Modified SPARC Instruction Simulator (SIS) to Support Experimental Tagging Architectures

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Saeede Zakeri

Augest 2014

Major Professor: Jim Alves-Foss, Ph.D.

Authorization to Submit Thesis

This thesis of Saeede Zakeri, submitted for the degree of Master of Science with a Major in Computer Science and titled "**Modified SPARC Instruction Simulator (SIS) to Support Experimental Tagging,**" has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor _____Date_____
Dr. Jim Alves-Foss

Committee Members _____Date_____
Dr. Robert Rinker

_____Date_____
Dr. Daniel Conte de Leon

Computer Science Department Administrator _____Date_____
Dr. Gregory Donohoe

Discipline's College Dean, College of Engineering _____Date_____
Dr. Larry Stauffer

Final Approval and Acceptance by the College of Graduate Studies

_____Date_____
Dr. Jie Chen

Abstract

This thesis is a part of an Air Force Research Laboratory (AFRL) project focused on developing a framework based on existing security tagging techniques. These techniques are developed to prevent or lower the overhead of known security vulnerabilities. Critical to prevent these security vulnerabilities is choosing techniques that target the most disruptive attacks such as buffer overflow, out of bound memory accesses and uninitialized memory access. For instance, over the last five years, buffer overflow vulnerability was the cause of non-deterministic failures and security breaches. The other main vulnerability is insufficient validated user inputs, which lead to dangerous security flaws such as format string, SQL command injection and path traversal. The flaws are exploitable when user input is passed to vulnerable programs without sufficient validation. Buffer overflows and Insufficient validated user inputs are among the top 25 software errors according to the CWE/SANS list of “Top 25 Most Dangerous Software Errors”.

This study presents research focused on evaluating hardware implementation of security tagging techniques. Run time hardware based taint tracking is an effective technique which controls data propagation during execution of an application. These techniques associate security tags with user provided data and track tags during program execution. To keep tags updated, the techniques intercept every attempt to access application and process data. A detection of a misuse of data will result in a security exception. Each technique has a different approach for intercepting and isolating instructions to prevent possibly difficult classes of vulnerabilities. The focus of this research is to understand the nature of security tagging as a foundation for developing a framework for simulating the different security tagging techniques to determine how well they can detect software flaws and vulnerable programs with minimum overhead. An additional objective of this work is to compare the efficiency of different proposed tagging technique. The AFRL project uses ERC32 (radiation-tolerant 32-bit RISC Processor) which is a SPARC variant based computer systems. Accordingly a SPARC Instruction Simulator is adopted to develop security techniques.

Acknowledgements

This project would not have been possible without support of many people. I would like to express my gratitude to my advisor, Dr. Jim Alves-Foss, whose expertise, understanding, and patience, added considerably to my graduate experience. I would like to thank the other members of my committee, Dr. Robert Rinker and Dr. Conte de Leon, for agreeing to serve on my committee.

I would like to thank the department chair, Dr. Gregory Donohoe, and Mrs. Darby Baldwin, Mrs. Rhonda Zenner, Ms. Arvilla Allen, and other staff members in the department of Computer Science for their help during my study in the department. I would like to thank all the staff of the College of Graduate Studies for their help and support throughout my study at UI.

I wish to acknowledge the United States Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA), for supporting me during the course of my graduate studies through grant number FA8750-11-2-0047.

I am indebted to all of the team members who worked with me in the past and present. I would like to thank Cindy Song, Stu Steiner and Abhay Patil for their input at various stages of my research.

Last, but certainly not least, I would also like to thank my family for the support they provided me through my entire life and in particular, I must acknowledge my husband and best friend, Mahdi, without whose love, encouragement and editing assistance, I would not have finished this thesis.

Dedication

This thesis is dedicated to my husband, Mahdi Salavatian. You are the love of my life, my strength and support. I also want to dedicate this to my amazing parents and beloved grandma.

Table of Contents

Table of Figures

Table of Tables

Chapter 1. Introduction

Computer security has become a very important economic and social problem. There has been a lot of research conducted to develop new ways to protect systems over the past three decades. This research has been conducted with the goal of preventing the ever-growing catastrophic effects of security vulnerabilities. According to estimates, cyber security attacks directly cost US companies tens of billions of dollars a year and much more in indirect cost to companies and individuals. "General Keith Alexander, Chief of the U.S. Cyber Command and Director of the National Security Agency, points out that the United States saw a 17-fold increase in cyber attacks between 2009 and 2011." [1].

Every year new security vulnerabilities and attacks emerge. Today almost everything relies on worldwide network communications, so having vulnerable code will cause serious worldwide impacts and losses of billions of dollars. In the past two years several companies and government organizations have been victims of hacker's attacks. Some of these attacks exploited software vulnerabilities which caused buffer over flow and out of bound memory accesses. Other attacks exploited vulnerabilities at a high level such as SQL (Structured Query Language) injection, command injection, CSS (Cross Site Scripting) and so on. These vulnerabilities are further discussed in section 1.1.

As an example of vulnerable code we can mention a bug called Heartbleed which was publicly announced on April 1st, 2014. This bug is a good sample of how a vulnerable code can have a worldwide affect. Heartbleed is a security bug in Open Secure Socket Layer (SSL) library. This bug can be exploited if either client or server uses a vulnerable OpenSSL instance. National Cyber security and Communications Integration Center (NCCIC) states that using Heartbleed vulnerability, attackers can decrypt previously encrypted information and stole servers' private keys and users session cookies and passwords [2].

Over half a million secure web servers were vulnerable to this bug. Several groups called the Heartbleed bug "catastrophic". Forbes cyber security columnist Joseph Steinberg wrote, "Some might argue that [Heartbleed] is the worst vulnerability found since commercial traffic began to flow on the Internet." [3] .

There are tools such as firewalls and anti-viruses that can be used to reduce the damage caused by Heartbleed types of attacks. But research shows that the security tools do not provide a reliable protection against ever-increasing attacks and viruses that exploit low level programming errors.

One approach to enhance security is to use hardware-bases security tagging techniques to cope with the security vulnerabilities. These techniques usually deploy security tags to support memory access control. In this thesis we evaluate the use of security tags associated with data to reduce the damages of vulnerable code.

This current Research is a part of an Air Force Research Laboratory Project focused on the framework development of different security tagging techniques inside SPARC Instruction Simulator. The goal of this thesis is to develop and implement a framework to test and evaluate hardware-based security tagging techniques. In this chapter section 1.1 introduces some of the common problem areas and basic concepts of security tagging and security tagging schemes. Section 1.2 introduces the motivation and objectives of this research. Section 1.3 concludes with an overview of the remaining parts of the thesis.

1.1. Problem Area

Table 1-1 lists the 2011 Common Weakness Enumeration/SysAdmin, Audit, Network, Security (CWE/SANS) top 25 most dangerous software weaknesses [4] .

Rank	ID	Category	Name
1	CWE-89	1	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
2	CWE-78	1	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
3	CWE-120	2	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
4	CWE-79	1	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
5	CWE-306	3	Missing Authentication for Critical Function
6	CWE-862	3	Missing Authorization
7	CWE-798	3	Use of Hard-coded Credentials
8	CWE-311	3	Missing Encryption of Sensitive Data
9	CWE-434	1	Unrestricted Upload of File with Dangerous Type
10	CWE-807	3	Reliance on Untrusted Inputs in a Security Decision
11	CWE-250	3	Execution with Unnecessary Privileges
12	CWE-352	1	Cross-Site Request Forgery (CSRF)
13	CWE-22	2	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
14	CWE-494	2	Download of Code Without Integrity Check
15	CWE863	3	Incorrect Authorization
16	CWE-839	2	Inclusion of Functionality from Untrusted Control Sphere
17	CWE-732	3	Incorrect Permission Assignment for Critical Resource
18	CWE-676	2	Use of Potentially Dangerous Function
19	CWE-327	3	Use of a Broken or Risky Cryptographic Algorithm
20	CWE-131	2	Incorrect Calculation of Buffer Size
21	CWE-307	3	Improper Restriction of Excessive Authentication Attempts
22	CWE-601	1	URL Redirection to Untrusted Site ('Open Redirect')
23	CWE-134	2	Uncontrolled Format String
24	CWE-190	2	Integer Overflow or Wraparound
25	CWE-759	3	Use of a One-Way Hash without a Salt

Table 1-1: 2011 CWE/SANS top 25 most dangerous software errors

These weaknesses are divided into three categories:

1- insecure interaction between components

2- risky resource management

3- porous defenses

Each category as well as its relevant weakness and rankings are shown in Table 1-1.

The weaknesses in the insecure Interaction between components category 1, are caused by improper data exchange between systems, programs or processes. As we can see in Table 1-1, this category largely deals with SQL injection attacks and CSS attacks.

The second category weaknesses are caused by the improper handling of systems resources. This category includes different buffer overflow attacks, directory traversal, format strings and so on. The third and last category of these weaknesses mostly deals with misused defensive techniques [4].

As we can see in Table 1-1, the first category of attacks exploits vulnerabilities at a high level such as Structured Query Language (SQL) injection, command injection, Cross Site Scripting, etc. These vulnerabilities allow malicious users to launch attacks by executing arbitrary code or stealing sensitive data. SQL injection is a technique for exploiting web applications which ask for user's data in SQL queries. Web applications provide the ability for users to store and retrieve information to and from databases over the internet. This information includes user's credentials, bank account information and so on. SQL injection attacks can send SQL commands to the back end database through web application. The attacker can then view, alter or remove user's data through these SQL commands.

Weaknesses in the second and third categories in Table 1-1 are related to buffer overflow attacks, out of bound memory accesses and so on. These categories are also dangerous and harmful. The newly emerged Heartbleed vulnerability can be classified in the out-of-bounds memory access attack group. The vulnerable code for this attack is shown in Figure 1-1 [5].

```
    C Code:
    if (hbtype == TLS1_HB_REQUEST)
    unsigned char *buffer, *bp;
    int r;
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
    bp = buffer;
    /* Enter response type, length and copy payload */
    *bp++ = TLS1_HB_RESPONSE;
     s2n(payload, bp);

    memcpy(bp, pl, payload);
     r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payloa
d + padding);
```

Figure 1-1: Heartbleed code vulnerability

In the SSL Heartbeat protocol one of the SSL users sends a request to the other with a payload of data and a size. The end user is supposed to copy the payload and send it back. However the size of the payload specified in the message is never checked. Since attackers can lie, they can request a return of a large amount of data and the end user will comply, copying data from main memory including possible encryption keys and user passwords. This is an example of category 2 vulnerabilities that could cause at least 500 Million dollars [6].

Companies use defensive and preventive measures to prevent or lower the cost of corruption or theft of their information. Defensive measures include firewalls, cryptography, Intrusion detection Systems and antivirus programs. Preventive measures include penetration testing, authentication and verification.

The problem in most of the mentioned defensive measures is that they usually take a look at the symptoms of the attack rather than its source; we know that it's almost impossible to write perfectly secure code. Some hardware developers have come up with the idea of using hardware based protection to prevent these attacks or if not preventing them, at least to stopping the attacks from going farther. Among the proposed techniques, security tagging schemes exist to prevent attacks by adding security tags to the data. One example of security tagging schemes is Dynamic Information Flow Tracking (DIFT).

The DIFT technique was developed to prevent buffer overflow and format string attacks. DIFT was proposed to prevent the majority of attacks that change the flow of programs in order to gain control or unauthorized access. It prevents attacks by associating security tags to data, marking data as malicious and tracking it as it goes through the system. Malicious data is defined as any data that comes from malicious I/O. By assigning a security tag, the DIFT security engine can track data that attempt to transfer control of the system. Based on the implementation dependent rules in the DIFT tag engine, if tainted data is used in a way that it changes the control flow of the program or logic of the program, the tag engine hardware triggers an exception and pops back to the operating system. DIFT will be described in further details in Chapter 2 [7].

Several techniques have been developed based on hardware protection. However most of the existing techniques use simple applications which are tested in small scale.

The designed tests only show that the detection occurs. The tests do some simulations to determine performance, overheads, but most of the simulation and experimentation is done at the register transfer level which is really slow. Accordingly, the developed techniques have never been evaluated in full experiments. Having the capability of comparing and contrasting the logic, security behavior and functionality of the techniques can fulfill this shortcoming. This capability can be achieved by running known vulnerable applications through multiple security techniques. We accomplished this by simulating the techniques in an instruction simulator.

1.2. Research Objectives

The purpose of this thesis is to develop a framework, in which we can plug-in different hardware tagging schemes. The framework is used to evaluate the effectiveness and functionality of tagging schemes. The proposed framework has been used to implement three different tagging schemes: Dynamic Information Flow Tracking (DIFT), memory Bound Checking (BC) and Uninitialized Memory Checking (UMC). As described before, the DIFT technique is developed to associate data with security tags. In this simulated the DIFT tagging engine. The engine will be responsible for moving and checking tags through the execution of each instruction. The engine also includes an exception handler in the case of security exception.

The UMC technique targets uninitialized memory. This technique intercepts each memory access to check if memory is initialized during loading a value from memory. UMC associates each memory word with a tag. Memory tags are first all initialized to zero, indicating that memory block is not initialized. Then during each memory store instruction, memory tags for the corresponding memory location will be set. During loading a value from memory, the tag value will be checked to prevent uninitialized access. This technique is described in further detail in Chapter 6 [8].

The BC technique has characteristics of both DIFT and UMC. Each memory word and processor register is associated with a tag. These tags get propagated based on the BC tag engine rules. During memory access, tags are checked to see if the tag value falls in the memory boundary range. If not, the BC tag engine will cause security exception. We will describe this technique in further detail in Chapter 5 [9].

The objectives of this thesis are as follow:

- **Objective 1:** Introduce DIFT, UMC and BC security techniques. To be able to implement these techniques, we need to look in depth in each of these techniques. We will give an extensive background on each of these techniques at the instruction level in Chapter 2.
- **Objective 2:** Apply these techniques to SPARC (Scalable Processor Architecture) using SIS (SPARC Instruction Simulator). In this objective we group SPARC instructions to different categories for each technique. We then implement tagging rules for each technique. We also introduce new instructions to control the tag engine and manipulate tags for registers and memory for each technique. Having the new instructions gives us more control over the tag engines.
- **Objective 3:** Develop a framework for testing different security tagging techniques in the instruction level hardware simulator SIS. After implementing each technique we develop test cases to test techniques and compare the results.

Implementing all three techniques in one framework makes it easy to compare each technique to another. This framework helps estimate specific features and the upsides and downsides of each technique.

1.3. Thesis Overview

This thesis is organized as follow. Chapter 2 provides a background and survey on DIFT, UMC and BC security tagging schemes as well as a overview of how to add these techniques to the framework. Chapter 3 gives details about the SPARC architecture and SIS. In Chapter 4 we describe DIFT technique implementation in SIS. In Chapter 5 we describe BC implementation in SIS and in Chapter 6 we describe UMC implementation in SIS. We discuss different experiments, tests, results and evaluation of each technique in Chapter 7. Then we give the conclusion and future work in Chapter 8.

Chapter 2. Background and Framework Overview

In this Chapter we present an extensive background for DIFT, BC and UMC security tagging techniques. We then describe the whole design idea of the framework and how we can integrate each technique to the simulator. We introduce each technique by describing how it works at the source code level. Each technique is capable of detecting specific sort of security attacks. For instance DIFT is capable of detecting buffer overflow attacks. UMC is capable of detecting uninitialized memory accesses. BC is capable of detecting out of bound memory and illegal memory accesses. Having these techniques implemented in a framework, we will be able to detect and prevent vast majority of security vulnerabilities.

2.1. Dynamic Information Flow Tracking Background

DIFT, designed by Suh et al. [7], is a hardware security tagging technique which assigns security tags to data and checks the data manipulation in instructions to restrict the use of untrusted input. DIFT was developed to prevent a vast class of software security vulnerabilities such as buffer overflows. DIFT prevents security attacks by identifying and restricting the malicious information flow at runtime. DIFT implements a security tagging engine which includes the logic for book keeping and checking of security tags.

To take control of a program, attackers modify the contents of memory in a vulnerable program space with either malicious code or a pointer to a malicious code. To modify content of memory, they need to insert a value from input channels. DIFT marks untrusted input and output channels with a one bit tag. By tracking the untrusted data, it traps malicious manipulation of the data. In general, malicious use of the data can be defined as any disallowed manipulations of the data based on the permitted security policies. DIFT defines malicious use of data as using tagged data as a

jump or branch target address. DIFT security policy generates a trap in the case of this use. If the operation is not allowed based on the permitted security policy, the trap handler terminates the operation.

DIFT rules are defined to prevent executing of malicious code or transforming control of a program using malicious data. The key concern here is how to identify malicious code from legitimate code. The operating system makes the decision which means it initializes the untrusted channels as malicious data. Then operating system tracks malicious data.

2.1.1. DIFT initialization and propagation phase

To identify legitimate and malicious data, DIFT uses a one bit tag for each register in the processor and each byte of memory. Memory address and registers' tags are initialized to zero in the initialization phase of the tag engine. The operating systems tags data with the value one only if the data comes from a malicious input source.

The DIFT tag engine defines a set of rules to propagate tags through program execution. These rules can be categorized into three groups. The first group is defined as instructions which have either one or two operands and none of the operands are tagged. The result of running first group of instructions is an untagged value. The second group of instructions has at least one tagged operand. The result tag for the second group of instructions is a tagged value. Finally the third group of instructions has two operands which are both tagged. Running the instructions result in a tagged value.

The above three groups of propagation rules can apply to ALU, Load and Store instructions. Not all of the SPARC instructions propagate DIFT tags. We will describe in details how propagation rules are implemented for each set of SPARC instructions in Chapter 4.

2.1.2. DIFT checking phase

The DIFT tag engine defines two policies for instructions' tag checking. First policy tracks Load and Store instructions' tags. Loading from and storing, using an address in a register that is

marked as malicious is not allowed based on DIFT rules. Therefore every time load or store instructions get executed, the tag engine will intervene to check tags of source operands.

The second policy tracks Branch and Jump instructions. DIFT rules stops execution of the instruction which uses a tagged value as jump target address. These policies do not apply to other instructions such as ALU instructions. ALU instructions can be executed with having tagged operands. Recall that the tags are still propagated.

2.2. Memory Bound Checking technique Background

The main focus in the BC technique is on memory faults that occur by memory accesses through pointers. Illegal Memory Accesses (IMA) can be classified as memory faults which arise when a memory region is accessed with a pointer that is not initially assigned for that region. BC also covers vulnerabilities such as out of bound memory accesses [9].

The BC technique uses limited number of tags to associate with data. Tags can be 1-bit, 4-bits or 8-bits long. The BC technique associates tags with memory blocks and pointers. When the memory “m” is allocated, BC associate tags with “m”. Later when a pointer “p” is created, which points to “m” address, “p” is tainted with the same taint mark that is associated with the memory “m”. During execution of program, pointer tags get propagated. Finally when a memory region is accessed using a pointer, the BC technique checks both tags for memory and pointer in order to see if they match or not. The more bits BC uses, the more likely it will detect inappropriate use of pointer.

Memory taints never get propagated but pointer taint mark gets propagated as the program executes. However the BC tag engine separates memory locations’ tags from pointers’ tags. The process of implementing the BC technique breaks down into initialization, propagation and checking steps.

2.2.1. BC initialization phase

There are two type of memory allocation: static and dynamic. Static memory allocation can be referred to as defining global and local variables. In static memory allocation the technique identifies the memory location used for storing variables. Therefore it can initialize the taint marks for the specific memory locations. In dynamic memory allocation the technique identifies the amount of memory that should get the taint marks through communication with the allocation functions such as malloc. After that, the specific memory locations get the same fresh taint marks. No too adjacent allocated regions get the same taint mark.

In pointer initialization, pointers get the taint mark base on the memory location they point to. For example pointers that point to dynamically allocated memory intercept the function call to malloc and use the return value to get the same tag as the dynamically assigned memory. Since finding out the starting addresses of this type of memory is straight forward, initializing the taint marks for pointers that point to statically allocated memory is easier than dynamically allocated memory. In both cases after finding out the address for memory location, pointer and memory location will get the same taint mark.

2.2.2. BC propagation phase

The BC technique doesn't define rules for control instructions, so, data flow instructions are responsible for propagating tags in this technique. The BC technique treats tag propagation for memory and pointer differently. Memory location tags are never propagated. Memory location's tags get initialized at the beginning of program and cleared at the end of program or start and end of a function call for variables on the stack. During the deallocation of a memory location, this technique intercepts the function call to lower level memory-deallocation function to figure out the proper address in order to erase the taint marks. Also the associated pointer taint mark for this memory address should get erased. Otherwise it conveys a concept like a dangling pointer. Calling the deallocation function with an initial address as a parameter frees dynamically allocated memory, so

intercepting this function gives us the memory location whose taint mark should be erased. Returning from a function call will also deallocate statically allocated memory, so the intercepting function exits gives us the memory location whose taint mark should be erased.

Pointer taint marks are responsible for propagation rules during execution of the program. Every operation has a propagation rule defined for calculating the result of the taint mark based on the operands' taint marks. The rules for pointer tag propagation are developed based on patterns found in the software subjects and underlying machine languages. These rules are finely developed to remove cases that cause false negatives because of the underlying language and function implementation.

These rules handle different sets of instructions differently. For example, in addition and subtraction operations, if both operands are not tainted, then the result remains untainted. In some cases where just one of the operands is tainted, then the taint mark will propagate and the resulting taint mark gets the taint mark of the tainted operand. In cases that both operands are tainted, operations decide the propagation rule. For example during the addition operation, the result tag is the sum of the two operands' taint marks [9].

In operations like multiplication and division the result is never tainted. In bitwise AND operation if both operands are tainted or untainted the result is untainted. But if just one of the operands is tainted the result gets the taint mark of the tainted operand [9].

Propagation rules for bitwise NOT can be derived from addition and subtraction propagation rules. Accordingly the result gets the negative value of taint mark of the operand. In bitwise OR and XOR instruction the result is always untainted [9].

2.2.3. BC checking phase

The checking rules for BC technique intercept any memory access. If the taint mark of the memory location and its corresponding pointer location are the same, it is considered as a valid

memory access. Any scenario other than this is considered as an illegal memory access and it throws security exception.

Since the main goal in this technique is to implement the approach in hardware, the number of taint bits plays an important role on the performance overhead and design complexity of hardware. The problem with a small number of bits for the taint mark is that some memory locations have the same taint marks. So it causes an undetected IMA. The probability of detecting IMAs with having 4 bits of taint mark is 94% [9]. Also by using different strategies the probability of two memory locations having the same taint mark can be reduced. This is feasible by assigning different taint marks for adjacent regions of memory.

2.2.4. BC implementation at Hardware

When a technique is implemented in hardware, there is not enough information about statically allocated memory. So by reducing the precision of the technique, the technique taints all of the statically allocated region of the memory with same taint mark, Therefore the technique is not capable of detecting IMA's in local variables, but it still can detect IMA's between statically allocated variables and dynamically allocated memory.

2.3. Uninitialized Memory Checking Background

Reading from or writing to uninitialized memory is another type of error that causes the system to stop working or crash. Due to the nature of these errors, they are most likely to go undetected. Another type of error is a memory leak which happens because a block of memory has not been released. Memory leaks also produce errors that are hard to find and also harder to fix. The UMC technique helps find memory leaks and access errors such as reading from uninitialized memory or out of bound access of arrays. This technique helps find errors that happen at runtime [8].

The way this technique is implemented is that it will keep a tag for each memory address. It then intervenes each memory access and looks for the tag value of the corresponding memory address tag. Then the technique decides whether the access is authenticated or not.

UMC calls a specific function before any memory access function such as load and store. The implemented function for tags is responsible to make sure that memory address tag is the same as expected tag. This technique holds two bits representing state code for each byte in memory. This technique represents three different states that the memory can be in.

1. ***Unallocated state:***

   The unallocated state represents the unallocated memory. In this state memory bytes can neither be read from nor written to.

2. ***Allocated and initialized state:***

   The memory that is allocated and initialized is in this state. Each memory byte which is in this state is allowed to be written to and read from.

3. ***Allocated but uninitialized state:***

   In this state we are able to write to allocated memory, but we are not able to read from it.

2.3.1. UMC initialization phase

UMC keeps one bit tag for each byte of memory. The tag value shows the state of that memory location. These tags are initialized at the beginning of the program with tag value zero. Tag value zero indicates that the specific memory region is not initialized or allocated.

2.3.2. UMC propagation phase

Executing store instruction changes the memory state from either unallocated or allocated but not initialized state to the allocated and initialized state. During executing store instruction, UMC

technique sets the tag value of memory. This instruction is the only one that can propagate tags and set tags for memory regions.

2.3.3. UMC checking phase

The check function intervenes with every access to the variables through the load instruction and compares the status tag with the expected tag. Loading a value can only be performed from allocated and initialized state. If a memory location is in this state, it has tag value of one. If memory is in any other state, meaning that its tag is zero, UMC technique stops the execution of instruction and halt program.

2.4. Framework Overview

We have added a new module to the SIS simulator – the tagging engine. This module is designed to simulate the behavior of the tagging coprocessor. Within the SIS simulator we have added hooks to enable execution of the coprocessor. The first set of hooks defines the operations of CPOP instructions (coprocessor operations). The CPOP1 format instructions are used to control operation of the coprocessor (e.g., turn it on and off). The CPOP2 format instructions are used to manipulate the specific tags (e.g., set a tag of a register or memory location). In addition, we have a hook from the simulator that calls the `tag-dispatch-instruction()` function. This function is the framework used to specify the execution behavior of each instruction, or more specifically the execution behavior of the tag engine for each instruction. This function is called prior to actual execution of the instructions to ensure that the simulator simulates the throwing of a security exception prior to completion of the instruction. For each instruction, as illustrated in the following chapters, we implement specific tag checking and propagation rules.

To add techniques to the SIS simulator, we chose a tagging mechanism and defined tag formats for it. We determined the items that need to be tagged and formulas for tag propagation and

checking for each technique. Then we assigned a rule that specifies how the tag engine can initiate its work.

We specified list of data structures need for the tag engine part of each mechanism. This data structure includes the storage of tags, management of tag engine and placeholder for tag engine state information. We characterized a process which is used for initialization of data structures and variables of tag engine. We then defined set of functions that can have access to the data structure for necessary modifications.

We determined tag propagation and checking rules in terms of data structure for the tag engine based on the chosen mechanism. We defined the propagation and checking rules that shows how tags can be access and what functions can operate on tags. We then specified API's for both propagation and checking functions that include tag manipulation and propagation operation.

In some cases tag checking rules cause a trap. Checking rules for each technique is implemented differently to be able to detect different a set of attacks. So running different instructions in different technique may lead to trap caused by tag checking rule of tag engine.

To handle traps we defined a trap interface, we specified set of information and the procedure to pass the information to the trap handler. We defined trap specific interface function API and it's data structure.

We designed the interface for RTEMS to communicate with the simulator and the memory addresses for communicating with the trap engine. We defined set of control functions and data needed for the control functions. We then specified the API's for interface within the tag engine for each control function and the process that can access the control functions and communicate with them. We write library for supporting the new control functions and API's for interface within the tag engine for each control functions.

To do the testing and evaluation we conducted a design review for data structures, APIs and list of functions. We conducted a unit testing on all modules and functions. These tests are standalone tests to ensure each module behave correctly. We then designed set of generic test suits that can be used for each tagging technique.

2.5. Conclusion

In this chapter we have discussed three different hardware-based tagging techniques that we implement in our simulator. There are server techniques that have been discussed in the literature, but most of them are similar to the once review here. We then bring a whole overview of the framework and how each technique can be integrated to the simulator.

Chapter 3. Background in SIS

In Chapter 2 we introduced DIFT, MBC and UMC security tagging techniques. We gave an overview of how these techniques work and how they can prevent security vulnerabilities. To implement these techniques at the assembly instruction level, it's necessary to know the details of the underlying system where these techniques are implemented. In this thesis the tagging techniques are implemented in the Scalable Processor ARChitecture (SPARC) using the SPARC Instruction Simulator (SIS). SIS is a SPARC instruction simulator which is capable of emulating ERC32 (radiation-tolerant 32-bit RISC Processor). ERC32 is a SPARC variant based computer system implements SPARC version 7 [10].

This chapter provides details about SIS, SPARC instruction level and assembly level details. It also gives details about important features and components of the ERC32. The purpose of using ERC32 is that it provides high performance computation for embedded real time devices. The simulated ERC32 only implements memory and application specific peripherals, although other functionalities are supported by the core. ERC32 support neither MMU nor cache memory. Therefore it needs to access memory directly to run store and load instructions.

3.1. SPARC Instruction Simulator (SIS)

SIS is a SPARC Instruction simulator which simulates the CPU board for the ERC32 based computer systems. ERC32 based computer system is a 32-bit RISC processor which implements SPARC Version 7. All the ERC32 instructions have a 32-bit constant length [11].

The ERC32 incorporates different functionalities implemented in its own computing core. The main ERC32 board functionalities are ERC32 core and ERC32 peripherals. ERC32 core includes Processor and MEmory Controller (MEC) while ERC32 peripherals include EDAC, wait

state generator, timer, interrupt handler, watch dog and UART. ERC32 core Processor consists of three components including IU, FPU and implementation dependent CoProcessor (CP). These components form the 32-bit embedded ERC32 processor [12].

The SIS simulator simulates ERC32 processor, MEC and 32-bit wide instructions. The IU part of the processor is responsible for computing portion of ERC32 and is explained in Section 4.2. FPU executes single and double precision floating point instructions. FPU instructions can be executed concurrently with the IU instructions. The Implementation dependent CP can also be accessed through specific instructions. CP implementation dependent registers can also be accessed through CP instructions. All of these three units in ERC32 processor can work concurrently. Different naming convention exists for each processor component registers. Registers that are used in the IU called “r” registers, those that are used in the FPU “f” registers and those that are used in CP are called “c” registers. There are also control and status registers implemented to keep track of the status of events in the processor [12].

SIS simulates RAM and PROM for ERC32. By default the ERC32 chip has 32 MB RAM and 4 MB PROM which is used as the default amount for SIS simulated RAM and PROM. SIS provides functionalities to simulate ERC32 different applications. Since RAM and ROM sizes are configurable through SIS, these applications can update processor memory sizes.

To run applications using SIS, we can attach SIS to the GNU DeBugger (GDB) like a remote target so it can be used to debug application through GDB [11].

Two versions of SIS has been developed to make it capable of simulating different applications. These versions are named SIS and SIS64. SIS is capable of simulating time up to 2^{32} clock ticks which can run about 5 minutes at 14MHz frequency. SIS 64 is capable of simulating time up to 2^{64} clock ticks which provides almost unlimited time of simulation but it is 20% slower than the previous version of SIS. Running ERC32 applications with security tag engines does not require unlimited time of simulation, so we chose SIS version for our implementation.

3.2. Integer Unit (IU)

Instruction execution and computation is done in the IU. The IU is capable of executing one instruction in each cycle. The IU contains 136 general purpose registers and manages the operations that occur in the processor. The IU keeps track of the Program Counter (PC) during execution of each instruction. It is also responsible for performing integer arithmetic and logic instructions and computes memory addresses for load and store instructions. All of the ALU instructions are register to register operations; and only load and store instruction access memory. The following section describes the register model, data types of the IU, control and status registers [13].

3.2.1. Integer Unit General Purpose and Windows Register

There are total of 140 32-bit registers available for the IU. While 136 32-bit registers are general purpose registers, the rest of them are control/status registers. General purpose registers are divided into 8 global registers and 128 window registers. The 128 window registers are then divided into 8 sets of windows registers on a circular stack. The circular stack contains 24 r registers. The SPARC register file model is known as a register window [10].

There are 32 general purpose registers visible to a program at any given time:

- %g0 to %g7 global registers for storing global data.
- %l0 to %l7 local registers for storing local data.
- %i0 to %i7 in registers for storing incoming arguments.
- %o0 to %o7 out registers for storing arguments to subroutines.

The 8 global registers are mapped to physical registers and the remaining 24 registers are mapped to one of the overlapping register windows. Each register window has local registers, in registers and out registers. These registers are shown in Table 3-1.

Register Name	Register Number
Ins	r[24] to r[31]
Locals	r[16] to r[23]
Outs	r[8] to r[15]
Globals	r[0] to r[7]

Table 3-1: SPARC register set

Global registers are shared among all of the windows; with %g0 register hardwired to zero. Local registers belong only to the current window and they are not shared among other windows. They are usually used for storing temporary and local values. The current PC is stored in %l1 and Next PC (NPC) is stored in %l2 when a trap occurs.

The in and out registers in the register window are shared with adjacent windows. They are used for passing parameters and storing incoming arguments. Among the in registers %i0 is used to store the return argument, %i6 is used for storing frame pointer (%fp) and %i7 is used for storing the return address. Within the out registers, registers %o0-%o5 are used to store the arguments that are passed to a function. %o6 stores the stack pointer (%sp) and %o7 stores the return address. There exists a Current Window Pointer (CWP) which always points to the current active window. This register changes during executing TRAP, SAVE and RESTORE instructions to adjust its content in order to point to the right window [10].

Upon a subroutine call, the return address will be stored in %o7. The new window will be activated and the out registers of the current window become in registers of the next window as shown in Figure 3-1 (adapted from SPARC International Inc.). Save instruction decrements the CWP by one to activate next window. The subroutine's calling procedure's out-registers becomes the callee's procedure in the registers. In this way parameters passed directly. A program needs its own register window to return from a subroutine. The RESTORE instruction increments CWP to restore the caller's window. In this way the previous window becomes the current window. By the nature of

circular stack, the last register window is adjacent to the first one as shown in Figure 3-2 (adapted from SPARC International Inc.).

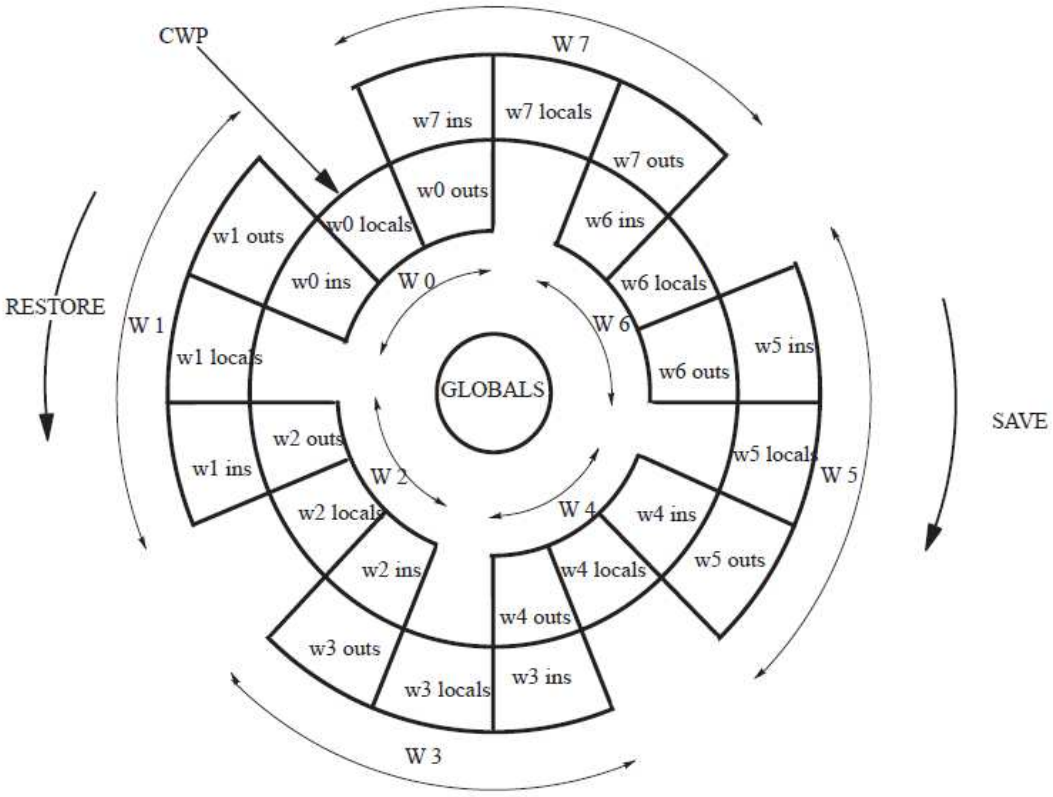

Figure 3-1: Cicular Stack of Register Window

3.2.2. Integer Unit Control and Status Registers

The control/status registers are 32-bit registers including:

- Processor state registers (PSR)
- Windows Invalid Mask (WIM)
- Trap Base Register (TBR)
- Multiply/Divide register(Y)
- Program Counter (PC, nPC)

At any given time the program has access to the current window through CWP field of Processor State Register (PSR). Register window overflow and underflow can be detected with Windows Invalid Mask (WIM) register.

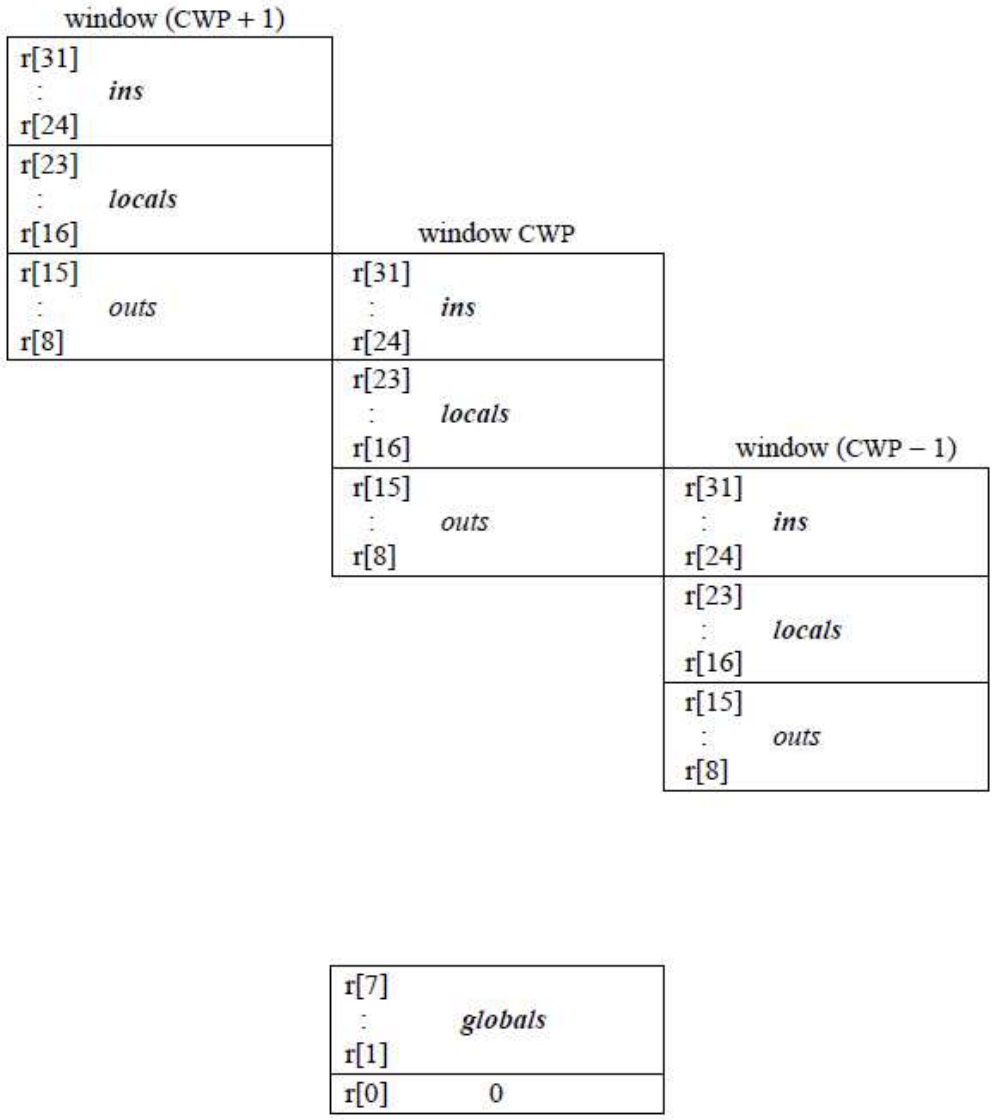

Figure 3-2: Change of the Register Window

4. Processor State Register (PSR)

The Processor State Register holds data that controls the processor or shows the status of the processor. There are several instructions that can modify the PSR. These instructions include SAVE and RESTORE.

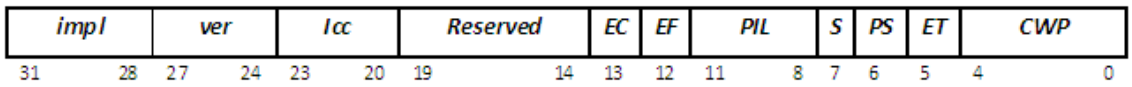

Figure 3-3: Processor State Register

Each of the implementation and version fields are 4 bits long and hold information regarding to the processor implementation number and Integer Unit version number respectively.

The icc field holds the four condition codes for the Integer Unit. There are several arithmetic and logic instructions whose mnemonic end with cc. These instructions can modify the icc bits in PSR. There are also BICC and TICC instructions that cause control transfer based on different bits of icc. The icc field itself contains bits for negative (N), zero (Z), overflow (V) and carry (C) flags. These bits are set or reset based on the results of arithmetic and logic instruction. The icc fields are shown in Figure 3-4.

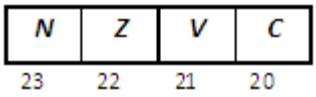

Figure 3-4: Icc bits

The reserved bits (bits 14 through 19) are reserved for future use and they are all set to zero at this time. The Enable Coprocessor (EC) bit, which is bit 13, indicates if the coprocessor is enabled or not. If it is not enabled or the coprocessor does not exist then this bit is set to zero. The Enable Floating Point Unit (EF) flag is bit 12 and it indicates if the floating point unit is enabled or not. Like the EC, if the Floating Point Unit (FPU) is not enabled or the processor does not support FPU then this bit will set to zero. The Processor Interrupt Level (IPL) identifies the level of current the

interrupt. The processor will accept any interrupt which has the priority equal or greater than the value that IPL defines. The Supervisor (S) bit is set to 1 when the processor is in the supervisor mode. The Previous Supervisor (PS) bit holds the value of the S bit when a trap occurs. Enable Traps (ET) bit indicates if traps are enabled or disabled. In the case of disabled traps, all the traps will be ignored. The CWP holds the index of the current active window 0 to 31. These 5 bits CWP can be modified by TRAP, SAVE and RESTORE instructions.

5. Trap Base Register (TBR)

This register provides the address of the trap table. Branch instructions read contents of this register when flow of program execution causes trap.

6. Windows Invalid Mask Register (WIM)

The WIM register is controlled by operating system's software and used in the hardware to determine register window overflow or underflow. WIM is 32 bits wide while each bit corresponds to one of the 32 register window. If one bit is set in the WIM, it indicates that the corresponding register window is invalid.

The trap occurs in the case of register window overflow or underflow, in the case of execution of SAVE or RESTORE/RETT instruction respectively. In the case of a trap, the CWP points to an invalid window indicated in the WIM register. Every time SAVE, RESTORE, or RETT instruction executes, CWP compares the decremented and incremented CWP against the WIM to check for window overflow or underflow.

3.3. Data Types

The IU unit of the ERC32 supports eleven data types. The three basic data formats and supporting width for each format are shown in Table 3-2. Single precision floating points uses 32 bit format; double precision floating points uses 64 bit format. The minimum size of each register is 32 bit wide. If the data written to the register is less than 32 bits, it's written to memory starting from

LSB. Depending whether the data is signed or unsigned, the remaining bits are zero extended or sign extended. For 32 bit data, whether it is unsigned or signed, data is simply loaded from or stored to the memory. Double word operands read from or load to two consequent registers [10].

The organization of data in memory follows the big-Endian convention which means lower addresses contains the higher order bytes.

DATA FORMATS	**SUPPORTING WIDTH**
signed Integer	Byte(8 bits), Halfword (16 bits), Word (32 bit),Tagged Word(30 bits with 2 bit tag), Doubleword (64 bit)
unsigned integer	Byte(8 bits), Halfword (16 bits), Word (32 bit), Tagged Word(30 bits with 2 bit tag), Doubleword (64 bit)
floating point	Word (32 bit), Tagged Word(30 bits with 2 bit tag), Doubleword (64 bits), Quadword(128 bits)

Table 3-2: SPARC Data Types

3.4. IU Instruction Set

The ERC32 processor reads an instruction from a specific memory address provided by the PC. Instructions can be executed, annulled or trapped by the processor. In the case of a trap occurrence, the operating system forwards control based on the trap table. Trap table contains the trap handler addresses. The trap handler's base address is set by the operating system. In the case that an instruction's execution doesn't cause a trap, the address for the next instruction is copied from

NPC to PC. The normal address for the next instruction is generated by incrementing the PC address by 4. This address is copied to the NPC register and is used in the next cycle of instruction execution.

3.5. Instruction Format and Addressing

Instructions are categorized into three different format categories. The first format is CALL instruction. The second format is BRANCH and SETHI instructions. The third format is logical and arithmetic instructions as well as memory instructions. Format 3 also supports floating point and coprocessor instructions. We will not give details about Floating Point instructions since the tested security techniques are not applied to Floating Point instructions. However, since we want to use the coprocessor as a major component for the security tag engine in our design we give details of the coprocessor instructions. Formats for all instructions are shown in Figure 3-5 through Figure 3-9 [11].

The address for operands of each instruction are either located in the instruction itself or calculated from displacement bits. The OPeration code (OP) field is a 2-bit field which determines the instruction types and encodes the 3 major instruction formats. OP2 is a 3-bit field which encodes instructions in format 2. The instruction encodings through OP and OP2 are shown in Table 3-3.

Format	OP	OP2	Instruction
1	1	Unimplemented	CALL
2	0	0	Unimplemented
		1	Unimplemented
		2	Bicc
		3	Unimplemented
		4	SETHI
		5	Unimplemented
		6	FBcc
		7	CBcc
3	2	Unimplemented	Memory instructions
3	3	Unimplemented	Arithmetic, logical, shift and remaining

Table 3-3: OP and OP2 encoding

The other fields in the instructions shown in Figure 3-5 through Figure 3-9 are encoded as below:

- **Rd field:** is a 5-bit field stores the address for source or destination register. This address is used by IU, FPU or CP.

- **imm22 field:** It's a 22-bit field is constant and is used by SETHI instruction to calculate the destination register.

- **cond:** It's a 4-bit field that chooses the condition code for branch instructions.

- **a:** It's a i-bit field in the branch instructions that annuals the instruction execution based on the type of branch.

- **op3:** It's a 6-bit field that encodes format 3 instructions.

- **i:** it's a 1-bit field that selects the second operand for arithmetic and load/store instructions.

- **disp22 and disp30:** these are 22-bit and 30-bit fields. They are used as PC-relative displacement for call or branch instructions.

- **Address Space Identifier (ASI):** it's an 8-bit field which is used by load/store alternate instruction. ASI is sent to the system memory for memory accesses. It is used to control supervisor/ user mode accesses to memory instruction and data.

- **rs1:** It's a 5-bit field. It shows the address of first source operand in r, f or c register.

- **rs2:** It's a 5-bit field. It shows the address of second source operand in r, f or c register when i field is 0.

- **simm13:** it's a 13-bit field. It has 13-bit immediate value used in the case that I field is 1 as the second source operand.

- **opf:** it's a 9-bit field that encodes floating point instructions or coprocessor instructions.

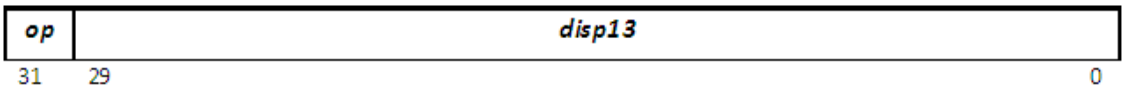

Figure 3-5: Format 1 (op = 1): CALL Instruction

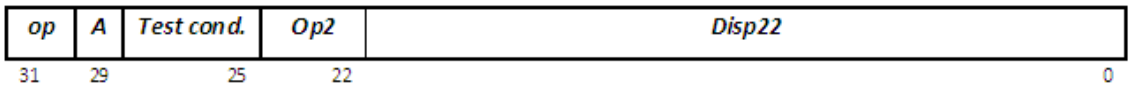

Figure 3-6: Format 2 (op = 0): BRANCH Instruction

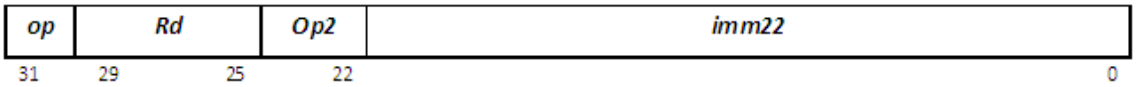

Figure 3-7: Format 2 (op = 0): SETHI Instruction

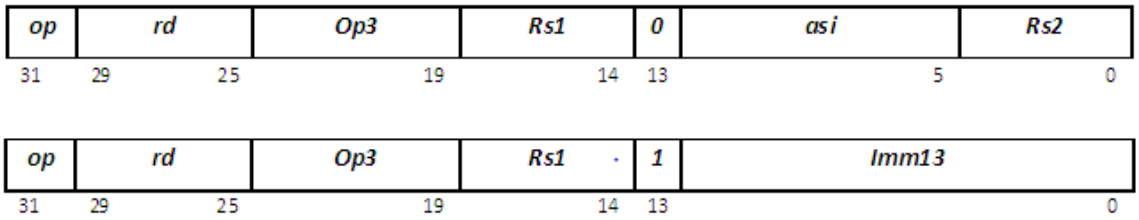

Figure 3-8: Format 3 (op = 2, 3): Integer and load/store Instructions

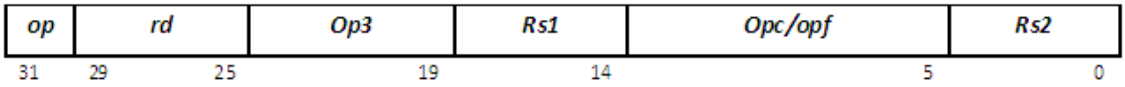

Figure 3-9: Format 3 (op = 2): FP/CP Instructions

SPARC instructions can be further categorized into 6 different categories:

- Load/Store instructions – Format 3, opcode 3
- Arithmetic and logic instructions - Format 3, opcode 2
- Control transfers – Format 1 and 2, opcode 0 and 1
- Floating Point Instructions – Format 3, opcode 2
- Coprocessor instructions – Format 3, opcode 2

Since SPARC is a load/store architecture, load and store instruction are the only instructions which have access to memory. There are three different scenarios in which memory addresses for load and store instructions can be generated. The first scenario uses two registers indicated by rs1 and rs2 fields of an instruction. The value in these two registers is added to create the address for load and store instructions. The second scenario uses an immediate value. In this scenario the i field is 1, rs1 field is used as the first source operand and the imm13 used as the second source operand. The address is calculated by adding the content of rs1 to the sign-extended value of imm13 field of the instruction. The third scenario is a special case of the second scenario. In this case we want to create the address by only using imm13 field in the instruction. For the third scenario the imm13 field value will be added to the rs1 register which is set to %g0. Since %g0 is hardwired to 0 the resulting address value will be only the sign extended value of imm13 field. In this scenario we can have an absolute addressing mode.

We can also use the program counter to calculate the address. The CALL and BRANCH instructions use the program counter as one of the sources for generating the target address. As we saw in Figure 3-5, in the CALL instruction format, the 30-bit displacement is the second source for calculating the address. Note that the ERC32 is a delayed control transfer machine. The PC gets the NPC before the control transfer instruction. After the address is calculated, NPC gets the new address. This means that the instruction following the call or branch instruction is executed before the call or branch is taken.

As we see in instruction format 2 in Figure 3-6, the BRANCH instruction uses the PC as well as a 22-bit displacement to calculate the target address.

3.5.1. LOAD/STORE instructions

Load and store instructions move data to/from registers from/to memory. As discussed earlier the address for load and store instruction is calculated based on different fields of the instruction. The destination field defines where the result is going to be loaded from or stored to.

This field can be any of the system registers or FP and CP registers. There are also two special load/store instructions, SWAP and LDSTUB (atomic load and store instruction). These instructions are atomic instructions meaning they cannot be interrupted. SWAP instruction swaps the contents of a register with a word in memory. LDSTUB reads from memory into a register and fills out the memory location with 1's.

3.5.2. Arithmetic and Logic instructions

Arithmetic and logic instructions take two operands as a source, perform the specific operation on them and save the result in the specified destination register. These two operands can be either two registers in the case that i field is 0, or it can be one register and a 13-bit immediate value in the case that i is equal to 1.

Most arithmetic and logic instructions can be categorized into two sub-categories. The first category can set the icc bits in the icc field of the instruction as well as performing the instruction. This category of instructions has cc at the end of the name of each instruction. The second category only performs the arithmetic and logic calculation and never touches the icc bits.

Arithmetic and logic instructions are divided into arithmetic, logic, shift, SETHI, multiply, divide and tagged add/subtract. In this study, these tagged instructions are defined by SPARC and they are not related to the security tags.

The SETHI instruction is used to create a 32-bit constant value, by using SETHI along with an arithmetic instruction. The 22-bit immediate value in the SETHI instruction is loaded in the upper bits of destination register.

3.5.3. Control Transfer

The Control Transfer instructions set the value of NPC to the desired target address. There are five different sets of instruction in this category, which include conditional Branch, call, Jump, trap, return from trap, SAVE and RESTORE. ERC32 also supports delayed control transfer.

Branch instructions use the icc bits, set by arithmetic instruction, to decide to take branch or not. There are two branch instructions BA (Branch Always) and BN (Branch Never) that are not decided based on the icc bits. The result of these two instructions are always or never branch, respectively. Traps are also occur or not based on the condition codes. In the case of the occurrence of a trap the following sequence happens. Traps are disabled, the state of the processor is saved, and current windows pointer is changed to point to the next windows. The address of the trap base register is copied to PC and the NPC gets the value of PC+4. In the case of returning from a trap, CWP is restored, the return address is calculated and trap conditions is enabled. State of the processor is restored and NPC gets the target address.

The address for the target of a CALL instruction is calculated base on the rules we've discussed. Jump however uses two registers as operands or one register plus 13-bit displacement to calculate the target address. The return address of the CALL instruction will be stored in %o register of the current window. The return address for the JUMP instruction will be stored in the register specified by the rd field of the instruction. SAVE instruction is used to save the current window of the caller and the RESTORE instruction restores the called window.

3.6. MEmory Controller (MEC)

The MEC is designed to interface FPU and IU to memory and I/O devices. It supports concurrent error detection and handling. MEC includes necessary system functions such as: [14]

- memory interface to RAM ranging from 256KB to 32MB
- memory interface to PROM ranging from 128KB to 4MB
- System clock
- I/O interface
- Address decoding
- EDAC

- wait state generator
- 2 32-bit timers
- interrupt handler
- watch dog
- two UARTs
- Block protection
- test and debug support

The MEC can be reprogrammed to interface with different sizes of RAM ranging from 256KB to 32MB. The default value for the RAM size is 256KB. By using MEC_MCR register, we can divide RAM size up to 8 different blocks of memory. Each block is composed of 32-bit data, parity bit and 7-bit check code. The default number of blocks is one.

MEC registers are writeable in the supervisor mode, but they can be read in the user mode. MEC registers are all 32-bit registers. Each bit or group of bits can be used to perform functionality. In some of the MEC registers not all 32-bits are used. The bits that are not used in these registers are marked as reserved bits and will hold a fixed value which is generally a zero. These bits can be read but they are write-protected.

3.7. Coprocessor

The SPARC architecture uses the IU as the main processing core, but the capability of adding two coprocessor extensions is also provided. These extensions can be implemented by using instruction set extensions. The coprocessor extensions are designed so that it can operate concurrently with the IU and FPU. To support the user-defined coprocessor, the coprocessor should include an internal register set and a status register as defined by the SPARC architecture.

The coprocessor register model is defined by SPARC architecture. A coprocessor has up to 32 x 32-bit registers called c registers. All of the operands for the coprocessor instruction are loaded

from “c” registers and results are stored in them. Using coprocessor load and store instructions, the content of these registers can be loaded or stored, to or from memory [11].

The processor can also execute coprocessor instructions which are defined by CPOP1 and CPOP2 opcodes. CPOP1 and CPOP2 opcodes define instructions that can perform calculation inside coprocessor. CPOP1 and CpOP2 instructions are encoded via type 3 format.

Coprocessor control/status registers includes Coprocessor State Register (CSR) and Coprocessor Deferred-Trap Queue (CQ). CSR contains the status of coprocessor and can be checked upon execution of Coprocessor instructions. Coprocessor exception deferred trap is handled using CQ.

3.8. Conclusion

In this chapter we discussed features of SPARC instruction simulator that simulates ERC32 processor. We introduced different categories of SPARC instructions set and its register window. We discussed ERC32 simulated components such as IU, FPU and CP. SIS has lot of functionalizes implemented in it. We chose set of functionalities that can be used by each tagging technique.

Chapter 4. Implementation of DIFT Tagging Schemes in SIS

Chapter 2 provided background on three different security tagging techniques including DIFT. This chapter explains the implementation details for the DIFT technique for SPARC. DIFT is implemented at the instruction level in SIS. In our work, we classify the SPARC instructions base on the rules for DIFT. The tagging rules are defined for each group of instructions as classified below:

- Group 1 defines the BRANCH and CALL instructions
- Group 2 defines 35 ALU instructions
- Group 3 defines LOAD, STORE and SWAP instructions
- Group 4 defines CPOP1 and CPOP2 instructions
- Group 5 defines the rest of the instructions

Base on the rules for the DIFT technique, all of the memory locations and registers are initialized with zero tag value represents untainted tag value. During the instructions execution, tags are retrieved, manipulated, set in registers' tags and saved in the memory tags. To perform get and set of the tag values for the registers and memory locations key functions have been used. The key functions are called from inside the propagation and checking functions to modify or return the tag values from the tag data structure [7].

In our framework we decided that tags should be checked before execution of the instruction. This allows any security exception to prevent instruction execution. To simulate this, we created the `tag_dispatch_interface()` function. The `tag_dispatch_interface()` function contains the propagation and checking rules needed for all five instruction groups introduced above. `Tag_dispatch_interface()` function is called at the beginning of the

`dispatch_instruction_interface()` function of SIS so that the tags are checked and propagated before execution of instructions.

In a real machine this would be accomplished in parallel with an error preventing completion of the instruction. Since error detection happens before completion of the instruction, a close evaluation of the hardware implementation is provided.

4.1. DIFT Initialization of Tag Engine

The DIFT technique keeps a 1-bit location tag for each word in the memory and each register. The data structure for tags in this technique is shown Figure 4-1. This structure will be changed for each tagging technique.

```
C code:
Typedef  struct tag
{
               Char dift_tag ;

               } tag_t ;

typedef struct
{
              tag_t  r[128] ;
              tag_t  g[8] ;
              tag_t  pc, cc, cwp, y;
              } UI_TAGS ;

UI_TAGS   tags ;
```

Figure 4-1: DIFT tag data structure

4.2. DIFT Propagation Rules

Propagation rules are rules that are responsible for the tags propagation in different instructions. Propagation rules are defined for the Groups 2, 3 and 4 of the SPARC instruction as explained hereafter. Each separate tagging technique will have its own propagation rules.

4.2.1. Rules for Group 2 Instructions

Group 2 includes arithmetic, logic and shift instructions as shown in Table 4-1. Propagation rules for this group of instructions are defined in Table 4-2. The propagation rules state that using tainted data in arithmetic, logic and shift instructions produce tainted result. All of the instructions in this group can either have one or two operands. In the case of two operands, rules are described in Table 4-2. In the case of having one operand, the result's taint mark is the taint mark for the only operand.

For example, executing instruction SUB %g1, %g2, %g3, subtracts the content of %g1 from the content of %g2 and stores the result in %g3. If either of %g1 or %g2 is tainted then %g3 is tainted. For instructions which modify the condition codes, propagation rules are exactly the same regardless of the result of the icc bits evaluation.

Opcode	Name
SMUL (SMULCC) UMUL (UMULCC)	Signed Integer Multiply (and modify icc) Unsigned Integer Multiply (and modify icc)
SDIV (SDIVSCC) UDIV (UDIVCC)	Signed Integer Divide (and modify icc) Unsigned Integer Divide (and modify icc)
XNOR (XNORCC) XOR (XORCC)	Exclusive Nor (and modify icc) Exclusive Or (and modify icc)
OR (ORCC) ORN (ORNCC)	Inclusive Or (and modify icc) Inclusive Or Not (and modify icc)
ANDN (ANDNCC) AND (ANDCC)	And Not (and modify icc) And (and modify icc)
SUB (SUBCC) SUBX (SUBXCC) TSUBCC TSUBCCTV	Subtract (and modify icc) Subtract with Carry (and modify icc) Tagged Sub and modify icc Tagged Sub and modify icc and Trap on
ADD (ADDCC) ADDX (ADDXCC) TADDCC TADDCCTV	Add (and modify icc) Add with Carry (and modify icc) Tagged Add and modify icc Tagged add and modify icc and Trap on
SSL SRL SRA	Shift Left Logical Shift Right Logical Shift Right Arithmetic

Table 4-1: Implemented ALU instructions

TADDCC, TADCCTV and TSUBCC, TSUBCCTV instructions are the same as ADDCC and SUBCC respectively except that tagged instructions' result depends on the result of ADDCC and SUBCC instructions. If bits at location 0 or 1 of any of the operands are not zero, the tag overflow occurs. Recall this is not a security tag. Also, if the result from an addition instruction causes overflow then tag overflow occurs, which results is setting the overflow bit in PSR. In case that it does not cause tag overflow, the overflow bit in PSR is cleared and the result of the calculation is stored in the destination register. The difference between TADDCC and TADDCCTV or TSUBCC and TSUBCCTV is that in TADDCCTV and TSUBCCTV is that if the tag overflow occurs, execution of TADDCCTV and TSUBCCTV instructions cause a trap and the contents of the destination register and icc bits remain unchanged. The propagation function for these instructions is called before the tag overflow evaluation to make sure tag overflow doesn't interfere with the propagation rules. It would be useful to modify the propagation rules such that register clearing operations including XOR %r1,%r1,%r1 do not propagate tags. However register cleaning operation in runtime is not specified in the DIFT specification, therefore they are not implemented here.

Op1 taint Mark	Op2 taint Mark	Result taint Mark
1	0	1
0	1	1
1	1	1
0	0	0

Table 4-2: Rules for ALU instructions

4.2.2. Rules for Group 3 instructions

Group3 includes LOAD, STORE, Atomic load store and SWAP instructions. Group 3 can be divided into three different sub groups according to the similarity of the DIFT propagation and checking rules. The sub groups are:

- Subgroup 1 defines 12 LOAD instructions

- Subgroup 2 defines 8 STORE instructions
- Subgroup 3 defines LDSTUB and SWAP instructions

Following is the detailed description of each subgroup:

- **Subgroup 1 defines 12 LOAD instructions**

Opcode	Name
LDD (LDDA)	Load Double word (from Alternate space)
LDSB (LDSBA)	Load Signed Byte (from Alternate space)
LDUB (LDUBA)	Load Unsigned Byte (from Alternate space)
LDSH (LDSHA)	Load Signe Halfword (from Alternate space)
LDUH (LDUHA)	Load Unsigned Halfword (from Alternate
LD (LDA)	Load Word (from Alternate space)

Table 4-3: Implemented LOAD instructions

Table 4-3 shows the Load instruction in the SPARC architecture. Load instructions copy a byte, half word, word or double word from memory to the register rd. Load instructions are in Format 3 instructions of SPARC. In case of a zero “i” field, the effective address to load from is calculated by adding contents of the registers rs1 and rs2. If the “i” field equals one, the effective address is calculated by adding the contents of register rs1 to the sign extended simm13. All of the registers are 32-bits wide, so in the case of loading a byte or half word, the value is right justified in the destination register. The rest of the bits in the rd register are sign extended if the instruction is signed load. In the case of executing unsigned load the rest of the bits are filled with zeroes.

Checking rules for Load instruction is defined in such way that if load instructions’ address is calculated using one register, the DIFT tag for that register is checked. If the value of the tag is tainted then it causes a security exception. If the address is calculated using two registers, then DIFT tags of both registers should be checked, and if either one of them is tainted, then program will cause a security exception.

In the case that no exception occurs, the execution of the instruction proceeds to propagating the tag. Propagation rules for the Load instruction set the tag of the register rd with the tag value of the data structure of the memory address.

The Load Double Word instruction copies a double word from memory into a register pair. The LDD instruction loads the contents of the most significant word into the register rd and the less significant word into the register rd+1.

Checking rules for LDD is the same as other load instructions. Checking rules checks the tag value of the source register which is used to calculate the target address. In addition source register causes an exception if the tag value is tainted. However the tag propagation rules are different. The tag value of the register rd gets the tag of the most significant memory address, and the tag for the register rd+1 gets the tag of memory address+4. This choice was made since LDD is often used as an optimization for the copying of large amount of contiguous data, where some adjacent words are differently tagged.

- **Subgroup 2 defines 8 STORE instructions**

Opcode	Name
ST (STA)	Store Word (from Alternate space)
STB (STBA)	Store Byte (from Alternate space)
STH (STHA)	Store Halfword (from Alternate space)
STD (STDA)	Store Doubleword (from Alternate space)

Table 4-4: Implemented STORE instructions

Store Instructions, which are in subgroup 2, are shown in Table 4-4. These instructions store the contents of the register rd into the specified memory address. ST stores a word from rd into a word of memory, STB stores the least significant byte of rd into memory and STH stores the lease significant half word of rd into memory. The address used to store the result is calculated either by using contents of the register rs1 or both registers rs1 and rs2. If rs1 is the only register that is used to calculate the address, DIFT checking rules checks the tag for register rs1 to make sure that the calculated target address does not come from tainted value. If rs1's tag value is tainted then a

security exception occurs. If registers rs1 and rs2 are used to calculate the target address, both of the tags of these two registers is checked and if either one of them is tainted, the program throw an exception.

If the checking phase passed in the two scenarios explained before, the program proceeds to the propagation phase. In the propagation phase the memory address gets the tag value of the register rd.

Store Double Word instruction, stores a value from the registers rd and rd+1 to the memory at the effective address and effective address+4 respectively. The effective address is the address of the most significant. The effective address is calculated using register rs1 or rs1 and rs2. The address of less significant word is effective address+4. Therefore rs1 or rs1 and rs2 should be checked. If the rs1 (in first case) or rs1 and rs2 (in second case) have zero tag value, tag propagation takes place. Tag propagation copies the tag values of the registers rd and rd+1 to the effective address and effective address+4 respectively.

- **Subgroup 3 defines LDSTUB and SWAP instructions**

Opcode	Name
LDSTUB (LDSTUBA)	Atomic Load-Store unsigned Byte (from
SWAP (SWAPA)	Swap r Register with Memory (from Alternate

Table 4-5: Implemented LOAD-STORE and SWAP instructions

Subgroup 3 instructions are shown in Table 4-5. The LDSTUB instruction loads a byte from memory to the register rd and then writes 1 to all of the bits in the address specified in the instruction. The address to load from and store to, is either calculated using rs1 or rs1 and rs2. In the checking rules for LDSTUB the tag value for the rs1 or rs1 and rs2 are checked. If any of the tag values of rs1 or rs2 is tainted the program causes a security exception, otherwise tags are propagated. Propagation rules are different based on the nature of the LDSTUB. For the load part of the instruction, register rd gets the tag value of the specified memory byte. In the store part of the

instruction, since the memory byte is filled with 1's which is a constant value, the tag of the memory byte is cleared.

The SWAP instruction, like LDSTUB, consists of two instructions executed atomically. It swaps the content of the register rd with a word in the memory. At first the effective address of the specific word in the memory is calculated by using the register rs1 or registers rs1 and rs2. The SWAP instruction loads content of the memory to a temporary register, following by storing content of the register rd to memory which after the temp value is stored in the register rd. Checking takes place for the registers which used to produce the memory address. The same thing happen for the tag values, after calculating the effective address, the tag value for this address is stored in the temporary value. The tag value of register rd will get overwritten to the tag value of the memory address and the temporary tag value will be stored as the tag value of register rd.

4.2.3. Rules for Group 4 instructions

CPOP1 and CPOP2 instructions constitute group 4 instructions. CPOP1 and CPOP2 instruction are in Format 3 in the SPARC architecture instruction format. Although the implementation of the CPOP is not in the SPARC architecture, we used these instructions by hard coding functionalities. We used the Coprocessor to implement the tag engine and tag related function inside the tag engine. The 9 bit opc field of the coprocessor provides the ability to define up to 2^9 different instructions for each one of the CPOP1 and CPOP2 instructions. In our implementation we used CPOP1 to control tag engine by turning it on or off. We defined a set of instructions as shown in Table 4-6.

CPOP1	Opc Field Value	Address in HEX Format
CPOP_TURN_ON_TAGGIN	0	0x81B00000
CPOP_TURN_OFF_TAGGI	1	0x81B00020

Table 4-6: New CPOP1 instructions

The two inline assembly functions `_rtems_tag_enable()` and `_rtems_tag_disable()` are created as an interface so that the CPOP tag engine can be turned

on and off respectively from users' programs. If either of these instructions is executed by the user, `tag_dispatch_interface()`executes the cpop1 instruction. In the beginning of execution of each instruction in the `tag_dispatch_interface()` the cpop variable is checked. If cpop1 is not set it means that no tagging instruction is allowed to execute unless the tag engine is on. If the user requests to turn off the tag engine, the cpop variable value is set to false and all of the register tags are cleared. If execution of an instruction causes an exception at any point of the execution in the `tag_dispatch_interface()`, the cpop value will set to false and the program execution will be forwarded to security exception handling.

CPOP2 instructions are responsible for manipulating tag values in the tag engine. These instructions can set, clear and return the tag value for a specific memory location in the DIFT tag engine.

Since these instructions are responsible for programming the coprocessor, CPOP1 and CPOP2 instructions always execute whether the cpop flag is true or false.

CPOP1	Opc Field Value	Address in HEX Format
CPOP_SET_DIFT_TAG	0	0x87B84002
CPOP_CLEAR_DIFT_TAG	1	0x87B84022
CPOP_GET_MEMORY_TA	2	0x87B84042

Table 4-7: New CPOP2 instructions

Executing CPOP2 instruction will yield to execution of the cpop function. Cpop function takes the opc value of the CPOP2, the address whose tag needs to be manipulated and the result register as an argument. Each value of the opc during the execution of cpop2 function provides separate scenarios of the tag manipulation. These scenarios can be described as setting the tag for a specific address, clearing the tag of a specific address, and returning the tag value of a specific memory location.

CPOP2 instructions are executed as a result of any of the following function calls from the user's program:

- `_rtems_set_dift_tag(addr)`

- `_rtems_clear_dift_tag(addr)`
- `_rtems_get_memory_tag(addr)`

These three functions use the inline assembly routines to set, clear and return the tag values for the specific address provided by the user.

4.2.4. Rules for Group 5 instructions

The rest of the instructions are floating point instructions, trap instruction and SETHI instruction. Since we didn't implement any of the tagging techniques for the floating point instruction, execution of these instructions neither causes a security exception nor modifies the result tag. The trap instruction follows the same rule as the floating point instruction meaning that it doesn't have any effect on the tag engine. However executing the SETHI instruction modifies the result tag. As described in the SIS background chapter, executing SETHI instruction results in writing a constant value to the destination register. Since the value written is a constant, the register gets the tag value of zero. However, there are no checking rules for the SETHI instruction.

4.3. DIFT checking rules

Checking rules check the tag for specific instructions base on the DIFT rules. In the case of illegitimate access, the checking function will cause a security exception. Checking rules are defined for Group 1 of the SPARC instruction as defined in the remainder of this section.

4.3.1. Rules for Group1 instructions

Group 1 instructions are shown in Table 4-8. CALL and Branch instructions are in Format 1 and Format 2 of the SPARC instructions respectively. JMPL and RETT instructions are both in Format 3 of the SPARC instructions. According to DIFT rules, data that carries tag value of one is tainted data. If the calculated target address for Group 1 instructions is tainted, then execution of

these instructions is not allowed and state of the system will change to illegitimate state, which causes a security exception. This group of instructions never propagates tags.

Opcode	Name
CALL	Call and Link
Bicc	Branch on integer condition code
JMPL	Jump and Link
RETT	Return from Trap

Table 4-8: Implemented CALL, BRANCH, JUMP and RETURN instructions

In the Group 1 instructions the register NPC gets the result target address. This value is passed to the `tag_check(npc)` function. This function checks the NPC's tag. If the NPC has a tainted tag value, execution of the instruction is forwarded to the exception handler. Different scenarios for each instruction in Group 1 are defined here under:

CALL Instruction: The CALL instruction stores content of the PC into r[15] which is %o7. Since CALL is Format 1 instruction of the SPARC, it takes the value of the disp30 field of the instruction. It attaches 2 zero bits at the beginning of dsip30 to make it 32 bits. Then it stores the value as a jump address in the NPC. After calculating jump target address, `tag_check_CALL()` is called. The calculated address is passed as an argument to this function so it can be evaluated.

Branch Instruction: Conditional branch instructions contain Branch Always (BA) and Branch Never (BN) instruction. BN never takes the branch which means it is a NOP. BA always takes the branch regardless of the status of icc field. Other branch instructions evaluate the condition code in the icc register by using comparison and creates a result based on the result of the comparison. This result can be either true or false. If the result of evaluating icc for a Bicc instruction is true then `tag_check_branch(npc)` function is called.

JUMP Instruction: JMPL instruction stores the contents of PC into a register specified in the rd field of the Format 3 instructions. NPC gets the value of r[rs1]+r[rs2] in the case that the "i" field is zero, or r[rs1]+sign-extended[simm13] if the "i" field is 1, as a jump target address. After calculating jump target address, `tag_check_JMPL()` is called to evaluate the target address's tag value.

RETT Instruction: RETT instruction is used to return from a trap handler. NPC gets the value of the r[rs1]+r[rs2] in the case that i field is zero, or r[rs1]+sign-extended[simm13]. After calculating jump target address, `tag_check_RETT()` is called and it checks the contents of the tag for the target address.

As we mentioned, the Group 1 of instructions never sets a tag for any specific address so they only check for the illegitimate use of data.

4.4. Conclusion

In this chapter we introduced instruction level rules for the DIFT tagging engine. We defined the implemented data structure for the DIFT. We categorized SPARC instructions into different groups and defined propagation and checking rules for each group. We then introduced the set of new API's that added to the DIFT tag engine inside coprocessor. These API's are used to enable or disable tag engine operations or manipulate tags.

Chapter 5. Implementation of Memory Bound Checking technique in SIS

The Memory Bound Checking technique was implemented by using the coloring scheme introduced by Clause et al. [9]. The BC technique assigns a tag to each memory location and pointers. As we described in the background chapter, during memory allocation, the BC technique assigns the same tag value of location and pointer color for the allocated memory. During the program execution, the pointer tags is propagated and checked whenever a memory location is needed to be accessed. If the pointer and memory tags do not match, program execution is forwarded to the exception handler.

We implemented the Bound Checking technique in the SIS. The coprocessor maintains 4-bits tag for pointer color for each register and word in memory. It also keeps 4-bits tag for location color for each memory location. The tags set by special instructions implemented in the coprocessor. Based on the instruction execution, the tags are propagated. The tag result is checked to see if the pointer color matches the location color of the memory location. If these two values don't match the coprocessor throws an exception. A detailed description of implementing the BC technique is explained in this chapter.

5.1. BC Initialization of Tag Engine

The BC technique keeps a 4-bit location color and pointer color tags for each word in memory. It also keeps a 4-bit pointer color tag for each register. The data structure for tags in this technique is in Figure 5-1:

```
C code:
Typedef  struct tag
{
               Char location_color ;
               Char pointer_color ;
               } tag_t ;

typedef struct
{
              tag_t  r[128] ;
              tag_t  g[8] ;
              } UI_TAGS ;

UI_TAGS   tags ;
```

Figure 5-1: MBC Tag data structure

As it's shown in Figure 5-1, pointer color and location color are assigned to each memory location and register. Location color for registers is not initialized or used during the program execution, since location color is only associated with memory locations.

We defined a value for initializing untainted location and pointer colors. This value can be any number outside the range of 0 to 15. In the beginning of the coprocessor initialization, all of the words in memory and register location and pointer colors will be set to this value. Also we should mention that untainted value is different from tag value 0. Tag values can be any value between 0 and 15.

One of the key points in initialization and propagation of the tags is their range. Tags are stored in 4-bit Char locations so they cannot go below zero nor above 15. To ensure that, every time a tag value is going to get set, we use mod 16 operation to prevent overflows or underflows.

5.2. BC Propagation rules

Tag propagation rules are implemented based on the rules described in Clause et al. [9] paper. Based on the described propagation rules, we classified the SPARC instructions into 7 groups:

- Group 1 defines MUL, DIV, OR and XOR ALU instructions.

- Group2 defines SUB ALU instructions
- Group3 defines ADD ALU instructions
- Group 4 defines AND ALU instructions
- Group 5 defines LOAD instructions
- Group 6 defines STORE instructions
- Group 7 defines SWAP and LDSTUB instructions
- Group 8 defines CPOP1 and CPOP2 instructions
- Group 9 defines the rest of instructions

A common scenario exists among propagation rules for all groups of instructions. It is possible that none of the operands are tainted. This case should be handled appropriately to prevent any non deterministic behavior in the code.

5.2.1. Rules for Group 1 instructions

Opcode	Name
SMUL (SMULcc)	Signed Integer Multiply (and modify icc)
UMUL (UMULcc)	Unsigned Integer Multiply (and modify icc)
SDIV (SDIVcc)	Signed Integer Divide (and modify icc)
UDIV (UDIV)	Unsigned Integer Divide (and modify icc)
MULScc	Multiply Step (and modify icc)
DIVScc	Divide Step (and modify icc)
IOR (ORcc)	Inclusive-Or (and modify icc)
IORN (ORNcc)	Inclusive-Or Not (and modify icc)
IXOR (XORcc)	Exclusive-Or (and modify icc)
IXNOR (XNORcc)	Exclusive-Nor (and modify icc)

Table 5-1: Implemented MUL, DIV, OR and XOR instructions

Op1 taint Mark	Op2 taint Mark	Result taint Mark
t_{op1}	Untainted	Untainted
Untainted	t_{op2}	Untainted
t_{op1}	t_{op2}	Untainted
Untainted	Untainted	Untainted

Table 5-2: Rules for Group1 Instructions

Multiply, Division, OR and XOR instructions are shown in Table 5-1. These instructions are either conditional or unconditional. No matter if the input is tainted or not, the result will always be untainted. This logic is shown in Table 5-2. Executing any of the Group1 instructions will make a call to their specific propagation functions. This function will remove the tag for the specific register's pointer color.

5.2.2. Rules for Group 2 instructions

Opcode	Name
SUB (SUBcc)	Subtract (and modify icc)
SUBX (SUBXcc)	Subtract with Carry (and modify icc)
TSUBcc (TSUBccTV)	Tagged Subtract and modify icc (and Trap on overflow)

Table 5-3: Implemented SUB instruction

Op1 taint Mark	Op2 taint Mark	Result taint Mark
Untainted	Untainted	Untainted
top1	Untainted	top1
Untainted	top2	top2
top1	top2	top1 - top2

Table 5-4: Rules for Group 2 instructions

Arithmetic instruction Subtract is shown in Table 5-3. Instruction such as `SUB %g2, %g1, %g3` will subtract the value in `%g2` from the value in %g1 and will store the result value in `%g3`. According to the rules shown in Table 5-4, the pointer color of `%g1` and `%g2` are checked to calculate the result pointer color of the `%g3`. In immediate type SUB instruction, the result pointer color is the pointer color of the register. We also treat the instructions that change the conditional code the same as the usual instructions, so the rules for these two types of instructions are the same.

5.2.3. Rules for Group 3 instructions

Opcode	Name
ADD (ADDcc)	Add (and modify icc)
ADDX (ADDXcc)	Add with Carry (and modify icc)
TADDcc (TADDccTV)	Tagged Add and modify icc (and Trap on overflow)

Table 5-5: Implemented ADD instruction

Op1 taint Mark	Op2 taint Mark	Result taint Mark
Untainted	Untainted	Untainted
t_{op1}	Untainted	t_{op1}
Untainted	t_{op2}	t_{op2}
t_{op1}	t_{op2}	$t_{op1} + t_{op2}$

Table 5-6: Rules for Group 3 instructions

ADD instructions forms Group 3 of instructions shown in Table 5-5. These instruction propagation rules shown in Table 5-6 are similar to propagation rules described for SUB instructions. The main difference is caused by the way the technique calculates the result register's pointer color. In this group we add the two pointer color values of the register operands and in the case of immediate value, the result pointer color is the pointer color of the register, whether it is an untainted register or register with a pointer color.

5.2.4. Rules for Group 4 instructions

Opcode	Name
IANDN (IANDcc)	And Not (and modify icc)
IAND (IANDcc)	And (and modify icc)

Table 5-7: Implemented AND instruction

Op1 taint Mark	Op2 taint Mark	Result taint Mark
Untainted	Untainted	Untainted
t_{op1}	Untainted	t_{op1}
Untainted	t_{op2}	t_{op2}
t_{op1}	t_{op2}	Untainted

Table 5-8: Rules for Group 4 instructions

Group 4 instructions shown in Table 5-7. In instruction `AND %g1, %g2, %g3` if both of the `%g1` and `%g2` are tainted or untainted, `%g3` will be untainted. In cases that one of `%g1` and `%g2` is tainted, `%g3` will be tainted if it points to an address in the same memory area as the tainted operand. The rules for this group of instruction are shown Table 5-8.This can be implemented by using a heuristic. For example a check should be done to see if the first 16-bits of the value of `%g3` and tainted register match each other. Otherwise it will be a situation such as masking a value by performing AND to an operand and `0x0000` value.

5.2.5. Rules for Group 5 instructions

Opcode	Name
LDSB (LDSBA)	Load Signed Byte (from Alternate space)
LDSH (LDSHA)	Load Signed Halfword (from Alternate space)
LDUB (LDUBA)	Load Unsigned Byte (from Alternate space)
LDUH (LDUHA)	Load Unsigned Halfword (from Alternate space)
LD (LDA)	Load Word (from Alternate space)
LDD (LDDA)	Load Doubleword (from Alternate space)

Table 5-9: Implemented LOAD instruction

Table 5-9 shows Load instructions in SPARC architecture. Load instructions copy a value from memory space to a register. For example, the instruction `LD [%fp-16], %o1` loads content of the memory space `[%fp-16]` to `%o1` register. The destination register pointer color gets the pointer color of the address which is loaded a value from, whether this value is an untainted value or any color in the range of 0 to 15.

We classified the Load Double word instruction in the same group as Load instructions. These instructions are basically the same in our implementation. Load Double word instructions move a double word from memory into a register pair. To implement these instructions, we perform two calls to the load propagation function. In the first call the pointer color of the memory address is copied to the pointer color of the specified register. In the second call the pointer color of the adjacent memory location is copied to the pointer color of the next register in the register pair.

5.2.6. Rules for Group 6 instructions

Opcode	Name
STB (STBA)	Store Byte (into Alternate space)
STH (STHA)	Store Halfword (into Alternate space)
ST (STA)	Store Word (into Alternate space)
STD (STDA)	Store Doubleword (into Alternate space)

Table 5-10: Implemented STORE instruction

Table 5-10 has a list of Store instructions in the SPARC architecture. Store instructions copy a value from a register into memory. Store instructions propagation rules can be implemented the way similar to Load instructions, except that the pointer color of the specific register will be copied to the pointer color of the address. Store Double word instructions also copy a double word from register pair into the memory. In this case same as the load double word, we use the same propagation rules twice, so that the pointer color of the register pair is copied to the pointer color of the adjacent memory locations.

5.2.7. Rules for Group 7 instructions

SWAP and LDSTUB instructions are from the type of load-store instructions. These instructions are implemented as an atomic load and store instructions. We treat these two instructions the same. We actually didn't implement separate propagation rules. We used the load propagation rules for the specific address and register, then we used the store propagation rules for the address and a fixed untainted value for the pointer color of the register, so that storing a fixed value results in an untainted value in memory.

Opcode	Name
LDSTUB (LDSTUBA)	Atomic Load-Store Unsigned Byte (in Alternate space)
SWAP (SWAPA)	Swap r Register with Memory (in Alternate space)

Table 5-11: Implemented Load/Store and SWAP instruction

5.2.8. Rules for Group 8 instructions

CPOP1 and CPOP2 are implemented for UMC to control the tag engine. The coprocessor should be enabled at the beginning of execution of the UMC technique. The CPOP1 instruction is responsible for enabling or disabling coprocessor. If execution of an instruction causes a security exception, then coprocessor is turned off, so no other instruction can produce another exception and gets executed after the security exception. Tag engine can be turned on and off from user's program by calling `_rtems_tag_enable()` and `_rtems_tag_disable()` functions. Calling the `_rtems_tag_enable()` function from the user's program will cause the execution of the `tag_dispatch_interface()`. However calling the `_rtems_tag_disable()` function will abort execution of `tag_dispatch_interface()`.

The CPOP2 instructions are responsible for manipulating the memory location tags and also memory and pointer color tags. These instructions are technique specific, meaning that they are hardcoded individually for each security tagging technique. We used the instructions provided in the Table 5-12 to set/clear the BC tag or to return the tag value of a specific memory address.

CPOP1 and CPOP2 instructions always execute, whether the cpop flag is true or false, since these instructions are responsible for programming the coprocessor.

CPOP2 instructions are executed as a result of any of the following function calls from the user's program:

- `_rtems_set_pointer_color`
- `_rtems_set_location_color`
- `_rtems_clear_bc_pointer_color`

- `_rtems_clear_bc_location_color`
- `_rtems_get_memory_pointer_color`
- `_rtems_get_memory_location_color`
- `_rtems_set_register_pointer_color`

CPOP2	Opc Field Value	Address in HEX
CPOP2_SET_POINTER_COLOR	5	0x87B840A2
CPOP2_SET_LOCATION_COLOR	6	0x87B840C2
CPOP2_CLEAR_POINTER_COLOR	7	0x87B840E2
CPOP2_CLEAR_LOCATION_COLOR	8	0x87B84102
CPOP2_GET_MEMORY_LOCATION_COLOR	9	0x87B84122
CPOP2_GET_MEMORY_POINTER_COLOR	10	0x87B84142
CPOP2_SET_REGISTER_POINTER_COLOR	11	0x87B84162

Table 5-12: New CPOP2 instructions

The above function calls are hardcoded functionalities that added to the tagging technique so we can manipulate tags from user's program. In a real implementation, the compiler would insert these calls into the code. Execution of any of the instructions from user's program sets the opc and address field of the CPOP2 instruction. Then the arguments will be passed to a `cpop2()` function. Based on the argument the `cpop2()` function decides to set/ clear the location color/pointer color, or return the memory color.

5.2.9. Rules for Group 9 instructions

The rest of the instructions are floating point instructions, trap instruction, Branch and Call instructions. Since we didn't implement any of the tagging techniques for the floating point instruction, execution of these instructions will neither cause a security exception nor modify the result tag. Trap instruction also follows the same rule as the floating point instruction means it doesn't have any effect on the tag engine. Also base on BC's rules, control dependency instructions are disregarded. Therefore executing any of the control dependency instructions such as Branch and Call have no effect on the flow of tag engine code.

5.3. BC checking rules

Checking rules for BC intercepts any memory access whether it is Load or Store through a pointer. If memory location and the pointer which is used to access, have the same tag value, checking is passed. Conversely, if memory location and pointer have different taint mark, program is halted and it throws exception.

The load checking rule checks the tag value of the effective address for load instruction. The effective address is calculated by adding the contents of two operand registers, if i field is zero, or it is calculated by adding the content of register operand and sign extended immediate value, if i field is one. The load checking rule extracts the pointer color associated with these two register operands and adds them. In the next step the checking rules check to see if the specified address location color matches the calculated pointer color. If the two values match then this access is considered legal access. We should also mention that there are some special cases that both the value of the location color of the address and calculated pointer color are the same but they are both untainted. We count this scenario as an illegal access so it throws an exception in this case for both load and store instructions. Checking rules for store instruction is also the same as load. Pointer color tags for registers that are used to calculate the store address get checked. If they are the same as the location color of memory which the result is going to be store into, then the access considered as a legal access. In both load and store instructions, if pointer color and memory location color do not match, IMA occurs and the program execution will be forwarded to the exception handler.

5.4. Conclusion

In this chapter we defined the instruction level implementation of BC technique tag engine. We defined the data structure for memory locations and registers in the tag engine. We categorized SPARC instructions into different groups and defined propagation and checking rules for each group. We then defined the set of API's that can enable or disable the tag engine. API's also are capable of manipulating tags for registers and memory locations.

Chapter 6. Implementation of Uninitialized Variable tagging technique in SIS

Uninitialized Memory checking is a common technique implemented to prevent reading from an unutilized memory. Chapter 2 gives background on this technique. In this chapter we describe the implementation of a UMC technique in SIS. A one bit tag is associated with each memory location for the UMC technique, which is initialized to zero at the beginning of execution of the program.

Writing a value to a specific location using store instruction set the tag for the location. Loading a value from the memory location performs a check on this tag to make sure it is initialized. The tag value equal to zero indicates that the memory location is uninitialized; therefore the load will result in a security exception. The UMC implementation configures SIS so that execution of the load and store instructions is forwarded to the tag engine. The tag engine also implements some intermediate instructions to set or get the tag value of the specific memory locations.

6.1. UMC Initialization of Tag Engine

Figure 6-1 shows the data structure of tags in the UMC technique. The tag value of a memory location can be extracted by translating the memory address to the address of the corresponding 1-bit tag in the tag engine data structure. For ease of implementation in the simulator, we assigned a character to tags but we only use 1-bit of the character. The 1-bit memory address's tag is extracted by adding an index value to the base address of tag data structure. The index value is equal to its corresponding memory index value from the memory base address. The functions that perform the address resolution for setting and getting tag values are `get_umc_tag()` and `set_umc_tag()` respectively.

```
C code:

struct tag
{
     char umc_tag;
};
typedef struct tag tag_t;
tag_t tag_ram[(RAM_END - RAM_START)>>2];
```

Figure 6-1: UMC tag engine data structure

Implementing the UMC technique requires defining propagation and checking rules for load and STORE instructions. The following list classifies each group of instructions for the UMC technique.

- Group 1 defines load instructions
- Group2 defines store instructions
- Group3 defines load/store instructions
- Group 4 defines the rest of instructions

The UMC memory is the only place where affected by carrying, initializing and clearing tags. Thereafter registers are not affected with the tags and do not need associated tagging rules.
In the beginning of the SIS execution, the one-time initialization function assigns the zero tag values for each RAM memory location.

6.2. UMC propagation Rules

Tag propagation rules are implemented for store and load/store instructions which are categorized in group 2 and 3 SPARC instructions. UMC tag propagation rules set the tag value of the address accessed through store instruction.

6.2.1. Rules for Group 2 instructions

The store instructions set the tag for memory address. This group of instructions takes the address of instruction and they set the tag value for the specific address to one. There are no

checking rules implemented for this group of instructions, so whenever the system wants to write a value to a specific memory address, the tag value for the address is set.

Opcode	Name
STB (STBA)	Store Byte (into Alternate space)
STH (STHA)	Store Halfword (into Alternate space)
ST (STA)	Store Word (into Alternate space)
STD (STDA)	Store Doubleword (into Alternate space)

Table 6-1: Implemented STORE instructions

6.3. UMC checking rules

Checking rules for UMC intercepts any memory load instruction. If memory location that is used to load a value from is tagged, checking is passed. Conversely, if memory location is not initialized hence not tagged, the program throws an exception. Checking rules are implemented for Group 1 instructions. Also the load part of load/store instruction also gets checked using the same set of rules for load.

6.3.1. Rules for Group 1 instructions

As described before, the load instruction checks the memory location's tag value to see if it is initialized or not. Instructions in this group never initialize or modify a tag, they only check the tags. The checking rules for this group take the address from the instruction, translate the address to get the tag value and check its tag value. If the tag value is not one, meaning the memory location is uninitialized; it throws a security exception and doesn't load the value.

Opcode	Name
LDSB (LDSBA)	Load Signed Byte (from Alternate space)
LDSH (LDSHA)	Load Signed Halfword (from Alternate space)
LDUB (LDUBA)	Load Unsigned Byte (from Alternate space)
LDUH (LDUHA)	Load Unsigned Halfword (from Alternate space)
LD (LDA)	Load Word (from Alternate space)
LDD (LDDA)	Load Doubleword (from Alternate space)

Table 6-2: Implemented LOAD instructions

6.3.2. Rules for Group 3 instructions

Group 3 instructions include the atomic load/store instruction and the swap instruction. The proper way to handle the LDSTUB instructions is to check the tag value of the specific memory location that is going to be used to load the value. Then LDSTUB sets the tag for the memory location that will be written to. The swap instruction also consists of a load and store instruction. In the load part, both memory locations should be checked. If either location is uninitialized, the swap instruction is halted and security exception is raised. This group of instructions is the only group that has both propagation and checking rules implemented. SWAP instruction, however, doesn't need the propagation rules since both the tags have to be set.

Opcode	Name
LDSTUB (LDSTUBA)	Atomic Load-Store Unsigned Byte (in Alternate space)
SWAP (SWAPA)	Swap r Register with Memory (in Alternate space)

Table 6-3: Implemented LDSTUB and SWAP instructions

6.3.3. Rules for Group 4 instructions

CPOP1 and CPOP2 are implemented in UMC to control the tag engine. The coprocessor should be enabled at the beginning of the UMC technique execution. The CPOP1 instruction is responsible for enabling or disabling the coprocessor. If the execution of an instruction causes a security exception, then the coprocessor is turned off. By turning off the coprocessor no other instruction can produce another exception and all will be executed executed after the security exception. Also, as we discussed before, tag engine can be turned on and off from user's program by calling `_rtems_tag_enable()` and `_rtems_tag_disable()` functions. Calling the `_rtems_tag_enable()` function from the user's program causes the execution of the `tag_dispatch_interface()`. However calling the `_rtems_tag_disable()` function aborts execution of `tag_dispatch_interface()`.

There are also some instructions added using the CPOP2 instruction format. These instructions are technique specific, meaning that they are hardcoded individually for each security tagging technique. We used the instructions provided in Table 6-4 to set or clear the UMC tag or to return the tag value of a specific memory address.

CPOP1 and CPOP2 instructions always execute, whether the cpop flag is true or false, since these instructions are responsible for programming the coprocessor.

CPOP1	Opc Field Value	Address in HEX Format
CPOP_SET_UMC_TAG	3	0x87B84062
CPOP_CLEAR_UMC_TAG	4	0x87B84082
CPOP_GET_MEMORY_TAG	2	0x87B84042

Table 6-4: New CPOP2 instructions

CPOP2 instructions are executed as a result of any following function calls from the user's program:

- `_rtems_set_umc_tag(addr)`
- `_rtems_clear_umc_tag(addr)`
- `_rtems_get_memory_tag(addr)`

The above function calls are hardcoded functionalities that are added to the tagging technique so we can manipulate tags from user's program. Execution of any of the instructions from user's program sets the opc and address field of the CPOP2 instruction. Then the arguments are passed to a `cpop2()` function. Based on these arguments the `cpop2()` function decides to set or clear the UMC tag or return the memory tag.

6.3.4. Rules for Group5 instructions

All of the instructions except the Groups 1 to 4, doesn't have any effect on the UMC tagging technique, so execution of these instructions simply return from the `tag_dispatch_instruction()` without modifying any tag.

6.4. Conclusion

In this chapter we defined the instruction level implementation of UMC technique tag engine. We defined the data structure for memory locations and registers in the tag engine. We categorized SPARC instructions into different groups and defined propagation and checking rules for each group. We then defined the set of API's that can enable or disable the tag engine. API's also are capable of manipulating tags for registers and memory locations.

Chapter 7. Evaluation and Analysis

The assumption of increasing the probability of catching malicious attacks by adding security tags to the data is the foundation of current study, yet a set of test suits is needed to support the basic assumption. The evaluation of this work with respect to several hardware and software issues has been done by:

- Testing prevention of known vulnerable programs and real world vulnerabilities by running them against each related techniques. Each technique is developed to prevent special types of attacks. These attacks are described in the background section of each technique. For example, we can validate the effectiveness of DIFT technique against different types of buffer overflows. To demonstrate this, we have performed a series of tests on the buffer overflow vulnerable code using DIFT technique to evaluate its effectiveness.

- Evaluating the effectiveness of propagation and checking instructions that have been developed based on the rules for each technique. This evaluation can be done by designing a technique-based test suite and running it against the related technique.

- Running the RTEMS test suites for each of the techniques. In this way we are able to compare the performance of DIFT, BC and UMC on the same set of test suites.

7.1. Implementation Testing and test case design

In this section, we describe each set of tests that has been developed to validate each technique based on the desired implementation notes. We picked different groups of instructions as

described in each technique and ran the test against them. We also defined a set of propagation and checking rules for each technique.

7.1.1. DIFT Rule- verification Testing

As we discussed in Chapter 4, DIFT rules are defined for 5 different categories of instructions. We designed a set of test suites to cover all propagation and checking rules for these 5 categories. We also break down the testing into more precise testing. Precise testing feeds in all the permutation of input tags to the test suite, calculates the output tag and compares it with the desired output tag. Figure 7-1 shows a sample of test implemented for one of the category of instructions in DIFT.

We test Category 1 branch and call instructions by implementing test cases that include decision statements. We also implemented different test cases with multiple outcomes of decision statements to verify that all propagation and checking rules for Category 1.

Category 2 which defines ALU instructions has the most instructions among the DIFT categories. Although running each of the instruction in this category forwards the execution to the same propagation and checking routine, we implemented different test cases to validate the consistency in each instruction testing result. Each test case has a set of instructions which belong to this category. Then we compared the results of each instruction with the desired DIFT result.

Category 3 defines rules for load and store instructions. We tested this category by manipulating arrays with both tagged and untagged values. In Figure 7-1 we can see a sample of code that has both load and store instruction implemented. In Figure 7-2, we included the assembly code for the highlighted portion of the C code. Through the highlighted code the tag engine is turned on and a tagged value tries to access one element of an array. This case causes an exception since this test case violates the DIFT rules.

C code:

```
volatile uint32 value1 = 2;

int main(void)
{
  int array[3];

  tag_t tag1,tag2;
  tag_t new_tag;

  _rtems_set_dift_tag((addr_t)&value1);


  printf("Turning on the tag engine \n");

  asm(CPOP_TURN_ON_TAGGING);

     array[value1] = 4;

  asm(CPOP_TURN_OFF_TAGGING);

  return 0;
```

Figure 7-1: Sample C code for Testing

Assembly code:

```
! 25 "test.c" 1
      .word 0x81B00000
! 0 "" 2
      .loc 1 27 0
      mov   4, %g2
      ld    [%l0+%lo(value1)], %g1
      sll   %g1, 2, %g1
      add   %fp, %g1, %g1
      st    %g2, [%g1-12]
      .loc 1 29 0
! 29 "test.c" 1
      .word 0x81B00020
```

Figure 7-2: Corresponding Assembly code for the test

Results:

```
in function tag_check_st_imm address for storing is: 1 and
dift tag for source1 is:1
EXCEPTION HAPPEND BECAUSE OF STORE INSTRUCTIONS WITH SOURCE2
AS IMMEDIATE
Unexpected trap (40) at address 0x020012B4
```

Figure 7-3: Result of running C code inside DIFT tag engine

7. Buffer overflow detection with DIFT

We ran different scenarios of buffer overflow tests to determine the effectiveness of DIFT. We took samples of the buffer overflow code from "Diagnostic Test Suite for evaluating buffer overflow detector" from software assurance reference database [15].

We also ran a couple of tests in a case that a tagged value has been used indirectly to calculate the result address for storing data. DIFT rules also catch this type of tests. A sample of the buffer overflow tests is shown in Figure 7-4 and the corresponding result for the code is shown in Figure 7-5.

C code:

```
volatile uint32 value1 = 123;
volatile uint32 value2 = 456;
int main(void)
{
  tag_t tag1,tag2;
  tag_t new_tag;
     _rtems_set_dift_tag((addr_t)&value1);

 uint32 *value1_ptr = &value1;
 _rtems_set_dift_tag((addr_t)&value2);

 asm(CPOP_TURN_ON_TAGGING);
     *(value1_ptr + value2) = 3;
 asm(CPOP_TURN_OFF_TAGGING);
}
```

Figure 7-4: Buffer Overflow test

Results:

```
in function tag_check_st dift tag for source1 is:0 and dift
tag for source2 is:1
EXCEPTION HAPPEND BECAUSE OF STORE INSTRUCTIONS WITH SOURCE2
AS REGISTER
Unexpected trap (40) at address 0x020012C0
```

Figure 7-5: Buffer Overflow result

7.1.2. UMC Rule- verification Testing

Test suites are developed to evaluate UMC rules that are described in Chapter 6. One set of test suites relates the tag propagation by initializing a variable through other initialized variable. In Figure 7-6, the C code related to this example is shown. Figure 7-7 describes the assembly code for the code section highlighted in Figure 7-6.

Note that using non volatile variables causes the system to do some optimization to remove the need of initializing variables. To make sure that we fully tested the functionality of UMC testing we should disable any compiler optimization by using volatile variables.

C code:

```
int main (void)
{
volatile uint32 value1 ;
  value1 = 123;
  int new;

  _rtems_set_umc_tag((addr_t)&value1);

 asm(CPOP_TURN_ON_TAGGING);

new = value1;

asm(CPOP_TURN_OFF_TAGGING);
}
```

Figure 7-6: UMC tag propagation test

Assembly code:

```
      .word 0x81B00000
! 0 "" 2
      .loc 1 19 0
      ld    [%fp-4], %g1
      st    %g1, [%fp-8]
.LLVL1:
      .loc 1 21 0
! 21 "init.c" 1
      .word 0x81B00020
```

Figure 7-7: UMC tag propagation Assembly code

Results:

```
In function _rtems_set _umc_tag, value1 address is:33719404
cpop is set to true
cpop is set to false

the result tag for new is:1
TEST PASSED
```

Figure 7-8: UMC tag propagation result

The other set of tests for UMC are designed base on initializing arrays. We initialized the first element in an array using `_rtems_set_umc_tag()`function. We used the first element in the array to initialize the next element. Each element in an array is initialized with the content of previous array element. We initialized the first element in an array and checked the tag value of the last element to see how UMC tags get propagated. This test verifies a memory address initialization results in initializing its corresponding tag value. Once a memory address is initialized; the corresponding tag keeps its value throughout the program.

7.1.3. MBC Rule- verification Testing

Bound checking technique rules are some sort of combination of UMC rules and DIFT rules. Memory access rules are implemented like UMC while the other instructions' rules are implemented like DIFT with modification.

Local variables are stored below the frame pointer [%fp] and are accessed with negative offsets. Parameters start at [%fp + 68]. In the following test we implemented a case to copy content of one array element to the other. Both arrays are initialized with the same location color and pointer color. We used an inline assembly code in our test. The general format of using Inline assembly code is shown in Figure 7-9.

Assembly code:

```
Asm (“assembly code”
: output operands
: input operands
:list of clobbered registers
);
```

Figure 7-9: Inline Assembly code general format

In the inline assembly, having input/output operands and clobbered registers is optional. In the inline assembly section of the code we performed a simple instruction which doesn’t affect any other instruction. Running the instruction informs GCC about using all registers except frame pointer and stack pointer. So GCC moved all active data from program out of these registers.

The assembly code in Figure 7-11 only has the corresponding code in between turning on and off the tag engine. As we can see, it accesses the frame pointer to load a value. Since the frame pointer hasn’t been initialized with the proper pointer color tag, the program throws exception. The result is shown in Figure 7-12.

C code:

```
int main (void)
{

int a[30];
int b[30];

int i;

for (i=0 ; i<10 ; i++)
{
_rtems_set_location_color((addr_t)&a[i],3);
}

 for (i=0 ; i<10 ; i++)
{
_rtems_set_location_color((addr_t)&b[i],3);
}

   asm volatile("add %%l0,%%l0,%%l0\n\t"

         ::: "g0","g1","g2","g3","g4","g5","g6","g7",

         "l0","l1","l2","l3","l4","l5","l6","l7",

         "i0","i1","i2","i3","i4","i5","i7",

         "o0","o1","o2","o3","o4","o5","o7");


asm(CPOP_TURN_ON_TAGGING);

a[1] =b[1];

asm(CPOP_TURN_OFF_TAGGING);


exit(0);

}
```

Figure 7-10: sample BC code improper handling tag for Frame Pointer

Assembly code:

```
      .word 0x81B00000
! 0 "" 2
      .loc 1 56 0
      ld    [%fp-236], %g1
      st    %g1, [%fp-116]
      .loc 1 58 0
! 58 "init.c" 1
      .word 0x81B00020
```

Figure 7-11: Assembly code for corresponding C code

Results:

```
EXCEPTION HAPPEND BECAUSE OF LOAD INSTRUCTIONS WITH SOURCE2
AS IMMEDIATE
...
 Hit vector 0x28!! @ PC = 0xopc is equal to:1
cpop is set to false
20012E4 with NPC = 0x20012E8
```

Figure 7-12: Result of improper handling tag for Frame Pointer

In the next test, we slightly modified the above code by adding highlighted piece of inline assembly code in Figure 7-13. The code in Figure 7-13 initializes frame pointer with the same color tag value as both arrays.

C code:

```
int main (void)
{
int a[30];
int b[30];
int reg = 30;
int op2 = 3;
int i;

int reg = 30;
int op2 = 3;

for (i=0 ; i<10 ; i++)
{
_rtems_set_location_color((addr_t)&a[i],3);
}

 for (i=0 ; i<10 ; i++)
{
_rtems_set_location_color((addr_t)&b[i],3);
}

   asm volatile("add %%l0,%%l0,%%l0\n\t"

         ::: "g0","g1","g2","g3","g4","g5","g6","g7",

         "l0","l1","l2","l3","l4","l5","l6","l7",

         "i0","i1","i2","i3","i4","i5","i7",

         "o0","o1","o2","o3","o4","o5","o7");


       asm("mov %0, %%g1\n\t"
            "mov %1, %%g2\n\t"
            CPOP_SET_REGISTER_POINTER_COLOR
             :
            :"r"(reg),"r"(op2)
            :"g1","g2");


 asm(CPOP_TURN_ON_TAGGING);

a[1] =b[1];

asm(CPOP_TURN_OFF_TAGGING);
}
```

Figure 7-13: C code proper handling tag for Frame Pointer

Results:

```
TEST PASSES !!
```

Figure 7-14: Results of proper handling tag for Frame Pointer

The highlighted code in Figure 7-13 is the part that was added to the previous test case. As shown in Figure 7-14, this test case passed.

Another set of test cases relates to uninitialized memory. This test initializes an array then it tries to access array element located outside boundaries of array.

C code:

```
int main (void)
{
int i;

int a;

volatile int b[10];
int reg, op2;

for(i= 0 ; i<10 ; i++)
{

_rtems_set_pointer_color((addr_t)&b[i],3);
_rtems_set_location_color((addr_t)&b[i],3);
b[i] = 5;
 }

reg = 30;op2 = 3;

asm(CPOP_TURN_ON_TAGGING);
a = b[11];

asm(CPOP_TURN_OFF_TAGGING);

exit(0);

}
```

Figure 7-15: C code for out of bound memory access

```
Results:
EXCEPTION HAPPEND BECAUSE OF LOAD INSTRUCTIONS WITH SOURCE2
AS IMMEDIATE
...
 Hit vector 0x28!! @ PC = 0xopc is equal to:1
cpop is set to false
2001310 with NPC = 0x2001314
```

Figure 7-16: Results of out of bound memory access

This is a good test case that shows how out of bound memory access can be tracked and captured using memory bound checking technique.

7.2. Performance evaluation

To evaluate performance of each tagging technique, we added tagging statistics to SIS for each technique. The common ground of all techniques is propagation and checking of memory and registers tags during execution of instructions. So we added statistic counters for tag propagation, tag check, memory tag check and memory tag set. Running each test case gives statistics for each one of the values. We gathered the statistics data. The results show what percent of instructions are tag propagation or tag check. Also what percentages of instructions try to access memory to either check or set the tag values.

7.2.1. Performance Evaluation for DIFT

We ran tests for DIFT, BC and UMC on the same set of RTEMS applications. These applications include check for uboot support in bsp and print application. We ran each set of test for each technique. Then we calculate the percentage of tag propagation, tag check, memory tag set and memory tag check for each application. We calculate the average of each of the statistics through different applications. The results of study for DIFT, UMC and BC techniques are shown in Figure 7-17 through Figure 7-19 respectively. The Figures shows on the Y-axis the instructions percentage for each technique. The X-axis shows set of instruction categories. The Figures show how each technique spend resources.

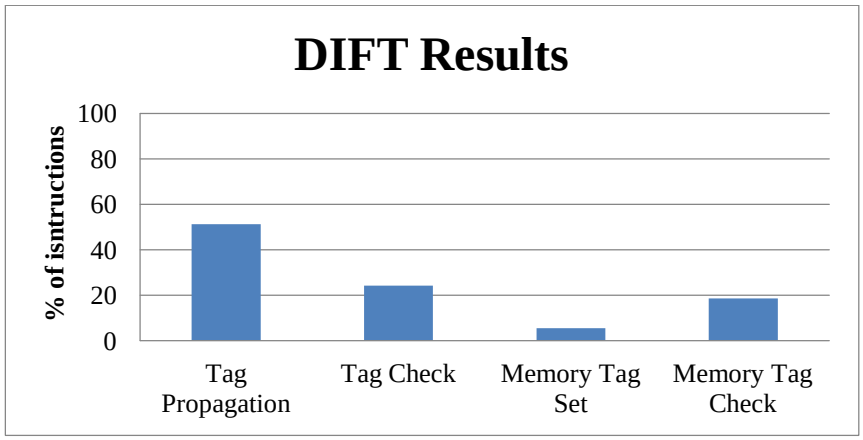

Figure 7-17: Performance evaluation for DIFT running RTEMS applications

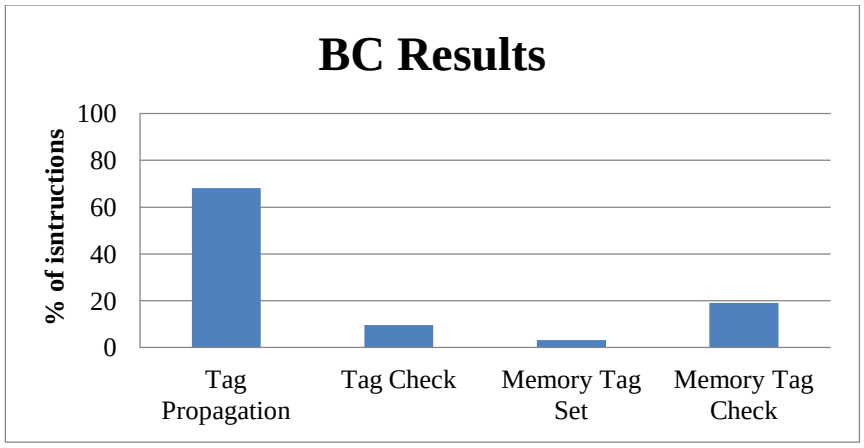

Figure 7-18: Performance evaluation for BC running RTEMS applications

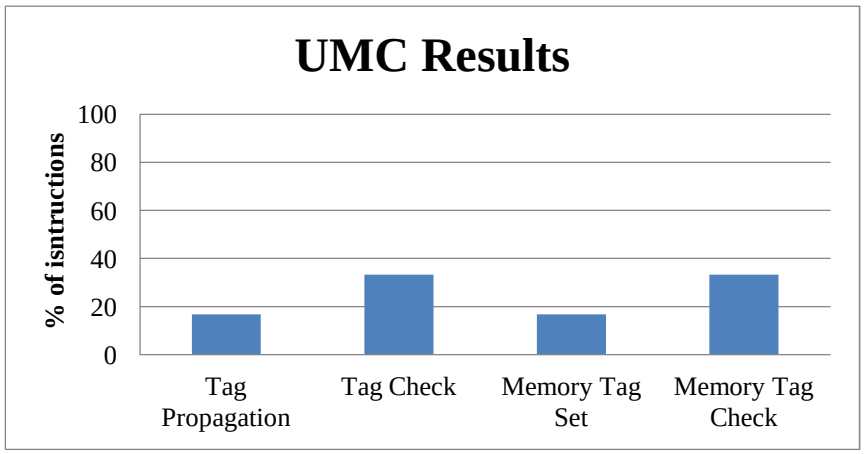

Figure 7-19: Performance evaluation for UMC running RTEMS applications

The highest percentage of instructions that executed in both DIFT and BC's tag engine are tag propagation. UMC tag propagation stat is the same as UMC memory tag set instructions. Also in UMC memory tag check is the same as tag check since loading a value from memory is the only instructions that uses tag check.

7.3. Analysis and Results

Figure 7-20 shows on the Y-axis the tag engine overhead percentage for each technique. The X-axis shows different techniques. To calculate the overhead of running each technique, we first ran RTMES applications and captured number of executed instructions in the absence of security techniques. Then we enabled each technique and calculated RTEMS applications instruction execution in the existence of each technique. Running the tests gives us the amount of overhead each technique has. DIFT has overhead of 23.6 %, UMC has 7.9% and BC has 41.5%.

As we see BC has the highest overhead, since it has propagation and checking rules for memory access checking as well as dataflow checking. UMC has only memory access checking so it has the lowest amount of overhead among all techniques. DIFT has dataflow checking which includes propagation of tags through ALU instruction and checking of tags through jump instructions. Since BC has both features of UMC and DIFT, BC's overhead is close to UMC and DIFT overhead combined.

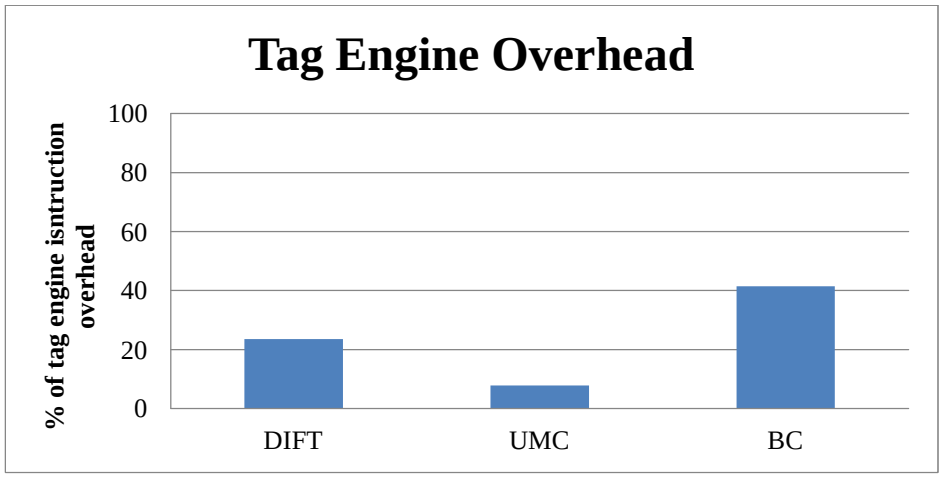

Figure 7-20: Tag engine overhead

7.4. Conclusion

In this chapter we showed set of test suites that test propagation and checking rules for each technique. We then showed the result of running the real world attacks such as buffer overflow and out of bound memory access for each technique. We then showed the result of running all techniques on the same set of application. We represent the overhead of running each technique and then compared the result of all technique.

Chapter 8. Conclusion

8.1. Conclusion

Today with the explosive growth of computer and network technology, we rely on computer systems to manage sensitive data and personal information. In recent years promising mechanisms and techniques developed to enhance computer and network system security. Some of the existing security techniques track instruction flow at run time.

Our goal was to design a framework which has run time security techniques to prevent runtime attacks. As a first step toward this goal, this thesis gives an extensive background on the most common run time security techniques such as DIFT, BC and UMC which are implemented to prevent buffer overflows and illegal memory accesses. We then give a description of SPARC Instruction Architecture and how security techniques can be implemented in SPARC using SPARC instruction Simulator. The validation of security technique implementation is completed by running test suites and RTEMS applications.

We developed tagging schemes that will be implemented the ERC32 processor, which is a SPARC variant. The understanding of SPARC instruction architecture gives us a better view of how each technique can be implemented for different instructions. According to each technique, we divide instructions into different logical groups and then defined propagation and checking rules for them. Dividing instructions into different groups gives us the ability to design test suites for each group as well. So we were able to check how each technique propagates and check security tags.

We validate each technique by running test suites that are designed specifically for it. Recall that each technique is capable of detecting set of attacks. We ran attacks for each technique to see if the technique is able to capture it.

To evaluate performance of each tagging technique, we added tagging statistics for tag propagation, tag check, memory tag check and memory tag set. Running each test case gives statistics for each one of the values.

We ran tests each technique on the same set of RTEMS applications as well. Then we calculate the average of each of the statistics of percent of instructions that got executed in the tag engine to the whole executed instructions through different applications.

BC requires the biggest data structure among all techniques. It assigns each memory and register with 4-bit location tag and 4-bit pointer tag. DIFT and UMC assign a 1-bit tag for memory and registers. Running the test suites show that BC is capable of detecting more attacks. It also has the highest overhead among all techniques. DIFT is capable of detecting buffer overflow attacks and it comes in the second place in the matter of instruction overhead. UMC has the lowest overhead and it is capable of detecting uninitialized memory accesses.

In summery the work presented in this thesis provides insight to the run time security techniques. While we hope to see these techniques implemented in a frame work as an architecture solution, we expect this research to be a useful resource for studying and evaluating security techniques.

8.2. Future Research

The previous section has summarized the work of this thesis. However, as in any research effort, there still are a number of areas where further work could enhance the prevention of misusing code vulnerabilities. Runtime taint tracking will continue to be an important tool for the security research. The evaluation that is conducted in the previous chapters revealed several opportunities to improve the implemented framework. These are discussed below:

1. **Run the SIS on a multi core processor or on multi-processor system**

SIS simulator supports a single core processor. We wish to modify it to support simulation of multi-core processor. This involves the design of multi-core simulator. Multi-core simulator will include memory model, inter processor communication and simulating time.

2. **Implement security tagging rules for trap instructions and trap handler**

We currently didn't implement propagation and checking rules for Ticc and RETT (trap instructions). We need to evaluate these instructions to verify if we are able to implement rules for them. At the same time we need to investigate what causes trap in the system to make sure that rules are implemented correctly.

3. **Implement security tagging rules for floating point instructions**

We have assigned tagging rules to Integer Instructions in the SPARC architecture. But we didn't add the support for floating point instructions. We need further effort to implement tag engine rules for DIFT, BC and UMC and evaluate each technique in the existence of Integer instructions' rules as well as floating point instructions.

4. **Add the capability of having a network or simulated network to test techniques in the existence of network accesses.**

At this time each test suite implemented for the technique manually set the tag values for memory locations and registers. We wish to expand the simulator and add the capability of network simulation to it. Having this capability, we are able to send traffic from network channels, and for DIFT, we can label couple of network channels as malicious input. In this way we can verify how DIFT can handle malicious IO channels.

5. **Implement these techniques for other instruction architectures such as ARM**

At this time these technique are only implemented for SPARC instruction set. We wish to implement each technique for other instruction architectures such as ARM. This requires modifying each technique to support different architectures.

6. **Implement tag cache simulator and add it to the tag engine for each technique**

Accessing tags in the memory is neither cheap nor fast. By implementing the tag cache simulator, we can store the tags for the recent accessed memory locations and registers and access them as needed. I already implemented the simulator cache inside SIS. By adding the tag cache simulator we can recalculate the overhead of memory tag set and check.

Chapter 9. Bibliography

1] B. R. Rowe and I. D. Pokryshevskiy, "Economic Analysis of Inadequate Cyber Security Technical Infrastructure," with Albert N. Link, University of North Carolina at Greensboro, Douglas S. Reeves, North Carolina State University, 3040 E. Cornwallis Road, Research Triangle Park, NC 27790, February 2013.

2] "Heartbleed OpenSSL vulnerability," National Cybersecurity and Communications Integration Center, April 2014.

3] J. Steinberg, "Massive Internet Security Vulnerability -- Here's What You Need To Do," 10 April 2014. [Online]. Available: http://www.forbes.com/sites/josephsteinberg/2014/04/10/massive-internet-security-vulnerability-you-are-at-risk-what-you-need-to-do/. [Accessed May 2014].

4] B. M. (MITRE), M. B. (SANS), A. P. (SANS) and D. K. (SANS), "2011 CWE/SANS Top 25 Most Dangerous Software Errors," The MITRE Corporation, 13 September 2011. [Online].

5] D. S. Henson, "Support for TLS/DTLS heartbeats," 31 December 2011. [Online].

6] S. M. Kerner, "Heartbleed SSL Flaw's True Cost Will Take Time to Tally - See more at: http://www.eweek.com/security/heartbleed-ssl-flaws-true-cost-will-take-time-to-tally.html#sthash.qLJdmngZ.dpuf," 19 April 2014. [Online].

7] G. E. Suh, J. W. Lee, D. Zhang and S. Devadas, "Secure program execution via dynamic information flow tracking," *Acm Sigplan Notices,* vol. 39, no. 11, ACM 2004.

8] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *In Proc. of the Winter 1992 USENIX Conference*.

9] J. Clause, I. Doudalis, A. Orso and M. Prvulovic, "Effective memory protection using dynamic tainting," in *In proceeding of the 22nd International Conference on Automated Software Engineering, 2007*.

10] *The SPARC Architecture Manual,* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

11] *sis - SPARC instruction set simulator version 3.0.1,* European Space Research and Technology Centre, 1999.

12] M. Ramström, J. Höglund, B. Enoksson and R. Svenningsson, "32-BIT MICROPROCESSOR AND COMPUTER DEVELOPMENT PROGRAMME," 1997.

13] *TSC691E Integer Unit User's Manual for Embedded Real time 32–bit Computer (ERC32)*, TEMIC Semiconductors.

14] M. Ramström, B. Törnberg and R. Svenningsson, *MEC Device Specification*, TEMIC Semiconductors, 1997.

15] "NIST National Institute of Standards and Technology Test Suits," [Online]. Available: http://samate.nist.gov/SARD/testsuite.php.