FORMAL FRAMEWORK AND TOOLS TO DERIVE EFFICIENT APPLICATION-LEVEL DETECTORS AGAINST MEMORY CORRUPTION ATTACKS

BY

FLORE QIN-YU YUAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Adviser:

Professor Ravishankar K. Iyer

Abstract

Memory corruptions are a major part of security attacks observed nowadays. Many protection mechanisms have been proposed to fight against them. These techniques can be broadly classified into two categories: those that focus on preventing vulnerabilities from being exploited (canary value, libsafe) and those that focus on preventing important data (e.g. return address, critical variable) from being overwritten by attackers (IFS, taintedness tracking, WIT, random memory layout). As the range of vulnerabilities increases, we believe that protecting all vulnerabilities with specific techniques begins to be unrealistic. That is why we want to focus on the second category. This thesis proposes to use an existing formal tool, SymPLAID, to find the minimum set of critical memory locations one needs to protect. The analysis results are also used to derive selective detectors which are guaranteed to detect a given attack model. We demonstrate the methodology by deriving application specific detectors which are guaranteed to detect all attacks where the attacker's goal is to corrupt the application's end result by modifying one memory location. Very few, well placed detectors are needed to get a 100% coverage for the given attack model.

Acknowledgments

This project would not have been possible without the help and support of the DEPEND group and my adviser Professor Ravishankar Iyer. I would also like to thank Professor Zbigniew Kalbarczyk for his constant feedback, advice and support. Special thanks to Karthik Pattabiraman, who was a great source of inspiration and mentored me throughout my master's work.

This work was supported in part by NSF grants CSN-05-51665 and the Department of Energy under Award Number DE-OE0000097, and Boeing Corporation as part of ITI Boeing Trusted Software Center.

Table of Contents

Chapter 1: Introduction 1
Chapter 2: Approach Overview 5
2.1 Symbolic fault injection 5
2.2 Tool chain flow 6
Chapter 3: Definitions and Attack Model 8
3.1 Attack model 8
3.2 Terminology 9
3.2.1 Definitions 9
3.2.2 Properties of the critical code sections 9
Chapter 4: SymPLAID 14
4.1 Input 14
4.2 Implementation 16
4.3 Error injection 17
4.3.1 Error injection mechanism 18
4.3.2 Injection rules and equations 19
4.4 Error propagation 22
4.5 Constraint solver 23
Chapter 5: SymPLAID Validation 26
5.1 Analysis of SymPLAID's result 27
5.2 False positive rate validation 29
5.2.1 Injected values 30
5.2.2 Injection result analysis 30
Chapter 6: Deriving Detectors 33
6.1 Identifying critical variables and critical sections 33
6.1.1 WuFTP 34
6.1.2 OpenSSH 37
6.2 Deriving detectors 38
6.2.1 Detector overview and implementation 38
6.2.2 Detector evaluation 42
6.3 On scaling the analysis 44
6.3.1 Technique 1: A practical approach. 44
6.3.2 Technique 2: Reducing the number of injections needed 45

Chapter 7: Discussion of Other Attack Models 50
7.1 Control flow attacks 50
7.1.1 Diverting the control flow to another legal path 50
7.1.2 Diverting the control flow to an illegal path 51
Chapter 8: Related Works 54
8.1 Preventing vulnerabilities from being exploited 54
8.2 Enforcing an application or system's correct behavior 54
Chapter 9: Conclusion and Future Work 56
9.1 Conclusion 56
9.2 Future work 56
9.2.1 Scalability of SymPLAID 56
9.2.2 Studying new attack models within the framework 57
9.2.3 Adapting SymPLAID for x86 58
9.2.4 Building detectors 58
References 59

Chapter 1: Introduction

Memory corruption attacks represent a major part of security attacks reported in recent years [1]. Applications written in unsafe languages like C or C++ are still numerous and are vulnerable to attacks exploiting memory errors like buffer overflow and underflow [2], [3] dangling pointers [4] or double frees [5].

As for defensive techniques, the most complete solutions often incur either high runtime or memory space overhead (e.g. duplication of critical data). Furthermore, coverage and false positive rates are often determined by testing the protection mechanism against known attacks [6], [7] and/or by theoretical ad-hoc deductions [7], [8].

Testing a security technique's coverage is a very hard task. Different methodologies can be used but each of them suffers from limitations. Based on our readings and experience, validation techniques can be classified into two broad categories: experimental validation and formal validation. (*Table 1* summarizes the discussion presented below.)

- **Experimental validation techniques** rely on testing the protection mechanisms against either crafted attacks or real, already known attacks.
    - As there is no database of real attacks available for research purposes, it is rather difficult to gather a large set of attacks to test the protection mechanism against. Bug and vulnerability reports usually do not contain enough information to allow an attack to be reproduced. Therefore, reproducing a large set of such attacks can be very time consuming.
    - Crafting attacks can also be very time-consuming as one has to understand what vulnerabilities need to be introduced in the system or applications and how to exploit them.

Table 1: Classification of validation techniques and their limitations

Categories		Limitations
Formal Validation	Writing proof by hand / Ad-hoc reasoning	- Difficult to make exhaustive - Ad-hoc reasoning does not supply rigorous proofs
	Using model checkers and theorem provers	- Need expertise in formal methods - Need to model the protection mechanism and the environment - Scalability problem due to software state explosion
Experimental Validation	Using real attacks	- No attack database available for research purpose - How many attacks are needed to get an accurate idea of coverage? - Which attacks should be chosen?
	Using crafted attacks	- Time-consuming - How many attacks are needed to get an accurate assessment of coverage? - Which attacks should be chosen?

  - Furthermore, such validation can only guarantee the effectiveness of the protection mechanisms for the tested attacks. It is hard to determine which attacks need to be tested against and how many of them are needed. The guarantee obtained by experimental validation only applies to the tested attacks unless proved otherwise. For example, showing that a detection mechanism can detect a certain format string attack does not prove that the detection mechanism covers all format string attacks.

- **Formal validation techniques** rely on reasoning and proofs to validate a protection mechanism.
  - Manual proofs and ad-hoc reasoning are usually not sufficient to guarantee the coverage of a protection mechanism. Indeed, it is difficult to enumerate all possible cases and situations that can happen and reason about them. It is particularly difficult for the human mind to think about corner cases.

Unfortunately, in the case of security, zero-days attacks that exploit hard to find vulnerabilities are frequent.

- Model checkers and theorem-provers are good alternatives to ensure a complete analysis of the coverage of a protection mechanism. However, this usually requires expertise in formal methods and effort to model the protection mechanism, the execution environment, and the attacks. Furthermore, as models become more complicated and detailed, formal methods usually do not scale well due to state explosion issues.

As an alternative, this thesis explores the following idea: instead of designing detectors and then verifying their coverage, why not use a formal environment to help derive detectors which, by design, achieve the coverage expected?

The formal environment used in this thesis is an existing tool: SymPLAID [9], developed in our research group. SymPLAID has a built-in machine model and a symbolic fault-injector. We leverage SymPLAID to analyze applications' behavior under fault in order to find memory locations that can be corrupted to achieve an attacker's goal (e.g. get into the system using invalid password). The analysis result is then used to derive protection mechanisms against memory corruption attacks. We implemented and placed the derived detectors within the formal framework itself, and we show that only a few, well placed detectors are needed to achieve high coverage for the attack models we studied. The approach followed is described in Chapter 2 and the attack model studied is fully defined in Chapter 3.

The main contributions of this thesis are the following:

i. Leverage an existing formal tool to determine critical data and code sections to protect. The idea is to use exhaustive symbolic fault injections to identify the set of memory addresses which an attacker can corrupt to achieve his/her goal.

ii. Design detectors to protect identified critical data and code sections.

iii. Model and verify the coverage of the proposed detectors within the formal framework.

iiii. Optimize the analysis time needed to find critical variables/critical code sections based on properties of the critical code sections.

Chapter 2: Approach Overview

This section presents an overview of the approach adopted to *(i)* identify what to protect (e.g. what data and/or computation) and where to place the protection mechanisms, and *(ii)* validate constructed detectors.

2.1 Symbolic fault injection

We leverage an existing formal tool, SymPLAID, to inject symbolic faults and analyze the behavior of an application under faults. SymPLAID runs on top of the Maude model checker [10] and models applications' execution at the assembly level (MIPS assembly in our case) using rewriting logic [11]. SymPLAID-supported symbolic fault injection introduces a single error ("err" symbol) per execution in one memory location and propagates the error's consequences using symbolic execution. The results from the symbolic fault injections are then used to determine memory locations that need to be protected and to design and place protection mechanisms.

Fault injection can be used to find memory locations which need to be protected in order to prevent malicious attacks. Independently of the vulnerabilities exploited [12] (buffer overflow, format string, third party libraries etc.), all memory corruption attacks involve the corruption of at least one memory location. Injecting symbolic faults allows us to consider all possible corruptions at the value in a given memory location and thus conduct an exhaustive analysis of the application's behavior under faults. It is important to note that one of the main assumptions made in this work is that all memory corruption attacks can be modeled by targeted fault injections.

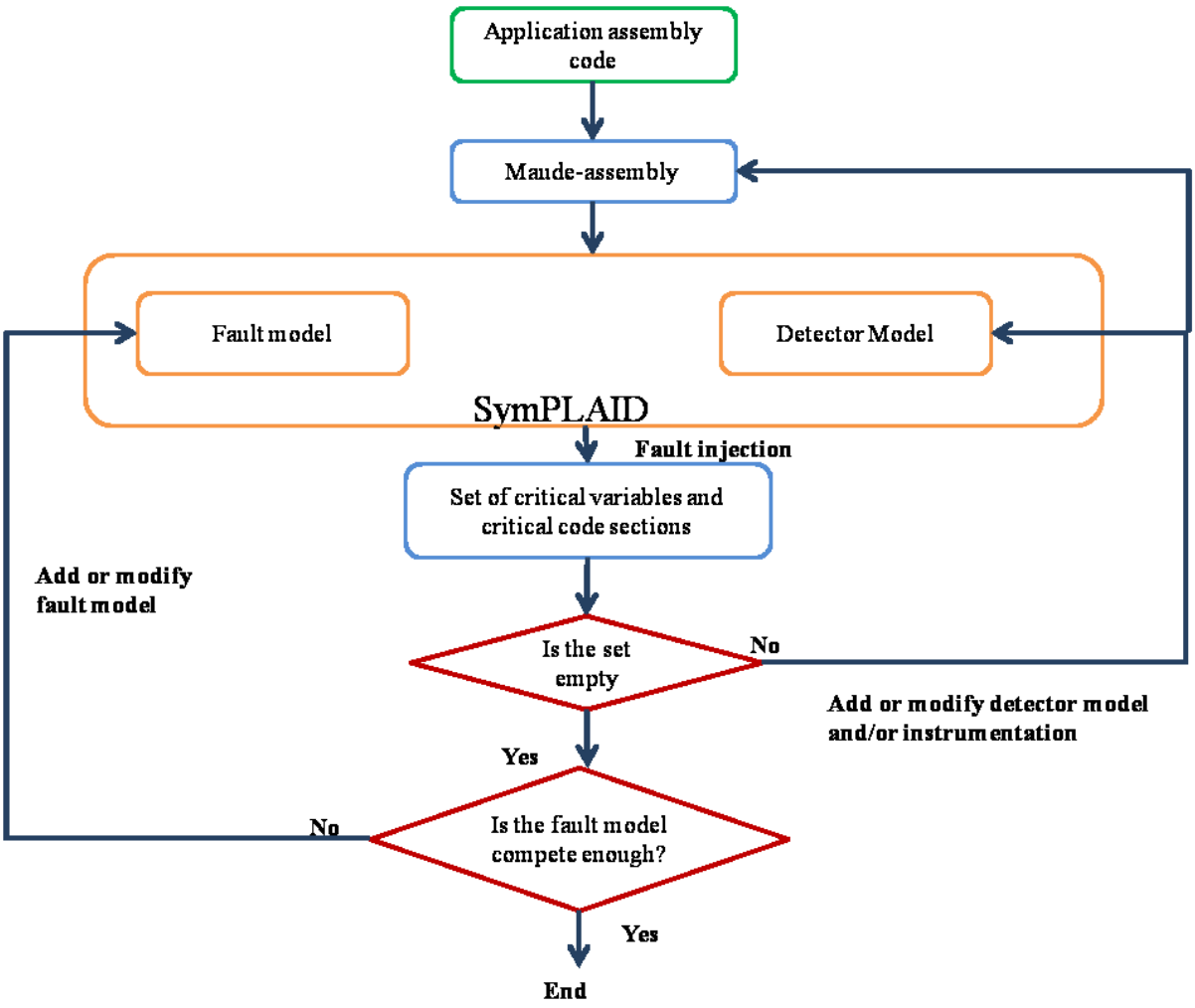
2.2 Tool chain flow

The process of going from SymPLAID's formal analysis to the identification of data and code to protect, and finally to the design and validation of detectors is partially automated. The complete tool chain flow used in this study is illustrated in *Figure 1* and described in *Table 2*.

Table 2: Tool chain flow description

	Automated /manual	**description**
STEP 1: Translation of the assembly code into equations	Automated	- The user supplied application's assembly is translated into an equational representation of the code defined by SymPLAID ("Maude assembly"). - Go to step 2.
STEP 2: Fault Injection	Automated	- The "Maude assembly" is fed to SymPLAID. - Fault injection is conducted using the fault model (initially only one memory location can be corrupted by the attacker) and the detector model (initially no detector) implemented in SymPLAID. - Go to step 3.
STEP 3: Parsing SymPLAID's result	Automated	SymPLAID's results are parsed in order to determine which memory location can be successfully corrupted by an attacker to achieve his/her goal (e.g. get authenticated with invalid login information). - If no memory location can be successfully corrupted by an attacker to achieve his/her goal then the application is resilient to the studied attack model. Go to step 5. - If there are memory locations that can be successfully corrupted by an attacker to achieve his/her goal then go to step 4.
STEP 4: Refining the detector model	Manual	- Use SymPLAID's results as an indication of what to protect and where to insert protection. - Implement or refine the detector model. - Instrument the application if necessary. - Go to Step 2.
STEP 5: Refining the attack model	Manual	- If the user determines that it is necessary to consider a more advanced fault model (e.g. several memory locations can be corrupted by the attacker) then add or refine the fault model and go to step 2.

Figure 1: Tool chain flow



As shown in *Table 2*, the symbolic fault injections as well as the analysis of the injections' results are automated. The only manual effort required from the user is the implementation, within the framework, of the detector models and of more advanced attack models if needed. We show in following chapters that adding or refining detector models within the framework requires a little effort and expertise in formal methods.

Furthermore, once implemented, detector models and attack models can be reused across analysis, which again reduces the manual effort required from the user.

Chapter 3: Definitions and Attack Model

This chapter defines the threat model considered in the work and introduces basic definitions and properties used throughout the thesis.

3.1 Attack model

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

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

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

Integrity of an application: In this work, it is considered that the attacker's goal is to alter the integrity of data produced by the application, i.e. corrupt data produced by the application. Therefore, we do not consider attacks targeting the confidentiality of data, where the goal is to steal information from the application.

Furthermore, it is assumed in this work that the attacker does not want to crash the application. Any attacks leading to a system exception being raised or a system crash are considered to be "detectable" attacks and thus are not considered.

3.2 Terminology

The following terms are used throughout this thesis.

3.2.1 Definitions

***Critical variable (or critical memory location)*:**

A critical variable or critical memory location is a memory location that can be corrupted by the attacker to achieve a particular goal (e.g. alter the application's output or login with an invalid password).

***Critical code sections associated with critical variable M*:**

A critical memory location is always associated with one or several critical code sections. A critical code section associated with the critical variable *M* is the **longest sequence of consecutive** (in terms of execution order) **assembly instructions** during the execution of which *M* can be corrupted by the attacker to achieve his/her goal. In other words, each critical code section encompasses all consecutive instructions during the execution of which *M* can be corrupted.

Note that for each possible execution flow, a set of critical code sections is derived. Therefore, critical sections can overlap for different execution flows.

3.2.2 Properties of the critical code sections

In this section, we introduce and prove two lemmas which characterize unique properties of the critical code section.

The following notations are introduced to enable the reasoning.

Notations:

- **Critical Variable**: *m*
- **Critical code section associated with m:** C_m
- **Execution flow:** $E = S_0 \xrightarrow{I_0} S_1 \xrightarrow{I_1} \ldots \xrightarrow{I_i} S_i \xrightarrow{I_{c0}} S_{i+1} \xrightarrow{I_{c1}} \ldots \xrightarrow{I_{cn}} S_{i+n+1} \ldots \xrightarrow{I_f} S_f$

  where I_i are instructions and S_i are machine states (e.g. memory, register, pc) and such that $\{I_{cj}\ , \forall j \in [0, n]\ \} = C_m$
- **Memory corruption**: $S_i[MEM([m(d) \leftarrow d'])]$ denotes the fact that a corrupted value *d'* is injected into memory location *m* to replace the legitimate data *d* during state S_i.
- **Memory content**: $S_i[MEM([m = d])]$ denotes the fact that at state S_i the memory location *m* contains the data *d*.
- **Original data in memory m**: *d*
- **Predicate associated with the critical variable m**:

  $P_m : Execution\ Flow \times \mathbb{N} \times value \times value \rightarrow boolean$ such that: $(P_m(E, i, d, d') = true) \Leftrightarrow$ (During the execution E, $S_i[MEM([m(d) \leftarrow d'])] \Rightarrow$ the attacker can achieve his/her goal)

  Intuitively, if we denote I_i the instruction executed to transition from S_{i-1} to S_i , the predicate P_m returns true if and only if I_i is part of a critical section associated with *m*.

Lemma 1: A critical code section associated with the critical memory *M* always starts with a write instruction to *M* and does not contain any other write to *M*.

Proof:

1. **Proof that a critical code section always starts with a write instruction to the associated critical memory location.**

Hypothesis 1: Ic_0 is **not** a write instruction to m

Hypothesis 2: I_{c0} is the first instruction in C_m.

Let us prove by contradiction that Ic_0 must be a write to m.

- As $I_{c0} \in C_m$, then $\exists d'$, $P_m(E, i+1, d, d') = true$.
- Let us now assume that the attacker has injected d' in m after the execution of I_i: $S_i[MEM([m(d) \leftarrow d'])]$.
- As I_{c0} is not a write instruction into m (hypothesis 1) then $S_i[MEM([m(d) \leftarrow d'])] \Rightarrow Si{+}1MEMm{=}d'$. (The data in memory address m has not been changed.)
- However we know that $P_m(E, i+1, d, d') = true$ and therefore $S_{i+1}[MEM([m(d) \leftarrow d' \Rightarrow$ the attacker can achieve his/her goal.
- By transitivity of the implication operator we have $S_i[MEM([m(d) \leftarrow d'])] \Rightarrow$ the attacker can achieve his/her goal. Therefore, $I_i \in C_m$, which is in contradiction with the hypothesis 2.
- Therefore Ic_0 must be a write to *m*.

2. **Proof that there is only one write instruction to *m* within each critical code section associated to *m*.**

Hypothesis 1: Let us suppose that there exists an instruction $I_{cj} \neq I_{c0}$ which is a write instruction to *m*.

- We denote by S_{cj-1} the machine state before the execution of I_{cj} and by S_{cj} the machine state after the execution of I_{cj}.
- We denote by *d* the value written into memory m by I_{cj}.

- Let us suppose that the attacker injects in state S_{cj-1} a malicious value: $S_{cj-1}[MEM([m(d) \leftarrow err])]$
- I_{cj} writes the value d in m; therefore $S_{cj}[MEM([m = d])]$ overwriting the value introduced by the attacker.
- Therefore I_{cj-1} cannot be in a critical section associated with m.
- However, I_{cj-1} is executed after I_{c0} and before I_{cj} and a critical code section is constituted of consecutive instructions; therefore by definition I_{cj-1} is in the critical code section. This contradicts the previous statement.
- Therefore hypothesis 1 does not hold.

Conclusion: This proves that there can be only one write instruction within each critical section and it is the first instruction.

Lemma 2: The instruction following the last instruction of the critical section is always a read from the critical memory location (critical variable).

Proof:

Hypothesis 1: I_m is the last instruction in the critical section and *I'* is the instruction executed right after I_m.

Hypothesis 2: Let us suppose that *I'* does not read the critical memory m.

I_m is in the critical section that implies that there exists a value *d'* such that the attacker can achieve his goal by writing *d'* into memory location m after I_m is executed. That also implies that there exists an instruction, I_r, executed after I_m that reads m and that the corrupted value *d'* propagates at least until I_r is executed (if the corrupted value is never read, then the attacker cannot alter the behavior of the application).

Now, we have assumed that *I'* is an instruction that does not read *m*; therefore corrupting *m* after *I'* with the value *d'* has the exact same effect as corrupting *m* after I_m: the corrupted value *d'* is going to propagate until I_r and the attack will succeed. Therefore *I'* should be in the critical section which is contradictory with the hypothesis 1 (I_m is the last instruction in the critical section). Therefore the instruction following the last instruction of the critical section is always a use of the critical memory.

Intuitively, the lemmas show that a data in memory can only be corrupted between the time it is written in memory and the time it is used. Also, as soon as the data is written into memory and as long as it is not used (loaded into registers), any changes to the data will be reflected at the next read instruction.

Chapter 4: SymPLAID

SymPLAID is a program level formal framework developed in our research group at the Coordinated Science Laboratory. SymPLAID runs on top of the Maude model checker. It has been designed to find application vulnerabilities to insider attacks [9] using symbolic model checking. In this thesis, we leverage the symbolic fault-injection capabilities of SymPLAID along with the flexible design of the machine model in order to design, model and verify detection mechanisms. Critical variables along with associated critical code sections are identified using SymPLAID's symbolic fault-injector. Based on the injection results, detectors are then designed, modeled and verified within the formal framework.

SymPLAID consists of the following components (all written using Maude functional modules or Maude system modules):

- A machine-specific front-end which describes machine properties such as the number of available registers, word sizes, instruction sizes, address calculation, the instruction set and how the instructions execute.

- A machine-independent back-end which contains the concrete evaluation model, the error model, the fault injection mechanism, and application model.

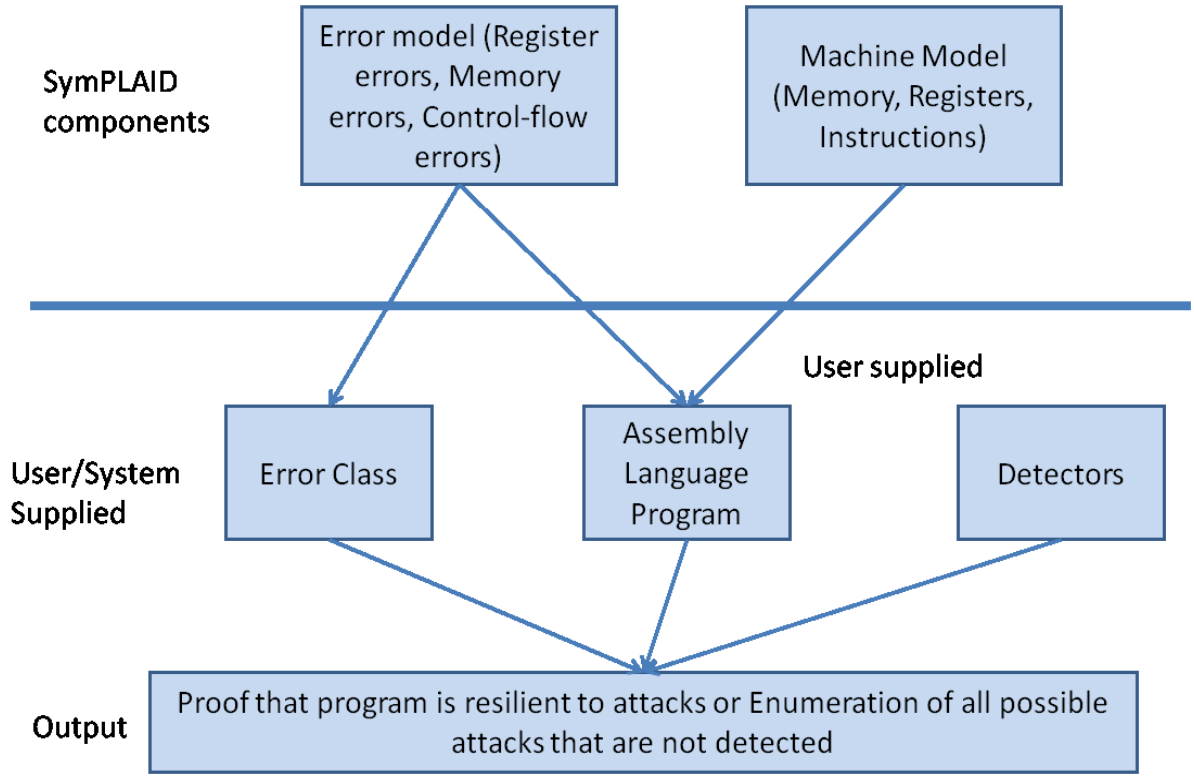
Figure 2 is derived from [13] and describes the overall design flow of SymPLAID

4.1 Input

The application model is close to the actual MIPS assembly language, so there is no manual effort needed to formalize the analyzed applications. A script automates the translation of the assembly program into a Maude functional module [14]. *Figure 3* shows a simple factorial program translated into a formal module using the translation script in

Perl. The formal module declares the application's functions, labels and base addresses as operators (Line 2, 3 *Figure 3*). The definition of each operator is then written using equations.

Figure 2: Conceptual design flow of SymPLAID



Each function operator (here the *factFunc* operator) involves two equations:

- A first equation indicates the base memory address of the text segment associated with the function (Line 4, *Figure 3*).

- A second equation defines a raw representation of the function's assembly code in Maude which is simply a succession of "labeled instructions" (Line 5, *Figure 3*).

First is the label (address) of the first instruction to execute (e.g. *factLoc* in the given example) [1] (Line 21, *Figure 3*).

[1] If an application has several functions, *first* would correspond to the label of the first function to be executed.

Input is the input value to the application. In *Figure 3*, the input value is 5 so the application should compute factorial 5 (Line 22, *Figure 3*).

The *relocate* operator (Line 21, *Figure 3*) grants every instruction a text segment address calculated based on the machine's instruction size.

Figure 3: Factorial program translated into formal module

```
--- Program to calculate factorial of a number
   mod FACTORIAL is
0     including ERROR-EXECUTE .
1      including RELOCATE .
2     op factFunc : -> Code .
3     op factLoc : -> Label .
4     eq factLoc = 100 .
5     eq factFunc = ---- Function to compute factorial of a number
6         [ 0 | ( ori $(2) $(0) #(1) ) ] --- initial product
7         [ 1 | ( ori $(6) $(0) #(1) ) ] --- initial product
8         [ 2 | ( read $(1) ) ] --- initial i
9         [ 3 | ( mov $(3) $(1) ) ] --- copy of input
10        [ 4 | ( ori $(4) $(0) #(1) ) ] --- for comparison
11        [ 5 | ( sgt $(5) $(3) $(4) ) ] --- start of loop
12        [ 6 | ( beqii $(5) #(0) #(13) ) ]
13        [ 7 | ( mult $(6) $(6) $(1) ) ]
14        [ 8 | ( mult $(2) $(2) $(3) ) ]
15        [ 9 | ( subi $(3) $(3) #(1) ) ]
16        [ 10 | ( beqii $(0) #(0) #(5) ) ] --- loop backedge
17        [ 11 | ( prints "Factorial = " ) ]
18        [ 12 | ( print $(2) ) ]
19        [ 13 | ( halt ) ] .
20     eq program = relocate( adjustSize(factFunc), factLoc ) .
21     eq first = factLoc .
22     eq input = & 5 .
23     eq timeout = 100 .
   endm
```

4.2 Implementation

SymPLAID is implemented using rewriting logic. Equations are used to model deterministic actions such as the program execution or memory and register lookups... Rules are used to introduce non deterministic transitions when injecting errors.

A machine state describes the states of the registers, memory, program counter etc. during the execution of an application. The whole application's execution consists of going

from a machine state to another through transitions. *Figure 4* lists the fields which define the machine state.[2]

In other words, transitions modeling the fault-free execution of an application are written with equations and transitions involving errors are written with rules.

Figure 4: Machine state description

- *PC*: the label of the instruction to be executed next.
- *out:* application output
- *inp:* application input
- *regs:* register state
- *mem:* memory state
- *temp:* temporary code
- *ex:* exceptions thrown
- *bkpts:* breakpoint list
- *step:* number of rewriting steps left
- *numInsts:* the number of instruction executed
- *cons:* constraints on the fault injected
- *log:* (fault log) records when the fault has been injected and when it has been activated (optional)
- *hist:* records every random jumps due to error in the program counter. (optional)

4.3 Error injection

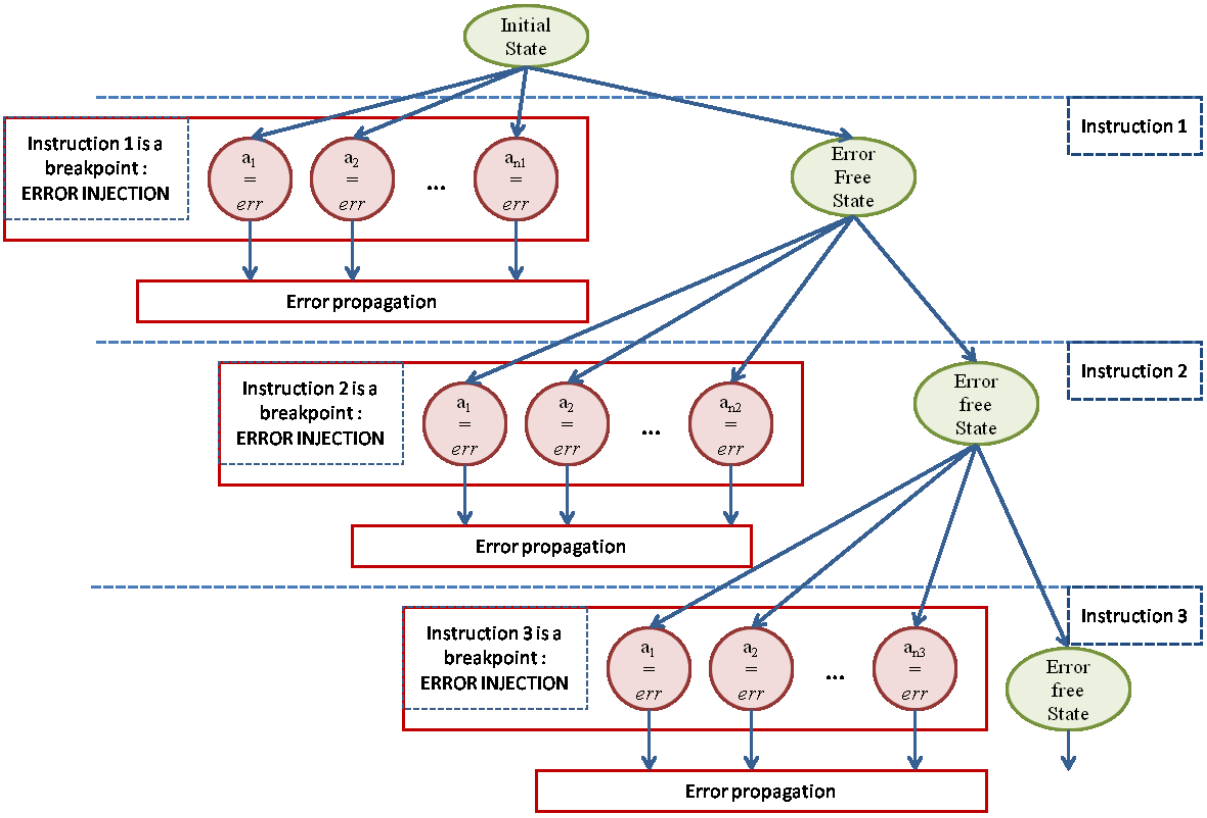
The error injection mechanism mimics the behavior of existing fault injectors like NFTAPE [15]. A breakpoint is set at a given application's execution point. When the breakpoint is reached, the *err* symbol is injected at one memory location. Then, the application's execution resumes propagating the error.

[2] The machine state is easily extensible.

4.3.1 Error injection mechanism

At each run, the user is able to specify where to inject the symbolic error (into register, memory or program counter). In this thesis, as we wish to model memory corruption attacks, we only consider memory errors. The user is also able to specify when to inject the error by setting breakpoints. By default, SymPLAID exhaustively and successively injects errors at every available memory location and at every possible breakpoint. *Figure 5* illustrates the exhaustive fault injection process. Within each state, a_i denotes memory location i and $a_i = err$ denotes the fact that the memory location a_i contains the error symbol. Note that there is at most one error injected on each path of the state diagram shown below.

Figure 5: State diagram of error injection of all memory addresses at all possible breakpoints



The initial state is a state where no instruction has been executed yet. After instruction *1* is executed, $n_1 + 1$ states can be reached from the initial state (n_1 being the number of memory locations available). In n_1 state, a unique symbolic error *err* is injected. In the first state, an error is injected into memory location 1, and in the i^{th} state an error is injected into memory location i. Once injected, each error is propagated independently from the others. The $(n_1 + 1)^{th}$ state is an error-free state from which new states are "forked" after the execution of the 2^{nd} instruction.

Table 3 shows some of the injection commands defined in SymPLAID. The Search command defined in Maude enables us to explore all reachable states from the initial state given the specified injections. The next section discusses the set of rules and equations that control how errors are injected.

Table 3: Basic commands to inject memory errors using SymPLAID and Maude

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

4.3.2 Injection rules and equations

This section describes the main equations and rules used to perform the symbolic fault injection. Other equations and rules may need to be modified or added depending on the error injection model chosen, but equations and rules presented here are the starting point for any

error injection model modification. In particular, the state model, the error propagation rules and the constraint solver may need to be modified accordingly. [3]

Notations: Let *C* be a Code, *lb* and *L* Labels, *I* an Instruction, *In* the input, *MemError* an errorType (memory error), *M* a memory, *a* an address and *v* a value.

Equation 1:

eq allMemoryErrors (C,lb,In)=allError(C,lb,In,MemError) .

Explanation: SymPLAID enables the injection of symbolic fault into memory, register and PC and this equation allows the definition of an unique "initialization rule" (rule 1 defined below) and one "injection start equation" (equation 2 defined below) for all three types of injections. To enable the injection of a new fault model, it is sufficient to define a new *errorType* corresponding to the new fault model and add an equation similar to equation 1. Rule 1 and equation 2 can then be reused.

Rule 1 (Initialization rule):

rl allErrors([L,I]C,lb,In,Type) => injectStart([L,I]C,lb,In,Type,L) .

Explanation: In Maude, rules are used to introduce nondeterminism. If the analyzed code, *C*, is composed of n instructions $\{[L_0, I_0][L_1, I_1] \ldots [L_n, I_n]\}$ this rule is equivalent to: $\forall i \in [0, n]$; from the initial state S_0 we can either transition to a state where L_i is set to be a breakpoint or a state where L_i is not set to be a breakpoint. That allows the model checker to consider all possible breakpoints when looking for all reachable states: each instruction, *I*, can be set to be a breakpoint or not. For example, to model an attack where the attacker can

[3] In this work we did not implement any new error model in SymPLAID. We only point out in this section rules and equations that may need to be modified.

modify *n* different memory addresses from *m* different breakpoints, this is the equation to change in order to add breakpoints. (Note: A conditional rule exists to limit the choice of the breakpoint pcs to be between an min and a max label. It is mostly similar to the rule given above except that we also check that $min < L_i < max$.)

Equation 2 (Injection starts equation):

eq injectStart(C,lb,In,Type,L)= injectError({ C, < fetch(C, st),init(st, bkpts(L) inp(In) step(noStep) log(Empty))> },type, L) .

Explanation: Equation 2 allows an injection campaign to start given that one breakpoint pc, *L*, has been chosen.

Rule 2 (Error injection rule):

crl [do-injection]: injectError({ C , < I , PC(L) Mem((a = v) M) bkpts(BL) S > },MemError, L) =>{ C, < I, PC(L) Mem([a <- err] M) bkpts(removeBkpt(L, BL)) S > } if (not isTerminal(L) and (L in BL)) .

Explanation: When a breakpoint pc, $L_{i0,}$ is reached and it is not the end of the execution, this conditional rule is equivalent to the following:

Denote by n_{i0} the number of memory locations used in the application's memory space at the breakpoint L_{i0}.

Let $\{a_0, a_1, \ldots a_{ni0}\}$ be the set of memory addresses used in the memory space of the application at the breakpoint $L_{i0.}$

$\forall j \in [0, n_{i0}]$ from the state S = *{ C , < I , PC(L) Mem((*a_j *= v) M) bkpts(BL) S > }* a transition exists to a state S' where:

- An error is injected into memory address a_j and the breakpoint L_{i0} is removed from the list of breakpoint: S' = *{ C, < I, PC(L) Mem([a_j <- err] M) bkpts(removeBkpt(L_{i0}, BL)) S >}*

- or there is no error injected into memory address a_j.

The equations and rules presented above allow the model checker to find all reachable states when considering all possible memory injections of 1 memory address at a particular breakpoint. To introduce more complex attacks when an attacker can corrupt m memory addresses at a particular breakpoint, this is the rule to modify.

4.4 Error propagation

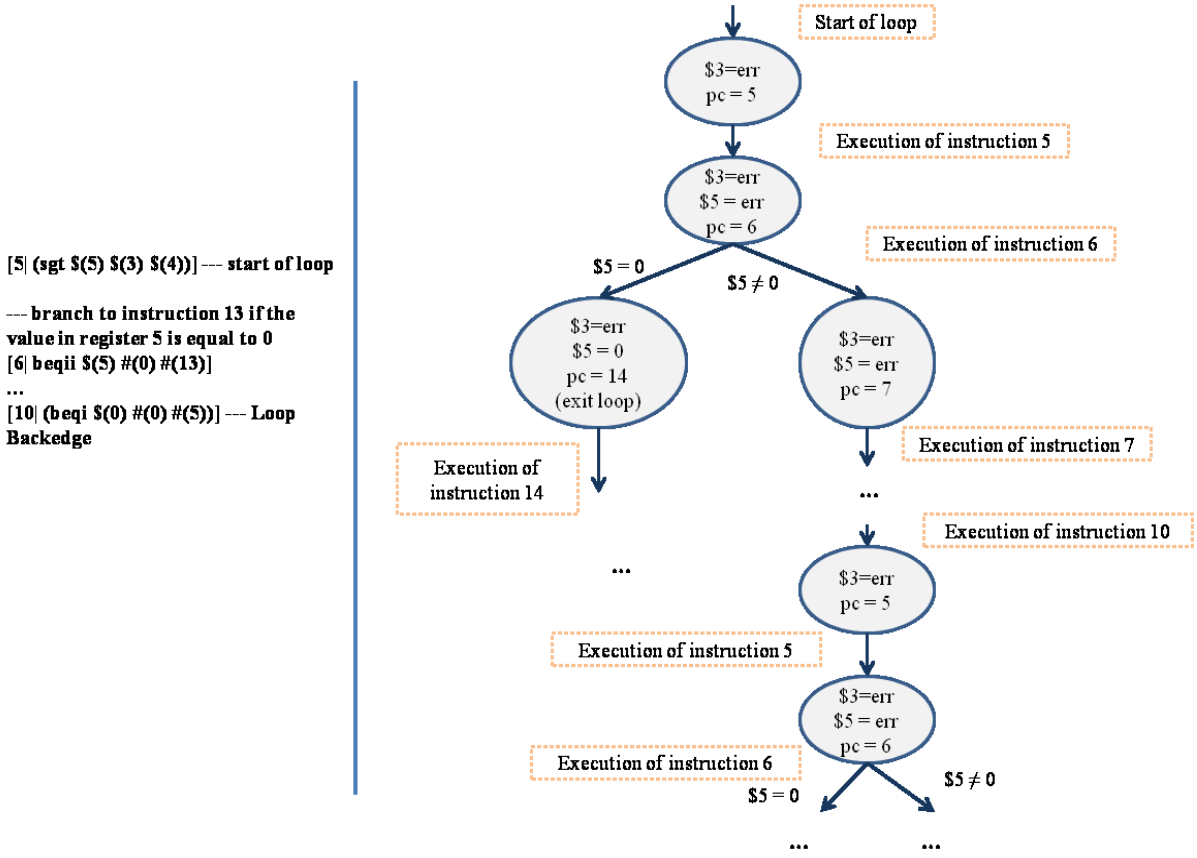
Once the error is injected into the program, the error is propagated. In this paragraph, we describe how error is propagated during the application's execution. Basically every operation involving err will return error expressions merging err symbols (e.g. *err + 2 = err*, *err + err = err*).

The interesting part of the error propagation is how the tool handles branching or comparison involving the err symbol. In the cases where branching and comparison are not involved, merging expressions into err is sufficient. But if we need to evaluate a predicate in order to determine how to branch, merging expressions into *err* is not possible anymore. SymPLAID uses rules to fork the execution when evaluating a predicate involving the *err* symbol: in other words a predicate containing an *err* symbol is evaluated to both true and false.

Take the example of the factorial module presented in *Figure 3*. Say an error propagated into *register 3 ($3)*. After instruction 5 executes, the error propagates into *register 5 ($5)*. When evaluating whether the value in *register 5 ($5)* equals to 0, SymPLAID forks

the execution. *Figure 6* illustrates this example by showing part of the state diagram representing all reachable states.

Figure 6: Example of propagation of err into a branch predicate



However, in this case, the error in *register 3* continues to propagate; therefore, at the next loop iteration, *register 5* will still contain an error (whose value is not constrained). That means that we will fork forever, and that is why SymPLAID uses bounded search (the number of rewriting steps allowed to reach a state is bounded) to ensure that the analysis ends.

4.5 Constraint solver

The error abstraction enables us to reason about a whole category of errors without having to provide a special value for it. The major problem with this abstraction is the presence of false positives. To limit the number of false positives, SymPLAID has a built-in

constraint solver. The constraint solver keeps track of constraints SymPLAID can identify on an err symbol. Consider the following dummy instructions and assume that register 5 contains the err symbol:

```
--- branch if the value in register 5 is greater than 0
[ 2 | ( bgtzali $(5) #(0) #(4) ) ]
[ 3 | (subi $(5) $(5) $(1) )]
--- branch if the value in register 5 is less than 2
[ 4 | ( bltzali $(5) #(3) #(7) ) ]
```

Note that each time register 5 is used in an instruction the constraint on register 5 is updated. Furthermore, whenever possible the constraints are merged. But, if we have a case of incompatible constraints (e.g. $5 ≤ −1 and $5 ≥ 3 in our case, *Figure 7*), the state with the incompatible constraints is discarded (red state in *Figure 7*).

Figure 7: Example of constraint propagation

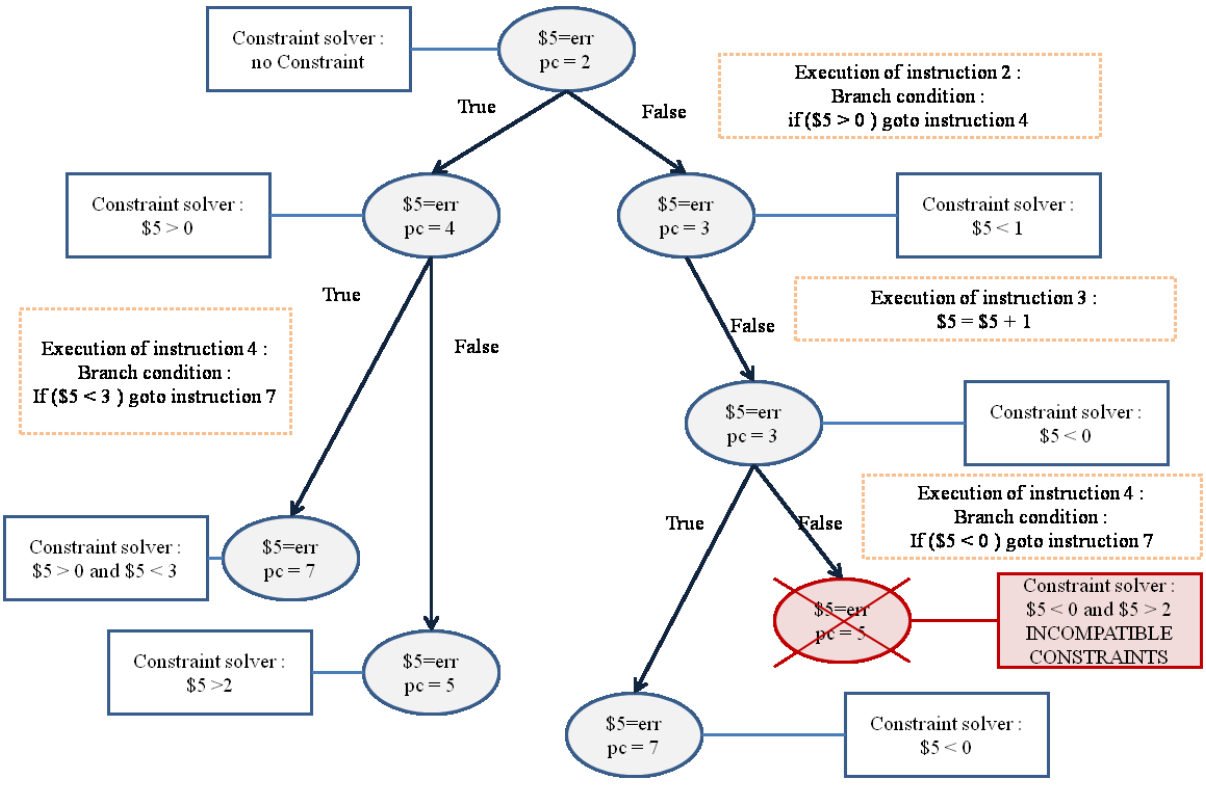


Figure 7 presents the state diagram associated with the above instructions.

The final constraint on the *err* symbol is given to the user as a part of SymPLAID's results. That indicates the values or range of values the *err* symbol should carry in order to achieve a given result. In other words, the constraints on *err* represent values or range of values an attacker should inject in order to achieve a given goal.

Chapter 5: SymPLAID Validation

Before using the results from SymPLAID's analysis to identify critical variables and critical code sections, we first evaluated the false positive rate of the tool. The goal was to ensure that injecting a fault, on real systems, in any variable identified as critical would allow the attacker to achieve his/her goal.

This section describes and explains the results obtained from the evaluation of SymPLAID. The application chosen for this analysis is the sorting algorithm, Bubblesort,[4] from the Stanford benchmark. In this evaluation, SymPLAID was asked to return all memory corruptions that would lead the application to return a corrupted output (e.g. non-sorted list, corrupted value in the output) without system crash or any system exception being raised. A tuple *{breakpoint pc: I , memory location to corrupt: m, value to inject: d}* is considered to be a false positive in the following cases:

- Injecting, on a MIPS system, the value *d* in memory address *m* at the breakpoint *I* does not impact the normal behavior of Bubblesort. In other words, Bubblesort returns the correct sorted list.
- Injecting, on a MIPS system, the value *d* in memory address *m* at the breakpoint *I* results in a system crash and/or system exception being thrown.

To evaluate the false positive rate of SymPLAID, an automated framework has been set up to allow the user to:

- generate scripts and run parallel SymPLAID analysis of a given program on the Trusted-ILLIAC cluster [16]. Each cluster node performs one job which

[4] Most false positives identified by this analysis were due to features in SymPLAID that are independent of the application analyzed (constraint solver, additional state fields). Therefore we believe that the results presented in this section are valid for any application.

consists of one SymPLAID's analysis. Each job is in charge of injecting memory errors in all memory addresses available in the application's memory space at n consecutive breakpoint instructions (n defined by the user).

- parse and filter the results (states) returned by SymPLAID.
- inject all tuples {breakpoint pc, memory location, fault value} found by SymPLAID during the execution of the analyzed application in the SimpleScalar [17] simulator to evaluate the false positive rate of SymPLAID.

5.1 Analysis of SymPLAID's result

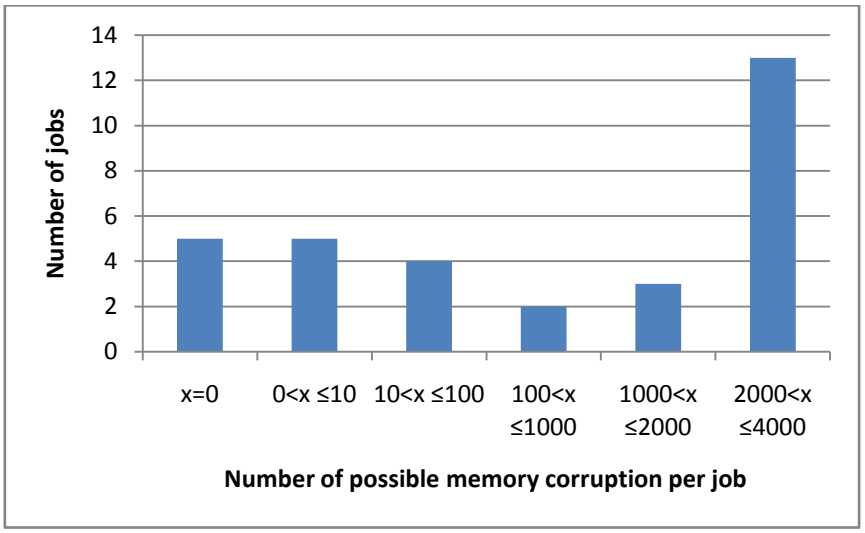
The analysis of Bubblesort in SymPLAID ran on the Trusted-ILLIAC cluster. Bubblesort consists of 96 assembly instructions. As an input, a list of 10 numbers from 0 to 9 was provided to Bubblesort. SymPLAID was asked to find all memory corruptions that could lead the program to finish without crashing or throwing any exceptions.

The analysis was divided into 32 subtasks (running on 32 nodes), each of which had to inject faults within 2 or 3 consecutive instructions (using the 3rd command presented in Table 3). The longest jobs took about 8 hours and were cancelled due to "node failures."[5] On average, though, jobs took less than 1 hour to complete, which seems to be a reasonable analysis time given that each job corresponds on average to the injection of all 2^8 possible values in each memory location (about 30 memory locations in this case) at 2 or 3 consecutive breakpoints. That would correspond at least to 23 * 30 * 2 = 15360 injections with a regular fault injector. We present, in a later chapter, methodologies to reduce the number of injection needed while preserving the completeness of the analysis.

[5] We think that those node failures were caused by an excessive memory usage from the model checker.

SymPLAID generates as output all reachable states given the fault injections. We filter the states returned by SymPLAID in order to select only those with an incorrect output (unsorted list/ wrong number of output numbers/ wrong list of number in the output). From each state obtained after filtering we identify a tuple which represents a possible memory corruption attack: {Breakpoint pc, memory location corrupted, constraint on the value to inject}. Such a tuple is interchangeably called in the rest of this thesis a solution, tuple or possible memory corruption. For Bubblesort, SymPLAID returned 48,622 possible memory corruptions or solutions. *Figure 8* represents the number of possible memory corruptions identified by SymPLAID per job.

Figure 8: Number of possible memory corruptions per job



From *Figure 8* we can see that:

- Five jobs did not return any solution, which means that altering any data in memory after the instructions analyzed in those five jobs would either lead to a program crash, an exception being thrown or have no consequence for the application's behavior. Therefore no protection is needed during the execution of those instructions.

- Thirteen jobs out of 32 gave more than 2000 possible memory corruptions. That means that attacks which corrupt any memory location when the instructions analyzed in those 13 jobs are being executed have better chance to succeed. That indicates that "critical code sections", as defined earlier, exist within an application. In other words, some sections of an application are more sensitive to memory corruption.

5.2 False positive rate validation

We wrote Python scripts to automatically:

- derive from the dat returned by SymPLAID, tuples representing possible memory corruptions (or solutions): {Breakpoint pc, memory location corrupted, constraint on the value to inject}.
- perform automated fault injection, based on the information contained in each tuple, in the SimpleScalar simulator.
- gather and parse the injections' results.

SimpleScalar [17] is an open source computer architecture simulator. In particular, SimpleScalar simulates MIPS architectures. An injector [6] implemented as a part of SimpleScalar's debugger DLite was used to perform the injections.

All 48,622 solutions were injected using within SimpleScalar. The whole fault-injection process being automated, all injections were performed within 4 hours.

[6] The injector was implemented previously in our research group.

5.2.1 Injected values

SymPLAID does not always return a precise value to inject but instead gives constraints on the value to inject. *Table 4* shows the values[7] we choose to inject given SymPLAID's constraints.

Table 4: Value injected based on SymPLAID's constraints

SymPLAID's constraint	Value injected
err=x	x
x<err<y	x+1
x<err	x+2
err<y	y-2
no constraint	random

5.2.2 Injection result analysis

Figure 9 shows the results of the fault injections in SimpleScalar. The red and blue sections represent false positives.

After analyzing the injection results we found the following:

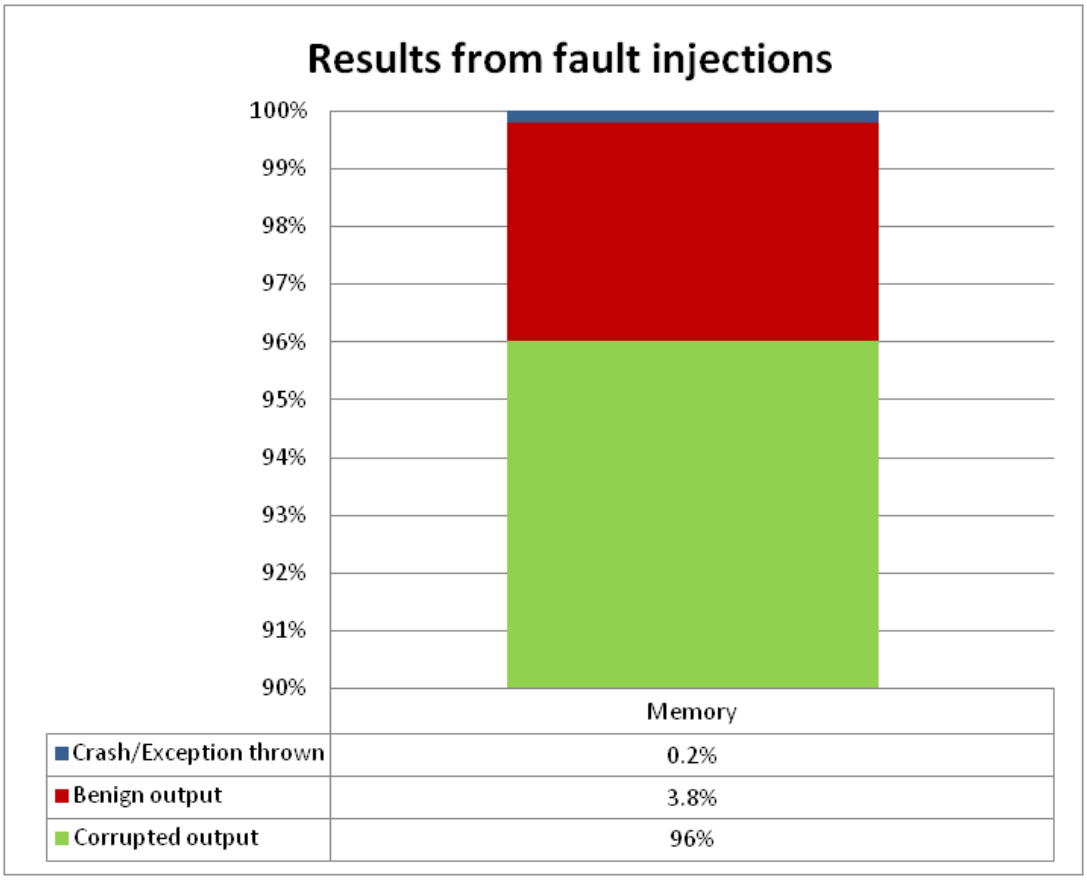
- Within all 48,622 possible memory corruptions reported by SymPLAID there are fully redundant tuples, which means that they share the same breakpoint PC, the same constraint on the value to inject and the same memory locations where the fault is injected. In *Figure 4*, which describes the machine states' fields, we can note that there is a *Step* field and a *numInsts* field. Those fields represent the number of rewriting steps the model checker went through in order to reach the reported state. If those fields differ in 2 final states, the model checker reports 2 different solutions even if all other fields are identical. It is possible to get rid of all the redundant tuples by adding a new operand in the module defining a State:

[7] The values chosen are arbitrary.

getSubstate. The getSubstate operand would return all the fields composing a State except for the *step* and *numInsts* fields. Issuing the command: *search getSubstate(allMemoryErrors(program,first,input)) =>! (S:State)* returns all different reachable substates. After filtering all redundant tuples, we reduced the number of solutions to 4833 distinct solutions

- Analyzing all 4833 distinct solutions allowed us to find that most false positives were due to the constraint solver not being able to determine a correct constraint on the value to inject.
- The rest of the false positives are due to loops. When asking SymPLAID to inject fault at a breakpoint pc x, SymPLAID is going to inject a fault each time

Figure 9: Results from the fault injections



the instruction x is reached. For example if we have a loop and that instruction x is executed twice, then SymPLAID is going to assume 2 distinct sets of injection runs: the first set of injections will inject err at all possible memory locations when instruction x is reached the first time, and the second set of injections will inject err at all possible memory locations when instruction x is reached the second time. While injecting in SimpleScalar, though, we always inject the error the first time instruction x is reached. which lead to discrepancies between the 2 sets of results.

As a conclusion to this evaluation, we found that the false positive rate is reduced from 4% down to 2% when false positives introduced by the constraint solver[8] are ignored. In that case, more than 98% of solutions identified by SymPLAID actually lead to a corruption of the application's output and can be used to identify critical variables and critical code sections.

[8] As our goal is to find critical variables and critical code section, the constraint on the value to inject is not important for our work. Therefore, we can ignore the false positives introduced by the constraint solver without adding false positives in the identification of critical variables and critical code sections.

Chapter 6: Deriving Detectors

This chapter describes: (i) how to identify critical variables and critical code sections using SymPLAID's symbolic fault injector, (ii) the critical variables and critical code sections identified for the two benchmark applications (*WuFTP* and *OpenSSH*), (iii) the design, implementation and coverage evaluation of detectors derived in our formal framework for the two target applications.

6.1 Identifying critical variables and critical sections

Critical variables and critical sections are, as defined in Chapter 3, respectively, memory locations that can be altered by an attacker to achieve a particular goal and the time frames during which an attack can occur. We use SymPLAID to perform an exhaustive fault injection campaign, where we inject all possible memory errors (represented as symbolic errors) at all possible breakpoints (corresponding to instructions within the program) and examine the resulting effect on the application's execution leveraging the symbolic fault-injector. All benign faults (no observable corruption) and all faults leading to crashes or exceptions being raised are then filtered.

The finding of critical variables and critical section by running SymPLAID is demonstrated on two application stubs, namely the authentication stub of *OpenSSH* and the authentication and log stub of *WuFTP*.

We assume in the following sections (6.1.1 and 6.1.2) that the attacker's goal is to alter the application's behavior. To do so, he/she has the capability of corrupting a single memory location during the execution of the application.

6.1.1 WuFTP

WuFTP is an open-source implementation of a File Transfer Protocol (ftp) server written in C [18]. The WuFTP stub we consider in this work translates into 252 assembly instructions and performs the following tasks: get the user name, get the password, if the password is correct display “correct login. Enter path” and log the path provided by the user. If the password is wrong then it displays “Incorrect Password” and exits. As an input for the analysis, we provided the application with a wrong password. The injection results are then filtered to only retain those where the attacker got authenticated even if the entered password was incorrect.

The analysis in SymPLAID ran on the Trusted-ILLIAC cluster divided into about 130 parallel jobs. All tasks finished within 4 hours without error. SymPLAID’s results were then parsed in order to identify all memory locations that can be altered in order to achieve the previously stated goal. All identified memory locations are logged as critical variables and we record all breakpoint pcs where the fault had been injected to identify each variable's critical section.

In order to have a high level understanding of what corresponds to each critical variable we map each critical variable (or critical memory location) to higher level variables (C like variables) which are listed in *Table 5*. The table also lists, for each variable, its critical section within each function. For example, for the memory location associated with the string of the *correct password*, its critical section within the function *main* is between the first instruction of main and the 38th instruction.

After analyzing SymPLAID’s results, we found 5 high level variables, corresponding to 6 different memory locations, that can be modified by an attacker in order get authenticated with an invalid password (*Table 5* first column).

Table 5: Critical variable analysis done on WU-FTP stub (mapping to high level variables)

Name of Critical Variables	Critical section [pc min - pc max]			Stack/heap
	Main	Auth-response	Check-auth	
string of the correct password	[1 - 38]	[1 - 15]	[1 - 16]	Heap
Pointer to the string "Incorrect Password" to be compared with "login correct"	[42 - 45]	N/A	[26 - 26]	Stack
Pointer to the user entered password	N/A	[9 - 15]	[6 - 14]	Stack
Pointer to the correct password	N/A	[12 - 16]	N/A	Stack
Decision making data	N/A	N/A	[17 - 20]	Stack

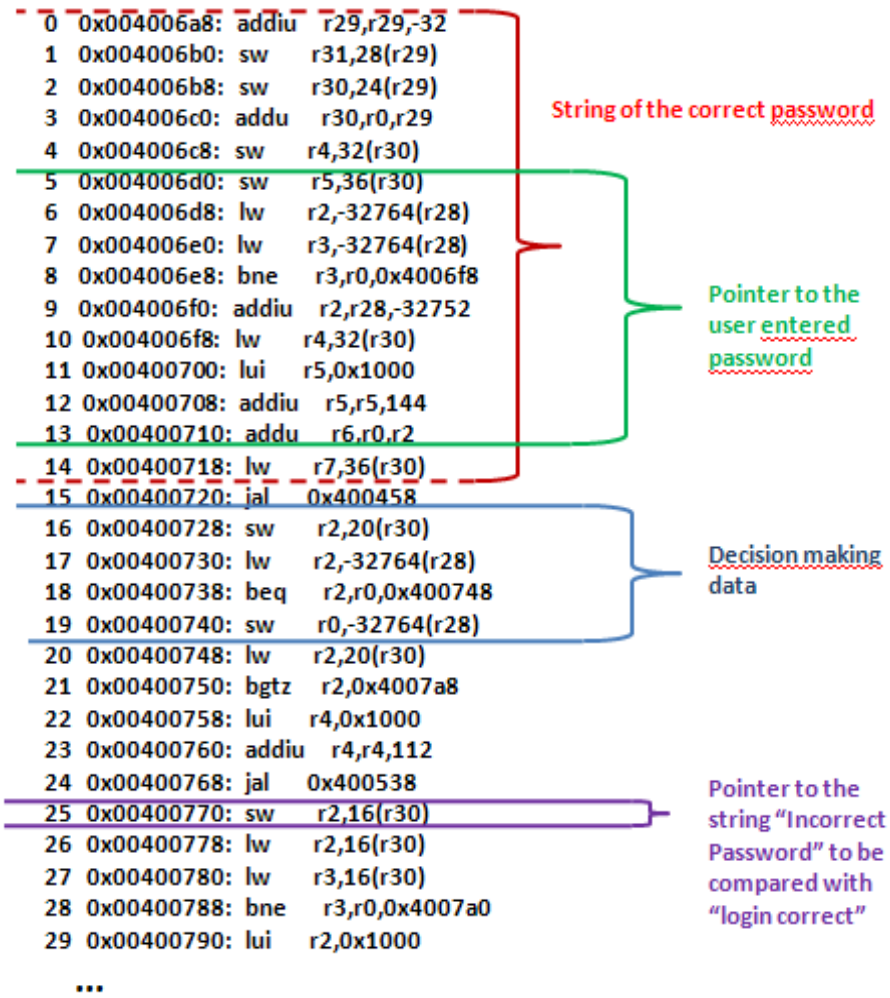
Moreover, for each critical variable, critical code sections, which correspond to the time frame where an attack can happen (*Table 5* column 2,3,4) were identified. The analysis guarantees that outside of those critical sections, critical variables are safe and thus do not need to be protected. The width of each critical section provides an understanding of how easy or likely it is for an attacker to achieve the corruption of a given critical variable. Indeed, if a critical variable can only be modified between instruction *i*, which stores the value into memory, and instruction *i+1*, which reads the value from memory, it is very hard for the attacker to achieve such an attack. To succeed, the attacker has to find a way to stop the execution of the application and corrupt the memory location before it is read (e.g. using a debugger). On the other hand if a call instruction to a third party library is found within the critical section or if an instruction getting input from the user is found within the critical section, critical variables may be easily modified by the attacker.

Figure 10 shows part of the *check-auth* function and the critical code section associated with each critical variable. The critical code section of the string of the correct password is limited by dotted lines in this figure. That is because the critical code section starts within the *check-auth* function's caller function (*main*) and ends within the function

called by check-auth at line 15 (*auth-response*). For all other critical variables, we can see that all critical code sections start with a write instruction to the associated critical variable and end before a load instruction from the critical variable.

Also note that while the stub contains 6 functions—*Main*, *Auth-response*, *Check-auth*, *Store*, *Auth-approve* and *Auth-value*—SymPLAID did not find any solutions within 3 of them: *store*, *auth-approve* and *auth-value*. Therefore, in order to achieve the stated goal, an attack cannot occur within those functions. In other words: those functions do not need to be protected.

Figure 10: Critical code sections associated with critical variables in *check-auth* function



6.1.2 OpenSSH

OpenSSH is a freely available C implementation of a Secure Shell (SSH) [19]. We focus in this work on a stub which contains the authentication mechanism of *OpenSSH*.

The same analysis as the one done on the *WuFTP* stub was performed on the *openSSH* authentication stub. The *openSSH* stub translates into 473 assembly instructions and performs the following tasks: get the user name, get the password, if the password is correct return 1 and if the password is wrong return 0.

The analysis was first performed with a valid username and a wrong password. The results were then filtered in order to keep only cases where the attacker got authenticated despite providing an invalid password.

The analysis in SymPLAID ran on the Trusted-ILLIAC cluster and was divided into about 250 parallel tasks. Most of the tasks finished within 1 day without error. Eight tasks took more than 4 days after which we stopped the analysis.[9] SymPLAID's results were then parsed in order to identify all memory locations that can be altered in order to achieve the previously stated goal.

The result of the analysis is shown in Table 6. Again, the critical memory locations are mapped in this table to higher, C-level, variable for the sake of clarity.

For *OpenSSH*, again only small number of critical variables to protect was found.[10] The string of the correct password in particular seems to be a major attack point as it can be corrupted in order to achieve the stated goal throughout the execution of the whole application.

---

[9] No obvious reasons were found to explain why the analysis did not complete.
[10] As mentioned in [20], there is one spurious attack SymPLAID finds because the studied ssh stub has its own malloc implementation. We do not consider those spurious attacks in our analysis.

Table 6: Critical variable analysis done on SSH stub

Name of Critical Variables	Number of instructions in each function after which a successful corruption can happen							Stack/heap
	Main	Sys-auth-Password	Auth-password	Allowed-user	getpwnam	Getpwnamallow	xcrypt	
string of the correct password	[1 - 41]	[1-15]	[1-17]	[1 – 10]	[1 – 21]	[1 - 14]	N/A	Heap
Pointer to the correct password	N/A	[20-46]	N/A	N/A	N/A	N/A	N/A	Stack
Pointer to user entered password	N/A	[41 - 42]	N/A	N/A	N/A	N/A	[8/31]	Stack
Decision making data	N/A	[47 - 48]	[24-32]	N/A	N/A	N/A	N/A	Stack

The same analysis was performed on the *openSSH* with an invalid username and a random password as inputs. The analysis showed that only the string of the correct password could be corrupted in order to be authenticated. Furthermore the critical section is very small: only 2 instructions within the main function (from instruction 0 to instruction 3).

6.2 Deriving detectors

Once critical variables and critical sections are determined, it is possible to design detectors using SymPLAID's framework. We demonstrate that it is easy to model and verify detectors within the framework. A simple detector model was designed, placed and validated on the two benchmark application stubs presented earlier: *WuFTP* stub and the *OpenSSH* authentication stub. It is important to note that the whole design, placement and validation process was fairly quick: 2 days to complete for the *WuFTP* stub and 4 days for the *OpenSSH* authentication stub.

6.2.1 Detector overview and implementation

A critical variable at memory address m can only be successfully overwritten within its critical section $[I_0, I_m]$. We have proved in section 3.2.2 that I_0 is the only write instruction

to *m* within the critical section and the instruction executed after I_m uses *m*. Let us denote R_m the instruction executed right after I_m.

Detection technique:

The detection technique proposed in this thesis duplicates the value of memory address *m* into a secure memory region before I_0 and compares the value of memory address *m* with the stored duplicated value before R_m. Any corruption of the data stored in *m* will be detected when compared to the stored duplicated value.

The detectors described above have been implemented within SymPLAID.

6.2.1.1 Secure memory region

The secure memory region is modeled by adding in the state model, described in section 4.2, *Figure 4*, a new attribute: *CritMem*. *CritMem* is defined as a subtype of memory and inherits all the attributes and operators from the initial memory model. The only difference between the secure memory region and the initial memory section is that we do not allow fault injection into the secure memory region. In other words, corrupted values are allowed to propagate in the secure memory region, but the attacker cannot directly alter data in this memory section. In practice, that kind of secure memory region could be achieved by encrypting stored data, or using memory randomization or special hardware.

The following equations have been added in the secure memory model in order to model a fetch from the secure memory and the comparison of both the original and duplicated data.

```
a: address , v: value, CM: CritMem
op _in_: Memory Memory -> bool .
eq (a = v ) in (a=v)CM = true .
eq (a = v) in CM = false [owise] .
```

Access to the secure memory is done by adding 3 new instructions in the instruction set: *Encoding instructions, decoding instructions and invalidate instructions.*

6.2.1.2 Encoding instructions

Encoding instructions duplicate data and store them into the secure memory. The following conditional equation (*equation 3*) describes the execution of an encoding instruction which stores the value of register *rs* into secure memory.

Equation 3 Encoding instruction execution:

ceq [encode-exec] : { C , < (encode rs a), PC(pc) regs(R) mem(M) critMem(CM) S > } = { C , < fetch(C, next(pc)), updateState(updateStore(PC(next(pc)) regs(R) mem(M) critMem(CM[a <- (R[rs])]) makeProgress(S), a , rs, R)) >}

if (canMakeProgress(S, pc, timeout)) .

Explanation: The equation first verifies that the execution of the application does not end. If there is a *next pc* to execute and the number of rewriting steps has not reached the maximum number defined by the user (cf. section 4.4 for bounded rewriting), the secure memory field (*critMem*) is updated. The update consists of storing the value contained in register *rs* (*R[rs]*) at address *a*.

6.2.1.3 Decoding instructions

Decoding instructions fetch duplicated data from the secure memory section and compare them with the original data stored in memory. If a mismatch is found, then the original data has been corrupted and an exception is thrown.

Equation 4 Decoding instruction execution:

ceq [decode-exec] : { C , < (decode a), PC(pc) regs(R) mem(M) critMem(CM) S > } = { C, (if ((a = M[a]) in CM)

then < fetch(C, next(pc)) , PC(next(pc)) regs(R) mem(M) critMem(CM) makeProgress(S)>

else < throw (detException(a)) , PC(next(pc)) regs(R) mem(M) critMem(CM) makeProgress(S) > fi)} if (canMakeProgress(S, pc, timeout)) .

Explanation: Equation 4 describes the execution of a decoding instruction which verifies that the data associated with the memory address *a* is the same as the duplicated data stored in the secure memory. If a mismatch is found, a new type of exception, *detException*, is thrown, ending the execution of the application.

6.2.1.4 Invalidate instructions

For data stored on the stack, we added an invalidate instruction which invalidates the duplicated data in the secure memory section when functions return.

6.2.1.5 Example of implementation

The detectors proposed in this thesis can be implemented in either software or hardware. This paragraph presents an idea of how the detectors can be implemented in hardware. The secure memory region can be implemented in a special, secure hardware. During encoding phases, the secure hardware keeps track of where duplicated data are stored in the secure memory (e.g. using a table). During the decoding stage, the main hardware sends to the secure hardware the memory address *a* along with the data stored in *a*. The secure hardware then can find the duplicated data corresponding to memory address and check whether both duplicated data and the data sent by the main memory are identical. If both data are identical, then the hardware sends back a signal allowing the execution of the application; otherwise an exception is raised.

6.2.2 Detector evaluation

The coverage of the detectors derived using our formal framework is evaluated in the context of the *WuFTP* and the *OpenSSH* applications (presented in section 6.1.1 and 6.1.2). The two applications are instrumented with the detectors and SymPLAID based analysis is repeated to measure the coverage.

Encoding and decoding instructions (defined in paragraph 6.1.2.1 and 6.1.2.2) are placed in the applications following the critical code sections derived by the formal analysis. However, critical code sections are determined for given executions and therefore for given application inputs.

Therefore, to ensure that no instrumentation is missing, the instrumented application and SymPLAID's formal fault injection need to be executed against different inputs in order to exercise all possible execution paths.

A missing encoding instruction will lead to a false positive (an alarm being raised even if the application is not attacked). To detect a missing encoding instruction, the instrumented application is executed against multiple inputs in order to exercise all possible execution paths. If an exception is thrown during one of the executions, a corresponding encoding instruction can be added until all runs end without throwing any exceptions.

A missing decoding instruction results in a false negative (attack that is not detected). To ensure that no decoding instructions are missing and that all encoding / decoding instructions are correctly placed in the application's source code, SymPLAID's symbolic fault injection analysis is performed again on the instrumented application while feeding different inputs to the application.

The analysis' time of the instrumented application is greatly reduced (less than 30 min instead of 4 hours for the non-instrumented application). Indeed, by adding detectors all the

final states with corrupted outputs are reduced into one single state where the attack is detected (Exception thrown). That reduces the state search space for SymPLAID as the tool is asked to search for final states where no exceptions are thrown.

This set of analysis uncovers holes in the protection mechanisms proposed above. Indeed, because decoding instructions were inserted before the actual read instruction, SymPLAID detected that there is still a small window left for an attacker to exploit a *time-of-check-time-of-use vulnerability*. This window can be removed by placing decoding instructions after the read instructions and by changing the decoding instructions to compare the value already loaded into the register and the duplicated value stored into the secure memory. The following equation (*equation 5*) defines the modified decoding instruction execution.

Equation 5 Decoding instruction execution modified:

ceq [decode-exec] : { C , < (decode a rs), PC(pc) regs(R) mem(M) critMem(CM) S > } = { C, (if ((a = R[rs]) in CM)

then < fetch(C, next(pc)) , PC(next(pc)) regs(R) mem(M) critMem(CM) makeProgress(S)>

else < throw (detException(a)) , PC(next(pc)) regs(R) mem(M) critMem(CM) makeProgress(S) > fi)} if (canMakeProgress(S, pc, timeout)) .

After the proposed modification, SymPLAID's analysis was repeated on the instrumented code of *WuFTP* and *OpenSSH*. This time SymPLAID could not find any successful attack in any of the protected applications. We are guaranteed that our instrumented application is now resilient to all attacks assuming the attacker can only modify a single memory location.

6.3 On scaling the analysis

The *WuFTP* stub was rather small so we were able to wait for the whole critical variable analysis and the exhaustive fault injection to end before embedding detectors. For the *openSSH* authentication stub, however, it took us 4 days to identify the critical variables. Two techniques are proposed to accelerate the analysis and the detector placement process and to improve the scalability of the overall approach.

6.3.1 Technique 1: A practical approach.

The first technique is a practical technique which is based on the following ideas:

- It is not necessary to wait for all jobs to complete before identifying critical variables and critical code sections. The 2 properties demonstrated in section 2.2.2 along with partial fault injection results can be used to identify part of the critical code sections.
- SymPLAID's analysis is quicker after detectors are placed. That is because we eliminate from the set of reachable states all the states which result in application misbehavior (all possible corrupted outputs).

We use the following procedure to identify critical variables and critical code sections, and to place our detectors:

- Launch SymPLAID on the non-instrumented application and let it run until some final state with a corrupted output field is found (cf definition of a state in *Figure 4*, paragraph 4.2). The states are then parsed to identify the memory location where the fault has been injected and the instruction after which the fault has been injected. Each memory location identified is a critical variable, and each instruction identified by the analysis is part of the critical code section associated with the critical variable.

- For a given critical variable *m*, if the instruction *I* has been identified as a critical instruction, find the last write to m before *I* (I_W) and the first read to *m* after *I* (l_R). This is done by introducing some equations within SymPLAID to keep track of all read and write from and to a specific memory location. If the real critical section of *m* is $[I_0,I_m]$, we are sure that $[I_w, I_R[$ is included into $[I_0,I_m]$.
- Add an encoding instruction before I_w and a decoding instruction before I_R.
- Re-launch SymPLAID on the instrumented code until some final states with corrupted outputs are found. Then, add/replace instrumentation accordingly.
- Iterate this process until no more final states with corrupted outputs are found by SymPLAID.

Using this approach, the analysis and instrumentation time of the openSSH application have been reduced by a factor of 4 (from 4 days to 1 day). The fully instrumented application was then validated in SymPLAID. (Even after 4 days of analysis, no solution has been found.)

6.3.2 Technique 2: Reducing the number of injections needed

Technique 2 is based on a theoretical proof on how to reduce the number of injections. The following property is derived from Lemma 1 which we proved in section 3.2.2: all critical sections start with a write to the secure memory and finish right before a use of the secure memory.

6.3.2.1 Lemma and proof

Lemma 3:

Part 1: Consider an execution flow where I_{wa} is a write to memory location *a*. If I_{wa} is not in a critical section associated with *a* (meaning that corrupting *a* after the execution of I_{wa} does

not allow the attacker to achieve his/her goal) then all instructions executed after I_{wa} and before the next write to a are not part of any critical section associated with a for this execution flow.

Part 2: If I_{wa} is in a critical section associated with a, the following is true. If $\{I_{ra1}, I_{ra2}, \ldots I_{ran}\}$ denotes the set of all read instructions to memory location a executed after I_{wa} and such that changing the data in a right before I_{rai}, $i \in [0, n]$ allows the attacker to achieve his/her goal, then all instructions executed between I_{wa} and any of these I_{rai}, $i \in [0, n]$ are part of a critical section associated with a.

Proof: The proof is quite straightforward and derives from the previously demonstrated lemmas.

Hypothesis:

- Let I_{wa} be a write to memory location a.
- First suppose that I_{wa} is not in a critical section associated with a.

Three cases can happen:

1. There is no more write to a after I_{wa}.
2. The set of instructions executed between I_{wa} and the following write to a is empty.
3. The set of instructions executed between I_{wa} and the following write to a is not empty.

- Case 1: Suppose that there is no more write to the memory location a after I_{wa}. Let I be an instruction executed after I_{wa}. It can be proved that I cannot be in the critical section of a. If I is in a critical section associated with a, then the first instruction in the critical section is a write to a and it is the only one in the critical section. The most recent write to a is I_{wa}; therefore if I is in a critical section associated with a, then so is

I_{wa} which is not possible, as we assumed that I_{wa} is not part of any critical section associated with a. That proves that any instructions executed after I_{wa} are not part of any critical sections associated with a. In order words, after I_{wa} there is no way an attacker can corrupt a and achieve his goal (under the supposed attack model).

- Case 2: If the set of instructions executed between I_{wa} and the following write to a is empty, then there is nothing to prove as no instruction between I_{wa} and the following write to a can be in a critical section associated with a.
- Case 3: If the set of instructions executed between I_{wa} and the following write to a is not empty. Let us choose an instruction I in this set. Suppose that I is part of a critical section associated with a; then again, by the lemma demonstrated in paragraph 3.2.2, this critical section starts with a write to a which is also the only write to a in the critical section. However, the latest write to a was I_{wa} and I_{wa} is not part of any critical section associated with a. Therefore I cannot be in a critical section associated with a.

This concludes the proof of the first part of lemma 3: If I_{wa} is not in a critical section associated with a memory location a, then all instructions executed after I_{wa} and before the next write to a are not part of any critical section associated with a for this execution flow.

The second part of the lemma is straightforward to demonstrate and comes directly from the second lemma in section 3.2.2.

6.3.2.2 Optimized algorithm for identifying critical variables

Based on Lemma 3, we design the algorithm presented in *Figure 11* to find critical variables.

Figure 11: New algorithm to identify critical variables and critical code sections

```
For each possible execution flow (e.g. for different inputs): {
        - Find the set of all writes to memory W= {I_w1,...I_wn} within the application.
        For each I_wi in W {
                - Let a_i be the memory locations where I_wi writes to.
                - Set a breakpoint after I_wi
                - Inject a symbolic error into a_i and propagate the symbolic error using
                SymPLAID
                If ( no attack is reported by SymPLAID) {
                        - Go to the next write instruction.
                 }
                 Else {
                        - Find the set of all reads to a_i Ri = {I_R1,...I_Rm} which are executed
                        after I_w and before the next read to a_i.
                        - Let's suppose that the set is ordered such that I_Rj is executed before
                        I_Rk if j < k.
                        - New critical section for a_i = [I_wi, I_R1]
                        For all I_Rj in R_i {
                             - Set a breakpoint before I_Rj and inject a symbolic error into a_i
                             and propagate the symbolic error using SymPLAID.
                              If (no attack is reported by SymPLAID) {
                                    - Set of critical section for a_i = Set of critical section for a_i
                                    U New critical section for a_i.
                                    - Break and go to the next write instruction
                              }
                              Else { - New critical section for ai= [I_wi, I_Rj] }
                         }
                 }
          }
}
```

Notations:

- Number of instructions executed: Inst
- Average number of valid memory locations at each instructions: MemLoc
- Number of write instructions executed: WriteInst
- Number of read instructions executed: ReadInst

Using this algorithm we reduce the number of injections needed for every execution flow from approximately ***Inst * MemLoc*** injections to ***WriteInst + ReadInst*** injections

In the case of OpenSSH, we can reduce the number of injections needed from about 473 * 50 ~ 23,650 injections to about 300 injections, which is a significant improvement.[11] However, this technique involves finding all reads and writes to memory along with their relative execution order, which can usually be done using compiler analysis.

In conclusion, we have presented in this chapter two optimization techniques. The first technique, which is a practical technique applied on the *openSSH* application, achieved a 4x speedup. The second technique allows reduction by a factor of 100 (in the case of *openSSH*) in the number of symbolic fault injections required to achieve an exhaustive application analysis.

[11] We approximated the number of instructions executed during a given execution flow by the total number of instructions in the stub. We have also approximated the number of write instructions and read instructions to the total number of read/write instructions. Given that not all instructions are executed during a given execution flow, this is an over approximation. We also approximated the number of valid memory addresses after each execution by an average number of memory addresses used in the stub. The order of magnitude is, however, relevant.

Chapter 7: Discussion of Other Attack Models

In the previous chapters, we considered a specific attack model: namely, the attacker has the ability to modify a single memory word anytime during the execution of the application. This chapter discusses attacks that cannot be detected by the proposed detection mechanisms. In particular, control flow attacks (attacks where the execution flow of an application is diverted by the attacker) are classified and conditions under which such attacks happen are discussed.

7.1 Control flow attacks

Control flow attack can either change the flow of a given application from one legal flow to another legal flow or from a legal flow to an illegal flow.

7.1.1 Diverting the control flow to another legal path

First let us consider attacks which divert the control flow of the application from a legal path to another legal path.[12]

As an example, this simple fragment of code (extracted and simplified from the *openSSH* application) can be considered:

```
If (strlen(storedPassword)==0) Return authenticated;
else Continue authentication process…
```

[12] A legal path is a path that can be statically determined as possible under normal execution (without attack).

The fragment verifies whether there is a password associated with a given user. If there is no password associated with the user, then the user gets authenticated without having to give a password. At the assembly level the fragment translates into the following instructions (written in SymPLAID's format):

```
… //store arguments in registers
balr $(ret) (strlenAddr)   --- call strlen
 bneri $(2) $(0) (L34-0)  --- if the string's length is not 0 jump to label L34-0 otherwise continue execution
movi $(2) #(1)     --- put 1 in return register
beqii $(0) #(0) (L31-0)   --- jump to function's return code at label L31-0
L34-0 : … --- continue the execution process.
L31-0 : … ---function return code
```

A valid attack would be the following: after the call to *strlen*, the attacker changes the value in *register 2* to 0. Therefore, the stored *password's length* would be 0 and the attacker does not need to provide any password to get authenticated. A second attack could be the following: the *strlen* function is corrupted by an insider to return a wrong value. These attacks are very difficult to detect but also very difficult to execute. Indeed, in order to be able to divert the control flow to another valid control flow, the attack must happen at a branch instruction. In the example above, the attacks can be detected either by protecting the data in register 2 (or more generally return values) or, in the case of the second attack, checking libraries to ensure their integrity.

7.1.2 Diverting the control flow to an illegal path

Now let us consider attacks which divert the control flow of an application to an illegal path.

Changing the control flow of an application to an illegal path can be motivated by two goals: either the attacker wants to execute new code or he/she wants to skip the execution of existing instructions.

Executing "new code" can be done, for example, by code injection (e.g. buffer overflow exploit to launch a root shell) or by exploiting architectural dependent vulnerabilities (e.g. in x86 Linux, it is possible to execute system calls by jumping in the middle of existing opcodes [21], [22]) or by using a return-to-libc attack [23]. Doing so, however, will most probably crash the application [22] while the goal is to use the newly introduced code to corrupt the execution's environment (e.g. privilege escalation). Therefore we do not consider this kind of attack in this study as we want to focus on attacks which corrupt the behavior of the compromised application rather than crashing the application and causing denial of services.

Bypassing instructions can be done by modifying registers and memory locations to change the behavior of the attacked application. Two different goals can motivate an attacker to bypass instructions: either the attacker wants to bypass checking mechanisms (or instructions added to guarantee security and reliability properties) or he/she wants to bypass some essential functionalities within the application (e.g. escape the comparison of the stored password with the user provided password). Bypassing the checking mechanism in itself cannot achieve the attacker's goal. Indeed, in the case of the protection mechanism described in previous chapters, it is not sufficient to bypass the decoding instructions; the attacker also needs to corrupt something else to alter the behavior of the application. We argue that bypassing computational elements is weaker than modifying directly registers and memory locations. Indeed, bypassing computational elements and not crashing the application relies on the fact that registers and memory locations not assigned because of the attack, already

contain the desired value. For example, bypassing the comparison of the stored and the user-given passwords relies on the fact the result register/flag already contains the desired value. We argue that exhaustive fault injections into registers and memory locations can model the effect of such control flow attacks on the application's behavior. The following code segment illustrates the idea.

```
0:  movi $(2) #(0) --- put 0 in register 2
1:  sw $(2) 36$(30)  --- stores the value of register 2 in memory
2:  movi $(2) #(1) --- put 1 in register 2
3:  …
```

Consider the following control flow attack. The attacker skips *instruction 2* to force *register 2* to contain the value 0 instead of 1 before the execution *instruction 3*. By injecting the value *0* in *register 2* after the execution of the *instruction 2* it is possible to model the effect of such control flow attacks. If the application is not affected by such an injection, or if the application crashes, then we know that the control flow attack cannot succeed either.

Chapter 8: Related Works

Many studies propose protection mechanisms against memory attacks. We can broadly classify them into two categories: those which ensure that vulnerabilities are patched or cannot be exploited and those which aim to guarantee a correct behavior of the protected application or system.

8.1 Preventing vulnerabilities from being exploited

1. The use of canary value has been proposed by [24]. The authors propose to insert a canary value or a canary word at the end of each buffer on the stack. If a buffer overflow occurs, the canary value will be the first word to be overwritten and a failed verification of the canary value will detect such attacks. However, this technique can only protect against stack smashing attacks.

2. Libsafe [8] proposes to intercept, at runtime, calls to known vulnerable C library functions (e.g. strcpy, strcat, getpw, gets) and replaces them with safer functions which do not allow buffer overflow. However, the technique only protects a very limited set of functions.

3. Address space randomization techniques [25], [26], [27], [28] have been proposed to obfuscate the layout of the memory space, preventing attackers from identifying control data or critical data in memory. However, [23] claims that the randomization can be broken by repeated, undetected attacks on the system.

8.2 Enforcing an application or system's correct behavior

1. Program Shepherding [29] restricts the amount of executable code and control transfers (branches, call to library functions) based on a security policy. The idea is to prevent

attackers from executing injected code or to change the control flow of a highly privileged application in order to gain control of a system.

2. [21] proposes to enforce at runtime the control flow integrity of an application. The idea is to ensure that the software execution follows a path of a control flow graph determined ahead of time (source code, binary analysis or execution profiling).

Both techniques (Program Shepherding and Control Flow Integrity) aim to ensure that control flows cannot be diverted at runtime by attackers. However, they do not consider non-control data corruption attacks. [30] demonstrates that non-control data corruption is a serious threat and can be used to take control of an application. [31] shows that non-control data corruption attacks are realistic and practical on real world applications.

3. Information Flow Signature [32] guarantees the integrity of critical data by insuring the control flow and the data flow integrity within each critical data's backward slice. The technique, however, relies on users to identify critical data.

4. WIT [7] uses points-to analysis at compile time to compute the control-flow graph and the set of objects that can be written by each instruction in the program. It then enforces at run time to prevent instructions from modifying data that are not in the statically computed set. However, as the technique is not selective, it involves tagging all memory data and all write instructions. In the technique proposed in this thesis, we only focus on critical data identified by formal verification.

The detectors proposed in this study fit in this second category. The goal is to ensure that formally identified critical memory locations cannot be corrupted by attackers during the critical code sections. However, by designing detectors based on formal analysis, this work also provides strong guarantees about the detection mechanism's coverage.

Chapter 9: Conclusion and Future Work

9.1 Conclusion

This thesis has shown that it is possible to leverage existing formal tools to design and verify protection mechanisms which can achieve a very high coverage. More precisely, we evaluated an existing formal tool (SymPLAID) and used it to find critical variables and their associated critical code sections. From the critical variables and the properties of the critical sections, we designed a detection mechanism with high coverage. We implemented the designed detection mechanism within SymPLAID itself, instrumented two applications, namely the OpenSSH authentication stub and WuFTP logging stub, and proved that given the attack model we assumed, our detection mechanism protects all critical memory locations identified by SymPLAID. Furthermore, we proved several properties about the critical sections that allow us to scale the analysis process.

9.2 Future work

9.2.1 Scalability of SymPLAID

Two techniques to scale the analysis time of SymPLAID for single memory address injections are presented in this thesis. These techniques offer ways of reducing computation time while still providing exhaustive analysis. Note that the techniques described in this work do not improve the scalability of the tool itself. The first technique relies on properties of the critical code sections as well as on the proposed detectors to reduce the number of reachable states. The second technique leverages the critical code sections' properties to reduce the number of symbolic fault injections needed.

It is not clear that the techniques proposed in this work can be directly applied to improve the scalability of symbolic fault injections into registers, the program counter or

several memory locations. For example, as the program counter is updated and read at each instruction, both techniques presented in this work will be unable to efficiently scale an exhaustive fault injection into the program counter. In this case, the large analysis time is mostly due to the amount of reachable states introduced by one single error in the program counter. Indeed, if the program counter contains an *err* symbol, any instruction within the application's text segment can be fetched as the next instruction. That creates as many paths as the number of instructions in the application for the model checker to explore.

Model abstraction is widely used to address the scalability problem in formal verification [33]**.** The idea is to verify a less detailed model of the system to avoid the state explosion problem. It could be interesting explore how one can further abstract the machine model in order to keep the accuracy of the analysis and reduce the state search space.

It could also be interesting to see if there is a way of breaking the analysis of a large program into analysis of smaller pieces.

9.2.2 Studying new attack models within the framework

In this thesis, only the attack model of a single memory corruption per attack is considered. It would be interesting to consider an attack model where register values can be corrupted. It is not clear, however, how one could map those direct register corruptions to higher level attacks. To the best of our knowledge, attackers generally have to go through corrupting memory in order to corrupt register values (e.g. when registers are dumped into memory when a function call happens).

It would also be interesting to consider attack models where the attacker can introduce several errors. That would require some changes within SymPLAID, in particular in the constraint solver in order to allow several symbolic errors to propagate.

9.2.3 Adapting SymPLAID for x86

Currently, SymPLAID can only model the execution of MIPS applications. However, as most of the attacks and applications run on x86, it would be interesting to model the x86 instruction set and machine model to test whether the technique applied in this thesis can be transferred to applications running on x86.

9.2.4 Building detectors

Currently the detectors designed in this study have not been implemented either in software or hardware. A software approach would have to implement a "trusted memory section," which could be done, for example, by encrypting duplicated data. The detectors can be implemented in hardware as a new security module within the RSE framework [34] developed in our research group. The module would duplicate, store and check the integrity of data stored in critical variables.

Once they are implemented, we will be able to evaluate the performance of the proposed detectors.

References

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

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

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

[4] J. Afek and A. Sharabani, "Dangling Pointer: Smashing the pointer for fun and profit," Watchfire White Paper, Aug. 2007.

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

[6] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. Iyer, "Defeating memory corruption attacks via pointer taintedness detection," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, 2005, pp. 378-387.

[7] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008, pp. 263-277.

[8] T. K. Tsai and N. Singh, "Libsafe: Protecting critical elements of stacks," Avaya Labs, Tech. Rep., ALR-2001-019, 2001.

[9] K. Pattabiraman, Z. Kalbarczyk, and R. k. Iyer, "Discovering application-level insider attacks using symbolic execution," in *Proceedings of the IFIP International Conference on Information Security (SEC)*, 2009, pp. 63-75.

[10] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer, "Principles of Maude," *Electronic Notes in Theoretical Computer Science*, vol. 4, pp. 65-89, 1996.

[11] N. Marti-Oliet and J. Meseguer, "Rewriting logic as a logical and semantic framework," in *Electronic Notes in Theoretical Computer Science*, vol. 4, J. Meseguer, Ed. Elsevier Science Publishers, 2000.

[12] K. S. Lhee and S. J. Chapin, "Buffer overflow and format string overflow vulnerabilities," *Softw. Pract. Exper.*, vol. 33, no. 5, pp. 423-460, 2003.

[13] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. K. Iyer, "SymPLFIED: Symbolic Program-Level Fault Injection and Error Detection Framework," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, Anchorage, AK, June, 2008., pp. 472-481.

[14] M. Clavel et al., "Functional modules," in *All about Maude - A High-Performance Logical Framework*, S. B. Heidelberg, Ed. 2007, Springer, ch. 4, pp. 61-118.

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

[16] Trusted-ILLIAC. Information Trust Institute, University of Illinois at Urbana-Champaign. [Online]. http://www.iti.illinois.edu/content/trusted-illiac

[17] SimpleScalar LLC. (2004) [Online]. http://www.simplescalar.com

[18] WuFTP Server. [Online]. http://www.wu-ftpd.org

[19] OpenSSH. (2009) [Online]. http://www.openssh.com

[20] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. K. Iyer, "Discovering application-level insider attacks using symbolic execution," University of Illinois at Urbana-Champaign, Tech. Rep. UILU-ENG-09-2201.

[21] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, Alexandria, VA, USA, 2005, pp. 340-353.

[22] W. Gu, Z. Kalbarczyk, and R. K. Iyer, "Error sensitivity of the Linux kernel executing on PowerPC G4 and Pentium 4 processors," in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, 2004, p. 887.

[23] H. Shacham et al., "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2004, pp. 298-307.

[24] C. Cowan et al., "Protecting systems from stack smashing attacks with StackGuard," in *Proceedings of Linux Expo*, Raleign, North Carolina, U.S.A, 1999, [n.p.].

[25] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, 1997, p. 67.

[26] J. Xu, Z. Kalbarczyk, and R. K. Iyer, "Transparent runtime randomization for security," in *IEEE Symposium on Reliable Distributed Systems*, p. 260, 2003.

[27] S. Bhatkar, D. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *Proceedings of the 12th USENIX Security Symposium*, Washington,DC,USA, Aug 2003, pp. 105-120.

[28] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic memory saftey for unsafe languages.," in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Ottawa, Ontario, Canada, 2006, pp. 158-168.

[29] V. Kiriansly, D. Bruening, and S. Amarashinghe, "Secure execution via program shepherding," in *Proceedings of the 11th USENIX Security Symposium*, 2002, pp. 191-206.

[30] W. Young and J. McHugh, "Coding for a believable specification to implementation mapping," in *IEEE Symposium on Security and Privacy*, Los Alamitos, CA, USA, 2005, p. 140.

[31] S. Chen, J. Xu, E. Sezer, P. Gauriar, and R. K. Iyer, "Non-control hijacking attacks are realistic threats" in in *Proceedings of the 14th USENIX Security*, 2005, pp. 177-192.

[32] K. Pattabiraman, W. Healey, F. Yuan, Z. Kalbarczyk, and R. K. Iyer, "Insider attack detection by information-flow signature enforcement," Coordinated Science Laboratory, University of Illinois at Urbana-Champaign , Tech. Rep., 2009.

[33] Y. W. Hsieh and S. P. Levitan, "Model abstraction for formal verification," in *Proceedings of the Conference on Design, Automation and Test in Europe*, Le Palais des Congrès de Paris, France, 1998, pp. 140-147.

[34] N. Nakka, Z. Kalbarczyk, R. K. Iyer, and X. J, "An architectural framework for providing reliability and security support," in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, 2004, p. 585.