above equations are composable, and hence can be matched with a generic symbol S representing the "rest of the state". T*his allows new machine state elements to be added without modifying existing equations that do not manipulate the new state.

2. **Branch Instructions:** Consider the example of the *beq rs, v, l* instruction, which branches to the code label *l* if and only if the register *rs* contains the constant value *v*. The equation for *beq* is similar to the equation for the *addi* operation except that it uses the in-built if-then-else operator of Maude.

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

Note the use of the *isEqual* primitive rather than a direct == to compare the values of the register *rs* and the constant value *v*. This is because the register *rs* may contain the symbolic constant *err* and hence needs to be resolved accordingly (by the error model).

3. **Load/Store Instructions:** Consider the example of the instruction *ldo rt, rs, a* which loads the value in the memory location at the address given by adding the offset *a* to the value in the register *rs.* The equation for this instruction is as follows.

$$eq\ \{\ C\ ,\ <\ ldo\ rt\ rs\ a\ ,\ PC(pc)\ regs(R)\ mem(M)\ S\ >\ \}\ =\ \{\ C,\ <\ fetch(C,\ pc),\ C(next(pc))\ mem(M)\ regs(R[rt]\ <-\ M[a\ +\ R[rs]])\ >\ \}\ .$$

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

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

5. **Special Instructions:** These instructions are responsible for starting and stopping the program, e.g., *halt* and *throw* instructions to terminate the program.

The halt instruction sets the program counter of the program to *done*, to indicate to the *fetch* sub-model to stop fetching instructions. Its equation is given by:

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

The *throw* instruction is similar except that it sets the program counter to *crash* and prints the exception.

$$eq\ \{\ C,\ <\ throw\ e,\ PC(pc)\ out(O)\ S\ >\}\ \ =\ (\ PC(crash)\ ex(e)\ out(\ pc\ >>\ (\text{" : "}\ >>\ (toString(e2)\ >>\ O))\ )\ S\ )\ .$$

Both equations transform the super state prior to their execution into a machine state. This is because the model-checker (Section 5.4) searches for machine states only and the above transformations ensure that the terminating states of the program are returned by the search.

5.2 Error Model

The overall approach to error injection and propagation was discussed in Section 3.2. In this section, we discuss the implementation of the approach in Maude. The error model is divided into five sub-models as follows:

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

$$rl\ allErrors(\ [L\ |\ I]\ C,\ input,\ type\ )\ =>\ injectStart(\ [L\ |\ I]\ C,\ In,\ type,\ L\ )\ .$$
$$eq\ injectStart(C,\ input,\ type,\ pc)\ =\ injectError(\ \{\ C,\ <\ fetch(C,\ st),\ initState(input)\ bkpts(pc)\ >\ \},\ type\ )\ .$$
$$crl\ injectError(\ \{\ C,\ <\ I,\ PC(pc)\ bkpts(BL)\ S>,\ CtrlError)\ =>\ \{C,\ <I,\ PC(err)\ S>\}\ if\ pc\ in\ BL\ .$$
$$rl\ injectError(\ \{\ C,\ <\ I,\ Regs(\ R\ (r\ =\ v)\ )\ PC(pc)\ bkpts(BL)\ S>,\ RegError)\ =>\ \{C,\ <I,\ Regs(\ R\ (r\ =\ err)\ S>\}\ if\ pc\ in\ BL\ .$$
$$rl\ injectError(\ \{\ C,\ <\ I,Mem(M\ (a\ =v)\ )\ PC(pc)\ bkpts(BL)\ S>,\ MemError)\ =>\ \{C,\ <I,\ Mem(M\ (a\ =\ err))\ PC(pc)\ bkpts(BL)\ S>\}\}\ if\ pc\ in\ BL\ .$$

In the above equations, the *allErrors* function injects all possible faults of a certain type (control, register, or memory) into the program by systematically enumerating each location in the program and calling the *injectStart* function on the location. The *injectStart* function starts the program after setting a breakpoint at the location. When the program's execution reaches the breakpoint, the *injectError* function is invoked, which injects the fault by setting the corresponding location to *err*. In the case of control errors, the *err* location is the program counter. In the case of register errors, the *err* location can be any register in the register file. In the case of memory errors, the *err* location can be any initialized memory location.

Error Propagation sub-model: Once an error has been injected, it is allowed to propagate through the equations for executing the program in the machine model. The semantics of error propagation are also described by equations as shown below (*I* represents an integer below).

$$eq\ err\ +\ err\ =\ err\ .\quad eq\ err\ +\ I\ =\ err\ .\quad eq\ I\ +\ err\ =\ err\ .$$
$$eq\ err\ -\ err\ =\ err\ .\quad eq\ err\ -\ I\ =\ err\ .\quad eq\ I\ -\ err\ =\ err\ .$$
$$eq\ err\ *\ I\ =\ if\ (I==0)\ then\ 0\ else\ err\ fi\ .$$
$$eq\ I\ *\ err\ =\ if\ (I==0)\ then\ 0\ else\ err\ fi\ .$$
$$eq\ err\ /\ I\ =\ if\ (I==0)\ then\ throw\ divException\ \ else\ err\ fi\ .$$
$$eq\ I\ /\ err\ =\ if\ isEqual(err,\ 0)\ then\ throw\ divException\ else\ err\ fi\ .$$
$$eq\ err\ *\ err\ =\ if\ isEqual(err,\ 0)\ then\ 0\ else\ err\ fi\ .$$
$$eq\ err\ /\ err\ =\ if\ isEqual(err,\ 0)\ then\ throw\ divException\ else\ err\ fi\ \ .$$

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

Comparison Handling Sub-model: The rules for comparisons involving one or more *err* values are expressed as rewrite rules as they are non-deterministic in nature. For example, the rewrite rules for the *isEqual* operator used in section 5.1 are as follows:

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

The comparison operators involving err operands evaluate to either true or false non-deterministically. This

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

Constraint Tracking and Solving Sub-model: A new structure called the *ConstraintMap* is added to the machine state in Section 5.1. The *ConstraintMap* structure maps each register or memory location containing *err* to a set of constraints that are satisfied by the value in the location. An example of a set of constraints for a location is the following: *notGreaterThan(5) notEqualTo(2) greaterThan(0).* This indicates that the location can take any integer value between 0 and 5 excluding 0 and 2 but including 5. The constraints for a location are updated whenever a comparison is made based on the location if and only if it contains the value *err*. Constraints are also updated by arithmetic operations involving addition/subtraction. Multiplication/division operations are not considered. For a given location, it may not be possible to find an integer value that satisfies all its constraints (un-satisfiable constraints). The model-checker terminates the search when it comes to a state with un-satisfiable constraints.

The constraint solver determines whether a set of constraints is satisfiable. For example, the constraints *notGreaterThan(5) GreaterThan(10)* are unsatisfiable. The constraint solver also simplifies the constraints for a location. For example, the constraints *notGreaterThan(5) notGreaterThan(3)* can be simplified to the constraint *notGreaterThan(3).* The equations for the constraint tracking and solving sub-model are omitted due to space constraints.

Memory- and Control Handling Sub-model: Memory and control errors are non-deterministic and hence their semantics are expressed using rewrite rules as follows:

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

$$rl\ \ fetch(\ [L\ |\ I]\ C,\ E\ ) => beq\ \$(0)\ \#(0)\ L\ .$$
$$rl\ fetch(\ C,\ E\ ) => instException\ .$$

Errors in pointer values of loads: The program either retrieves the contents of an arbitrary location in memory or throws a memory or alignment exception as follows:

$$rl\ (\ (a = v1)\ M)\ [\ err\ ]\ => v1\ .$$
$$rl\ (\ M)\ [\ err\ ]\ => memException(0)\ .$$
$$rl\ (\ M)\ [\ err\ ]\ => alignException(0)\ .$$

Errors in pointer values of stores: The program either overwrites the contents of an arbitrary memory location, or throws an alignment exception. We do not consider writes to locations outside the set of locations defined in the program as such locations are not read by the program.

$$rl\ (a = v1)\ M)\ [\ err <- v2\ ]\ => (\ (a = v2)\ M)\ .$$
$$rl\ \ (\ (a = v1)\ M)\ [\ err <- v2\ ]\ =>\ \ M\ [\ a <- alignException(0)\ ]\ .$$

5.3 Detector Model

Error detectors are defined as executable checks in the program that test whether a given memory location or register satisfies an arithmetic or logical expression. For example, a detector can check if the value of register *$(5)* equals the sum of the values in the register *$(3)* and memory location *(1000)* at a given program counter location. If the values do not match, an exception is thrown and the program is halted.

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

We assume that the execution of a detector does not fail. This assumption is further considered in Section 5.5.

A detector is written in the following format:

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

The arguments of the detector are as follows:

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

$$Expr :: = Expr + Expr\ |\ Expr - Expr\ |\ Expr * Expr\ |\ Expr\ /\ Expr\ |@\ (c)$$
$$|\ !\ (Reg\ Name)\ |\ *(memory\ address)$$

Using the above notation, the detector introduced earlier would be written as: *det(4, $(5), == , ! ($3) + *(1000)).*

Detectors are implemented using equations, as their behavior is deterministic in the absence of errors. The equations for the detector's execution are independent of the equations in the machine model, and hence are not affected by errors introduced in the machine other than those that are present in the registers or memory locations used in the detector's expression. Execution of a detector also updates the constraints for the checked location in the *ConstraintMap* structure described in section 5.2.

The following equations evaluate an expression *e* used within a detector on the machine state S. In the equations below, *e1* and *e2* are expressions, a is an address, and *r* is a register name. *i* is an integer constant.

$$eq\ eval(\ e1 + e2,\ S\ ) =\ \ eval(e1,\ S) + eval(e2,\ S)\ .$$
$$eq\ eval(\ e1\ *\ e2,\ S\ ) =\ \ eval(e1,\ S)\ *\ eval(e2,\ S)\ .$$
$$eq\ eval(\ e1\ /\ e2,\ S\ ) =\ \ eval(e1,\ S)\ quo\ eval(e2,\ S)\ .$$
$$eq\ eval(\ e1\ -\ e2,\ S\ ) =\ \ eval(e1,\ S)\ -\ eval(e2,\ S)\ .$$
$$eq\ eval(\ !(r),\ regs(R)\ S\ ) = R[\ r\ ]\ \ .$$
$$eq\ eval(\ *(a),\ mem(M)\ S\ ) = M[\ a\ ]\ .$$
$$eq\ eval(\ @(i),\ S\ ) = i\ .$$

We define the *applyCheck* operation to evaluate a detector on a machine state. If the detector returns true (i.e., passes), the machine state is returned. Otherwise, a *checkException* is thrown. We consider the *applyCheck*

equation for a detector with the == operator below. The equations for other operations are similar.

eq < I, applyCheck(det(i, rd, ==, e) , regs(R) S) > =
if (isEqual(R[rd], eval(e, regs(R) S))) then < I, regs(R) S > else < (throw (checkException i)), regs(R) S > fi .

Note that we can represent any detector in the framework, as long as the detector can be written as an algebraic or logical expression. The detectors considered in many prior studies fall into this category [1], [2]. However, we cannot represent detectors that use timing information. This is an avenue for future investigation.

5.4 Model Checking

The exhaustive *search* feature of Maude is used to model-check programs [17]. The aim of the search command is to expose interesting "outcomes" of the program caused by a particular class of faults. The "outcome" is a user-defined function on the machine state described in Section 5.1 and must be specified as part of the *search* command. Note that while it is relatively straightforward to specify outcomes in terms of the program's final outcome or states, doing so at intermediate points of its execution requires knowledge of the assembly language code. For the programs we have considered however (i.e., *tcas*, replace), this has not been an issue.

As an example, the following search command obtains the set of executions of the program that will print a value of *1* without crashing, under all single errors in registers (one per execution). As mentioned in Section 5.1, only terminating program states are found by the search.

search allErrors(program, input, regErrors) =>! (S:MachineState) such that not (output(S) contains 1) and (getException(S)==0) .

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

Termination: In the absence of errors, most programs can be modeled as finite-space systems provided (1) they terminate after a finite amount of time or (2) they perform repetitive actions without terminating but revisit states. However, errors can cause the state space to become infinitely large, as the program may loop infinitely due to the error. This is not possible in practice, as the program data is physically represented as bits and there are only a finite number of bits available in a machine. However, the state space may become so large that it is impossible to explore fully in a reasonable amount of time.

In order to ensure that the model checking terminates in a reasonable time, the number of instructions that is allowed to be executed by the program must be bounded. This bound is referred to as the *timeout*. After the specified number of instructions is exceeded, a "timed out" exception is thrown and the program is halted. This functionality may be provided by a watchdog timer.

The timeout must be conservatively chosen to encompass the number of instructions executed by the program during all correct executions in the absence of errors. The question of how to choose an appropriate timeout is outside the scope of this paper. However, we find that the time taken by the search is not sensitive to the actual value of the timeout (provided it is finite). This is because the execution time of the search is dominated by rewrite rules. A conservative timeout potentially increases the number of equations executed by Maude, but does not affect the execution of rewrite rules.

5.5 Discussion

In this section, we consider some of the trade-offs made in the implementation of SymPLFIED and its impact on the scalability of the tool and the accuracy of the results.

Machine Model: The machine model makes a number of assumptions regarding memory accesses and control-flow instructions (see Section 5.1 for a detailed list). These assumptions limit the number of states that must be considered by SymPLFIED under an error. For example, by assuming that any access to uninitialized memory location results in an exception, we need to consider only initialized memory locations when performing a memory access with an erroneous address, thus ensuring that the analysis is independent of the environment. However, we may miss error outcomes in which the program continues executing after reading from uninitialized locations. Nonetheless, this is a reasonable assumption as the values in uninitialized locations are non-deterministic and the program may behave unpredictably after reading them.

Fault Model: We assume that errors can occur only in the architectural state of the processor. This assumption is reasonable as only faults that propagate to the architectural state can impact the application. However, the error may occur during the execution of the instruction (i.e., in the processor's pipeline), and may or may not manifest in its output (for example, errors in the memory stage of non-memory instructions may not have an impact on the instruction). Modeling such effects would require a detailed model of the processor's internals, which would blow up the state space explored by SymPLFIED. Therefore, we assume that any error during the execution of an instruction affects its target register (or memory address).

Detection Model: We assume that errors cannot occur during the execution of detectors. This is because we assume that at most one error occurs in the program, and an error in the detector means that the program is error-free (note that this does not preclude an error originating in the program and propagating to the detector). Therefore, an error in the detectors can at worst lead to the program being stopped and not executed to completion. Such errors can never lead to incorrect outputs or safety violations, which are the focus of SymPLFIED. Further, it is possible to realize this assumption in practice by implementing the detector on a separate piece of hardware called the Reliability and Security Engine (RSE) [24].

Other Limitations: We currently do not support interrupts and Direct Memory Access (DMA) operations in SymPLFIED. Further, memory mapped input-output or file-seeking operations are not supported. These constructs are typically used only by systems code and are platform dependent. We do not consider systems code in order to ensure that SymPLFIED is platform neutral.

Finally, while SymPLFIED symbolically considers all possible errors in the program, it needs a concrete input to perform the analysis. This is because SymPLFIED

explicitly enumerates every state of the program under a set of faults, and hence needs a concrete starting point to begin the exploration, which is provided by the input.

6 RESULTS

This section reports our experience in using SymPLFIED on the *tcas* application [22], which is widely used as an advisory tool in air traffic control for ensuring minimum vertical separation between two aircrafts and hence avoid collisions. In the final part of this section, we report the results of applying SymPLFIED on the *replace* program of the Siemens suite to understand the effects of scaling to larger programs. Our goal is to demonstrate the capabilities of the SymPLFIED framework for exposing gaps in error detectors rather than to identify specific vulnerabilities in the applications studied.

We have implemented SymPLFIED using Maude version 2.4. Our implementation consists of about 2000 lines of uncommented Maude code split into 35 modules. The core of SymPLFIED consists of about 54 rewrite rules and 384 equations.

We also built a custom translator to convert MIPS assembly language as represented by the SimpleScalar simulator's Portable Instruction Set Architecture (PISA) [21]. To use our framework, the developer needs to compile their code with SimpleScalar's compiler (gcc), and then run our translator on the assembly file . We also wrote scripts to translate the counter examples found by SymPLFIED to the MIPS assembly language.

***Tcas*:** The application consists of about 140 lines of C code, which is compiled to 913 lines of MIPS assembly code. This in turn is translated to 800 lines of SymPLFIED's assembly code (by our custom translator). Table 5 shows the functions in *tcas*, and the number of lines of code in them (both C and SymPLFIED assembly).

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

Table 5: Functions in *tcas* and their sizes

Function	LOC (source)	Lines of SymPLFIED's assembly
ALIM	3	14
Alt_sep_test	27	77
Initialize	6	15
Non_Crossing_Biased_Climb	16	44
Non_Crossing_Biased_Descend	16	44
Own_Above_Threat	3	12
Own_Below_Threat	3	12
Inhibit_Biased_Climb	3	12
Main	16	20

6.1 Experimental Setup

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

We directed SymPLFIED to search for runs in which the program did not throw an exception[5] and produced a value other than 1 under the assumption of a single register error during its execution. We chose a timeout value of 10000 instructions, which is more than 10 times the number of instructions executed in a fault-free run.

Optimization: The total number of injections performed by SymPLFIED is (800 * 32), since there are 32 registers in the machine, and each instruction in the program is chosen as a breakpoint for the injection. In order to reduce the state space of the model checker, we inject errors only into the registers used in each instruction of the program. Further, we inject the error just before the instruction that uses the register, and check whether the instruction is executed to measure the fault's activation. The effect of the injection is equivalent to injecting the register at an arbitrary code location such that the error is activated at the instruction.

Parallelism: The injections with SymPLFIED were started on a cluster of 150 AMD Opteron processors running at 2 GHz with 2 GB of RAM. The search command is split into multiple smaller searches, each of which sweeps a particular section of the program code looking for errors that satisfy the search conditions. Each node in the cluster can perform the smaller searches independently, and the results are pooled together to find the overall set of errors. The maximum number of errors found by each search task was capped at 1,000 and a maximum of 30 minutes was allotted for the task to complete.

Validation: We augmented the SimpleScalar simulator [21] with the capability to inject errors into the source and destination registers of all instructions in the program. For each register we injected three extreme values in the integer range as well as three random values.

6.2 SymPLFIED Injections into *tcas*

For the injections on the *tcas* application, SymPLFIED found only two cases where an output of 1 (upward advisory) is converted to an output of 2 (downward advisory). This advisory can potentially be catastrophic as it is difficult to distinguish from the correct outcome of *tcas, and can result in a mid-air collision if followed*.

Later in this section, we discuss the catastrophic outcomes in more detail. We first discuss the overall results of the injections excluding the catastrophic cases. These consisted of cases where (1) *tcas* printed an output of 0 (unresolved) in place of 1, (2) the output was outside the range of the allowed values printed by *tcas*, and (3) numerous cases where the program exited or aborted prematurely. We do not consider these cases as catastrophic because *tcas* is only an advisory tool and the operator can ignore the advisory if he or she determines that the output produced by *tcas* is incorrect. We also found violations in which the value is computed correctly but printed incorrectly. We do not consider these cases as catastrophic because the real implementation of *tcas* may have a different output method. Section 6.4 presents a detailed analysis of the cases uncovered by tcas.

Running Time: Of the 150 search tasks started on the cluster, only 85 tasks completed within the allotted time of 30 minutes. We report results only from the tasks that completed. Of the 85 tasks that completed, 70 tasks did not find any errors that satisfy the conditions in the search command. These 70 tasks completed within 1 minute (overall). The remaining 15 tasks completed and

[5] As such exceptions will result in crashes, and not incorrect outputs.

found errors. T*he time taken by all the completed tasks (including the one that found the catastrophic outcome) is less than 4 minutes (240 seconds), and the average time for task completion is 64 seconds (ranges from 1 second to 240 seconds).* Since we performed this study, we have made a number of improvements to the SymPLFIED framework and are able to run the framework on a single machine (core i-7 processor at 2.5 GHz) with 8 Gigabytes of RAM. On this machine, we decomposed the above search task into eighteen parallel tasks, which were started in parallel. All but two of the tasks completed within two hours (these tasks were terminated by us). Since the processor has four cores, this time corresponds to a total running time of 8 hours. Thus, the total execution time on a single machine was less than the total time on the cluster. We use the more conservative cluster times for comparison with SimpleScalar injections in Section 6.3.

Note that the running time of SymPLFIED is dependent on the efficiency of the Maude model checker. Because Maude uses explicit state exploration, it can incur very high memory overheads. More efficient model checking approaches can alleviate the overhead and allow us to scale to larger programs. We do not consider such approaches however, as the goal of SymPLFIED is to demonstrate the feasibility of reasoning about hardware errors at the software level. It is worth noting that even though we do not use the fastest model-checking approach, we were able to uncover catastrophic failures in the *tcas* application in less time than an equivalent fault injection campaign (Section 6.3).

Catastrophic outcome: In order to understand the catastrophic error that lead to the incorrect value of 2, we show an excerpt from the *tcas* code in Figure 3**3**. The code corresponds to the function *alt_sep_test*, which tests the minimum vertical separation between two aircrafts and returns an advisory. This function in turn calls the function *Non_Crossing_Biased_Climb()* and the *Own_Above_Threat()* function to decide if an upward advisory is needed for the aircraft. It then checks if a downward advisory is needed by calling the function *Non_Crossing_Biased_Descend()* and the function *Own_Below_Threat()*. If neither advisory nor both advisories are needed, it returns the value *0* (unresolved). Otherwise, it returns the computed advisory.

We note that the *tcas* application (and system) has been extensively verified and checked for safety violations [22]. Nevertheless, the application has no detectors in its code. As mentioned in Section 3.1, SymPLFIED does not need error detectors in order to analyze the application.

The error under consideration occurs in the body of the called function *Non_Crossing_Biased_Climb()* and corrupts the value of register *$31* which holds the function return address (this is the calling convention in the MIPS processor that SymPLFIED emulates [21]). Therefore, instead of control being transferred to the instruction following the call to the function *Non_Crossing_Biased_Climb()* in *alt_sep_test()*, the control gets transferred to the statement *alt_sep = DOWNWARD_RA* in the function. This causes the function to return the value *2* instead of the value *1*, which is printed by the program. An analogous case exists for the function *Non_Crossing_Biased_Descend*, which is also called by the *alt_sep_test* function. We do not discuss this case in the interest of space. We have verified that the errors exposed above are not false-positives by performing targeted injections into the registers at the locations identified by SymPLFIED using the augmented SimpleScalar simulator (Section 6.3).

Note that the above error occurs in the stack, which is part of the runtime support added by the compiler. Hence, in order to discover this error, we need a technique like SymPLFIED that can reason at the assembly language (or lower) level. This shows the value of modeling low-level details in reasoning about transient-error propagation in programs.

6.3 SimpleScalar Injection Results

We performed over 6000 fault injection runs on the *tcas* application using the modified SimpleScalar simulator to see if we can find the catastrophic outcome outlined above. Both SymPLFIED and SimpleScalar were run for the same amount of time. Recall that the SymPLFIED injections were run with 150 tasks on the cluster, and each completed task took a maximum time of 4 minutes. This constitutes 10 hours in total. In this time, SimpleScalar was able to inject 6000 faults into the *tcas* program. The results of the injections are summarized in column 2 of Table 6 (numbers within parentheses represent the absolute number of injections). The results show that the SimpleScalar injections were unable to uncover even a single scenario with the catastrophic outcome of '2'.

In order to uncover the catastrophic error scenario using random fault injections, not only must the error be injected into register *$31* in the *Non_Crossing_Biased_Climb* function, the address of the assignment statement must be chosen to be injected in register $31 in Figure 3**3**. Otherwise, the catastrophic scenario will not be exposed by the random injections.

We also extended the SimpleScalar based fault injection campaign to inject 41000 register faults to check if any of the injected errors lead to the catastrophic outcome. The fault injection campaign took about 35 hours to complete. Note that this corresponds to over three times the time taken by SymPLFIED. However, the campaign was still unable to find an error leading to the catastrophic outcome. The results are shown in column 3 of Table 6.

Table 6: SimpleScalar fault injection results

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

6.4 SymPLFIED Results: Analysis

In this section, we analyze the results obtained by SymPLFIED with the goal of understanding the dominant failure modes of the *tcas* application. Table 7 shows the results of the analysis. The first column of the table is the output produced by SymPLFIED, the second and third columns are the functions and the registers into which the errors were injected, and the fourth column is the total number of injections that resulted in the output.

The outputs in Table 7 fall into the following categories. First, there are invalid values such as 740 and 122, which are produced by the program due to the injected errors (i.e., values other than 0, 1 and 2). The second category corresponds to valid but incorrect outputs such as 0 and 2. These outputs are difficult to distinguish from the correct output of *tcas*, which is 1 for this input.

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

Figure 33: Portion of *tcas* code corresponding to the catastrophic outcome

Table 7: Results obtained by SymPLFIED on tcas

Output	Function(s)	Register(s)	No. of injections
740	initialize, Non_crossing_biased_climb, Alt_sep_test	$31	20
Err	Alt_sep_test	$2	9
Exit	All functions	$31	96
[1] Abort	[2] All functions	[3] $31	96
122	Alt_sep_test	$31	2
2	Non_Crossing_Biased_Climb, Non_Crossing_Biased_Descend	$31	20
0	All functions	$31 $29 $2, $3 $4, $5	254
9936	Non_Crossing_Biased_Climb, Non_Crossing_Biased_Descend, Alt_sep_test	$31	30
empty	All functions		96

,Third, the application may exit early, abort due to an assertion violation or produce no output. These categories have been represented by "Exit", "Abort", and "Empty". These are different from crashes as the application does not raise an exception, but instead exits gracefully.

Finally, the application may print the value Err if an injection is performed into register $2 in the alt-sep-test function (register $2 holds the return value).

We can see from Table 7 that most of the wrong outputs are caused by errors in the return address register $31. Therefore, incorrect return addresses are the dominant cause of non-crash causing errors in the *tcas* application. Further, the catastrophic outcome discussed in 6.2 was also due to corruption of the return address register. Therefore, in Appendix A, we design error detectors to check the return address register to detect these errors. In Appendix B, we investigate the resilience of the *tcas* program to memory errors, with and without detectors.

6.5 Application to Larger Programs

In order to evaluate the scalability of SymPLFIED, we analyzed the *replace* program using SymPLFIED. *replace* is the largest of the Siemens benchmarks suite [20]. Our custom translator translates the program to 1550 lines of Maude code spanning 22 functions.

Using the same experimental setup as described in Section 6.1, we ran SymPLFIED on the *replace* program to find all single register errors (that lead to an incorrect outcome of the program). The overall search was decomposed into 312 parallel tasks. Of these, 202 completed execution within the allotted time of 30 minutes. In 148 of the completed search tasks, either the error was benign or the program crashed due to the error, while 54 of the search tasks found error(s) leading to incorrect outcome. These tasks took 10 minutes on average to find the error. More details about the results for injections into *replace* may be found in the technical report version of this paper [27].

7 Conclusion

This paper presented SymPLFIED a modular, flexible framework for performing symbolic fault injection and evaluating error detectors in programs. We have implemented the SymPLFIED framework for a MIPS-like processor using the Maude rewriting logic engine. We demonstrate the SymPLFIED framework on a widely deployed application *tcas,* and use it to find a transient error that can lead to catastrophic consequences.

Acknowledgements: This research was funded in part by NSF grant CNS-05-51665 and CNS-04-6351. We thank the Gigascale System Research Consortium (GSRC) and the Motorola Center for Communications at the University of Illinois at Urbana-Champaign for their support. We thank Carol A. Bosley for editing assistance.

References

[1] M. Hiller, A. Jhumka, and N. Suri. On the placement of software mechanisms for detection of data errors. In *Proc. Int'l Conf. on Dependable Systems and Networks (DSN)*, pages 135-144, 2002.

[2] Pattabiraman, K., Kalbarczyk, Z., and Iyer, R. K. Automated Derivation of Application-aware Error Detectors using Static Analysis. In *Proc. of the 13th Intl. on-Line Testing Symposium,* 2007.

[3] W. Gu, Z. Kalbarczyk, R.K. Iyer, Z. Yang. Characterization of Linux Kernel Behavior under Errors. *Proc. International Conference on Dependable Systems and Networks (DSN)*, pp. 459-468, 2003.

[4] Arlat, J., et al. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Trans. Softw. Eng,* , 1990.

[5] H. Madeira, J. Carreira, J.G. Silva. Injection of Faults in Complex Computers. *IEEE Workshop on Evaluation Techniques for Dependable Systems.* San Antonio. Texas. October 1995

[6] D. Cyrluk. Microprocessor verification in PVS: A methodology and simple example. *Tech Report SRI-CSL-93-12,* 1993.

[7] R. S. Boyer and J S. Moore. "Program Verification". *Journal of Automated Reasoning* 1, 1 (1985), 17-23.

[8] Krautz et al., Evaluating coverage of error detection logic for soft errors using formal methods. In *Proc. of the Conf. on Design, Automation and Test in Europe (DATE),* 2006.

[9] Seshia, S. A., Li, W., and Mitra, S. Verification-guided soft error resilience. In *Proc. of the Conference on Design, Automation and Test in Europe* (DATE), 2007.
[10] A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *Intl. Conference on Distributed Computing Systems,* pages 436--443, 1998.
[11] Nicolescu, B. Gorse, N. Savaria, Y. Aboulhamid, E.M. Velazco, R. On the use of model checking for the verification of a dynamic signature monitoring approach. IEEE Trans. on Nuclear Science, Vol. 52, 5(2), pp. 1555-1561, 2005.
[12] Perry F., et al., Fault-tolerant Typed Assembly Language. *Proc. of Conf. on Prog. Lang. Design and Implementation (PLDI)*, 2007.
[13] Matthew L. Meola and David Walker. Faulty Logic: Reasoning about Fault Tolerant Programs. Proceedings of the *European Symposium on Programming (ESOP)*, 2010.
[14] King, J. C. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (Jul. 1976), pp. 385-394.
[15] W. Bush et al. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7), 2000.
[16] D. Larson and R. Hahnle. Symbolic Fault Injection, *International Verification Workshop (VERIFY), vol. 259, pp. 85-103, 2007.*
[17] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In *First Intl. Workshop on Rewriting Logic and its Applications,* 1996.
[18] E. Clarke, A. Biere, R. Raimi, Y. Zhu. Bounded Model checking using satisfiability solving. *In Formal Methods in System Design*, 2001.
[19] M. Clavel et al. The Maude Formal Tool Environment. *Springer Verlag LNCS*, Vol 4624, pp. 173-178, Aug 2007.
[20] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control flow-based test adequacy criteria. In Proc. *Intl. Conf. on Software Engineering (ICSE)*, pp 191–200, 1994.
[21] Burger, D. and Austin, T. M. 1997. The SimpleScalar tool set, version 2.0. *Comput. Archit. News* 25, 3, 1997.
[22] J. Lygeros and N.A. Lynch. On the formal verification of the TCAS conflict resolution algorithms. In *Proc. 36th IEEE Conf. on Decision and Control,* pp. 1829--1834, 1997.
[23] Federal Aviation Administration, TCAS II Collision Avoidance System (CAS) System *Requirements Specification,*1993.
[24] N. Nakka, Z. Kalbarczyk, R.K. Iyer, J. Xu. An architectural framework for providing reliability and security support, International Conference on Dependable Systems and Networks (DSN), pages 585-594, 2004.
[25] Kuperman, B. A., Brodley, C. E., Ozdoganoglu, H., Vijaykumar, T. N., and Jalote, A. Detection and prevention of stack buffer overflow attacks. *Commun. ACM* 48, 11 (Nov. 2005), 50-56.
[26] Hennessey and Patterson. Computer Organization and Design: The hardware-software interface, Morgan Kauffman, 2011.
[27] K. Pattabiraman, N. Nakka, Z. Kalbarczyk and R. K. Iyer, "SymPLFIED: Symbolic Program Level Fault injection and Error Detection", Technical Report (UILU-ENG-08-2205), University of Illinois (UIUC), January 2008.

Authors' Biographies

Karthik Pattabiraman received the M.S. and Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign (UIUC) in 2004 and 2009. He is currently an assistant professor at the University of British Columbia in electrical and computer engineering. His research interests include design of reliable and secure applications using static and dynamic analysis. Based on his dissertation work, Pattabiraman was awarded the *William C. Carter* award in 2008 by the IFIP Working Group on Dependability and the IEEE Technical Committee on Fault-tolerant Computing (TC-FTC). He is a member of the IEEE and the IEEE Computer Society.

Nithin Nakka received his B.Tech degree from Indian Institute of Technology, Kharagpur and his M.S. and Ph.D. degrees from the University of Illinois at Urbana-Champaign (UIUC), under the guidance of Professor Ravishankar Iyer. His areas of research interest include reliability and hardware implemented fault-tolerance. Nakka held positions as a research faculty in UIUC, with Professor Iyer, and at Northwestern University with Professor Alok Choudhary. He also worked for Motorola's mobile devices group. He is working for Nextest Systems.

Zbigniew T. Kalbarczyk is currently Research Professor at the Center for Reliable and High-Performance Computing in the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign. Kalbarczyk's research interests are in the area of design and validation of reliable and secure computing systems. Currently, he is a lead researcher on the project to explore and develop high availability and security infrastructure (including use of dedicated software and reprogrammable hardware) capable of managing redundant resources to foil security threats, detect errors in both the user applications and the infrastructure components, and recover quickly from failures. His research involves designing of techniques for automated validation and benchmarking of dependable computing systems using formal (e.g., model checking) and experimental (e.g., fault/attack injection) methods. Kalbarczyk served as a Program Chair of Dependable Computing and Communication Symposium (DCCS), a track of the International Conference on Dependable Systems and Networks (DSN) 2007, and Program Co-Chair of Performance and Dependability Symposium, a track of the DSN 2002. Kalbarczyk has published over 90 technical papers and is regularly invited to give tutorials and lectures on issues related to design and assessment of complex computing systems. He holds a PhD degree in computer science from the Technical University of Sofia, Bulgaria. He is a member of the IEEE, the IEEE Computer Society, and IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance.

Ravishankar K. Iyer is a George and Ann Fisher Distinguished Professor of Engineering at the University of Illinois at Urbana-Champaign. From 2008-2011, Iyer served as the Vice Chancellor for Research (Interim) on the campus. He holds appointments in the Department of Electrical and Computer Engineering and the Department of Computer Science. He is Director of the Center for Reliable and High-Performance Computing and the Chief Scientist at the Information Trust Institute. Iyer's research interests are in the area of dependable and secure systems. He has been responsible for major advances in the design and validation of dependable computing systems. He currently leads the TRUSTED ILLIAC project at Illinois, which is developing application-aware adaptive architectures for supporting a wide range of dependability and security requirements in heterogeneous environments. Professor Iyer is a Fellow in the AAAS, the IEEE and the ACM. He has received several awards including the Humboldt Foundation Senior Distinguished Scientist Award for excellence in research and teaching, the AIAA Information Systems Award and Medal for "fundamental and pioneering contributions towards the design, evaluation, and validation of dependable aerospace computing systems," and the IEEE Emanuel R. Piore Award "for fundamental contributions to measurement, evaluation, and design of reliable computing systems."