TECHNICAL UNIVERSITY OF CRETE

Efficient hardware support for Dynamic Information Flow Tracking (DIFT) in the LEON processor

by

Vaios Taxiarchis

A thesis submitted in partial fulfillment of the requirements
for the diploma thesis of Electronic and Computer Engineering

in the
Microprocessor and Hardware Lab
School of Electronic and Computer Engineering

THESIS COMMITTEE
Professor Dionisios Pnevmatikatos, *Thesis Supervisor*
Professor Apostolos Dollas
Assistant Professor Vasilis Samoladas

June 2016

"A common mistake people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools."

D. Adams

Abstract

Computer security is of growing importance due to the increasing reliance of computer systems in most societies. Software vulnerabilities can be seen as flaws or weaknesses in the system that can be exploited by an attacker in order to alter the normal behavior of the system. As a consequence, vulnerabilities in the production of software make necessary to have tools that can help programmers to avoid or detect them in the development of the code. Existing defenses, such as canaries or web application firewalls, often suffer from compatibility issues or are easily evaded by a professional attacker. Security defenses should focus on safety, speed, flexibility, practicality and end-to-end coverage. Recent researches have shown that Dynamic Information Flow Tracking (DIFT) is a promising technique for detecting a wide range of security attacks. DIFT tracks the flow of untrusted information within a programs runtime by extending memory and registers with tags. With hardware support, DIFT can provide comprehensive protection against input validation attacks with minimal performance overhead. Thus, in relation to our on-going research on vulnerability detection, this thesis presents the design and implementation of a hardware platform for DIFT, based on the synthesizable LEON processor. The specific platform is an extension of the LEON processor with additional instructions for data-flow integrity support. Specifically it can track tag information along data within the processor pipeline and through computations, if we install appropriate Linux-based operating system. The modified processor protects applications from low-level memory corruption exploits (such as buffer overflows or format string attacks) and can be extended so as to protect from high-level semantic vulnerabilities (such as SQL injections or cross-site scripting) in future work. The processor includes also support to trapping when unsafe data are used as pointers to prevent information leakage.

Keywords: Computer Security, Software Vulnerabilities, Dynamic Information Flow Tracking, Processor Architecture

Acknowledgements

During my undergraduate studies, I am deeply indebted to many people for their contributions towards this thesis.

First of all, I am profoundly grateful to my supervisor, Professor Dionisios Pnevmatikatos, for his persistent and patient mentoring, our excellent collaboration, but most for giving me the opportunity to work on this field of research for the purpose of this thesis. His continuous support and guidance helped me in all the time of developing and writing of this thesis.

I would also like to thank Professor Apostolos Dollas and Assistant Professor Vasilis Samoladas for accepting to be in my committee.

Furthermore, I would like to acknowledge Nick Christoulakis and Giorgos Christou from Foundation for Research & Technology-Hellas (FORTH) for their assistance whenever needed and for our cooperation during my thesis.

Finally, I would like to acknowledge my parents, Apostolia and Dimitris, as well as my beloved younger sister, Foteini for always being my role models in matters of morals and principles, for loving me and for supporting my decisions.

Contents

List of Figures

List of Tables

To my family...

Chapter 1

Introduction

It is widely recognized that computer security is a critical problem with far-reaching financial and social implications [42]. Despite significant development efforts in computer security, existing security tools do not provide reliable protection against attacks, such as worms and viruses that target vulnerabilities and sometimes target multiple vulnerabilities [43]. Apart from memory corruptions, such as Buffer Overflow Attacks, we need to focus on high-level exploits, such as Directory Traversal, Command Injection, SQL Injection and Cross-site Scripting (XSS). Cybercrime costs the U.S. economy hundreds of billions of dollars annually [44], but researches on system security and attack prevention is very timely.

In every computer system, we use applications that process untrusted input and as a consequence security exploits occur. Recent research has established Dynamic Information Flow Tracking (DIFT) [17, 18] as a promising platform for detecting a wide range of security attacks. DIFTs main idea is tracking the flow of untrusted information within a programs runtime by extending memory and registers with tags. Thus, DIFT associates tags with memory and resources in the system, and then uses these tags to maintain information about the reliability of the corresponding data. Tags are initialized in accordance with the source of the data. A typical tag initialization policy would be to mark data arriving from untrusted sources such as a tainted network, while keeping files owned by the user untainted.

Typical examples of DIFT implementations are JVM used to prevent Java servlet vulnerabilities and other DIFT-based implementations in hardware to prevent wide range of memory corruption exploits. Most scripting languages, such as PHP [19] and Java [20], Dynamic Binary Translators [21], and hardware platforms [18], use implementations of DIFT technique. DIFT is fast as evinced by some of the high-performance DIFT systems, while works on unmodified binaries or bytecode, without requiring any source

code or debugging information. However, software DIFT platforms are neither fast nor practical when protecting legacy binaries as they result in high performance overheads, such as 3x [25] up to 37x [26] and restrict applications to a single core. Hardware DIFT platforms are fast and practical, but in most case support only a single, fixed policy (not flexible and not safe). An ideal DIFT platform should satisfy all of these criteria, combining the best of both software and hardware approaches.

1.1 Thesis Contribution

We explore the potential of hardware DIFT, in order to provide comprehensive protection from a wide variety of attacks. We focus on low-level vulnerabilities, such as Buffer Overflow prevention using Pointer Injection (PI) policy, but we hope to continue this project to fully support high-level vulnerabilities as well. The scope of this thesis is to describe the design and implementation of a hardware platform for DIFT, based on the synthesizable LEON processor [38]. The specific platform is an extension of the LEON processor with additional instructions for data-flow integrity support. The contribution of this thesis is summarized in the following bullets:

- It presents an efficient hardware DIFT platform that prevents attacks on low-level vulnerabilities. This hardware platform with DIFT support provides transparent, fine-grain management of security tags and low performance overhead for user code and data.
- It explores the synthesizable LEON3 processor, describing the full functionality of the LEON3 core and the whole template design, based on the use of VHDL generics. It also presents the design and implementation of a hardware platform for DIFT with additional instructions for data-flow integrity support, based on the synthesizable LEON3 processor.
- It discusses the experimental evaluation of the hardware platform with DIFT support. We evaluated DIFT policies and successfully prevented low-level vulnerabilities, such as buffer overflow attacks, using tainted binaries written in C or Assembly language.

1.2 Thesis Organization

In Chapter 2 we present all the background material needed for this thesis. We present modern security vulnerabilities and existing defensive countermeasures. This chapter

describes each input validation vulnerability and any existing non-DIFT defenses. In Chapter 3 we provide an overview of DIFT and present the DIFT-related research. We analyze DIFT technique in different implementation methods, summarizing and comparing the best techniques and methodologies. In Chapter 4 we offer a detailed description of the basic architecture and functionality of the LEON3 processor, in which our thesis is based on. In Chapter 5 we present the design and implementation of hardware platform for DIFT, describing at first the installation and configuration of the host platform, and secondly the basic design of our architecture for DIFT support in the LEON3 processor. We describe the implementation of our scheme in the LEON3 processor, explaining all the necessary modifications done and we summarize and conclude the chapter comparing the simulating results by evaluating the whole platform. Finally, in Chapter 6 we present our conclusions and future work of this platform.

Chapter 2

Background

This chapter presents the input validation vulnerabilities in modern software. Input validation refers to the process of validating all the input to an application before using it. Input validation is absolutely critical to application security, and most application risks involve tainted input at some level. Many applications do not plan input validation, and leave it up to the individual developers. This is a recipe for disaster, as different developers will certainly all choose a different approach, and many will simply leave it out in the pursuit of more interesting development.

First, we describe the vulnerability itself, which secure programming techniques can be used to prevent the vulnerability, and subsequently what defensive countermeasures exist to prevent attacks on vulnerable applications. We describe all commonly exploited input validation security vulnerabilities, from low-level buffer overflow and format string attacks in legacy languages to high-level SQL injection and cross-site scripting in modern web applications. Existing defensive countermeasures evaluated according to requirements of ideal security solutions:

- Safety - How well a technique resists being defeated when faced with a competent attacker
- Speed - How high the performance overhead of a technique is
- Flexibility - How applicable a technique is to a range of different security flaws and vulnerabilities
- Practicality - How easy it is to apply a technique to modern software settings, where source code access may not be available and legacy code may be present
- End-to-end coverage How easy it is to scale beyond individual components, and offer full-system protection.

2.1 Buffer Overflow

In computer security and programming, a buffer overflow, or buffer overrun, is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations. This is a special case of the violation of memory safety. Buffer overflows can be triggered by inputs that are designed to execute code, or alter the way the program operates. This may result in erratic program behavior, including memory access errors, incorrect results, a crash, or a breach of system security.

Thus, they are the basis of many software vulnerabilities and can be maliciously exploited. Programming languages commonly associated with buffer overflows include C and C++, which provide no built-in protection against accessing or overwriting data in any part of memory and do not automatically check that data written to an array (the built-in buffer type) is within the boundaries of that array. Bounds checking can prevent buffer overflows.

2.1.1 Technical Description

A buffer overflow occurs when data written to a buffer also corrupts data values in memory addresses adjacent to the destination buffer due to insufficient bounds checking. This can occur when copying data from one buffer to another without first checking that the data fits within the destination buffer.

```
#include <string.h>

void foo (char *bar)
{  char  c[12];
   strcpy(c, bar);  // no bounds checking }

int main (int argc, char **argv)
{  foo(argv[1]); }
```

FIGURE 2.1: C code showing buffer overflow vulnerability. The canonical method for exploiting a stack based buffer overflow is to overwrite the function return address with a pointer to attacker-controlled data (usually on the stack itself) [1, 2].

An example of buffer overflow vulnerability is presented in Figure 2.1. This code takes an argument from the command line and copies it to a local stack variable c. This

works fine for command line arguments smaller than 12 characters, as you can see in Figure 2.2 (b). Any arguments larger than 11 characters long will result in corruption of the stack. The maximum number of characters that is safe is one less than the size of the buffer here because in the C programming language strings are terminated by a zero byte character. A twelve-character input thus requires thirteen bytes to store, the input followed by the sentinel zero byte. The zero byte then ends up overwriting a memory location that's one byte beyond the end of the buffer.

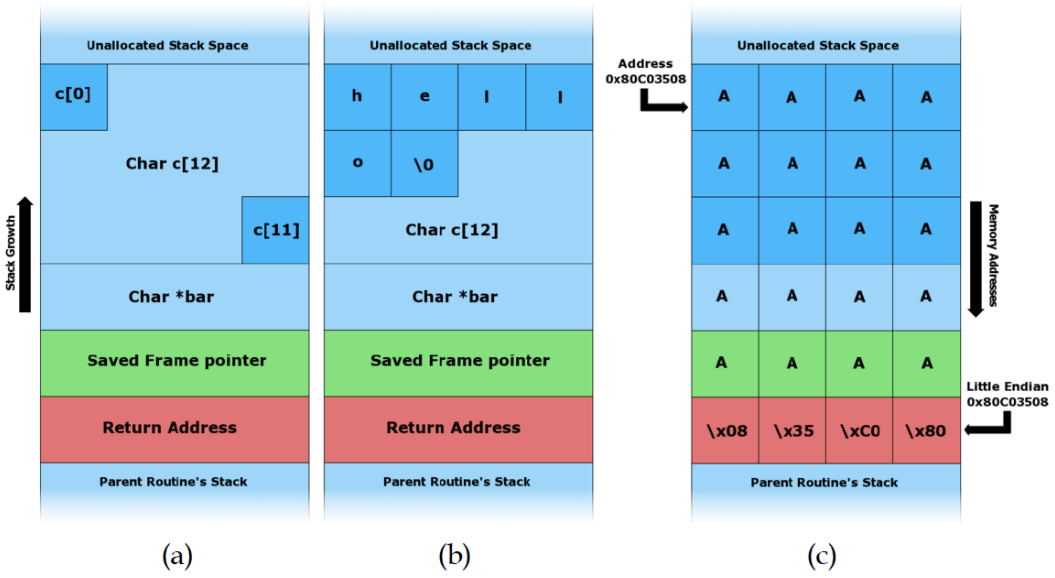

FIGURE 2.2: The program stack in foo() with various inputs.
(a) Before data is copied. (b) "hello" is the first command line argument.
(c) "AAAAAAAAAAAAAAAAAAAA\x08\x35\xC0\x80" is the first command line argument.

Notice in Figure 2.2 (c), when an argument larger than 11 bytes is supplied on the command line `foo()` overwrites local stack data, the saved frame pointer, and most importantly, the return address. When `foo()` returns it pops the return address off the stack and jumps to that address (i.e. starts executing instructions from that address). Thus, the attacker has overwritten the return address with a pointer to the stack buffer `char c[12]`, which now contains attacker-supplied data. In an actual stack buffer overflow exploit the string of `"A"`'s would instead be shellcode suitable to the platform and desired function. If this program had special privileges (e.g. the SUID bit set to run as the superuser), then the attacker could use this vulnerability to gain superuser privileges on the affected machine [1].

2.1.2 Countermeasures

Buffer overflow protection refers to various techniques used during software development to enhance the security of executable programs by detecting buffer overflows on stack-allocated variables, and preventing them from causing program misbehavior or from becoming serious security vulnerabilities. Modern countermeasures can be divided into the following categories: canaries, address space layout randomization, executable space protection and bounds checking.

Canaries

Canaries or canary words are known values that are placed between a buffer and control data on the stack to monitor buffer overflows [3]. When the buffer overflows, the first data to be corrupted will usually be the canary, and a failed verification of the canary data is therefore an alert of an overflow, which can then be handled, for example, by invalidating the corrupted data.

Canaries have been implemented in practice using a security-aware compiler. The most popular Linux [4] and Windows [5] compilers support stack-based canaries for preventing buffer overflows. Unfortunately, canaries are not *practical* because they require source code access, and they change the memory layout of the address space by inserting words before security critical data. Canaries are also not *safe* because they only protect certain designated security critical metadata.

Address Space Layout Randomization

Address Space Layout Randomization (ASLR) is a computer security technique involved in protection from buffer overflow attacks. In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries. This technique is deployed on Linux, Windows, OS X, Solaris, FreeBSD, OpenBSD, DragonFly BSD, Android and iOS systems.

ASLR is based upon the low chance of an attacker guessing the locations of randomly placed areas. Security is increased by increasing the search space. Thus, ASLR is more effective when more entropy is present in the random offsets. Entropy is increased by either raising the amount of virtual memory area space over which the randomization occurs or reducing the period over which the randomization occurs. The period is typically implemented as small as possible, so most systems must increase VMA space

randomization. However, ASLR is not completely *safe* and can be bypassed in many practical situations. Any attack that overwrites non-pointer data will bypass ASLR [6]. ASLR is also not *practical* as it breaks backwards compatibility with legacy applications. Executables must often be recompiled so that they can support randomized remapping and loading.

Executable Space Protection

Executable Space Protection is the marking of memory regions as non-executable, such that an attempt to execute machine code in these regions will cause an exception. It makes use of hardware features such as the NX bit. If an operating system can mark some or all writable regions of memory as non-executable, it may be able to prevent the stack and heap memory areas from being executable. This helps to prevent certain buffer overflow exploits from succeeding, particularly those that inject and execute code, such as the Sasser and Blaster worms. These attacks rely on some part of memory, usually the stack, being both writable and executable; if it is not, the attack fails.

This approach is not *practical* as it breaks backwards compatibility with legacy applications that generate code at runtime, which occurs in languages such as Objective C, as well as just-in-time (JIT) virtual machine interpreters such as the Java Virtual Machine. Furthermore, this technique is not *safe* at all, and in fact can be completely evaded by attackers. Executable Space Protection only prevent buffer overflow attacks that inject code, but buffer overflow exploits can be crafted that do not require code injection capabilities.

Bounds Checking

Bounds Checking is any method of detecting whether a variable is within some bounds before it is used. It is usually used to ensure that a number fits into a given type (range checking), or that a variable being used as an array index is within the bounds of the array (index checking). A failed bounds check usually results in the generation of some sort of exception signal. Because performing bounds checking during every usage is time-consuming, it is not always done. Bounds Checking elimination is a compiler optimization technique that eliminates unneeded bounds checking.

Bounds Checking elimination is a compiler optimization useful in programming languages or runtimes that enforce bounds checking, the practice of checking every index into an array to verify that the index is within the defined valid range of indexes. Its goal is to detect which of these indexing operations do not need to be validated at runtime, and eliminating those checks. One common example is accessing an array

element, modifying it, and storing the modified value in the same array at the same location. Normally, this example would result in a bounds check when the element is read from the array and a second bounds check when the modified element is stored using the same array index. Bounds Checking elimination could eliminate the second check if the compiler or runtime can determine that neither the array size nor the index could change between the two array operations.

Unlike other buffer overflow countermeasures, this technique is *safe* so long as assembly code is used with correct, manually-specified wrappers, and all memory allocation functions are annotated by the programmer. However, this technique has never been employed by industry as it is not *practical*. Bounds checking compilers require source code access and have serious backwards compatibility issues. Performance overheads can also reach over 2x [7, 8], which is too high for performance-intensive production deployments. Unlike other techniques discussed in this section, bounds checking compilers are not widely employed in practice.

2.2 Format String Attack

Format String Attack is a type of software vulnerability, discovered around 1999, that can be used in security exploits. Previously thought harmless, format string exploits can be used to crash a program or to execute harmful code. The problem stems from the use of unchecked user input as the format string parameter in certain C functions that perform formatting, such as `printf()`. A malicious user may use the %s and %x format tokens, among others, to print data from the call stack or possibly other locations in memory. One may also write arbitrary data to arbitrary locations using the %n format token, which commands `printf()` and similar functions to write the number of bytes formatted to an address stored on the stack.

2.2.1 Technical Description

The Format String exploit occurs when the submitted data of an input string is evaluated as a command by the application. In this way, the attacker could execute code, read the stack, or cause a segmentation fault in the running application, causing new behaviors that could compromise the security or the stability of the system. The attack could be executed when the application does not properly validate the submitted input. In this case, if a Format String parameter, like %x, is inserted into the posted data, the string is parsed by the Format Function, and the conversion specified in the parameters is executed.

```
#include <stdio.h>

int main ()
{
  printf ("%s%s%s%s%s%s%s%s%s%s%s%s");
}
```

FIGURE 2.3: C code showing format string vulnerability. This method increases the possibility that the program will read an illegal address, crashing the program and causing its non-availability.

An example of format string vulnerability is presented in Figure 2.3. For each %s, `printf()` will fetch a number from the stack, treat this number as an address, and print out the memory contents pointed by this address as a string, until a NULL character (i.e., number 0, not character 0) is encountered. Since the number fetched by `printf()` might not be an address, the memory pointed by this number might not exist (i.e. no physical memory has been assigned to such an address), and the program will crash. It is also possible that the number happens to be a good address, but the address space is protected (e.g. it is reserved for kernel memory). In this case, the program will also crash.

2.2.2 Countermeasures

Some safe languages try to prevent format string vulnerabilities by allowing the format function to determine the type of its arguments, and comparing the type that was given as an argument to the function to the type of argument that the format specifier expects. Static analyzers try to detect format string vulnerabilities by checking that the format string argument to a format function is either a literal string or by doing taint analysis. Taint analysis marks all user input as tainted and will report an error when a variable that is expected to be untainted is derived from a tainted value. However, this proposal is not *practical* as it requires source code access for static analysis and recompilation of the application.

Some library wrappers for format functions exist to try and prevent the exploitation of format string vulnerabilities. FormatGuard [9] attempts to detect invalid format strings by counting the arguments the format specifier expects on the stack and comparing them to the amount of arguments that were really passed, if less arguments are passed than expected an error has occurred. Libformat [10] checks if the format string that was supplied to the format function does not contain any %n specifiers if the string

is located in writable memory. Furthermore, these approaches are not completely *safe*, as format string attacks may still use format string specifiers other than %n to read arbitrary memory addresses, allowing for information leaks. Format string information leaks can be used to exfiltrate sensitive information, or to undermine other security defenses such as Address Space Layout Randomization [11].

2.3 Directory Traversal Attack

A Directory Traversal or path traversal consists in exploiting insufficient security validation (or sanitization) of user-supplied input file names, so that characters representing "traverse to parent directory" are passed through to the file application programming interfaces (APIs). The goal of this attack is to order an application to access a computer file that is not intended to be accessible. This attack exploits a lack of security (the software is acting exactly as it is supposed to) as opposed to exploiting a bug in the code. Directory Traversal is also known as the `../` (dot dot slash) attack, directory climbing, and backtracking. Some forms of this attack are also canonicalization attacks.

2.3.1 Technical Description

In web applications with dynamic pages, input is usually received from browsers through GET or POST request methods. A typical example of vulnerable application in PHP code is presented in Figure 2.4. As you can see in Figure 2.5, an attack against this system could be to send the HTTP request, generating a server response, which allows attacker to access restricted directories and execute commands outside of the web servers root directory.

```
<?php
$template = 'red.php';
if (isset($_COOKIE['TEMPLATE']))
   $template = $_COOKIE['TEMPLATE'];
include ("/home/users/phpguru/templates/" . $template);
?>
```

FIGURE 2.4: A typical example of vulnerable application in PHP code.

The repeated `../` characters after `/home/users/ phpguru/templates/` has caused `include()`[1] to traverse to the root directory, and then include the Unix password file `/etc/passwd`[2]. Unix `/etc/passwd` is a common file used to demonstrate directory traversal, as it is often used by crackers to try cracking the passwords. However, in more recent Unix systems, the `passwd` file does not contain the hashed passwords. They are, instead, located in the shadow file which cannot be read by unprivileged users on the machine. It is however, still useful for account enumeration on the machine, as it still displays the user accounts on the system.

```
GET /vulnerable.php HTTP/1.0
Cookie: TEMPLATE=../../../../../../../../../../etc/passwd

HTTP/1.0 200 OK
Content-Type: text/html
Server: Apache

root:fi3sED95ibqR6:0:1:System Operator:/:/bin/ksh
daemon:*:1:1::/tmp:
phpguru:f8fk3j1OIf31.:182:100:Developer:/home/users/phpguru/:/bin/csh
```

FIGURE 2.5: A HTTP request and server response showing the requested password.

2.3.2 Countermeasures

Directory traversal vulnerabilities are difficult to precisely detect because the application is using legitimate file open library or system calls, but the filename may have been unsafely determined by attacker input. Without the ability to tell which parts of the program filename came from untrusted input, reliable detection of these attacks remains a challenge. Existing solutions detect anomalous filenames, or attempt to mitigate the damage done by a directory traversal attack by limiting access to filesystem resources on a per-application basis.

[1] The include statement includes and evaluates the specified file. When a file is included, the code it contains inherits the variable scope of the line on which the include occurs. Any variables available at that line in the calling file will be available within the called file, from that point forward. However, all functions and classes defined in the included file have the global scope.

[2] passwd is a tool on most Unix and Unix-like operating systems used to change a user's password. The password entered by the user is run through a key derivation function to create a hashed version of the new password, which is saved. Only the hashed version is stored; the entered password is not saved for security reasons.

Access Control List

An access control list (ACL), with respect to a computer file system, is a list of permissions attached to an object. An ACL specifies which users or system processes are granted access to objects, as well as what operations are allowed on given objects [12]. Each entry in a typical ACL specifies a subject and an operation. For example, if a file object has an ACL that contains (Alice: read,write; Bob: read), this would give Alice permission to read and write the file and Bob to only read it. A successful directory traversal attack will still allow the attacker to access any file that the application is authorized to read or write, because this technique only prevents applications from accessing unauthorized files. Thus, this method is not completely *safe*, instead it only mitigates damage.

Root Directory

Root Directory is a specific directory on the server file system in which the users are confined. Users are not able to access anything above this root. For example, the default root directory of IIS on Windows is C:\Inetpub\wwwroot and with this setup, a user does not have access to C:\Windows but has access to C:\Inetpub\wwwroot\news and any other directories and files under the root directory (provided that the user is authenticated via the ACLs). The root directory prevents users from accessing sensitive files on the server such as `cmd.exe` on Windows platforms and the `passwd` file on Linux/UNIX platforms. This vulnerability can exist either in the web server software itself or in the web application code. In order to perform a directory traversal attack, all an attacker needs is a web browser and some knowledge on where to blindly find any default files and directories on the system. This approach is not *safe*, because attackers still can bypass this security level and access restricted directories.

2.4 Command Injection

Command Injection is an attack in which the goal is execution of arbitrary commands on the host operating system via a vulnerable application. Command Injection attacks are possible when an application passes unsafe user supplied data (forms, cookies, HTTP headers etc.) to a system shell. In this attack, the attacker-supplied operating system commands are usually executed with the privileges of the vulnerable application. Command Injection attacks are possible largely due to insufficient input validation and that makes them high-impact security vulnerabilities. This is a vulnerability that affects all languages.

2.4.1 Technical Description

Many confuse Command Injection with Code Injection, but the main difference is that code injection allows the attacker to add his own code that is then executed by the application. Additionally, in Code Injection, the attacker extends the default functionality of the application without the necessity of executing system commands. A typical example of Command Injection vulnerability is presented in presented in Figure 2.6. As we can see, a simple program accepts a filename as a command line argument, and displays the contents of the file back to the user. The program is installed `setuid` root because it is intended for use as a learning tool to allow system administrators in-training to inspect privileged system files without giving them the ability to modify them or damage the system.

```
int main(char* argc, char** argv) {
    char cmd[CMD_MAX] = "/usr/bin/cat ";
    strcat(cmd, argv[1]);
    system(cmd);
}
```

FIGURE 2.6: Simple program showing a command injection vulnerability. This method provides the attacker to recursively delete the contents of the root partition.

Because the program runs with root privileges, the call to `system()` also executes with root privileges. If a user specifies a standard filename, the call works as expected. However, if an attacker passes a string of the form `";rm -rf /"`, then the call to `system()` fails to execute `cat` due to a lack of arguments and then recursively the contents of the root partition deleted.

2.4.2 Countermeasures

Application developers prevent command injection vulnerabilities by writing filters that restrict user input to a whitelist of safe values -list of entities that are being provided a particular privilege, service, mobility, access or recognition. When used in a command execution statement, these safe values should result only in the execution of application-approved programs that cannot violate the system security policy. Current methods for addressing and preventing command injection attacks are the same as those used to protect against directory traversal attacks, described in detail in Section 2.3. In other words, command injection allows attackers to control what files are executed, while directory traversal allows attackers to control what files are read or written.

2.5 SQL Injection

SQL Injection is a code injection technique, used to attack data-driven applications, in which malicious SQL statements are inserted into an entry field for execution (e.g. to dump the database contents to the attacker) [13]. SQL Injection must exploit a security vulnerability in an application's software, for example, when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and unexpectedly executed. SQL Injection is mostly known as an attack vector for websites but can be used to attack any type of SQL database. SQL Injection vulnerability can occur in all languages, including high-level, strongly typed languages such as PHP and Python.

2.5.1 Technical Description

There are many technical implementations for SQL injection such as incorrectly filtered escape characters, incorrect type handling, blind SQL injection, conditional responses and second order SQL injection. A common form of SQL injection is incorrectly filtered escape characters and occurs when user input is not filtered for escape characters and is then passed into an SQL statement. This results in the potential manipulation of the statements performed on the database by the end-user of the application. An example of SQL injection vulnerability is presented in Figure 2.7. This SQL code is designed to pull up the records of the specified username from its table of users.

```
statement = " SELECT * FROM users WHERE name = '' OR '1'='1';"
```

FIGURE 2.7: A line of code illustrates SQL injection vulnerability. This method forces the selection of a valid username because the evaluation of '1'='1' is always true.

However, the variable of name is crafted in a specific way by a malicious user and the SQL statement may do more than the code author intended. If this code were to be used in an authentication procedure then this example could be used to force the selection of a valid username because the evaluation of `'1'='1'` is always true. SQL injection attacks occur when applications use untrusted input in a SQL query without correctly filtering the untrusted input for SQL operators or tokens.

2.5.2 Countermeasures

Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL injection vulnerabilities. Application developers prevent SQL injection attacks by filtering user input using vendor or language-supplied input filtering functions before using untrusted input in a SQL query. Nevertheless, SQL injection vulnerabilities are difficult to detect and defeat because from the database viewpoint, a SQL injection attack may appear only as just another SQL query.

Parameterized Statements

With most development platforms, parameterized statements that work with parameters can be used (sometimes called placeholders or bind variables) instead of embedding user input in the statement. A placeholder can only store a value of the given type and not an arbitrary SQL fragment. Hence the SQL injection would simply be treated as a strange (and probably invalid) parameter value. In many cases, the SQL statement is fixed, and each parameter is a scalar, not a table. The user input is then assigned (bound) to a parameter.

Escaping

A straightforward, though error-prone way to prevent injections is to escape characters that have a special meaning in SQL. The manual for an SQL DBMS explains which characters have a special meaning, which allows creating a comprehensive blacklist of characters that need translation. For instance, every occurrence of a single quote `(')` in a parameter must be replaced by two single quotes `('')` to form a valid SQL string literal. For example, in PHP it is usual to escape parameters using the function `mysqli_real_escape_string();` before sending the SQL query. Routinely passing escaped strings to SQL is error prone because it is easy to forget to escape a given string. This technique is not *safe* because creating a transparent layer to secure the input can reduce this error-proneness, if not entirely eliminate it.

Database Permissions

Limiting the permissions on the database logon used by the web application to only what is needed may help reduce the effectiveness of any SQL injection attacks that exploit any bugs in the web application. For example, on Microsoft SQL Server, a database logon could be restricted from selecting on some of the system tables which would limit exploits that try to insert JavaScript into all the text columns in the database. However, this

approach is not *safe*, because it only mitigates damage. Pattern Check and Hexadecimal Conversion are also a few more preventative measures, but we have not a magic wand that will protect from any SQL injection attack.

2.6 Cross-site Scripting

Cross-site Scripting (XSS) is a type of computer security vulnerability typically found in web applications. XSS enables attackers to inject client-side script into web pages viewed by other users. A cross-site scripting vulnerability may be used by attackers to bypass access controls such as the same-origin policy. Cross-site Scripting carried out on websites accounted for roughly 84% of all security vulnerabilities documented by Symantec as of 2007 [14]. The expression "cross-site scripting" originally referred to the act of loading the attacked, third-party web application from an unrelated attack site, in a manner that executes a fragment of JavaScript prepared by the attacker in the security context of the targeted domain (taking advantage of a reflected or non-persistent XSS vulnerability). XSS attacks affect all languages.

2.6.1 Technical Description

There is no single, standardized classification of cross-site scripting flaws, but most experts distinguish between at least two primary flavors of XSS flaws: non-persistent and persistent. Some sources further divide these two groups into traditional (caused by server-side code flaws) and DOM-based (in client-side code). The non-persistent (or reflected) XSS vulnerability is by far the most common type [15]. These holes show up when the data provided by a web client, most commonly in HTTP query parameters or in HTML form submissions, is used immediately by server-side scripts to parse and display a page of results for and to that user, without properly sanitizing the request [16]. A common example of a potential vector is a site search engine where user searches for a string and the search string will typically be redisplayed verbatim on the result page to indicate what was searched for. If this response does not properly escape or reject HTML control characters, a cross-site scripting flaw will ensue [14].

The persistent (or stored) XSS vulnerability is a more devastating variant of a XSS flaw and occurs when the data provided by the attacker is saved by the server and then permanently displayed on "normal" pages returned to other users in the course of regular browsing, without proper HTML escaping. A typical example of this is with online message boards where users are allowed to post HTML formatted messages for other users to read [14]. For example, suppose there is a dating website where members

scan the profiles of other members to see if they look interesting. For privacy reasons, this site hides everybody's real name and email. These are kept secret on the server and the only time a member's real name and email are in the browser is when the member is signed in, and they can't see anyone else's.

2.6.2 Countermeasures

XSS attacks are the most common form of web application vulnerability because security vendors cannot tell which bytes of an HTML document come from safe or trusted applications, and which bytes come from untrusted sources. Existing XSS defenses can be divided into two categories: contextual output encoding/escaping of string input and safely validating untrusted HTML input. The primary defense technique to stop XSS attacks is contextual output encoding/escaping. There are several different escaping schemes that must be used depending on where the untrusted string needs to be placed within an HTML document including HTML entity encoding, JavaScript escaping, CSS escaping, and URL (or percent) encoding. Performing HTML entity encoding only on the five XML significant characters is not always sufficient to prevent many forms of XSS attacks and because of this drawback, this approach is not completely *safe*.

On the other hand, stopping an XSS attack when accepting HTML input from users is much more complex using safely validating untrusted HTML input. For instance, operators of particular web applications (e.g. forums and webmail) allow users to utilize a limited subset of HTML markup. When accepting HTML input from users, output encoding will not suffice since the user input needs to be rendered as HTML by the browser. Untrusted HTML input must be run through an HTML sanitization engine to ensure that it does not contain XSS code. Besides content filtering, another imperfect method for XSS mitigation is the use of additional security controls when handling cookie-based user authentication. Typical examples are web applications rely on session cookies for authentication between individual HTTP requests, and because client-side scripts generally have access to these cookies, simple XSS exploits can steal these cookies.

Chapter 3

Dynamic Information Flow Tracking

Recent researches have shown that Dynamic Information Flow Tracking (DIFT) is a promising technique against software vulnerabilities. DIFT can be used to prevent a wide range of attacks including low-level software vulnerabilities such as Buffer Overflow Attacks and Format String Attacks and high-level threats such as Directory Traversal, Command Injection, SQL Injection and Cross-site Scripting. DIFT tracks the flow of untrusted information within a programs runtime by extending memory and registers with one or more tag bits. A tag bit is typically set if the register or memory word contains untrusted information received over the network or some other source. Tags are propagated at runtime according to a set of tag propagation rules. A security exception is raised if untrusted information is used unsafely, such as dereferencing a tainted pointer or executing a SQL query with tainted database commands in order to protect the system.

This chapter introduces DIFT and provides a thorough description of this method. In Section 3.1 we describe an overview of DIFT itself and its applications, while in Section 3.2 we analyze DIFT technique in different implementation methods. Subsequently, in Section 3.3 we describe related systems are implemented until now and the potential of DIFT as well as the challenges that remain in helping DIFT to reach its full potential as a security technique. Lastly, in Section 3.4 we summarize and conclude the chapter comparing the best techniques and methodologies.

3.1 Overview

Dynamic Information Flow Tracking (DIFT) is a technique for tracking and restricting the flow of information when executing programs in a runtime environment [17, 18]. Programs are represented as sequences of instructions. A runtime environment consists of memory, storage or I/O device resources and a program interpreter. Java Virtual Machine and Intel 32-bit processor are typical examples of runtime environments, the first implemented in software and the second implemented in hardware respectively. The runtime environment interprets program instructions in order to allow the program access to memory and storage resources.

In every computer system, we use applications that process untrusted input and as a consequence security exploits occur. Thus, this untrusted input must be validated before using it in a security-sensitive manner. The validation procedure should ensure that future use of the input will not result in a violation of the system security policy. Any breakdown in this chain of events, such as missing a validation check or incorrectly validating untrusted input, triggers more and more input validation security vulnerabilities. They could be applied just as well to high-level vulnerabilities such as SQL injection attack, as it does to low-level vulnerabilities like the common buffer overflow corruptions. Acknowledgement of the crucial role that information flow plays in all range of vulnerabilities makes DIFT potent and comprehensive technique. By tracking and restricting the flow of untrusted information of which have entered the system, DIFT policies can prevent user input from being used unsafely in a security-sensitive manner.

More specifically, DIFT associates tags with memory and resources in the system, and then uses these tags to maintain information about the reliability of the corresponding data. Tags are initialized in accordance with the source of the data. A typical tag initialization policy would be to mark data arriving from untrusted sources such as a tainted network, while keeping files owned by the user untainted. Tag propagation refers to the combining of tags of the source operands to generate the destination operands tag. As every instruction is processed by the program, the corresponding metadata must be performed by the runtime environment. For example, an arithmetic operation must combine the tags of the operands in accordance with the tag propagation policies and the data processing. Tag checks are performed also in accordance with the configured policies to check for security violations. A security exception is raised in the case of an unsafe use of untrusted information, such as the dereferencing of an untrusted pointer. But, it is essential to identify when a tainted operand can be safely accessed without raising an exception. Figure 3.1 shows the logical view of the system at the ISA level, where every register and memory location appears to be extended with a tag.

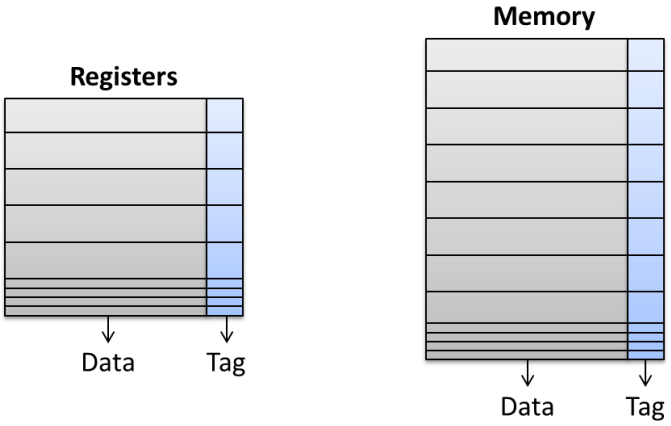

FIGURE 3.1: The logical view of the system at the ISA level, where every register and memory location appears to be extended with a tag.

Furthermore, it is worth noting that no other security technique has been shown to be applicable to such a wide spectrum of attacks until now. First of all, DIFT is practical since it does not require any knowledge about the internals or semantics of programs. Secondly, DIFT model is flexible because it has allowed for a myriad of implementations at various levels of abstraction. Typical examples of DIFT implementations are JVM used to prevent Java servlet vulnerabilities and other DIFT-based implementations in hardware to prevent wide range of memory corruption exploits. Most scripting languages, such as PHP [19] and Java [20], Dynamic Binary Translators [21], and hardware platforms [18], use implementations of DIFT technique. DIFT is fast as evinced by some of the high-performance DIFT systems, while works on unmodified binaries or bytecode, without requiring any source code or debugging information. Ultimately, DIFT has been shown to provide end-to-end protection on systems by securing both operating systems and programs against attacks [22].

3.2 DIFT Implementations

Academic researchers have had success in creating DIFT policies in a number of environments in order to prevent many kinds of input validation vulnerabilities, but existing DIFT policies, for instance buffer overflow policies, are unsafe and impractical, and have significant false positives and negatives in real-world applications. Moreover, there are high-impact vulnerabilities that have yet to be addressed by a DIFT policy, such as web authentication and other authorization vulnerabilities. It is important for the policy designer to use all available information and determine what constitutes information flow within the application, when designing a DIFT policy to protect applications. In other

words, DIFT systems provide configurable and flexible support for reasonable policies and policy designers use this support to find the best method of preventing security attacks. DIFT systems can be implemented in both software and hardware platforms. In this section, we will provide an overview of the major DIFT implementation types: compiler-based, bytecode rewriting, metaprogramming, modified language interpreter, dynamic binary translation, and hardware.

3.2.1 Compiler-based DIFT

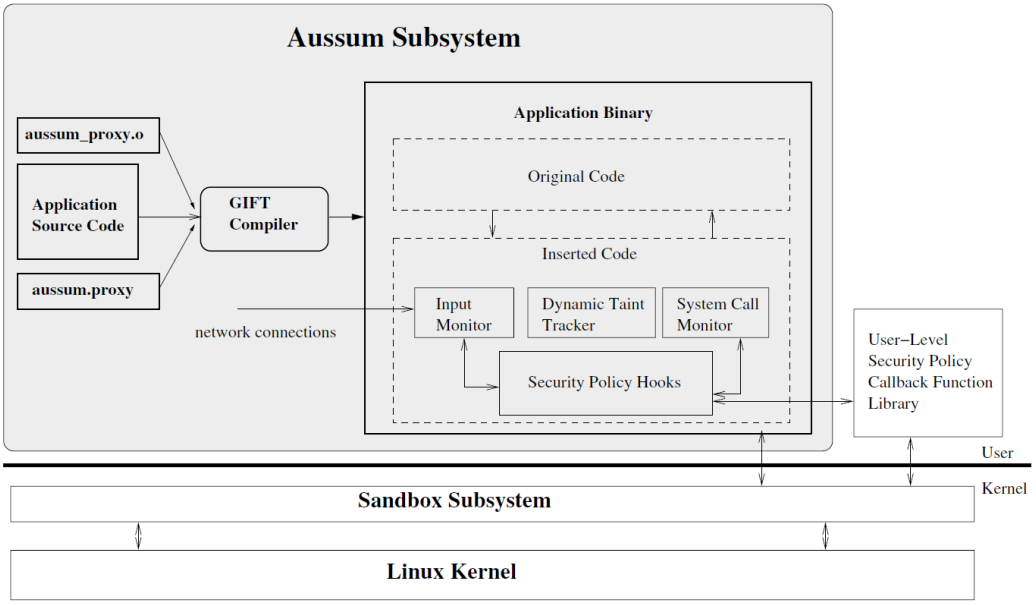

FIGURE 3.2: The Aussum compiler is built from the GIFT framework. The Input Monitor intercepts data read from a network connection and marks it according to a configurable security policy. The Dynamic Taint Tracker propagates the mark throughout the program. Finally the System Call Monitor marks files and system call arguments that are derived from network inputs.

Academic researchers have proposed implementing DIFT technique by modifying existing compilers [23]. Compilers transparently add DIFT operations to an application during the compilation process, emitting a DIFT-aware executable file. In this approach, the overhead of tracking information flow minimized, because compiler uses standard optimization techniques. All the existing researches have focused on the C programming language. While these prototypes implemented only a few fixed policies with minimal performance overheads, it would be easy to add support for arbitrary policies, providing a solution that is flexible and fast at the same time. Figure 3.2 illustrates the Aussum compiler [23], built from the GIFT framework.

However, this approach has a fundamental drawback, because requires the entire system to be recompiled with DIFT support, in order to accurately track taint for complete information flow tracking safety. Furthermore, developers must recompile their applications and distribute their binaries with the DIFT technique enabled. This approach seems to be impractical, as it requires source code access to all protected applications. Developers have to think about customers that do not want DIFT binaries, or develop and ship two versions of the same program, one with DIFT enabled and one without.

Additionally, any code not compiled by the DIFT compiler, such as code written in another programming language, assembly code, code from third-party libraries, system libraries, will have no support for DIFT. Compiler-based DIFT approach is not safe, because going back and annotating entire code-bases, especially for third-party libraries seems to be a very painful work (there are billions of lines of legacy code). Thus, these approaches will not be an acceptable solution in production environments, due to lack of safety and practicality.

3.2.2 Bytecode Rewriting DIFT

Researchers have also proposed DIFT implementations by statically or dynamically rewriting the application bytecode for languages which compile source code to a modifiable bytecode format, such as Java [17]. Classes used as tag sources, sinks and propagators are modified by inserting bytecode instructions to perform the DIFT operations required. In bytecode rewriting DIFT systems, the instrumented language must have bytecode file format that allows DIFT operations to be inserted before or after tag source, sinks, and propagators. The major bytecode formats used by high-level languages (e.g., Java and Microsoft .NET) meet this requirement.

DIFT prototypes based on bytecode rewriting have focused on object-granularity tags, rewriting the bytecode of specific classes, such as the String class in Java. However, there is nothing to prevent the DIFT system developer from rewriting all application and system library bytecode to provide byte-granularity taint tracking. This approach seems to be flexible and safe, if all relevant classes associated with DIFT operations are instrumented. Moreover, extremely low performance overhead measured in academic prototypes, such as bytecode rewriting on Java applications for object granularity tags [17].

Nevertheless, the performance of bytecode rewriting when implementing a fine-grained, per-character or per-byte information flow tracking policy is unknown. Any encountered overheads should be strictly less than implementing DIFT using a dynamic binary translator (explained in detail in section 3.2.5) as DBT-based DIFT approaches

perform very similar operations on low-level ISA[1] instructions but additionally incur the overhead of dynamic binary translation. Finally, these object-granularity prototypes are not completely safe as they do not track information flow across characters or character arrays.

3.2.3 Metaprogramming DIFT

Certain high-level languages with DIFT support use metaprogramming application programming interfaces[2] (APIs) to perform runtime modification, monkey patch, of all tag sources, sinks, and propagators [24]. High-level languages allow programs to modify existing classes at runtime by redefining or wrapping existing methods, or by adding additional fields. In this approach, DIFT can be implemented using constructs already provided by the language. Metaprogramming DIFT systems can be created without modifying the interpreter, performing bytecode rewriting or investing in hardware, allowing the designer to develop a program written in the target language that uses the metaprogramming APIs to modify all the relevant classes that are designated as tag sources, sinks, or propagators. Thus, the original applications run and the modified classes serve as the runtime environment for DIFT.

This approach, in contrast to compiler-based DIFT, is theoretically the least complex way to implement DIFT. There are currently no empirical evaluations of this approach, or available research prototypes, and thus the performance overheads are unknown. However, it seems to be flexible, providing safety, if all sources, sinks or propagators can be modified. This approach is not practical because it is applicable only for languages that allow runtime modification of all relevant taint sources, sinks and propagators via metaprogramming APIs. A typical example is Python programming language, which will not fit this requirement for most reasonable DIFT policies because system classes, such as String, cannot have their methods redefined or modified.

Ruby is an object-oriented, type-safe language and is the only mainstream language that fully supports metaprogramming DIFT [24]. In these schemes, all system types are objects, including primitives such as integers and all operations are method calls, even integer addition. Any object can be modified at runtime in arbitrary ways, including

[1] Instruction set, or instruction set architecture (ISA), is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification of the set of opcodes (machine language), and the native commands implemented by a particular processor.

[2] Application programming interface (API) is a set of routines, protocols, and tools for building software and applications. An API expresses a software component in terms of its operations, inputs, outputs, and underlying types, defining functionalities that are independent of their respective implementations, which allows definitions and implementations to vary without compromising the interface.

adding fields and wrapping or redefining existing methods. Ruby even allows for fine-granularity taint tracking, as integer addition and all other arithmetic operations are ordinary methods that can be modified or redefined using metaprogramming to insert taint tracking operations. No other mainstream language may be used to implement DIFT entirely via metaprogramming.

3.2.4 Language Interpreter DIFT

Modifying the language interpreter itself is another implementation strategy for DIFT. This can be implemented by adding the appropriate functionality to the interpreter, ensuring that information flow tracking occurs for all tag sources, sinks and propagators. This scheme can potentially provide better performance than interpreter-independent techniques, such as metaprogramming or bytecode rewriting, because these systems have direct access to the interpreter, allowing specific DIFT optimizations to be placed into the interpreter itself, most likely in native code.

Language interpreter DIFT strategies allow flexible, practical and safe DIFT policies. The cost depends only on the complexity of the interpreter and the scope of the changes required to support DIFT technique. A typical example is the PHP interpreter [19], in which academic researchers have added support for DIFT, without big implementation cost. Interpreters which do not perform JIT[3] compilation or dynamic code generation are usually cheap to modify, with low performance overhead, depending on DIFT support, while performing similar changes in a JIT-compilation interpreter, such as HotSpot, would be significantly more difficult. No researches have been done on the performance overhead of adding DIFT to a JIT-compilation interpreter, but performance overhead can be expected to be smaller than implementing DIFT using dynamic binary translation (DBT).

Providing a DIFT-aware interpreter requires the designer to maintain a fork of the interpreter, a costly maintenance operation, as adding DIFT support often requires changes to many different pieces of an interpreter, many of which are usually part of the core, such as character and integer handling. Thus, the downside to this approach seems to be the high maintenance costs.

[3] Just-in-time (JIT) compilation, also known as dynamic translation, is compilation done during execution of a program at run time rather than prior to execution. Most often this consists of translation to machine code, which is then executed directly, but can also refer to translation to another format.

3.2.5 Dynamic Binary Translation DIFT

Another approach for DIFT implementation in software is to use a Dynamic Binary Translator (DBT). Binary Translation (or Binary Recompilation) is the emulation of one instruction set by another through translation of binary code. Sequences of instructions are translated from the source to the target instruction set. In a typical DBT-based DIFT implementation, the application, or in some cases the entire system is run within a DBT. The binary translation framework maintains metadata, or state associated with the applications data. This metadata is important and is used to maintain information about the taintedness of the associated data. Every time DBT-based system performs binary translation, dynamically inserts instructions for DIFT. Every instruction from the application has an associated metadata instruction that manipulates the associated tainted values.

Since the security analysis is performed in software, the policies employed can be arbitrarily complex and flexible. Thus, we are able to use the same infrastructure for a wide range of policies, and this is a big advantage. Nonetheless, binary translation approach has considerably high performance overhead, because it is necessary to introduce a whole new instruction to manipulate the taint associated with the original programs instruction. Depending upon the application and policies in every platform, DBT-based DIFT systems have been shown to have extremely high performance overheads ranging from approximately 3x in LIFT platform [25] to 37x in other cases, such as TaintCheck implementation [26]. To successfully apply DIFT support to an entire system, we have to virtualize all devices, such as memory, operating system and all applications related to the DBT. Thus, virtualization for DIFT support increases overhead of the entire system significantly and restricts the wide-spread applicability of DBT-based DIFT technique.

Another fundamental drawback with DIFT solutions in binary translation frameworks is the lack of support for multi-threaded applications. It is essential, when executing multi-threaded applications, to ensure consistency between updates to both data and tags, so that all other active threads perceive these updates as atomic operations [18] and prevent race conditions. Despite this, recent researches have shown that hardware support for hybrid DIFT systems can use DIFT technique for multi-threaded applications efficiently, but this requires in depth hardware modifications. Metaprogramming, bytecode rewriting, and interpreter-based DIFT only protect and instrument software written in a single language. In contrast, we can easily apply DIFT technique to unmodified binaries, using DBT.

3.2.6 Hardware DIFT

Besides implementing DIFT in software [18, 27], DIFT technique can also be implemented in hardware. In this approach, the runtime environment is the CPU, while memory is represented by CPU caches and physical RAM, and resources map directly to I/O devices. Hardware is the ideal level for implementing DIFT support because it is the lowest layer of abstraction in a computer system. Thus, programs, binaries and executables run on top of the hardware. Furthermore, DIFT security policies can be applied to scripting languages, binaries, applications, or even operating systems. This is a very promising method that renders the protection completely independent of the choice of programming language, since all languages must eventually be translated to some form of assembly language that hardware can run automatically.

Hardware DIFT is implemented by extending all registers, memory, and CPU caches with tag bits. Tag propagation and checks are performed inside the CPU in parallel with instruction execution. Tags are propagated from source to destination operands for all instructions. A tag is cleared when data is validated or reset. The hardware raises an exception when an instruction, a jump target, or load/store address is tagged. These DIFT systems provide extremely low-overhead protection, even when applied to the whole operating system, because tag propagation occurs in parallel with the execution of the original data instruction. Additionally, the performance and the complexity challenges faced by whole-system dynamic binary translation are extremely different in hardware platforms and DIFT policies can be applied to the whole system efficiently. Existing hardware systems are also fast, as tag storage and tag propagation are provided by hardware.

Moreover, hardware DIFT systems may also extend the memory coherence protocols and provide low-overhead tag operations safely even in multi-threaded applications where different threads may concurrently update the data and tags of a memory location. Michael Dalton et al. implemented this method in their system prototype called Raksha [28] by ensuring atomic updates to both data and tags. Hari Kannan et al. made minor modifications to the coherence protocols to ensure that an atomic view of data and tags is always presented to other processors [29]. These techniques are the keys in ensuring the practical viability of the DIFT solution, since computer systems are migrating to multi-core environments. To conclude, unlike DBT-based DIFT systems, hardware DIFT systems have been shown to provide comprehensive support with low performance overheads against both low-level memory corruption exploits (such as buffer overflows or format string attacks and high-level web attacks such as SQL injections or cross-site scripting (XSS)).

Despite the fact hardware DIFT systems are ultra-fast with low performance overheads, in the most cases they seem to be inflexible. A significant drawback to hardware DIFT architectures is the usage of single fixed security policies to catch all types of attacks. Many new technologically computer Viruses, Worms and Trojans have become more damaging, because they target multiple vulnerabilities [30] and can bypass the protection offered by current hardware DIFT architectures, since they can protect against only one kind of exploit using a solitary or single security policy. Modern software becomes extremely complex and combined with the lack of flexibility restricts the ability of current hardware DIFT systems to handle such cases. For instance, the current buffer overflow policy in use by existing hardware DIFT implementations is impractical and unsafe due to real-world false positives and negatives.

3.3 Related Work

DIFT taints the data entering the system from untrusted sources, and dynamically tracks the propagation of this data during program execution. As a result, it is possible to detect situations where the tainted data is used in potentially insecure ways. Hardware-based systems can apply DIFT to any application, even those using self-modifying code, JIT compilation, or multi-threading. All existing hardware DIFT systems focus on memory corruption attacks using a single, fixed security policy.

One of the first hardware DIFT architectures for software attacks proposed in 2004 by Suh et al. [27]. They assumed that the tag is a taint bit that can be applied to memory elements as small as bytes. There are propagation policies for four categories of instructions, providing rules for each one (e.g. for an arithmetic instruction, the taint of the result receives the taint of the operand). Whenever a tainted data is used as an execution address, which is likely to characterize a memory corruption attack, the processor raises a security exception. They introduced also a multi-granularity policy, a page table structure that keeps tag data blocks as large as entire pages or words with only one taint bit, in order to reduce the overhead of tagging every byte of memory. It is only upon the first write operation on a data smaller than the granularity of the page it belongs to, that the OS refines the tagging granularity for this page. Their mechanism relies on TLBs to be extended in order to cache the tag type of memory pages. Finally, they introduced separate tag caches, to avoid extending each processor cache block with tag bits. Figure 3.3 illustrates the overview of DIFT protection scheme proposed by Suh et al. [27].

Another approach for hardware DIFT architecture, called Minos [31], proposed at the same year, by Crandall et al. and it seems to be quite similar to Suh et al. [27].

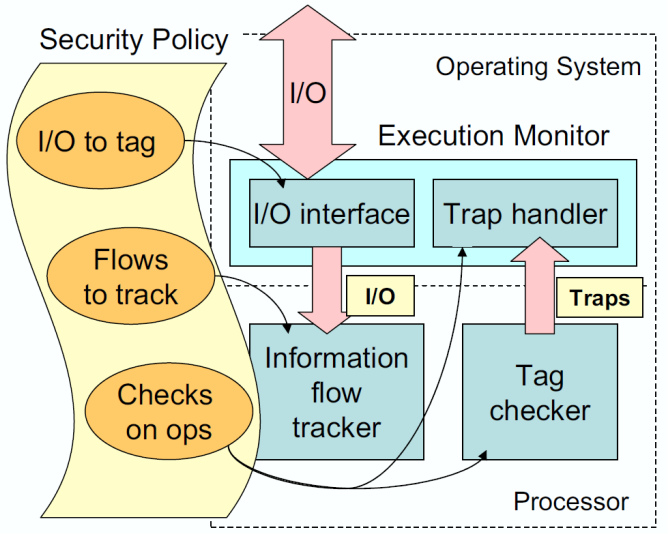

FIGURE 3.3: DIFT protection scheme proposed by Suh et al. consists of three major parts: the execution monitor, the tagging units (flow tracker and tag checker), and the security policy.

A basic difference between these two approaches is that Minos' researchers did not try to reduce the memory overhead induced by the tag bit, arguing that Moores law could easily absorb the overhead overtime. Thus, in this design, the memory, the common data bus and the whole memory chain of the processor (e.g. caches and registers) are extended with tag bits at the granularity of word. These two approaches introduced tag storage for DIFT in a completely different way, but the main focus has been toward more flexible propagation policies without complex hardware implementations.

Three years later, Dalton et al. proposed Raksha [28], the first approach targeting flexibility but mostly resembles the previous hardware DIFT architectures. It supports up to four programmable, independent and concurrent security policies each with its own set of rules for propagation and checks. Therefore, the software is able to control each policy by using pre-existing rules or defining custom rules for each class of instructions. This platform provides flexibility to detect various types of software attacks, such as low-level attacks (e.g. memory corruptions and format string attacks) or high-level attacks (e.g. SQL injections and cross-site scripting (XSS)).

An effort to avoid modifications to the design of processors has been made by Kannan et al. [29]. They described an architecture that performs all DIFT operations in a small off-core, attached coprocessor (based on the previous project called Raksha [28]). The coprocessor supports a strong DIFT-based security model by synchronizing with the main core only on system calls. They proposed a design that addresses the complexity, verification time, power, area, and clock frequency challenges of previous proposals for

DIFT hardware and no changes are necessary to the main processors pipeline, design, or layout. Figure 3.4 shows the pipeline diagram for the DIFT coprocessor proposed by Kannan et al. [29].

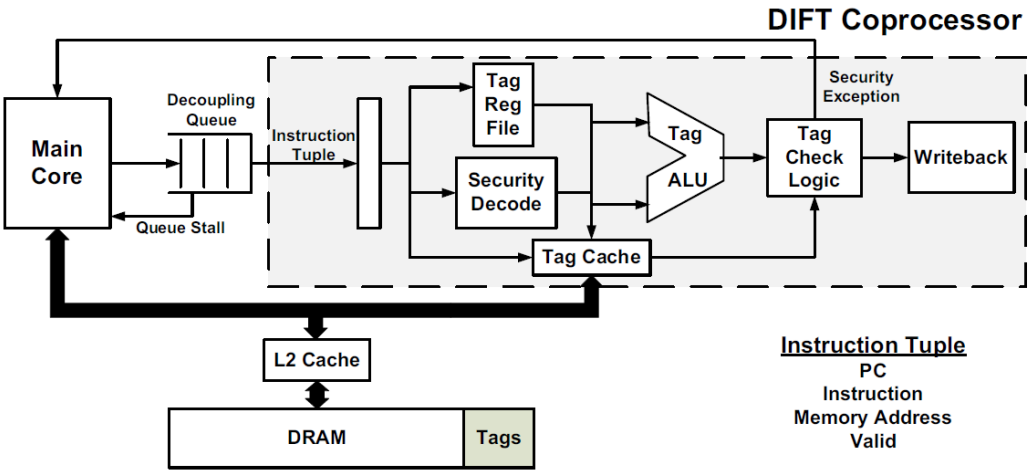

FIGURE 3.4: The pipeline diagram for the DIFT coprocessor proposed by Kannan et al. Structures are not drawn to scale.

Another approach targeting flexibility introduced in 2008 by Venkataramani et al. and called Flexitaint [32]. The main objective was to reduce the hardware impact on out-of-order processor architectures. Instead of placing the taint tag propagation logic along with the regular computation, they introduced a programmable accelerator, adding an extra stage at the back-end of the pipeline and leaving the other stages unmodified. In this new stage, the tag propagation is either performed by software custom rules or pre-fixed rules. Tags are stored as an array in a protected area of the virtual address space of each program, in order to avoid the need to modify the whole memory and the bus, allowing a full compatibility with conventional OS mechanisms (e.g. copy-on-write, copy-on-write, etc.). Finally, an additional L1 cache is introduced within the processor to cache taints.

Unlike the previous approaches, Vachharajani et al. proposed RIFLE [33], an architecture that focuses on the detection of information leakage. In RIFLE, information-flow security policies are enforced using a combination of binary translation and a modified architecture. Program binaries are translated from a conventional instruction-set architecture (ISA) to an information-flow secure (IFS) ISA. The translated programs are executed on hardware that aids information-flow tracking, and the executed programs interact with a security-enhanced operating system that is ultimately responsible for enforcing the users policy. This three part enforcement mechanism ensures that confidential data is not leaked and provides users with complete control over the confidentiality of

their data. Table 3.1 presents a summary of previously described approaches to DIFT and the specific type of target vulnerabilities.

Name	**Year**	**Tag width**	**Target vulnerabilities**
DIFT [Suh et al.]	2004	1bit/byte	Control and non-control data attacks
Minos [Crandall et al.]	2004	1bit/word	Control data attacks only
RIFLE [Vachharajani et al.]	2004	Unspecified	Information leakage
Raksha [Dalton et al.]	2004	4bits/word	Low-level and high-level attacks
Flexitaint [Venkataramani et al.]	2008	1-2bits/word	Unspecified
SHIFT [Chen et al.]	2008	1bit/byte-quadword	Low-level and high-level attacks
Coprocessor based on Raksha [Kannan et al.]	2009	4bits/word	Low-level and high-level attacks
GLIFT [Tiwari et al.]	2009	-	Information leakage
SIFT [Ozsoy et al.]	2011	Unspecified	Unspecified

TABLE 3.1: Summary of related works to DIFT.

Another direction has been explored, to implement DIFT in hardware, when Tiwari et al. [34] proposed GLIFT, an approach for tracking information-flow at gate-level, a level lower than the architectural level. GLIFT shows how to generate shadow flow tracking logic for each simple gate in a design, and eventually suggests composition rules between gates to handle more complex structures. Moreover, SHIFT [35], a low-overhead DIFT system proposed by Chen et al., detects a wide range of attacks. The key idea is to treat tainted state (untrusted data) as speculative state (deferred exceptions). Likewise, Ozsoy et al. introduced SIFT [36], another DIFT platform, where a separate thread performing taint propagation and policy checking is executed in a spare context of an SMT processor. However, the instructions for the checking thread are generated in hardware using self-contained off-the-critical path logic at the commit stage of the pipeline.

3.4 Summary

In this section, we will discuss about the applicability of DIFT system architectures to particular vulnerabilities (low-level and high-level). DIFT technique is flexible and can be implemented to all of the different layers of abstraction in both software and hardware platforms. For instance, the modified PHP interpreter with DIFT support, can easily detect high-level vulnerabilities written in PHP language, but they cannot address low-level corruptions, such as buffer overflows. Moreover, DIFT systems implemented a

particular layer of abstraction prevent vulnerabilities in all of the layers above them, and not vice versa. DIFT systems built in hardware will be able to address assembly or ISA-instruction level vulnerabilities, while DIFT systems built by modifying the JVM will prevent only Java application vulnerabilities.

However, when developers lower the layer of abstraction, such as by choosing a DBT-based DIFT platform over a JVM-based DIFT system, they cannot retain all the information about the applications they are protecting. Thus, developing DIFT architectures at the lowest possible level of abstraction does not always guarantee a better result. A typical example is the JVM that has semantic information about a Java class than the DBT, which cannot even resolve method names to memory addresses. The risk that developers have to take is that their systems may not have sufficient information about the target application to properly support the desired set of DIFT policies. They can defeat this drawback, by allowing trusted higher-level components to communicate with the DIFT system via an API, such as having the JVM pass class information to a dynamic binary translator via special ISA instructions.

Another factor that developers have to consider about is the type of language that interpreter uses. For example, if PHP SQL libraries are implemented as C extensions, a hardware DIFT system is able to protect PHP applications against SQL injections. Nevertheless, if the PHP interpreter created raw sockets and the SQL libraries manipulated those sockets entirely in PHP language, it would be more than difficult for the hardware DIFT system to interpose on calls to SQL library methods without the ability to map PHP function names to memory addresses at runtime. Thus, by modifying the PHP interpreter efficiently, developers could provide this semantic information to a low-level DIFT platform, such as a DBT-based or hardware DIFT architecture.

	Low-level Vulnerabilities		**High-level Vulnerabilities**		
DIFT System	**Assembly**	**C**	**C**	**Compiled**	**Interpreted**
Compiler DIFT	✖	✔	✔	✖	?
Bytecode DIFT	✖	✖	✖	✔	✔
Metaprogr. DIFT	✖	✖	✖	✔	✔
Interpreter DIFT	✖	✖	✖	✔	✔
DBT DIFT	✔	✔	✔	?	?
Hardware DIFT	✔	✔	✔	?	?

TABLE 3.2: The applicability of DIFT system architectures to particular vulnerabilities. There are two types of vulnerabilities. Low-level vulnerabilities: memory corruption attacks (e.g., buffer overflows, pointer dereferences and format string vulnerabilities). High-level vulnerabilities: API-level injection attacks (e.g., SQL injection, cross-site scripting, directory traversal and command injection). There are four types of languages: assembly, C, interpreted languages and JIT compiled languages. The question mark (?) indicates cases that are situation-dependent.

Ultimately, we present in detail the applicability of DIFT system architectures to particular vulnerabilities in Table 3.2. There are also some cases that are situation-dependent, such as the example we discussed above that may work on a particular language (e.g., PHP interpreter), but not necessarily on all other languages. On these occasions, we need a trusted component to be added to the system, maybe a higher-lever one, in order to efficiently communicate information to the specific DIFT platform. To sum up, if we are able to modify a JVM (or other JIT-compilation systems) for DIFT support with mappings of Java method names to memory addresses, we could prevent high-level vulnerabilities (such as SQL injection) using a hardware-based platform for DIFT, but without this extra modification, this system could not support Java language.

Chapter 4

The LEON processor

LEON is a 32-bit CPU microprocessor core, based on the SPARC-V8 RISC architecture and instruction set. The LEON project was started by the European Space Agency (ESA) in late 1997 to study and develop a high-performance processor to be used in European space projects [37]. The aim of the project was to provide an open, portable and non-proprietary processor design, capable to meet future requirements for performance, software compatibility and low system cost. Another objective was to be able to manufacture in a Single event upset (SEU) sensitive semiconductor process. This chapter introduces LEON3 and provides a detailed description of its architecture. In Section 4.1 we describe an overview of LEON3 architecture and GRLIB library, while in Section 4.2 we describe LEON3 processor. Lastly, in Section 4.3 we describe Memory Interface for PROM, memory mapped I/O and Static RAM accesses.

The LEON family includes the first LEON1 VHSIC Hardware Description Language (VHDL) design that was used in the LEONExpress test chip developed in 0.25 m technology to prove the fault-tolerance concept. The second LEON2 VHDL design was used in the processor device AT697 from Atmel (F) and various system-on-chip devices. These two LEON implementations were developed by European Space Research and Technology Centre (ESTEC), part of the European Space Agency (ESA). Gaisler Research (now Aeroflex Gaisler) developed the third LEON3 design, described in synthesizable VHDL. The full source code is available under the GNU GPL license, allowing use for any purpose without licensing fee. LEON3 is also available under a proprietary license, allowing it to be used in proprietary applications. The core is configurable through VHDL generics, and is used in system-on-a-chip (SOC) designs both in research and commercial settings. In this thesis we used the third LEON3 processor, a highly configurable processor, designed mainly for embedded applications, combining high performance with low complexity and low power consumption.

4.1 Overview

In this section, we offer a detailed description of the basic architecture and functionality of the LEON3 processor, in which our thesis is based on. The original manual of LEON3 [38] describes the full functionality of the LEON3 core and the whole template design, through the use of VHDL generics, while all the parts of the described functionality can be suppressed or modified to generate a smaller or faster implementations. The description provided in this chapter is organized in a top-down manner, starting with a view of the whole IP library GRLIB (LEON3 template design) and going down to the LEON3 integer pipeline (integer unit).

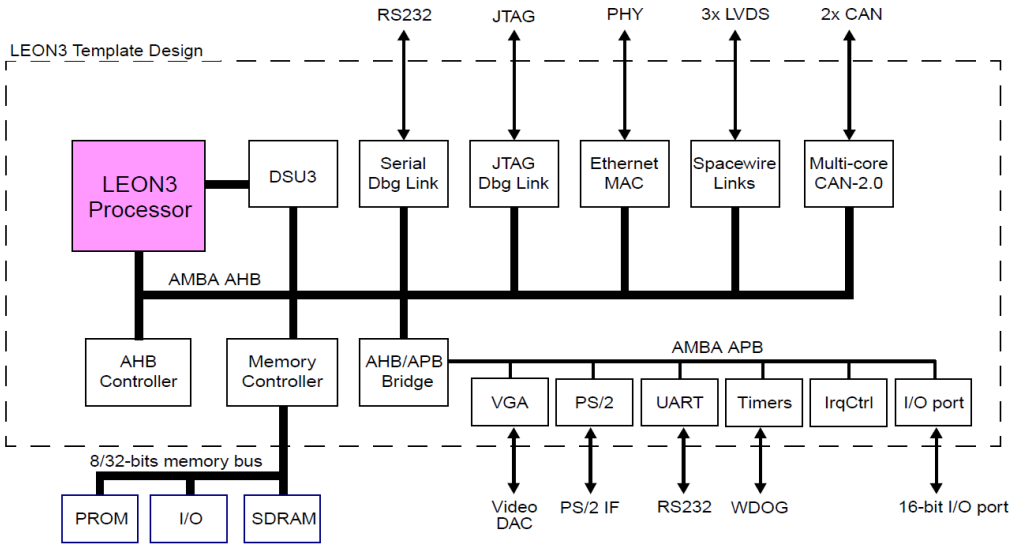

FIGURE 4.1: LEON3 template design block diagram.

4.1.1 Architecture

The LEON3 template design consists of the LEON3 processor and a set of IP cores connected through the AMBA AHB/APB buses (see Figure 4.1). LEON3 processor and other high-bandwidth devices are connected to the AMBA Advanced High-Speed bus (AHB), while external memory is accessed through a combined PROM/IO/SRAM/S-DRAM memory controller (described in detail in Section 4.2). The on-chip peripheral devices include three SpaceWire[1] links, Ethernet 10/100 Mbit MAC, dual CAN-2.0 interface, serial and JTAG debug interfaces, two UARTs, interrupt controller, timers and an I/O port. Furthermore, the design is highly configurable, and the various features

[1]SpaceWire is a spacecraft communication network based in part on the IEEE 1355 standard of communications. It is coordinated by the European Space Agency (ESA) in collaboration with international space agencies including NASA, JAXA and RKA.

can be suppressed if desired. Most parts of the design are provided in source code under the GNU GPL license[2]. The exception is the SpaceWire core, which is only available under a commercial license. For evaluation and prototyping, netlists suitable for the many boards are provided. The LEON3 processors and associated IP cores also exist in a fault-tolerant (FT) version. The FT cores detect and remove SEU errors due to cosmic radiation, and are particularly suitable for systems that operate in the space environment. The FT version of LEON3 and GRLIB is only licensed commercially by Gaisler Research.

4.1.2 The GRLIB library

GRLIB is a library of VHDL source codes of IP cores for designing a complete system on chip for the LEON3 processor [38]. The library is structured into directories that group IP cores according to the contributors company. There are more subdirectories with complete IP cores organized in packages in the directories. These directories contain VHDL source codes of packages for simulation and synthesis and files with package configurations. A package configuration is propagated to VHDL entities via generics. The library is managed by an automated tool based on GNU make (described in detail in Section 5.1) that manages configurations and compilation of packages for simulation and synthesis.

The LEON3 processor consists of many parts. The main parts of the processor implementation are placed in the `$GRLIB/lib/gaisler/leon3` directory. VHDL files with the SPARC V8 instruction codes and support functions for instruction disassembly during simulation are placed in the `$GRLIB/lib/grlib/sparc` directory. User designs with top-level files, that instantiate complete SoC designs, are located in the `$GRLIB/designs` directory. In this thesis we used the Xilinx ML50X design, which is located in the `$GRLIB/designs/leon3-xilinx-ml50x` directory. Overall, the LEON3 processor is distributed as part of the GRLIB IP library, allowing simple integration into complex SOC designs. GRLIB also includes a configurable LEON3 multi-processor design, with up to 4 CPU's and a large range of on-chip peripheral blocks.

[2]The GNU General Public License (GNU GPL or GPL) is a widely used free software license, which guarantees end users (individuals, organizations, companies) the freedoms to run, study, share (copy), and modify the software. Software that allows these rights is called free software and, if the software is copylefted, requires those rights to be retained (the GPL demands both).

4.2 LEON3 SPARC V8 processor

This section contains a brief description of the LEON3 SPARC V8 processor (see Figure 4.2). The processor core can be extensively configured through the xconfig graphical configuration program (described in detail in Section 5.1). In the default configuration provided by Gaisler Research, the cache system consists or 8 + 4 Kbytes I/D cache with cache snooping enabled. The LEON3 debug support unit (DSU3) is also enabled by default, allowing downloading and debugging of programs through a serial port or JTAG.

4.2.1 Overview

LEON3 is a 32-bit processor core conforming to the IEEE-1754 (SPARC V8) architecture [38]. It is mainly designed for embedded applications, combining high performance with low complexity and low power consumption. The LEON3 core has the following main features: 7-stage pipeline with Harvard architecture, separate instruction and data caches, hardware multiplier and divider, on-chip debug support and multi-processor extensions.

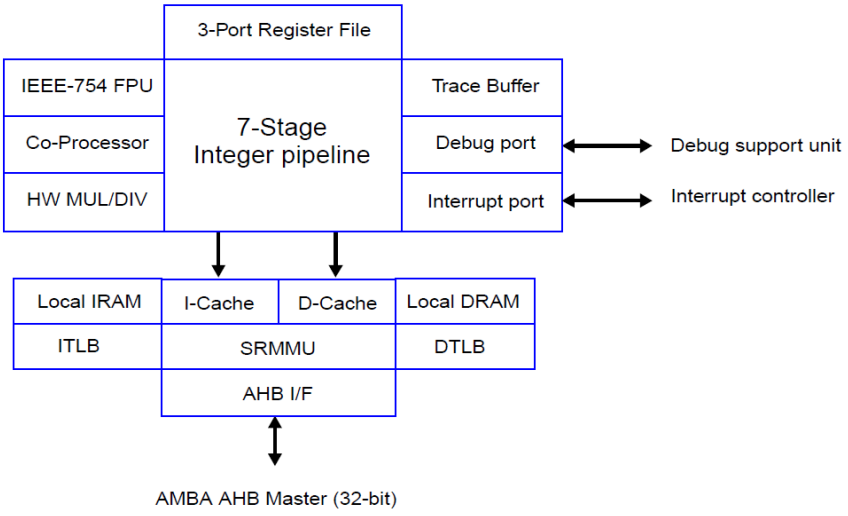

FIGURE 4.2: LEON3 processor core block diagram.

Integer unit

The LEON3 integer unit (or 7-Stage Integer pipeline in Figure 4.2) implements the full SPARC V8 standard, including hardware multiply and divide instructions. The number of register windows is configurable within the limit of the SPARC standard

(from 2 up to 32), with a default setting of 8. The pipeline consists of 7 stages with a separate instruction and data cache interface (Harvard architecture), described in detail in Section 4.2.2.

Cache sub-system

LEON3 has a highly configurable cache system, consisting of a separate instruction and data cache. Both caches can be configured with 1 to 4 sets, 1 to 256 Kbytes per set, 16 or 32 bytes per line. Sub-blocking is implemented with one valid bit per 32-bit word. The instruction cache uses also streaming during line-refill to minimize refill latency. Furthermore, the data cache uses write-through policy and implements a double-word write-buffer. The data cache can also perform bus-snooping on the AHB bus. A local scratch pad ram can be added to both the instruction and data cache controllers to allow zero-waitstates access memory without data write back.

Floating-point unit and co-processor

The LEON3 integer unit provides interfaces for a floating-point unit (FPU), and a custom co-processor. Two FPU controllers are available, one for the high-performance GRFPU (available from Gaisler Research) and one for the Meiko FPU core (available from Sun Microsystems). The floating-point processors and co-processor execute in parallel with the integer unit, and does not block the operation unless a data or resource dependency exists.

Memory management unit

A SPARC V8 Reference Memory Management Unit (SRMMU) can optionally be enabled. The SRMMU implements the full SPARC V8 MMU specification, and provides mapping between multiple 32-bit virtual address spaces and 36-bit physical memory. A three-level hardware tablewalk is implemented, and the MMU can be configured to up to 64 fully associative TLB entries.

On-chip debug support

The LEON3 integer pipeline includes functionality to allow non-intrusive debugging on target hardware. Up to four watchpoint registers can be enabled, to aid software debugging. Each register can cause a breakpoint trap on an arbitrary instruction or data address range. When the (optional) debug support unit is attached, the watchpoints can be used to enter debug mode. Through a debug support interface, full access to all

processor registers and caches is provided. The debug interfaces also allows single stepping, instruction tracing and hardware breakpoint or watchpoint control. An internal trace buffer can monitor and store executed instructions, which can later be read out over the debug interface.

Interrupt interface

LEON3 supports the SPARC V8 interrupt model with a total of 15 asynchronous interrupts. The interrupt interface provides functionality to both generate and acknowledge interrupts.

AMBA interface

The cache system implements an AMBA AHB master to load and store data to or from the caches. The interface is compliant with the AMBA-2.0 standard. During line refill, incremental burst are generated to optimize the data transfer.

Power-down mode

The LEON3 processor core implements a power-down mode, which halts the pipeline and caches until the next interrupt. This is an efficient way to minimize power-consumption when the application is idle, and does not require tool-specific support in form of clock gating.

Multi-processor support

LEON3 is designed to be use in multi-processor systems. Each processor has a unique index to allow processor enumeration. The write-through caches and snooping mechanism guarantee memory coherency in shared-memory systems.

Performance

Using two caches of 8 Kbytes each (8 Kbytes + 8 Kbytes = 16 Kbytes) and a 16x16 multiplier, the dhrystone 2.1 benchmark reports 1,500 iteration/ s/MHz using the gcc-3.4.4 compiler (-O2). This translates to 0.85 dhrystone MIPS/MHz using the VAX 11/780 value a reference for one MIPS.

4.2.2 LEON3 integer unit

The LEON3 integer unit implements the integer part of the SPARC V8 instruction set (Figure 4.3 shows a block diagram of the integer unit). The implementation is focused on high performance and low complexity. The LEON3 integer unit has the following main features:

- 7-stage instruction pipeline
- Separate instruction and data cache interface
- Support for 2 to 32 register windows
- Hardware multiplier with optional 16x16 bit MAC and 40-bit accumulator
- Radix-2 divider (non-restoring)
- Single-vector trapping for reduced code size

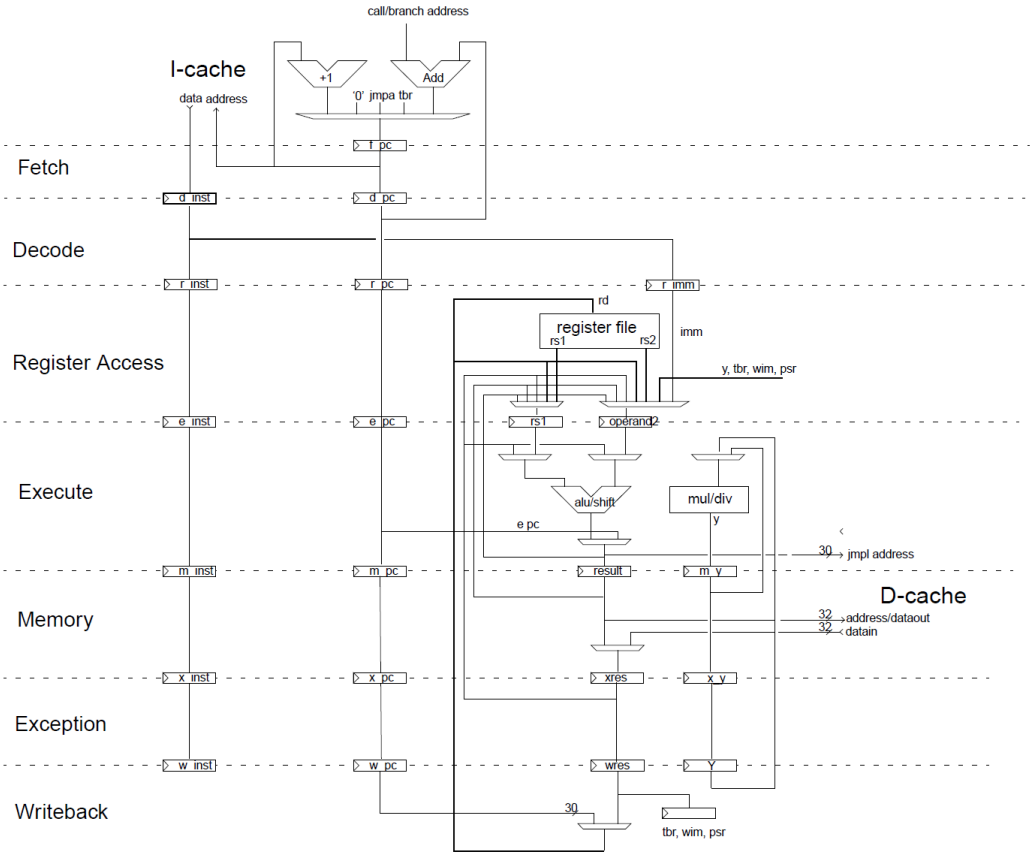

FIGURE 4.3: LEON3 integer unit datapath diagram.

The LEON integer unit uses a single instruction issue pipeline with the according 7 stages (Table 4.1 lists the cycles per instruction):

1. Instruction Fetch (FE): (a) If the instruction cache is enabled, the instruction is fetched from the instruction cache. (b) Otherwise, the fetch is forwarded to the memory controller. The instruction is valid at the end of this stage and is latched inside the IU.

2. Decode (DE): The instruction is decoded and the CALL and Branch target addresses are generated.

3. Register access (RA): Operands are read from the register file or from internal data bypasses.

4. Execute (EX): Arithmetic and Logical Unit (ALU), logical, and shift operations are performed. For memory operations (e.g., LD) and for JMPL/RETT, the address is generated.

5. Memory (ME): Data cache is accessed. Store data read out in the execution stage is written to the data cache at this time.

6. Exception (XC): Traps and interrupts are resolved. For cache reads, the data is aligned as appropriate.

7. Write (WR): The result of any ALU, logical, shift, or cache operations are written back to the register file.

Instruction	**Cycles**
JMPL, RETT	3
Double load	2
Single store	2
Double store	3
SMUL/UMUL	4*
SDIV/UDIV	35
Taken Trap	5
Atomic load/store	3
All other instructions	1

TABLE 4.1: Instruction timing. The table lists the cycles per instruction, assuming cache hit and no icc or load interlock. (* Multiplication cycle count is 5 clocks when the multiplier is configured to be pipelined.)

Additionally, Gaisler Research is assigned number 15 (0xF) as SPARC implementors identification. This value is hard-coded into bits 31:28 in the %psr register (Register Access or Write stage). The version number for LEON3 is 3, which is hard-coded in to bits 27:24 of the %psr register. Asserting the RESET input for at least 4 clock cycles resets the processor. Table 4.2 indicates the reset values of the registers, which are affected by the RESET input. All other registers maintain their value (or are undefined).

By default, the execution will start from address 0. This can be overridden by setting the RSTADDR generic in the model to a non-zero value. However, the reset address is always aligned on a 4 Kbytes boundary.

Register	**Reset value**
PC (program counter)	0x0
nPC (next program counter)	0x4
PSR (processor status register)	ET=0, S=1

TABLE 4.2: Processor reset values.

4.2.3 Instruction cache

The instruction cache can be configured as a direct-mapped cache or as a multi-set cache with associativity of 2 or 4 implementing either LRU or random replacement policy or as 2-way associative cache implementing LRR algorithm. Moreover, the set size is configurable to 1 to 64 Kbytes and divided into cache lines of 16 to 32 bytes. Each line has a cache tag associated with it consisting of a tag field, valid field with one valid bit for each 4-byte sub-block and optional LRR and lock bits. The instruction is fetched and the corresponding tag and data line updated, on an instruction cache miss to a cachable location, but in a multi-set configuration a line to be replaced is chosen according to the replacement policy.

Furthermore, if instruction burst fetch is enabled in the cache control register (CCR) the cache line is filled from main memory starting at the missed address and until the end of the line. At the same time, the instructions are forwarded (streaming) to the integer unit (IU). If the IU cannot accept the streamed instructions due to internal dependencies or multi-cycle instruction, the IU is halted until the line fill is completed. If the IU executes a control transfer instruction (such as branch/CALL/JMPL/RETT/TRAP) during the line fill, the line fill will be terminated on the next fetch. If instruction burst fetch is enabled, instruction streaming is enabled even when the cache is disabled. In this case, the fetched instructions are only forwarded to the IU and the cache is not updated. During cache line refill, incremental burst are generated on the AHB bus. Figure 4.4 demonstrates an instruction cache tag entry that consists of several fields (in two different situations). Only the necessary bits will be implemented in the cache tag, depending on the cache configuration. As an example, a 4 Kbytes cache with 16 bytes per line would only have four valid bits and 20 tag bits. Finally, the cache rams are sized automatically by the ram generators in the model.

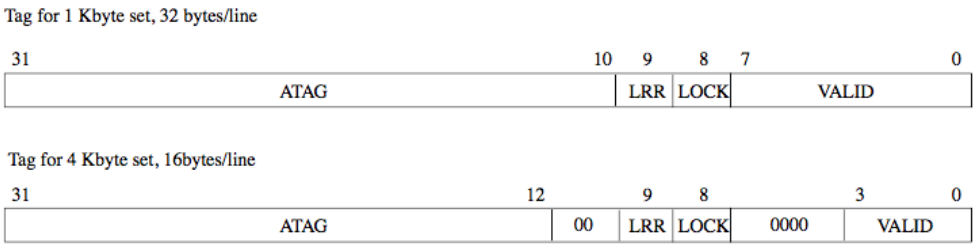

FIGURE 4.4: Instruction cache tag layout examples. Field Definitions: [31:10]: Address Tag (ATAG) - Contains the tag address of the cache line. [9]: LRR - Used by LRR algorithm to store replacement history, otherwise 0. [8]: LOCK - Locks a cache line when set. 0 if cache locking not implemented. [7:0]: Valid (V) - When set, the corresponding sub-block of the cache line contains valid data. These bits is set when a sub-block is filled due to a successful cache miss; a cache fill which results in a memory error will leave the valid bit unset. A FLUSH instruction will clear all valid bits. V[0] corresponds to address 0 in the cache line, V[1] to address 1, V[2] to address 2 and so on.

4.2.4 Data cache

The data cache can be configured as a direct-mapped cache or as a multi-set cache with associativity of 2 or 4 implementing either LRU or (pseudo-) random replacement policy or as 2-way associative cache implementing LRR algorithm. The set size is configurable from 1 to 64 Kbytes and divided into cache lines of 16 to 32 bytes. Each line has a cache tag associated with it consisting of a tag field, valid field with one valid bit for each 4-byte sub-block and optional lock and LRR bits. On a data cache read-miss to a cachable location 4 bytes of data are loaded into the cache from main memory. The write policy for stores is write-through with no-allocate on write-miss. But, in a multi-set configuration a line to be replaced on read-miss is chosen according to the replacement policy. If a memory access error occurs during a data load, the corresponding valid bit in the cache tag will not be set, and a data access error trap (tt=0x9) will be generated. Implemented traps and their individual priority described in detail in GRLIB document [38], written by Jiri Gaisler.

Moreover, the write buffer (WRB) consists of three 32-bit registers used to temporarily hold store data until it is sent to the destination device. The stored data replicated into proper byte alignment for writing to a word-addressed device, before being loaded into one of the WRB registers, for half-word or byte stores. The WRB is emptied prior to a load-miss cache-fill sequence to avoid any stale data from being read in to the data cache. A write error will not cause an exception to the store instruction, since the processor executes in parallel with the write buffer. The write cycle may not occur until several clock cycles after the store instructions has completed, depending on memory and cache activity. If a write error occurs, the currently executing instruction will take

trap 0x2b. (The 0x2b trap handler should flush the data cache, since a write hit would update the cache while the memory would keep the old value due the write error). Figure 4.5 demonstrates a data cache tag entry that consists of several fields. Only the necessary bits will be implemented in the cache tag, depending on the cache configuration. As an example, a 2 Kbytes cache with 32 bytes per line would only have eight valid bits and 21 tag bits. As the instruction cache, the cache rams are sized automatically by the ram generators in the model.

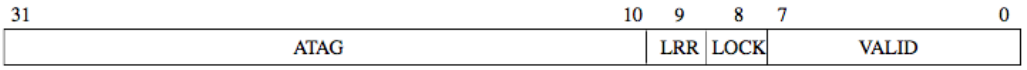

FIGURE 4.5: Data cache tag layout examples. Field Definitions: [31:10]: Address Tag (ATAG) - Contains the address of the data held in the cache line. [9]: LRR - Used by LRR algorithm to store replacement history. 0 if LRR is not used. [8]: LOCK - Locks a cache line when set. 0 if instruction cache locking was not enabled in the configuration. [3:0]: Valid (V) - When set, the corresponding sub-block of the cache line contains valid data. These bits is set when a sub-block is filled due to a successful cache miss; a cache fill which results in a memory error will leave the valid bit unset. V[0] corresponds to address 0 in the cache line, V[1] to address 1, V[2] to address 2 and V[3] to address 3.

In conclusion, cash sub-system has additional cash functionality, such as cache flushing, diagnostic cache access, cache line locking, and the LEON3 module contains a memory management unit (MMU) compatible with the SPARC V8 reference MMU can optionally be configured. The SPARC V8 manual and the GRLIB document discuss in detail the operation of MMU.

4.3 Memory interface

The memory controller handles a memory bus hosting PROM, memory mapped I/O devices, asynchronous static ram (SRAM) and synchronous dynamic ram (SDRAM). The controller acts as a slave on the AHB bus. The function of the memory controller is programmed through memory configuration registers 1, 2 and 3 (MCR1, MCR2 and MCR3) through the APB bus. The memory bus supports four types of devices: prom, sram, sdram and local I/O. The memory bus can also be configured in 8-bit or 16-bit mode for applications with low memory and performance demands. The controller decodes three address spaces (PROM, I/O and RAM) whose mapping is determined through VHDL-generics. Chip-select decoding is done for two PROM banks, one I/O bank, five SRAM banks and two SDRAM banks. Figure 4.6 shows how the connection to the different device types is made.

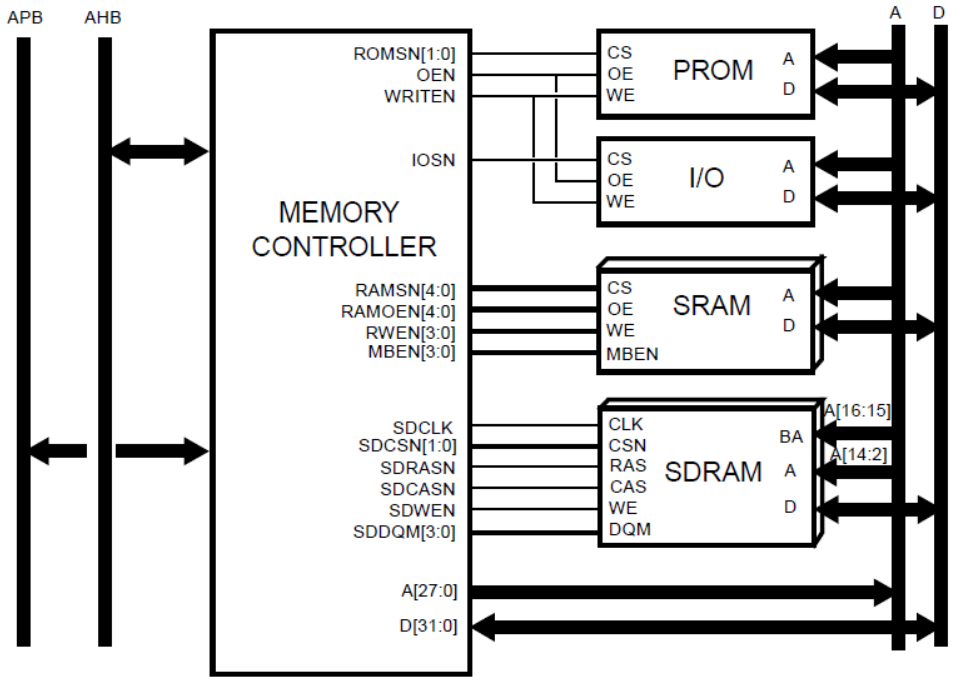

FIGURE 4.6: Memory controller conected to AMBA bus and different types of memory devices.

4.3.1 PROM access

Accesses to PROM have the same timing as RAM accesses, the differences being that PROM cycles can have up to 15 wait-states. Two PROM chip-select signals are provided, MEMO.ROMSN[1:0]. When the lower half of the PROM area as addressed MEMO.ROMSN[0] is asserted, while MEMO.ROMSN[1] is asserted for the upper half. When the VHDL model is configured to boot from internal PROM, all accesses to the lower half of the PROM area are mapped on the internal PROM and MEMO.ROMSN[0] is never asserted. Figure 4.7 shows a basic PROM read cycle.

4.3.2 Memory mapped I/O

Accesses to I/O have similar timing to ROM/RAM accesses, the main differences being that additional wait-states can be inserted by de-asserting the MEMI.BRDYN signal. The I/O select signal (MEMO.IOSN) is delayed one clock to provide stable address before MEMO.IOSN is asserted. Figure 4.8 shows a basic I/O read cycle.

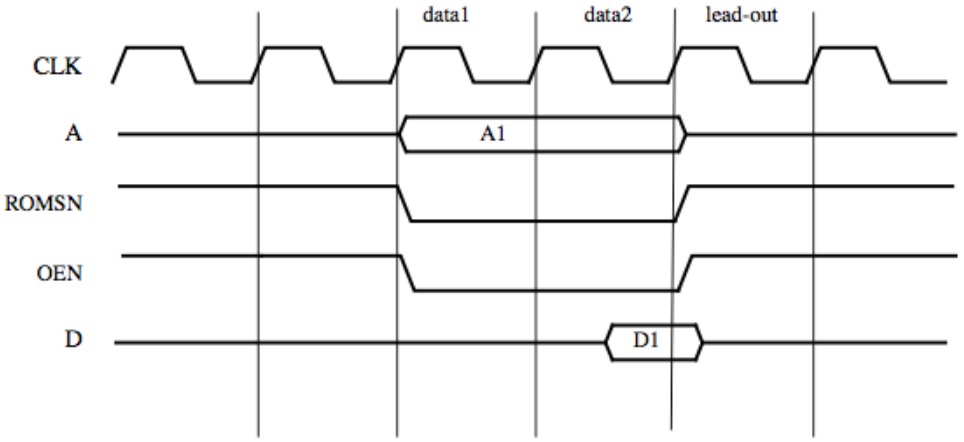

FIGURE 4.7: PROM read cycle.

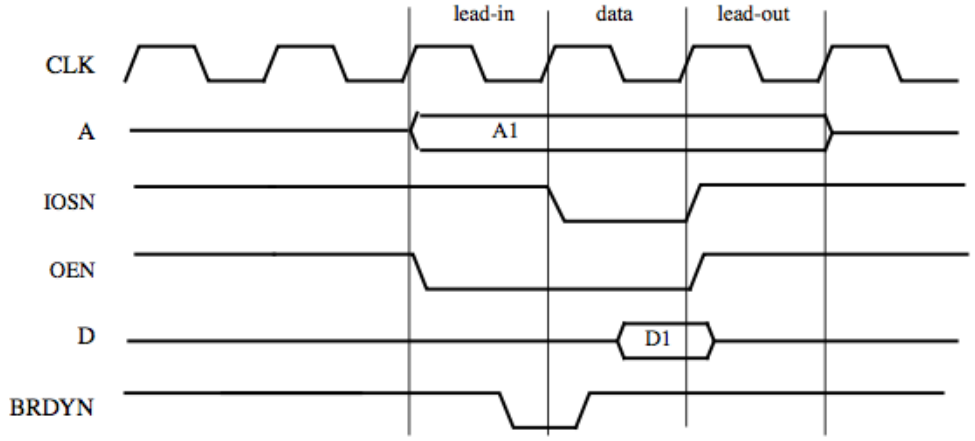

FIGURE 4.8: I/O read cycle.

4.3.3 Static RAM access

The Static RAM (SRAM) area can be up to 1 Gbyte, divided on up to five RAM banks. The size of banks 1 to 4 (MEMO.RAMSN[3:0]) is programmed in the RAM bank-size field (MCR2[12:9]) and can be set in binary steps from 8 Kbytes to 256 Mbytes. The fifth bank (MEMO.RAMSN[4]) decodes the upper 512 Mbytes. A read access to SRAM consists of two data cycles and between zero and three wait-states. Accesses to MEMO.RAMSN[4] can further be stretched by de-asserting MEMI.BRDYN until the data is available. On non-consecutive accesses, a lead-out cycle is added after a read cycle to prevent bus contention due to slow turn-off time of memories or I/O devices. Figure 4.9 shows the basic read cycle waveform (zero wait-state). For read accesses to MEMO.RAMOEN[n], a separate output enable signal (MEMO.RAMOEN[n]) is provided for each RAM bank and only asserted when that bank is selected.

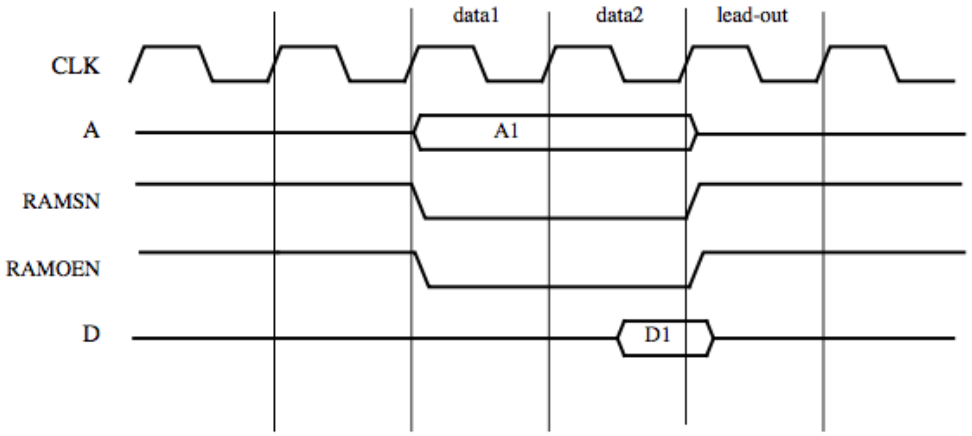

FIGURE 4.9: Static RAM read cycle (0-waitstate).

A write access is similar to the read access but takes a minimum of three cycles. Through an (optional) feed-back loop from the write strobes, the data bus is guaranteed to be driven until the write strobes are de-asserted. Each byte lane has an individual write strobe to allow efficient byte and half-word writes. If the memory uses a common write strobe for the full 16- or 32-bit data, the read-modify-write bit MCR2 should be set to enable read-modify-write cycles for sub-word writes. A drive signal vector for the data I/O-pads is provided which has one drive signal for each data bit. It can be used if the synthesis tool does not generate separate registers automatically for the current technology. This can remove timing problems with output delay. Figure 4.10 shows the basic write cycle waveform.

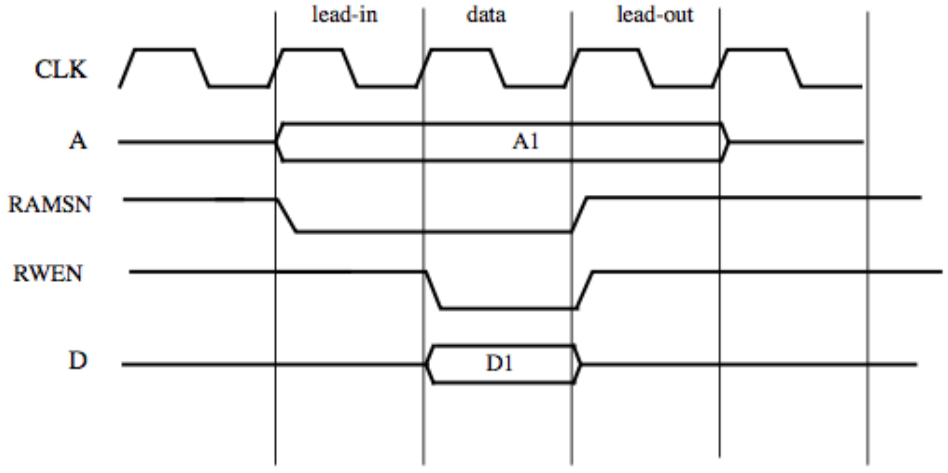

FIGURE 4.10: Static RAM write cycle.

Chapter 5

Design and Implementation of Hardware Platform for DIFT

This chapter presents the design and implementation of hardware platform for DIFT. In Section 5.1 we describe the installation and configuration of the host platform, while in Section 5.2 we present the basic design of our architecture for DIFT support in the LEON3 processor. Subsequently, in Section 5.3 we describe the implementation of our scheme in the LEON3 processor, explaining all the necessary modifications done. Lastly, in Section 5.4 we summarize and conclude the chapter comparing the simulating results by evaluating the whole platform.

5.1 LEON3 system setup

Before the implementation of our platform we must build the LEON3 system. This can be done by reading the official GRLIB manual and documentation [38]. This section offers a detailed description of the installation and configuration (setup) of the host platform which will be used to develop our idea for DIFT support on it.

5.1.1 Installation

GRLIB can be installed in any location on the host system because it is distributed as a gzipped tar-file. The following instructions unzip the tar-file:

```
gunzip -c grlib-com-1.5.0-bxxxx.tar.gz | tar xf  (or)
tar xvf grlib-com-1.5.0-bxxxx.tar.gz
```

GRLIB uses the GNU `'make'` utility to generate scripts and to compile and synthesis designs. It must therefore be installed on a Unix system or in a Unix-like environment, the one that we chose. Table 5.1 explains the file hierarchy of the distribution and lists the main directories.

Directory	**Description**
bin	Various scripts and tool support files
boards	Support files for FPGA prototyping boards
designs	Template designs
doc	Documentation
lib	VHDL libraries
netlists	Vendor specific mapped netlists
software	Software utilities and test benches
verification	Test benches

TABLE 5.1: File hierarchy of the distribution.

5.1.2 Directory organization

GRLIB is organized around VHDL libraries, where each IP vendor is assigned a unique library name. A unique subdirectory grlib/lib contains all vendor-specific source files and scripts, and each vendor is also assigned under the same subdirectory. The vendor-specific directory can contain subdirectories, to allow for further partitioning between IP cores etc. Other vendor-specific directories are also delivered with GRLIB, but are not necessary for the understanding of the design concept. Table 5.2 shows the basic directories delivered with GRLIB under `grlib-1.x.y/lib`.

Directory	**Description**
grlib	Packages with common data types and functions
gaisler	Cobham Gaislers components and utilities
tech/*	Target technology libraries for gate level simulation
techmap	Wrappers for technology mapping of marco cells (RAM, pads)
work	Components and packages in the VHDL work library

TABLE 5.2: Directories delivered with GRLIB under `grlib-1.x.y/lib`.

5.1.3 Building the host platform

GRLIB is design to work with a large variety of hosts. Our project has been developed on Microsoft Windows 7 operating system. Hardware has been synthesized and downloaded in Windows by using Cygwin [39] which is a Unix-like environment for Microsoft Windows. Figure 5.1 shows a typical session of Cygwin and X Windows graphical system, used for configuration of LEON3. The following host software must be installed for the GRLIB configuration scripts to work:

- Bash shell
- GNU make
- GCC
- Tcl/Tk-8.4
- Patch utility
- X Windows graphical system (required for Tcl/Tk on Cygwin)
- Xilinx ISE Design Suite 14.7

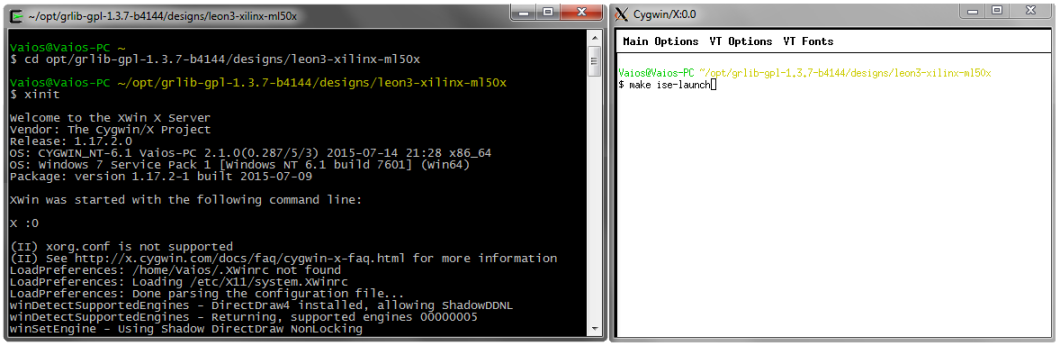

FIGURE 5.1: Typical session of Cygwin and X Windows graphical system (X server).

Now we will explain the basic steps to set up the paths to tools properly. For Xilinx ISE tools the XILINX environment variable must point at the installation of ISE. This can be done in the Cygwin shell by typing:

`export XILINX=c:\\Xilinx\\14.7\\ISE_DS\\ISE`,

which should lead to a print-out matching the Xilinx ISE installation. Paths to the EDA tools must be included in the PATH variable. The PATH variable must be set by typing:

`export PATH=$PATH:$XILINX/bin/nt`.

We also installed an X server (xorg-server, xinit packages), in order to run the graphical configuration tools that come with GRLIB. Lastly, we installed the base set of Xilinx libraries which are taken from the Xilinx ISE installation. The variable `\$XILINX` needs to be set like it is from the ISE initialization scripts by typing:

`export XILINX=/usr/local/xilinx/14.7/ISE_DS/ISE`.

UNISIM libraries are then installed with the command:

`make install-unisim`.

Implementing a LEON3 system is typically done using one of the template designs on the designs directory. GRLIB offers a wide range of LEON3 template designs, but we chose the LEON3 template design for the Xilinx ML50X design. Implementation is basically done in three basic steps:

- Configuration of the design using xconfig
- Simulation of design and test bench
- Synthesis and place&route

The template design is located in `$GRLIB/designs/leon3-xilinx-ml50x` directory, and is based on three files:

- `config.vhd` - A VHDL package containing design configuration parameters. Automatically generated by the xconfig GUI tool.
- `leon3mp.vhd` - Contains the top level entity and instantiates all on-chip IP cores. It uses `config.vhd` to configure the instantiated IP cores.
- `testbench.vhd` - Test bench with external memory, emulating the Xilinx ML50X board.

Each core in the template design is configurable using VHDL generics. The value of these generics is assigned from the constants declared in `config.vhd`, created with the xconfig GUI tool.

5.1.4 Configuration

The LEON3 processor needs to be configured according to the requirements of the project and the `leon3mp.vhd` file needs to be modified according to the configurations. The configuration of the LEON3 processor is done with the xconfig GUI tool. The xconfig GUI tool is launched by typing `'make xconfig'` in the Cygwin shell. Figure 5.2 shows a typical session of xconfig GUI tool used for configuration of LEON3 processor. In this menu we set the default values for LEON3 processor and we checked all other values to be set according to our architecture. For instance, in VHDL Debugging there is an option, called Accelerated UART tracing, which allows LEON3 to interact with I/O of our simulation software (iSIM and Xilinx ISE Design Suite). We enabled this option, in order to see in the simulation console the output of LEON3 processor (such as printf() output or sequence of instructions and program counter value every time).

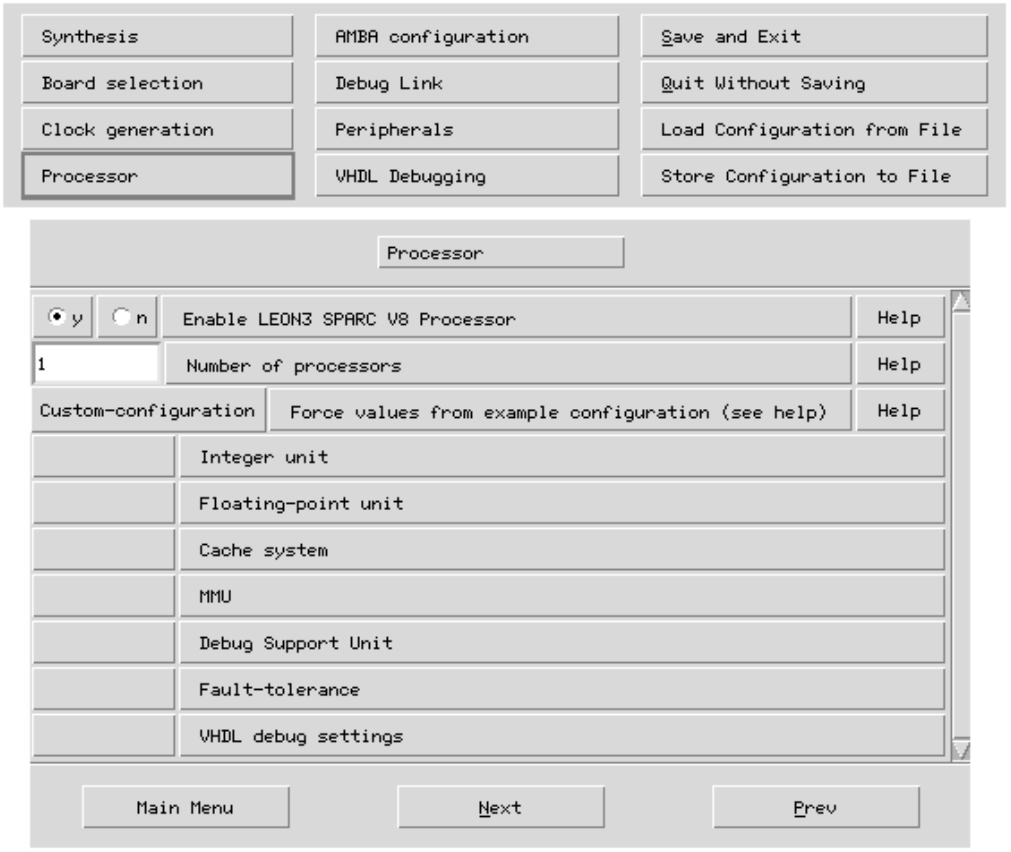

FIGURE 5.2: Typical session of xconfig GUI tool (processor configuration).

5.1.5 Software development

The LEON3 processor is supported by several free software tool chains, such as Bare-C cross-compiler system (BCC), RTEMS cross-compiler system (RCC), eCos real-time O/S for LEON, or Linux-build embedded linux. We used LEON Bare-C Cross Compilation System (BCC). BCC is a GNU-based cross-compilation system for LEON processors. The following components are included:

- GNU C/C++ cross-compiler (4.4.2 and 3.4.4)
- GNU Binutils (assembler, linker)
- Newlib embedded C library
- Bare-C run-time system with interrupt support
- GNU debugger (GDB)
- Windows (MinGW) and Linux hosts

BCC allows cross-compilation of C and C++ applications for LEON3. Previous versions of LEON (e.g., LEON2) are not supported, and this is the main reason we chose LEON3 for development. Using the GDB debugger, it is possible to perform source-level symbolic debugging, either on the TSIM simulator or on real target hardware using GRMON. Applications can be compiled to run from directly from PROM or first loaded into RAM before being executed. BCC helped us to compile source files to Assembly language and then to binary files, in order to test LEON3 platform. Compilation and debugging of applications is typically done in the following steps:

1. Compile and link program with gcc
2. Debug program on a simulator or remote target
3. Create boot-prom for a standalone application

BCC supports both tasking and non-tasking C and C++ programs. Compiling and linking is done in the same manner as with a host-based gcc. The produced binaries will run on LEON3 system, without requiring any switches during compilation. Ordinary C programs can be compiled without any particular switches to the compiler driver:

```
sparc-elf-gcc -msoft-float -g -O2 hello.c -o hello.exe
```

The default link address is start of RAM, for LEON is `0x40000000`. Other link addresses can be specified through the -Ttext option. For convenience purposes we used also the command `'make soft'` to build the binary files and `'make soft-clean'` to clean up them.

5.1.6 LEON3 simulation

In section 5.1.4 we described how to configure LEON3 processor properly and in this section we will show the simulation method of LEON3 processor. In this thesis we used Xilinx ISE Design Suite and iSIM simulator, in order to test the binaries (section 4.1.5 offers detailed description how to compile and build binaries for LEON3). We used also Cygwin and X Windows graphical system, to open LEON3 design in Xilinx ISE Design Suite, because LEON3 processor comes with a makefile and the only way to open that file is through the terminal. After entering the directory of LEON3 design using Cygwin (`'cd'` command), we used the following command to run Xilinx ISE Design Suite from LEON3 directory:

```
make ise-launch
```

Xilinx ISE Design Suite environment showed up and all the files from GRLIB and `testbench.vhd` file appeared in the Design Hierarchy window. We configured Xilinx ISE Design Suite to use iSIM simulator and change the compile order to manual. We clicked on `testbench.vhd` file and then in the Simulate Behavioral Model button. The iSim simulator window showed up, loading the default signals of LEON3 design. Figure 5.3 shows a typical session of iSIM simulator for LEON3 processor. After compiling the default C file (systest.c), we built the binary file that will run inside LEON3 processor.

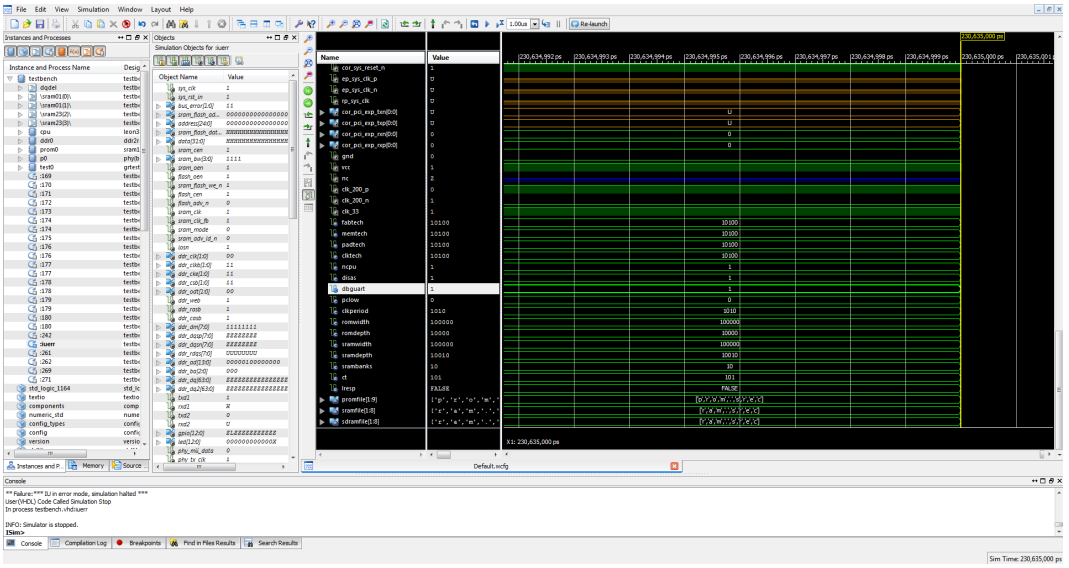

FIGURE 5.3: Typical session of iSIM simulator for LEON3 processor.

During the simulation, we can observe every instruction of the binary following by the specific time (measured in nanoseconds) that executed by the LEON3 processor. At the end of the simulation, the following text showed up, as a sign that the LEON3 processor was stopped (halted) and simulation finished successfully:

```
...
** Failure:*** IU in error mode, simulation halted ***
User(VHDL) Code Called Simulation Stop
In process testbench.vhd:iuerr

INFO: Simulator is stopped.
...
```

Lastly, we created new `.wcfg` files containing the signals that we wanted to observe every time we simulate a binary in LEON3 processor, mainly from Integer Unit (IU) and Memory.

5.2 Policies

In this section we offer a detailed description of existing DIFT policies for buffer overflow protection and how we designed ours. DIFT policies are one of the most important designing parameters that help researchers to create and develop efficient DIFT systems that protect both binaries and system from potential attackers.

5.2.1 Overview

Every time designers want to develop a DIFT system, they have to balance important design criteria such as performance overhead, supported languages, cost and backwards compatibility. First of all, they must think about policies and what range of attacks they want to target. Developing DIFT policies for each attack is very important aspect of every DIFT system and developers have to design a DIFT policy which avoids real-world false positives and minimizes false negatives. Additionally, after designing policies properly, they have to implement the DIFT system architecture to support their policies, safely, flexibly, and not costly. Once the applications and vulnerabilities have been identified, DIFT policies should be formulated to protect applications from security attacks that exploit the vulnerabilities.

To create our system, we followed these steps, in order to build a complete DIFT system that supports our policies and has prospects of further work for the future. In the following sections we will discuss about the policies that we chose for our platform, our system architecture and the modifications that must be done in the LEON3 processor, to support our DIFT scheme efficiently. This process should not be seen as a traditional "waterfall" development methodology. It is very likely to be iterative, as experience in later steps may cause previous design decisions to be reconsidered. A better understanding of DIFT policy design may point to new problems and vulnerabilities which can be addressed by DIFT. Similarly, tradeoffs made when deciding on the DIFT architecture may warrant changes to the policies or vulnerabilities addressed by the system. For these reasons we developed policies that can be reconfigured and changed for extra support or further development of our platform.

Ultimately, despite decades of research, buffer overflow attacks remain a critical threat to system security. For instance, the successful exploitation of a buffer overflow typically results in arbitrary code execution with the privilege level of the vulnerable application. Over the years researchers have proposed the use of Dynamic Information Flow Tracking technique to prevent buffer overflows in unmodified binaries, due to the flaws in modern buffer overflow protection systems, described in detail in Chapter 2. The

following section describes how to create efficient policies for buffer overflow prevention and other corruptions, presenting the shortcomings of currently available approaches. The most important step to preventing an attack is to define the vulnerability itself, specifying the definition in terms of information flow if possible.

5.2.2 Existing DIFT Policies

There are two major existing policies for buffer overflow protection using DIFT. The first one called bounds-check recognition (BR) and the second pointer injection (PI). The approaches are both based on DIFT using tags, but differ in tag propagation rules, the indication of attacks, and whether tagged input can ever be validated by application code. In the first policy (BR), most DIFT systems [18, 25, 28, 31], use a BR policy to prevent buffer overflow attacks by forbidding dereferences of untrusted information without a preceding bounds-check. A buffer overflow is detected when a tagged code or data pointer is used. Certain instructions, such as logical AND or comparison against constants, are assumed to be bounds-check operations that represent validation of untrusted input by the program code. Hence, these instructions untaint any tainted operands. However, the most critical drawback of BR-based policies is an unacceptable number of false positives with commonly used software. Input validation on binaries has an inherent false positive risk, as there is no debugging information available to disambiguate which operations actually perform validation. While the tainted value that is bounds checked is untainted by the DIFT system, none of the aliases for that value in memory or other registers will be validated. Moreover, even trivial programs can cause false positives because not all untrusted pointer dereferences need to be bounds checked. So, false positives occur during common system operations such as compiling files with `gcc` command or compressing data with `gzip` command. In practice, false positives occur only for data pointer protection. Finally, control pointer protection alone has been shown to be insufficient [6].

On the other hand, recent works in DIFT [40, 41] have proposed a pointer injection (PI) policy for buffer overflow protection. Rather than use the first approach (BR) and recognize bounds checks, PI enforces a different invariant. Untrusted information should never directly supply a pointer value, but tainted information must always be combined with a legitimate pointer from the application before it can be dereferenced. In practice, we have applications that frequently add an untrusted index to a legitimate base address pointer from the application's address space. However, existing exploitation techniques rely on injecting pointer values directly, such as by overwriting the return address, frame pointers, global offset table entries, or malloc chunk header pointers. A PI policy uses two tag bits per memory location, one to identify tainted data (Tag bit)

and the other to identify pointers (Pointer bit), in order to prevent buffer overflows attacks. PI-based policies do not provide specific scheme for untainting data, nor rely on any bounds-check recognition. The Pointer bit is set only for legitimate pointers in the application and propagated only during valid pointer operations such as adding a pointer to a non-pointer or aligning a pointer to a power of two boundary. Security attacks are detected if a tainted pointer is dereferenced and the Pointer bit is not set. The most important advantage of PI-based policies is that avoid the false positive and negative issues that appear in BR-based policies, because it does not rely on bounds-check recognition. Nevertheless, a significant drawback is that PI policy requires legitimate application pointers to be identified. Finally, the original proposal of PI policy was limited to simulation studies with performance benchmarks and only a few evaluations have been done on a wide range of large applications.

5.2.3 Our DIFT Policies

We designed our system to support DIFT with Pointer Injection (PI) policy, because after a research in previous DIFT systems and benchmarks PI policy seems to be the best option. The rules are intended to be as conservative as possible while still avoiding false positives. In our platform we are going to use two tag bits per word of memory and hardware register, since our policy is based on pointer injection. The taint (T) bit is set for untrusted data, and propagates on all arithmetic, logical, and data movement instructions. Any instruction with a tainted source operand propagates taint to the destination operand (e.g., register, memory). The pointer (P) bit is initialized for legitimate application pointers and propagates during valid pointer operations such as pointer arithmetic. A security exception is thrown if a tainted instruction is fetched or if the address used in a load, store, or jump instruction is tainted and not a valid pointer. Therefore, our PI policy allows a program to combine a valid pointer with an untrusted index, but not to use an untrusted pointer directly. The DIFT rules for tag propagation for buffer overflow prevention for Taint bit and Pointer bit are represented in Table 5.2 and Table 5.3 respectively.

Our propagation rules for the Taint bit (Table 5.2) is the most fundamental, because they allow to us implementing further policies based on the rules for the Taint bit. We set the Taint bit of every instruction, register or memory location for untrusted data which could be used later for low-level attacks such as memory corruptions, or even high-level attacks such as SQL injections. For example, an `add` instruction which adds two registers that one of them is tainted, gives a tainted result that taints the destination register as well (or sets the Taint bit of the destination register). Furthermore, the propagation rules for a `load` or `store` instruction set the Taint bit of the destination

register or memory location, if the source register or memory location was tainted (the Taint bit was set).

Instruction	**Syntax**	**Meaning**	**Tag Propagation**
Load	`ld [%r1+imm], %r2`	r2←[r1]	T(r2)←T([r1+imm])
Store	`st %r1, [%r2+imm]`	[r2]←r1	T([r2+imm])←T(r1)
Add/sub/or/and	`add %r1, %r2, %r3`	r3←r1+r2	T(r3)←T(r1) | T(r2)
Other ALU	`xor %r1, %r2, %r3`	r3←r1⊕r2	T(r3)←T(r1) | T(r2)
Sethi	`sethi imm, %r1`	r1←imm	T(r1)←0
Jump	`jmpl %r1+imm, %r2`	r2←PC PC←r1+imm	T(r2)←0

TABLE 5.3: The DIFT propagation rules for the Taint bit. T(r) refers to the Pointer bit (T) for register, instruction (r) or memory location of ([r+imm]). Table shows simplified instructions, real instructions are based on instruction set of SPARC V8.

Our propagation rules for the Pointer bit (Table 5.3) are derived from pointer operations used in real code. Any operation that could reasonably result in a valid pointer should propagate the Pointer bit. For instance, we propagate the Pointer bit for data movement instructions such as `load` and `store`, because every time a pointer is copied, the Pointer bit must be copied as well. Another typical example for Pointer bit use is the `and` instruction which is often used to align pointers. Thus, the propagation rule for the and instruction sets the Pointer bit of the destination register if one source operand is a pointer, and the other is a non-pointer. Moreover, we have to propagate the Pointer bit for instructions that may initialize a pointer to a valid address in statically allocated memory using the `sethi` instruction. This instruction sets the most significant 22 bits of a register to the value of its immediate operand and clears the least significant 10 bits, so we propagate the Pointer bit of the `sethi` instruction to its destination register. To initialize Pointer bit, we must scan any data segments for 32-bit values that are within the virtual address range of the current executable or shared library and set the Pointer bit for any matches, such immediate operand of the `sethi` instruction.

Instruction	**Syntax**	**Meaning**	**Tag Propagation**
Load	`ld [%r1+imm], %r2`	r2←[r1]	P(r2)←P([r1+imm])
Store	`st %r1, [%r2+imm]`	[r2]←r1	P([r2+imm])←P(r1)
Add/sub/or	`add %r1, %r2, %r3`	r3←r1+r2	P(r3)←P(r1) | P(r2)
And	`and %r1, %r2, %r3`	r3=r1|r2	P(r3)←P(r1) ⊕ P(r2)
Other ALU	`xor %r1, %r2, %r3`	r3←r1⊕r2	P(r3)←0
Sethi	`sethi imm, %r1`	r1←imm	P(r1)←P(instr)
Jump	`jmpl %r1+imm, %r2`	r2←PC PC←r1+imm	P(r2)←1

TABLE 5.4: The DIFT propagation rules for the Pointer bit. P(r) refers to the Pointer bit (P) for register, instruction (r) or memory location of ([r+imm]). Table shows simplified instructions, real instructions are based on instruction set of SPARC V8.

Lastly, the Check Rules for PI policy (Table 5.4) shows that a security exception is raised only if the condition in the rightmost column is true. In other words, if a tainted instruction is fetched in the instruction fetch stage, or if the address used in a load, store, or jump instruction is tainted and not a valid pointer (not legitimate pointer). This section describes our DIFT policies while the precious one discuss about the deficiencies of current DIFT-based buffer overflow approaches. The most promising techniques in the future must emphasize on analysis for reliably preventing buffer overflows in both userspace and the operating system, in order to build secure systems.

Instruction	**Syntax**	**Meaning**	**Check Rule**
Load	`ld [%r1+imm], %r2`	r2←[r1]	T(r1) & P(r1)'
Store	`st %r1, [%r2+imm]`	[r2]←r1	T(r1) & P(r1)'
Jump	`jmpl %r1+imm, %r2`	r2←PC PC←r1+imm	T(r1) & P(r1)'

TABLE 5.5: The DIFT check rules for PI policy.

5.3 Architecture

In this section we discuss about the main architecture of our DIFT system for LEON3 processor. We also describe the basic idea on how to create systems with DIFT support effectively, underlying the policies that we have already designed and described in detail in the previous section 5.2.3.

5.3.1 Overview

The study showed that the overhead of per-byte tags is unnecessary, so we defined tag bits per 32-bit word instead of per byte. Two variables with different security specifications almost never will be packed into a single word, if we consider the way language compilers allocate variables. Like most other DIFT architectures, our DIFT system does not track implicit information flow since it would cause a large number of false positives. In addition, unlike information leaks, security exploits usually rely only on tainted code or data that is explicitly propagated through the system. In this chapter we describe a DIFT prototype which is based on the LEON3 processor (Chapter 4 describes original LEON3 architecture and functionality in detail), a 32-bit VHDL synthesizable core developed by Gaisler Research [38]. In this section we discuss about how we modified LEON3 processor, to include all the security features that we described in section 5.2.

5.3.2 Hardware Architecture

The LEON3 processor fully implements the SPARC V8 standard while the SPARC Instruction Set Architecture (ISA) is very similar to other RISC ISAs (e.g., MIPS). However, the SPARC uses condition codes for branch instructions, and supports register windows rather than relying on frequent compiler-generated register saves and restores on function calls. LEON3 processor uses a single-issue, 7-stage pipeline (described in detail in Chapter 4). We started by modifying the VHDL code to add tags to all user-visible registers, as well as cache (both instruction and data cache) and memory locations. We also added support for the low-overhead security exceptions and extended all buses to accommodate tag transfers in parallel with the associated data. A simplified diagram of our system hardware is represented in Figure 4.4, showing the LEON3 processor 7-stage pipeline.

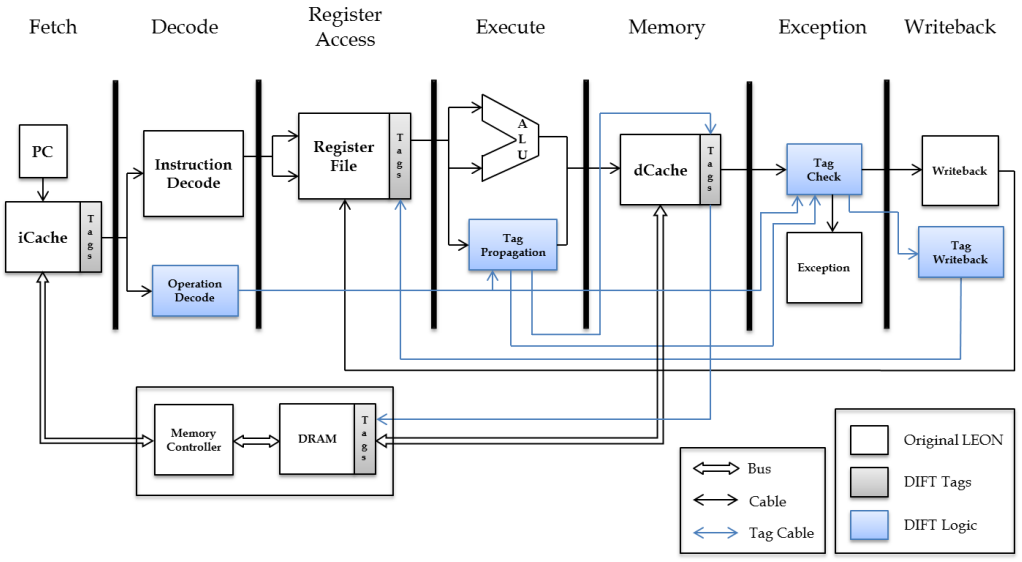

FIGURE 5.4: Simplified diagram of modified LEON3 processor (7-stage pipeline).

The LEON3 processor operates on tags as instructions flow through all of the stages of the pipeline. The Fetch stage is the first stage which every instruction is been processed. In this stage we check the tag of the instruction fetched from the Instruction Cache (iCache). If an instruction is tainted (Taint bit is set), an exception must be raised in the Check Tag module. The next stage is the Decode stage which decomposes each instruction and checks its opcode. We check the instruction type (e.g., arithmetic, logical, and data movement instructions) and we decide what propagation rules must be followed for every instruction. The Access stage reads the tags for the instruction operands from the Register File, including the destination operand. When an instruction completed this stage, we know the exact tag propagation and check rules to apply for

this instruction. It is very important to understand that the security rules applied for each of the tag bits are independent of one another. The next two stages, Execute and Memory, propagate source tags to the destination tag in Data Cache (dCache), in accordance with the active policies. The Exception stage performs any necessary tag checks and raises a precise security exception if needed. In the final Writeback stage, all register or state updates are performed. Ultimately, we gave great importance to the pipeline forwarding for the tag bits which is implemented similar to, and in parallel with, forwarding for regular data values.

Our current implementation of the memory system simply extends all cache lines and buses by four tag bits per 32-bit word. We used only the two of the four bits (Taint and Pointer bits), and we set the two extra for further modification (extending policies and security). A good try for future versions of this system, is to implement the multi-granular tag storage approach proposed by Suh et al. [27], where tags are allocated on demand for cache lines and memory pages that actually have tagged data (described in Chapter 3 in section 3.3 Related Work). This method will reduce tag storage overhead significantly, as tags have very high spatial locality in practice.

5.4 Implementation

In this section we describe the basic modification of the LEON3 processor, in order to implement a DIFT system. We also offer a detailed scheme of the original LEON3 processor and the modified LEON3 processor with the new signals, modules, registers or other logic. Original LEON3 processor described in detail in Chapter 4, and all of its basic elements and modules (e.g., registers and caches).

5.4.1 Overview

The LEON3 is an open source processor developed by Jiri Gaisler (Gaisler Research) and the LEON3 core is synthesized using VHDL and is intended for system-on-a-chip designs. We started configuring the LEON3 processor using the Cygwin tool [39] (described in detail in Section 5.1 and 5.1.3), which can be used to operate the LEON3 processor. The design suite is very similar to Unix-based environments and is used for both simulation and synthesis purposes. We used Cygwin tool only to open and configure the design, and we used Xilinx ISE Design Suite to modify the VHDL code and simulate the model. The system was compiled, synthesized, and programmed through Xilinx ISE Design Suite, using the GRLIB makefiles and the Xilinx PROM builder. The LEON3 processor is organized in GRLIB library and consists of many parts. User designs with top-level

files, that instantiate complete SoC designs, are located in the `$GRLIB/designs` directory (described in section 4.1.2). In our thesis we used the Xilinx ML50X design, which is located in the `$GRLIB/designs/leon3-xilinx-ml50x` directory. Figure 5.5 illustrates the LEON3 processor VHDL files organization, showing the Top-Level entity is consisted by the other entities.

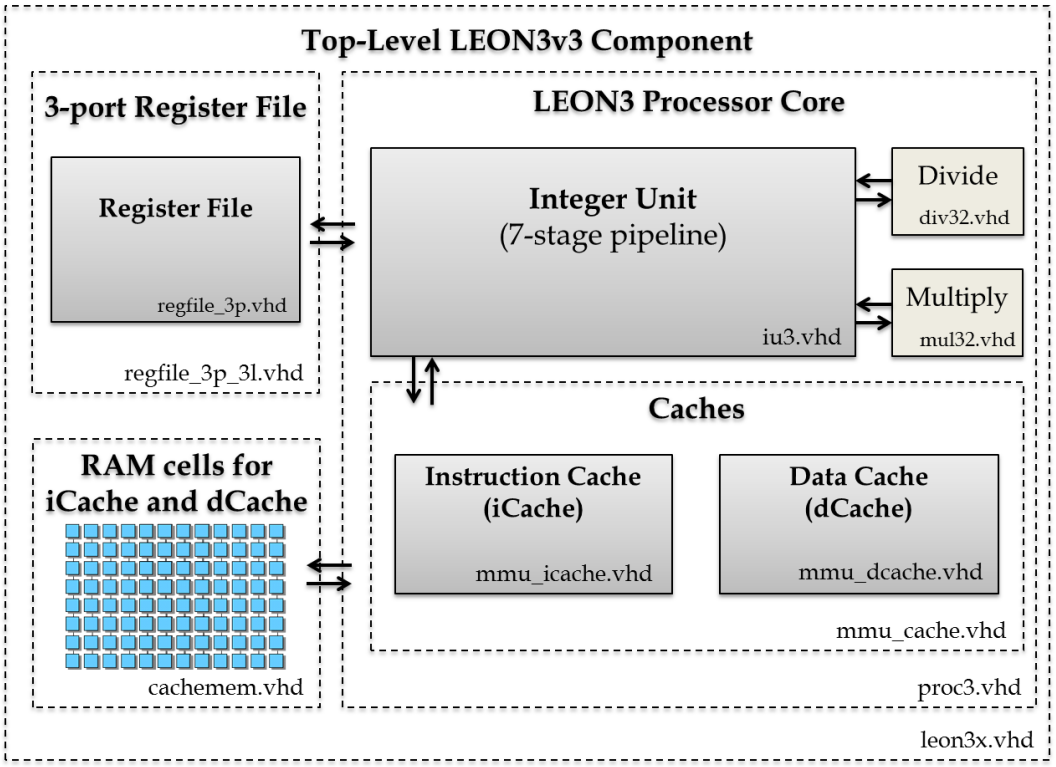

FIGURE 5.5: The LEON3 processor IP cores organization (.vhd files). Arrows in opposite directions indicate signal buses that exchange data between two entities.

As we can see, `leon3x.vhd` entity is the Top-Level LEON3v3 component with all options which contains the LEON3 Processor Core. The LEON3 Processor Core is the `proc3.vhd` entity and contains the Integer Unit (`iu3.vhd`), Caches (`mmu_cache.vhd`), multiply and divide entities (`mul32.vhd` and `div32.vhd`) which we did not modify at all. The most important entity is the iu3.vhd entity, because it operates all the instructions and propagates all the necessary tags. All of these files have library files that contain information about the signal types that they use. For example, `iu3.vhd` entity has a library file, called `libiu.vhd`, which contains the declaration of signal types that are used by `iu3.vhd` entity. Complete systems with many components written in VHDL, such as LEON3 processor, use libraries in order to organize better the signal types. Table 5.6 lists the LEON3 processor core (with pipeline, multiply, divide and cache controller units) inputs and outputs (`proc3.vhd`).

Inputs

Name	Type
clk	std_ulogic
rstn	std_ulogic
ahbi	ahb_mst_in type
ahbsi	ahb_slv_in_type
ahbso	ahb_slv_out_vector
rfo	iregfile_out_type
cramo	cram_out_type
tbo	tracebuf_out_type
fpo	fpc_out_type
cpo	fpc_out_type
irqi	l3_irq_in_type
dbgi	l3_debug_in_type

Outputs

Name	Type
holdn	std_ulogic
ahbo	ahb_mst_out_type
rfi	iregfile_in_type
crami	cram_in_type
tbi	tracebuf_in_type
fpi	fpc_in_type
cpi	fpc_in_type
irqo	l3_irq_out_type
dbgo	l3_debug_out_type

TABLE 5.6: List of LEON3 processor core (with pipeline, multiply, divide and cache controller units) inputs and outputs (proc3.vhd).

We did not change at all these signal types, but we added extra signals in the library file (`libiu.vhd`), to support tag transfer from and to Register File and Caches. These signals works like array of signals, for instance, a value of type `iregfile_in_type` can be written like `rfi.value`, by simplifying the writing of the VHDL code.

5.4.2 LEON3 Processor Core

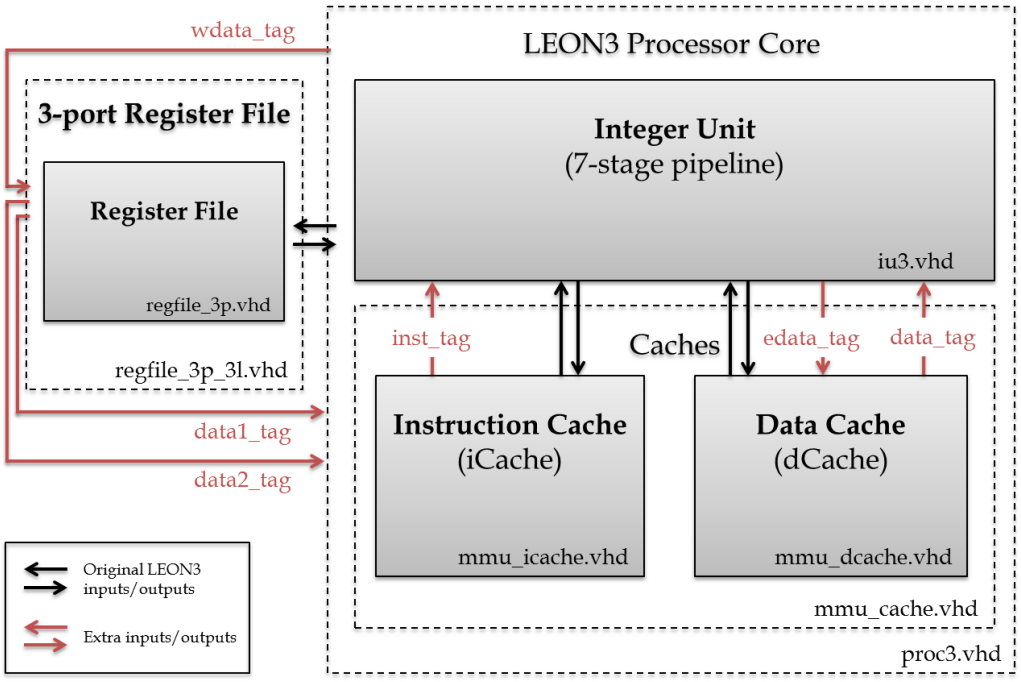

FIGURE 5.6: Modified LEON3 processor with Register File and Caches.

The LEON3 processor core uses the signals mentioned before (in section 5.4.1) and in this section we discuss about the modifications we made to support DIFT in the `proc3.vhd` entity. Figure 5.6 illustrates the modified LEON3 processor with the extra inputs and outputs. The signals which transfer the data (and tags) from Integer Unit to Register File and Caches are declared in the `libiu.vhd` entity.

Inputs

Name	Length	Description
raddr1	10 bits	Read address 1
raddr2	10 bits	Read address 2
waddr	10 bits	Write address
wdata	32 bits	Write data
wdata_tag	*4 bits*	*Write tag*
ren1	1 bit	Read enable 1
ren2	1 bit	Read enable 2
wren	1 bit	Write enable
diag	4 bits	Write data

Outputs

Name	Length	Description
data1	32 bits	Read data 1
data1_tag	*4 bits*	*Read tag 1*
data2	32 bits	Read data 2
data2_tag	*4 bits*	*Read tag 2*

TABLE 5.7: Register File inputs (iregfile_in_type) and outputs (iregfile_out_type). *Italics* indicate extra inputs/outputs.

At first, we added extra signals to support tag transfer and tag propagation, by editing the signal types in the `libiu.vhd` entity. This method allows developers to use these signals in every entity (such as `iu3.vhd` or `proc3.vhd`) simply by including this library. To support tag transfer in Register File, we added one input for write tag and two outputs for read tag 1 and read tag 2 respectively. Every time Integer Unit writes data to Register File, it writes also the data tag, and every time Integer Unit reads data from Register File, reads data tag as well.

Inputs

Name	Length	Description
rpc	32 bits	Raw address
fpc	32 bits	Latched address
dpc	32 bits	Latched address
rbranch	1 bit	Instruction branch
fbranch	1 bit	Instruction branch
inull	1 bit	Instruction nullify
su	1 bit	Super-user
flush	1 bit	Flush iCache
fline	29 bits	Flush line offset

Outputs

Name	Length	Description
data	32 bits	iCache data
data_tag	*4 bits*	*iCache tag*
set	2 bits	Set
mexc	1 bit	Mexc
hold	1 bit	Hold
flush	1 bit	Flush in progress
diagrdy	1 bit	Diagnostic access ready
diagdata	32 bits	Diagnostic data
mds	1 bit	Memory data strobe
cfg	32 bits	Cfg
idle	1 bit	Idle mode

TABLE 5.8: Instruction Cache (iCache) inputs (icache_in_type) and outputs (icache_out_type). *Italics* indicate extra inputs/outputs.

After modifying Register File inputs and outputs we continued with Instruction Cache (iCache) inputs and outputs, in order to support tag transfer as well. We did not changed iCahce inputs at all, but we added an extra output to transfer tag from iCache to Register File. Table 5.8 lists the Instruction Cache (iCache) inputs/outputs, while extra inputs/outputs indicated with Italics.

Inputs

Name	**Length**	**Description**
asi	8 bits	Asi
maddress	32 bis	Memory address
eaddress	32 bis	Execute address
edata	32 bis	Execute data
edata_tag	*4 bits*	*Execute tag*
size	2 bits	Size
enaddr	1 bit	Enable address
eenaddr	1 bit	Execute enable address
nullify	1 bit	Nullify
lock	1 bit	Lock
read	1 bit	Read
write	1 bit	Write
flush	1 bit	Flush
flushl	1 bit	Flush line
dsuen	1 bit	Dsu enable
msu	1 bit	Memory stage supervisor
esu	1 bit	Execution stage supervisor
intack	1 bit	

Outputs

Name	**Length**	**Description**
data	32 bits	dCache data
data_tag	*4 bits*	*dCache tag*
set	1 bit	Set
mexc	1 bit	Mexc
hold	1 bit	Hold
mds	1 bit	Memory data strobe
werr	1 bit	Write enable error
icdiag	32 bits	iCache diagnostic
cache	1 bit	Cache
idle	1 bit	Idle mode
scanen	1 bit	Scan enable
testen	1 bit	Test enable
hit	1 bit	Hit successfully

TABLE 5.9: Data Cache (dCache) inputs (dcache_in_type) and outputs (dcache_out_type). *Italics* indicate extra inputs/outputs.

Finally, we modified Data Cache (dCache) inputs and outputs, to support tag transfer to both Caches. We added an extra input to put data tag in dCache, and an extra output that reads data tag from dCache to Register File. Table 5.9 lists the Data Cache (dCache) inputs/outputs, while extra inputs/outputs indicated with Italics. Now we can transfer data with tag in the LEON3 processor core unit (`proc3.vhd`), which contains Register File and Caches.

5.4.3 Integer Unit

Integer Unit is the main entity of the LEON3 processor core, which operates all the instructions come from Instruction Cache (iCache) and stores data to Data Cache (dCache). We have started our modifications from this unit, in order to build a complete

DIFT system that defends against potential attacks. Integer Unit is a 7-stage pipeline located in the `iu3.vhd` entity, the largest GRLIB VHDL file with approximately 3500 lines of VHDL code. The entity declaration of LEON3 7-stage pipeline breaks the 7 stages into registers, as shown in Table 5.10. These registers are useful, because they help us to simulate the model and watch the values every single moment in each stage. Of course, we added extra signals and registers in each pipeline stage to propel data tag from one stage to another.

Name	**Type**
f	fetch_reg_type
d	decode_reg_type
a	regacc_reg_type
e	execute_reg_type
m	memory_reg_type
x	exception_reg_type
w	write_reg_type

TABLE 5.10: LEON3 processor pipeline separated in 7 registers (stages).

Every pipeline stage has many registers (all the same type for each stage) which store information about every instruction in every stage. The Fetch stage has 2 registers, called Program Counter (PC) and branch (standard logic). The Decode stage has 11 registers, such as PC, instruction (32-bit) and we added an extra register for instruction tag (4-bit). If the Taint bit of the instruction tag is set, the instruction cannot be processed and a security exception is raised. The next stage is the Register Access stage which has 24 registers, such as read address 1 and read address 2 (to read data from Register File). After that, is the Execute Stage which has 25 registers, such as op1 and op2 (32-bit data from Register file) and we added 2 extra registers for tag (4-bit data tag from Register File). In this stage, ALU operations are executed for both data and tag according to the policy rules (described in detail in section 5.2.3). By the end of this stage we know the exact result of the tag operation. In the following two stages, the Memory stage and the Exception stage which have 16 registers and 19 registers, such as result (32-bit), we added an extra register for tag result (4-bit) in each stage. These two registers store data for the tag after the Execute stage and after the Memory stage (if pipeline read data from dCache). The last stage of LEON3 processor is the Writeback stage which has 5 registers, such as result (32-bit) and we added an extra register for tag result (4-bit) which stores the tag that will be written to the Register File in the next cycle. Figure 5.7 illustrates the modified LEON3 processor (7-stage pipeline) with extra signals, registers and logic that we added for tag propagation and check. We extended cache storage to support the DIFT tags and we could expand DIFT tags storage to the whole memory of our design.

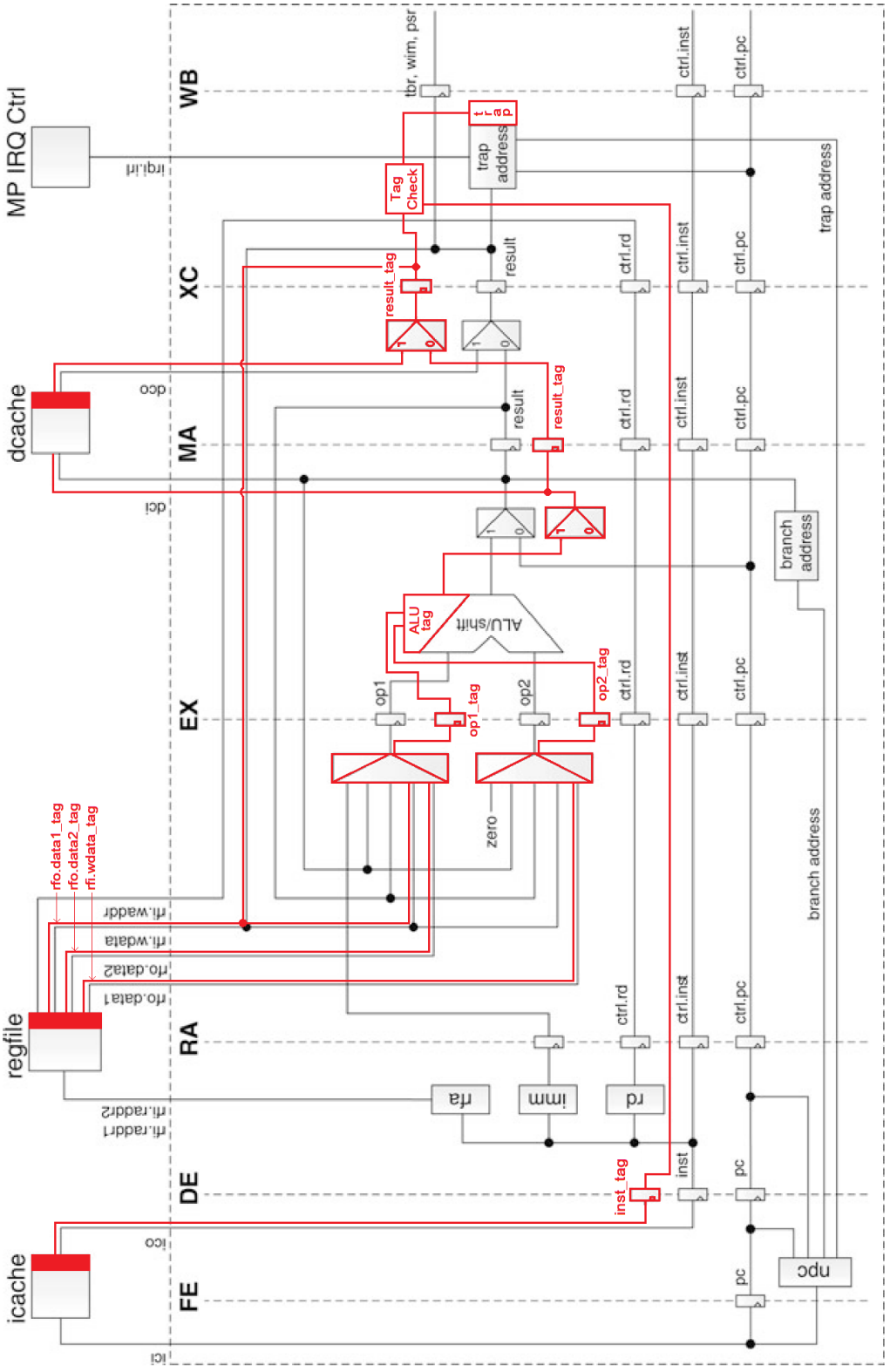

FIGURE 5.7: Modified LEON3 processor (7-stage pipeline) with extra signals, registers and logic that we added for tag propagation and check.

5.4.4 Extra DIFT instructions

Our DIFT architecture was designed to run a modified operating system (e.g., Linux-based), in order to work safely and be able to offer complete protection against unwanted attacks. It seems to be difficult for us to develop or modify an existing Linux-based operating system, to fully support DIFT capabilities. Thus, we do not proceed to that implementation and we introduce a smart modification to our existing architecture. We added extra DIFT instructions to LEON3 processor instructions set, to manipulate data tag, so we can simulate the model using binaries with extra DIFT data. The main LEON3 processor unit is the `proc3.vhd` entity and the 7-stage pipeline is the `iu3.vhd` entity. The decode stage, which decodes every SPARC V8 instruction, contained also in the `iu3.vhd` entity. However, instruction definitions according to the SPARC V8 are located in `sparc.vhd` entity.

		op3 [5:4]			
		0	**1**	**2**	**3**
op3 [3:0]	**0**	ADD	ADDcc	TADDcc	WRASR WRY
	1	AND	TSUBcc	ANDcc	WRPSR
	2	OR	ORcc	TADDccTV	WRWIM
	3	XOR	XORcc	TSUBccTV	WRTBR
	4	SUB	SUBcc	MULScc	FPop1
	5	ANDN	ANDNcc	SLL	FPop2
	6	ORN	ORcc	SRL	CPop1
	7	XNOR	XNORcc	SRA	CPop2
	8	ADDX	ADDXcc	RDASR RDY STBAR	JMPL
	9			RDPSR	RETT
	A	UMUL	UMULcc	RDWIM	Ticc
	B	SMUL	SMULcc	RDTBR	FLUSH
	C	SUBX	SUBXcc	*RD_REGTAG*	SAVE
	D			*WR_REGTAG*	RESTORE
	E	UDIV	UDIVcc	*RD_MEMTAG*	SMAC
	F	SDIV	SDIVcc	*WR_MEMTAG*	UMAC

TABLE 5.11: Map of arithmetic and logic opcodes of LEON3 processor ISA. *Italics* indicate extra instructions added to `sparc.vhd` entity.

In the V8 ISA, there are 6 instruction categories, such as control transfer, load/store, and arithmetic/logic/shift instructions. Every instruction is mapped on a specific 6-bit opcode and Table 5.11 shows the original LEON3 arithmetic and logic opcodes of ISA. Firstly, we classified the new instructions and secondly, for a given subset, each new instruction is mapped on an unimplemented opcode. This step was mandatory to minimize the area and path overhead of the decode stage. Every instruction has a 6-bit

opcode that is located the 6 middle bits of instruction, starting from 24^{th} bit down to 19^{th} bit. Unimplemented opcodes have no instructions mapped on them and they are shown as blank cells in the Table. Table 5.12 lists the new instructions added to the LEON3 SPARC V8 ISA (`sparc.vhd`).

Instruction	**Opcode**	**Example**	**Meaning**
Read Register Tag	101100	rd_regtag %r1, %r2	r2=Tag[r1]
Write Register Tag	101101	wr_regtag %r1, %r2	Tag[r1]=r2
Read Memory Tag	101110	rd_memtag [%r1+imm], [%r2]	[r2]=Tag[Mem[r1]]
Write Memory Tag	101111	wr_memtag [%r1+imm], [%r2]	Tag[Mem[r1]=Mem[r2]

TABLE 5.12: The new instructions added to the LEON3 SPARC V8 ISA (`sparc.vhd`). Tag(r) refers to the Tag (both Taint and Pointer bits) for register (r) or memory location of Mem([r+imm]).

We decided to create a few extra instructions because we wanted to change tag values simply by executing the binary file, and not by modifying the hardware design each time. This method allows us to test and simulate many different binary files with the same hardware design. In order to manipulate tags in both Register File and Cache, we have to add at least 4 extra instructions in the LEON3 processor ISA. We added 2 extra instructions (Read Register Tag and Read Memory Tag) for reading tags from Register File and Cache and 2 extra instructions (Write Register Tag and Write Memory Tag) for writing tags to Register File and Cache, respectively. Table 5.11 and Table 5.12 show the extra instructions added to `sparc.vhd` entity. Finally, we modified the pipeline of LEON3 processor properly, to identify these extra instructions as legal instructions (not illegal), by editing the `exception_detect` procedure.

5.5 Performance Evaluation

In this section we offer a detailed description of the performance evaluation of our modified LEON3 processor with DIFT support. This description includes both simulation and evaluation for our modified LEON3 processor with DIFT support. The simulation process of the original LEON3 processor described in detail in section 5.1.6.

5.5.1 Overview

It is obvious that simulating LEON3 processor is the most fundamental step in order to create the best DIFT system. Ideally, we have to edit a Linux-based Operating System by adding DIFT capabilities to it and simply run unmodified binaries on the LEON3 processor. The OS will be DIFT aware and propagate the right instructions to edit the tag values in hardware. We lack of this software, thus we are not able to run unmodified

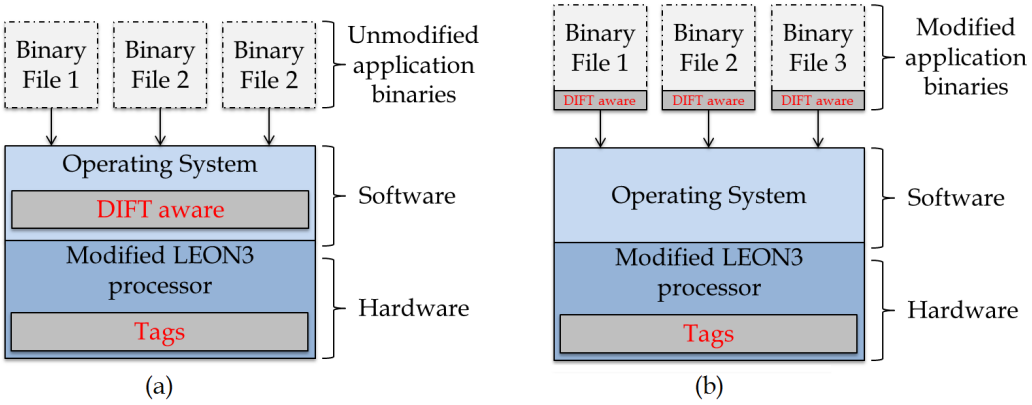

FIGURE 5.8: The system stack for the ideal DIFT platform and for the DIFT platform that we presented. (a) Ideal DIFT platform with DIFT-aware software and hardware with DIFT tags which runs unmodified binaries. (b) Our implementation of DIFT with hardware DIFT tags which runs modified binaries.

binaries on the modified LEON3 processor, but instead we manually edit the source code (in Assembly level) of our testing programs and the build the binary files again. When this procedure ends we are able to run DIFT-aware binaries on our modified processor. We added extra instructions in the instruction set of the modified LEON3 processor (described in detail in the previous section) to manually edit the tags of our variables. Figure 5.8 shows the system stack for the ideal DIFT platform (DIFT-aware software and hardware with DIFT tags) and our implementation of DIFT (only hardware with DIFT tags).

5.5.2 DIFT tags initialization

DIFT tags initialization is the most fundamental step to set values to DIFT tags that represents the taintedness (taint and pointer bits) of every word in the system (Register File and Memory). We did not develop Linux-based software to automatically set tag values for application binaries, but instead we created extra instructions to edit these values in hardware. Concerning the Taint bit, we must taint the environment variables and program arguments when a process is created, and also taint any data read from the filesystem or network. The Pointer bit must only be initialized for root pointer assignments, where a pointer is set to a valid memory address that is not derived from another pointer.

For better performance and safely use of the Pointer bit, we have to distinguish between static root pointer assignments, which initialize a pointer with a valid address in statically allocated memory (such as the address of a global variable), and dynamic root pointer assignments, which initialize a pointer with a valid address in dynamically

Tag bit	**Policies supported**	**Propagation Rules**	**Check Rules**
Taint bit	All (untrusted input)	Move (source only) Integer Arithmetic Logical	-
Pointer bit	PI-based Buffer Overflow	Move (source only) Integer Arithmetic Logical	Move (address) Integer Arithmetic Program Counter

TABLE 5.13: The tag initialization, propagation, and check rules for the security policies used by the modified LEON3 processor. The propagation rules identify the operation classes that propagate tags. The check rules specify the operation classes that raise exceptions on tagged operands.

allocated memory. If we chose to use Linux-based software (Operating System) to automatically initialize DIFT tags, we have to scan any data segments for 32-bit values that are within the virtual address range of the current executable or shared library and set the Pointer bit for any matches. To recognize root pointer assignments in code, we have to scan the code segment for `sethi` instructions. If the immediate operand of the `sethi` instruction specifies a constant within the virtual address range of the current executable or shared library, we have to set the Pointer bit of the instruction.

5.5.3 System Simulation

We decided to simulate the DIFT system using modified binaries which contain DIFT instructions for manipulating hardware tags. As we mentioned before in section 5.1.5, BCC allows cross-compilation of C and C++ applications for LEON3. For convenience purposes we used the command `'make soft` to build the binary files and `'make soft-clean` to clean up them. However this option compiles only C and C++ programs to executable files and we are not able to add extra DIFT instructions into the executable files. To avoid automatically compilation of executable files, we chose to make the compilation process manually. First, we compiled C and C++ programs to Assembly language, secondly we added the extra instructions in Assembly level and at the end we compiled Assembly instructions to executable file. To compile C and C++ programs to Assembly language we used the command:

```
sparc-elf-gcc -msoft-float -g -O2 hello.c S -o hello.S
```

and we compiled the file with Assembly instructions to executable file using the command:

```
sparc-elf-gcc -msoft-float -g -O2 hello.S -o hello.exe
```

By the end of this process we are able to run these binary files on our modified LEON3 processor to test our hardware design for DIFT support. Figure 5.9 shows an

example of a C program that we decided to run on our modified LEON3 processor. First, we wrote the program in C and then we compiled to the equivalent Assembly instructions. We added the extra instructions for DIFT and then we compiled this file to executable file. This example illustrates only the flow of tags from one variable to another by simple tainting (tag propagation rules). No security exception is raised because all pointers are valid and only one of the values is tainted.

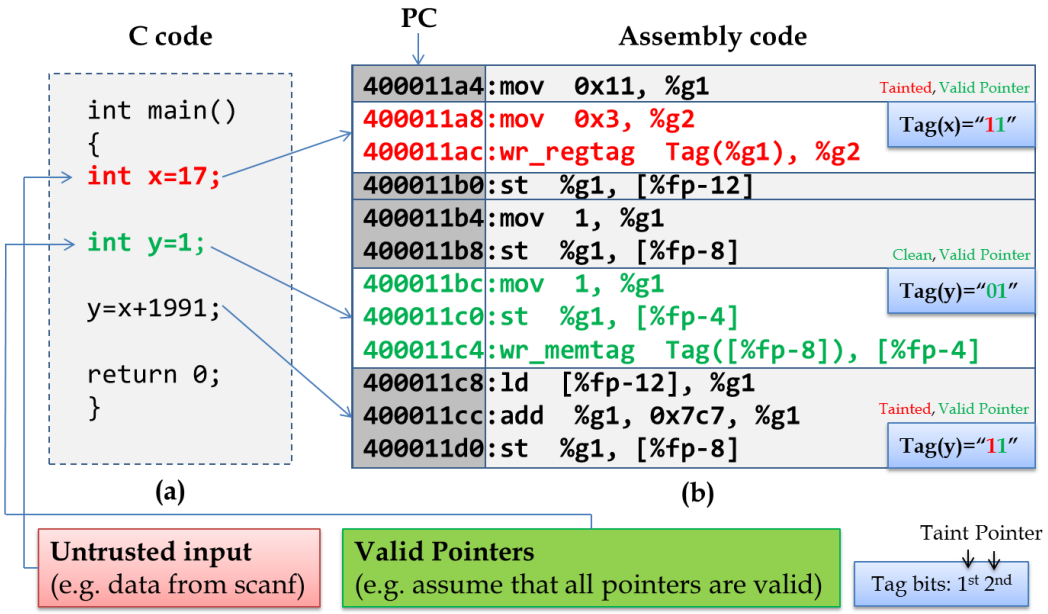

FIGURE 5.9: An example of a C program (a) and its equivalent Assembly instructions (b) with extra instructions for DIFT. The extra instructions for DIFT are written in color (red or green). LEON uses SPARC Instruction Set. We assumed that variable `x` is an untrusted input but all pointers are valid, thus we set the appropriate tag bits for all variables.

However, Figure 5.10 shows an example of a C program that we decided to run on our modified LEON3 processor to test if a security exception would be raised when necessary. We compiled the C program to executable file, exactly by the same procedure, but in this example we illustrate how a security exception is raised when an illegal instruction is executed on the modified LEON3 processor. This illegal instruction would be generated by the software (OS) that our processor runs, but in our implementation scheme we cannot provide this type of software to generate Assembly instructions automatically. The `load` instruction with Program Counter `400011c8` will not be executed by our processor and a security exception will be raised because the check rules for the PI-based policy is active (tainted and not valid pointer).

Another typical example for Pointer bit use is the `and` instruction which is often used to align pointers. Thus, the propagation rule for the and instruction sets the Pointer bit of the destination register if one source operand is a pointer, and the other is a

non-pointer. Moreover, we have to propagate the Pointer bit for instructions that may initialize a pointer to a valid address in statically allocated memory using the `sethi` instruction. The propagation rules for both Taint and Pointer bit described in detail in section 5.2.3. Previous example illustrates only the flow of tags and information, showing the propagation and check rules that will be applied in order to restrict the flow of instructions properly.

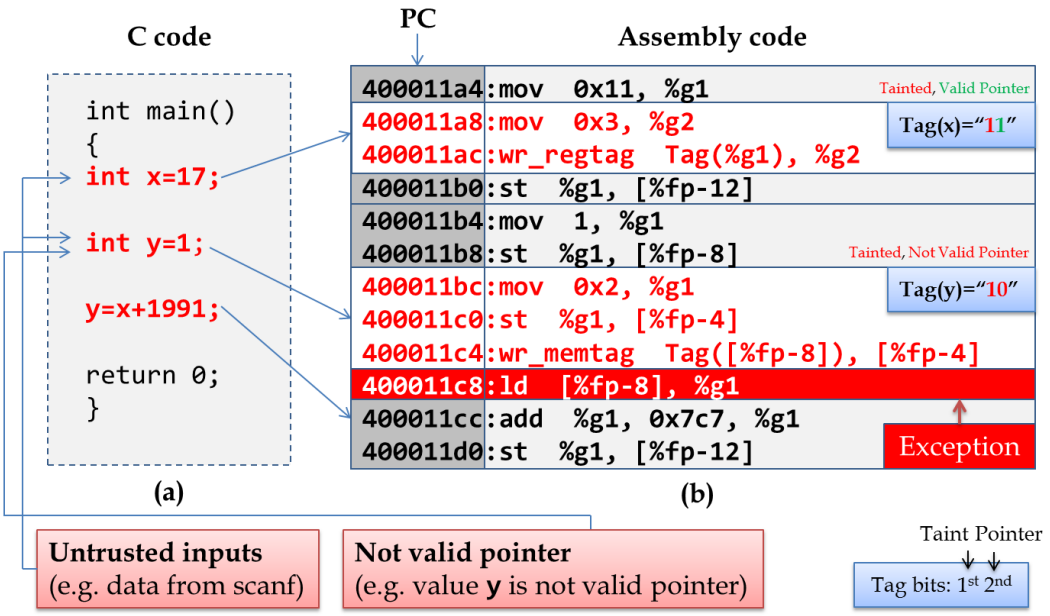

FIGURE 5.10: An example of a C program (a) and its equivalent Assembly instructions (b) with extra instructions for DIFT. The extra instructions for DIFT are written in color (red). LEON uses SPARC Instruction Set. We assumed that variables `x, y` are untrusted inputs, pointer `x` is valid and pointer `y` is not valid, thus we set the appropriate tag bits for all variables.

In order to simulate the model with modified binary files, we created new .wcfg files containing the signals that we wanted to observe each time, mainly from Integer Unit (IU) and Memory. These signals are array of signals and they are based on the pipeline stages of the LEON3 processor. The results of our simulation process are presented in the Appendix A.

5.5.4 System Evaluation

In this section we will offer a description of our system evaluation. We will show original LEON3 processor gate count and gate area, and the additional storage needed by our system to support DIFT technique successfully. This approach introduces an approximately 12% overhead in the memory system for tag storage. If we are going to download our hardware design onto an FPGA board with support for ECC DRAM, we have to use

the 4 bits per 32-bit word available to the ECC code to store the DIFT tags. The main core (including cache) gate count is nearly 93% of the LEON3 total gate count while our modifications were mainly in the processor core and not in the other components. Thus, our extra logic simply adds overhead in the LEON3 main processor and not in the other components.

	Gates	**Gate Area(μm^3)**
Modified LEON3 (total)	*353192*	*3310976*
Original LEON3 (total)	336374	3153311
Core (with cache)	311202	2917335
AHB Controller	7859	73676
AHB/APB Bridge	4624	43347
Memory Controller	6740	63183
UART	2236	20967
IRQ Controller	1174	11009
General I/O	1306	12246

TABLE 5.14: LEON3 gate count and gate area including Cache blocks.

Table 5.12 shows the gate count and the gate area in total for the original LEON3 and for the modified LEON3, while indicates also the gates of the main core processor with cache and other components. As we can see the extra overhead over the original LEON3 processor is approximately 5% while our clock frequency did not changed at all. Finally, Figure 5.11 shows the layout of LEON3 processor at 175MHz in UMC 0.18μm.

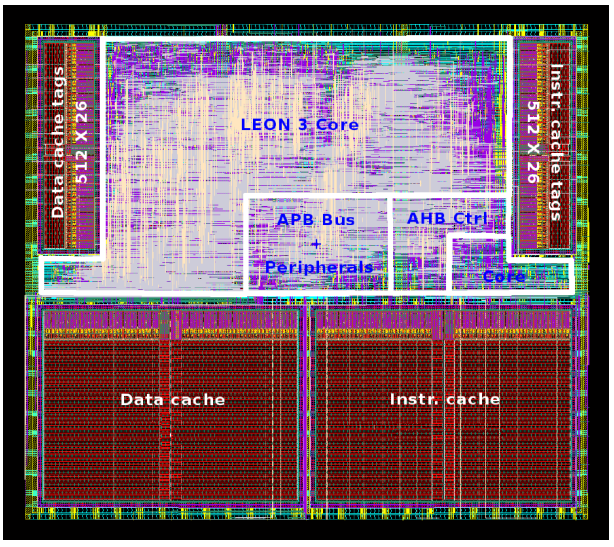

FIGURE 5.11: LEON3 processor at 175 MHz in UMC 0.18μm. 132 I/O pins. The layout is 2215.02μm by 1943.68μm (4.435mm^2).

Chapter 6

Conclusion & Future Work

In this chapter we present our conclusion and describe future work in Dynamic Information Flow Tracking technique. We also discuss about our implementation of DIFT.

6.1 Conclusion

Existing defenses on software security suffer from compatibility issues and are easily evaded by professionals, so security defenses should focus on safety, speed, flexibility, practicality and end-to-end coverage. We described a technique, Dynamic Information Flow Tracking (DIFT), and we concluded that this technique can be powerful and flexible for comprehensive prevention of many critical software security threats. This diploma thesis demonstrated that DIFT technique can be used to prevent a wide range of security attacks, according to the policies that we chose to implement.

Furthermore, we explored the synthesizable LEON3 processor and described how to design and implement a hardware platform for DIFT. This platform which is an extension of the LEON3 processor with additional instructions for data-flow integrity support, can track tag information along data within the processor pipeline and through computations, if we install appropriate Linux-based operating system. We proposed an efficient and cheap DIFT method based on the LEON3 processor. Our modifications in the original LEON3 processor were mainly in the processor core (7-stage pipeline) and not in the entire LEON3. We emphasized in the information flow (tag propagation and check) and not to build a complete system that is able to run unmodified binaries containing vulnerable codes.

Finally, we tested our DIFT platform and we presented the experimental evaluation of the hardware platform with DIFT support. We evaluated DIFT policies and successfully prevented low-level vulnerabilities, such as buffer overflow attacks, using tainted binaries written in C or Assembly language. The experimental results showed that our hardware design for DIFT can be successfully downloaded on an FPGA board. We could look forward to implement a complete DIFT platform, if we extend the whole LEON3 processor (containing RAM and other resources) to support DIFT tags.

6.2 Future Work

While there has been significant interest in DIFT in academia, several challenges remain to the widespread adoption of DIFT in the real world. It is essential to emphasize in studies to determine what security policies scale to enterprise environments, and what the necessary configurations are. Additionally, there has been very little research in exposing APIs to allow for system administrators to easily express their security policies in terms of DIFT mechanisms. Some web based vulnerabilities will benefit also from DIFT support in the language. Very little is known about the implications of adding DIFT support to an existing language.

The DIFT policies which we presented can be also used to protect the Linux kernel, providing the comprehensive kernelspace buffer overflow protection. Furthermore, they can be used in combination with a couple of extra policy bits in order to offer complete protection for both low-level and high-level vulnerabilities. Ours prospects for the future is to complete our hardware design by adding extra active policies and download the design onto an FPGA to evaluate the system in real time, using a Linux-based OS.

Appendix A

Appendix

Systematic control of complex circuits is necessary and "challenging". Testing is fundamental step to check the proper functionality of any system or hardware design. We decided to test our modified LEON processor by executing modified programs written in Assembly language. Integer Unit is a 7-stage pipeline located in the `iu3.vhd`. The entity declaration of LEON3 7-stage pipeline breaks the 7 stages into registers, as shown in Figure A.1. These registers are very useful, because they keep the processor core values every single moment in each different stage. Furthermore, we added extra signals and registers in each pipeline stage to propel data tag from one stage to another.

```
------------------------------------------------------------------------
  subtype word is std_logic_vector(31 downto 0);
  subtype pctype is std_logic_vector(31 downto PCLOW);
  subtype rfatype is std_logic_vector(RFBITS-1 downto 0);
  subtype cwptype is std_logic_vector(NWINLOG2-1 downto 0);
  type icdtype is array (0 to isets-1) of word;
  type dcdtype is array (0 to dsets-1) of word;
------------------------------------------------------------------------
  type registers is record
    f  : fetch_reg_type;
    d  : decode_reg_type;
    a  : regacc_reg_type;
    e  : execute_reg_type;
    m  : memory_reg_type;
    x  : exception_reg_type;
    w  : write_reg_type;
  end record;
```

```
------------------------------------------------------------------------
  type dc_in_type is record
    signed, enaddr, read, write, lock, dsuen : std_ulogic;
    size : std_logic_vector(1 downto 0);
    asi  : std_logic_vector(7 downto 0);
  end record;
------------------------------------------------------------------------
  type pipeline_ctrl_type is record
    pc    : pctype;
    inst  : word;
    cnt   : std_logic_vector(1 downto 0);
    rd    : rfatype;
    tt    : std_logic_vector(5 downto 0);
    trap  : std_ulogic;
    annul : std_ulogic;
    wreg  : std_ulogic;
    wicc  : std_ulogic;
    wy    : std_ulogic;
    ld    : std_ulogic;
    pv    : std_ulogic;
    rett  : std_ulogic;
  end record;
------------------------------------------------------------------------
  type fetch_reg_type is record
    pc     : pctype;
    branch : std_ulogic;
  end record;
------------------------------------------------------------------------
  type decode_reg_type is record
    pc    : pctype;
    inst  : icdtype;
    cwp   : cwptype;
    set   : std_logic_vector(ISETMSB downto 0);
    mexc  : std_ulogic;
    cnt   : std_logic_vector(1 downto 0);
    pv    : std_ulogic;
    annul : std_ulogic;
    inull : std_ulogic;
    step  : std_ulogic;
```

```
    divrdy: std_ulogic;
  end record;
-----------------------------------------------------------------------
  type regacc_reg_type is record
    ctrl  : pipeline_ctrl_type;
    rs1   : std_logic_vector(4 downto 0);
    rfa1, rfa2 : rfatype;
    rsel1, rsel2 : std_logic_vector(2 downto 0);
    rfe1, rfe2 : std_ulogic;
    cwp   : cwptype;
    imm   : word;
    ldcheck1 : std_ulogic;
    ldcheck2 : std_ulogic;
    ldchkra : std_ulogic;
    ldchkex : std_ulogic;
    su : std_ulogic;
    et : std_ulogic;
    wovf : std_ulogic;
    wunf : std_ulogic;
    ticc : std_ulogic;
    jmpl : std_ulogic;
    step  : std_ulogic;
    mulstart : std_ulogic;
    divstart : std_ulogic;
    bp, nobp : std_ulogic;
  end record;
-----------------------------------------------------------------------
  type execute_reg_type is record
    ctrl   : pipeline_ctrl_type;
    op1    : word;
    op2    : word;
    op1_tag    : std_logic_vector(3 downto 0);  -- op1 for DIFT
    op2_tag    : std_logic_vector(3 downto 0);  -- op2 for DIFT
    aluop  : std_logic_vector(2 downto 0);      -- Alu operation
    alusel : std_logic_vector(1 downto 0);      -- Alu result select
    aluadd : std_ulogic;
    alucin : std_ulogic;
    ldbp1, ldbp2 : std_ulogic;
    invop2 : std_ulogic;
```

```
    shcnt  : std_logic_vector(4 downto 0);      -- shift count
    sari   : std_ulogic;                        -- shift msb
    shleft : std_ulogic;                        -- shift left/right
    ymsb   : std_ulogic;                        -- shift left/right
    rd     : std_logic_vector(4 downto 0);
    jmpl   : std_ulogic;
    su     : std_ulogic;
    et     : std_ulogic;
    cwp    : cwptype;
    icc    : std_logic_vector(3 downto 0);
    mulstep: std_ulogic;
    mul    : std_ulogic;
    mac    : std_ulogic;
    bp     : std_ulogic;
    rfe1, rfe2 : std_ulogic;
  end record;
------------------------------------------------------------------------
  type memory_reg_type is record
    ctrl   : pipeline_ctrl_type;
    result : word;
    result_tag : std_logic_vector(3 downto 0);  -- result for DIFT
    y      : word;
    icc    : std_logic_vector(3 downto 0);
    nalign : std_ulogic;
    dci    : dc_in_type;
    werr   : std_ulogic;
    wcwp   : std_ulogic;
    irqen  : std_ulogic;
    irqen2 : std_ulogic;
    mac    : std_ulogic;
    divz   : std_ulogic;
    su     : std_ulogic;
    mul    : std_ulogic;
    casa   : std_ulogic;
    casaz  : std_ulogic;
  end record;
------------------------------------------------------------------------
  type exception_reg_type is record
    ctrl   : pipeline_ctrl_type;
```

```
    result : word;
    result_tag : std_logic_vector(3 downto 0);  -- result for DIFT
    y      : word;
    icc    : std_logic_vector( 3 downto 0);
    annul_all : std_ulogic;
    data   : dcdtype;
    data_tag   : dcdtype; -- data_tag for dift
    set    : std_logic_vector(DSETMSB downto 0);
    mexc   : std_ulogic;
    dci    : dc_in_type;
    laddr  : std_logic_vector(1 downto 0);
    rstate : exception_state;
    npc    : std_logic_vector(2 downto 0);
    intack : std_ulogic;
    ipend  : std_ulogic;
    mac    : std_ulogic;
    debug  : std_ulogic;
    nerror : std_ulogic;
    ipmask : std_ulogic;
  end record;
------------------------------------------------------------------------
  type write_reg_type is record
    s      : special_register_type;
    result : word;
    result_tag : std_logic_vector(3 downto 0); -- result for DIFT
    wa     : rfatype;
    wreg   : std_ulogic;
    except : std_ulogic;
  end record;
```

FIGURE A.1: LEON3 processor pipeline registers and records (`iu3.vhd`). Signals for DIFT tags indicated with comment next to the signal.

Testbench file (`testbench.vhd`) uses arrays of signals to help us observe the instruction flow in the LEON3 processor. The basic array of signals is `rin`, which contains all the LEON3 processor core registers. For the simulation process we used the iSIM simulator of the Xilinx ISE Design Suite using the Cygwin terminal. Figures A.2 to A.5 show the basic signals of LEON3 processor and our simulation results for the example C program illustrated in Figure 5.9.

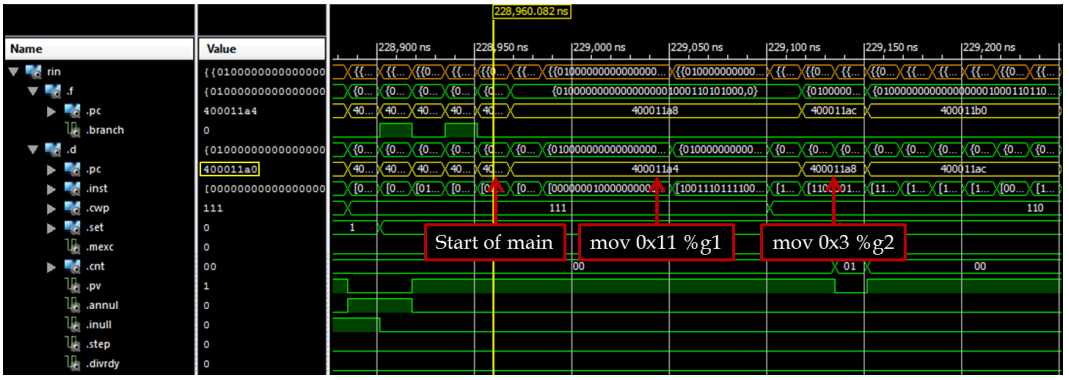

FIGURE A.2: Simulation screenshot in the Decode Stage showing the instructions arrived in the processor core.

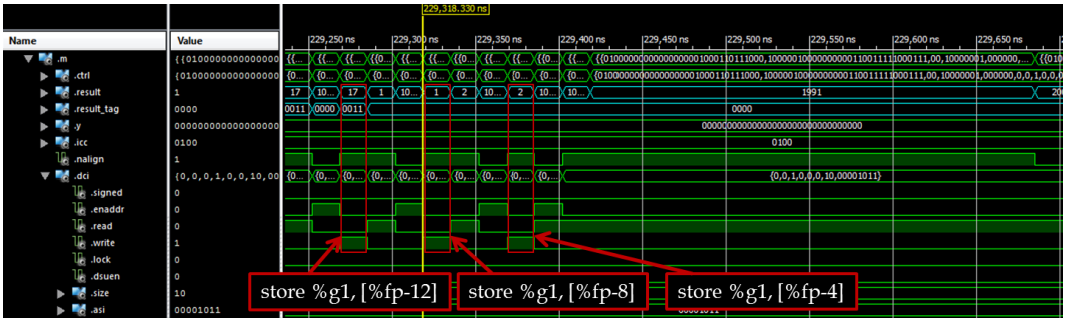

FIGURE A.3: Simulation screenshot in the Memory Stage showing the instructions storing to the Data Cache.

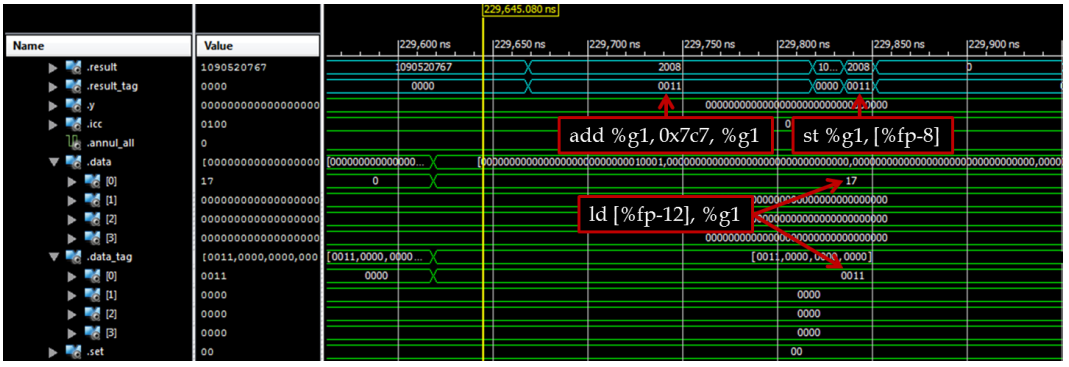

FIGURE A.4: Simulation screenshot in the Exception Stage showing the arithmetic instruction waiting for the reading of data with tag.

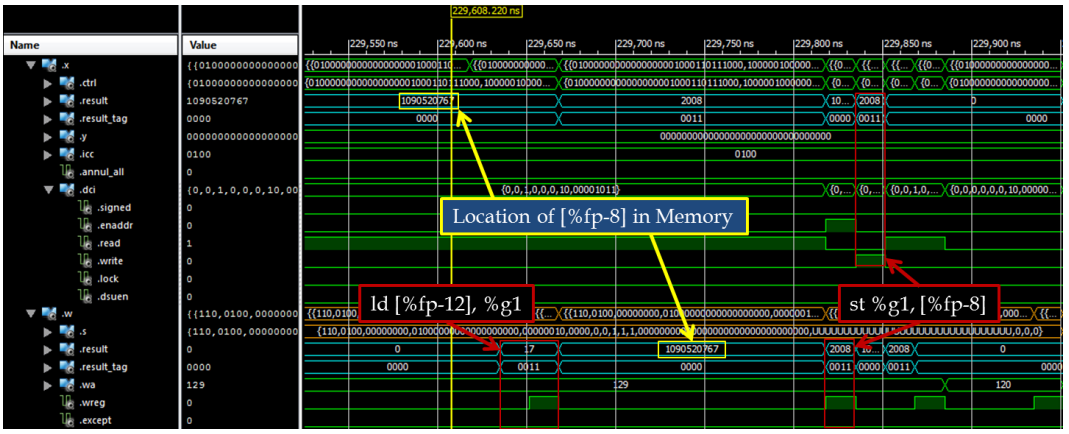

FIGURE A.5: Simulation screenshot in the Exception and Writeback Stages showing the simple tainting from the one valid pointer to another.

Now we are going to show the simulation results when a security exception should be raised. Figure A.6 shows our simulation results for the example C program illustrated in Figure 5.10. As we can see, a security exception raised when tainted data and invalid pointer combined together. The tag of the memory location `[%fp-8]` has Taint bit = '1' and Pointer bit = '0', thus the `load` instruction using this location is untrusted and must not executed.

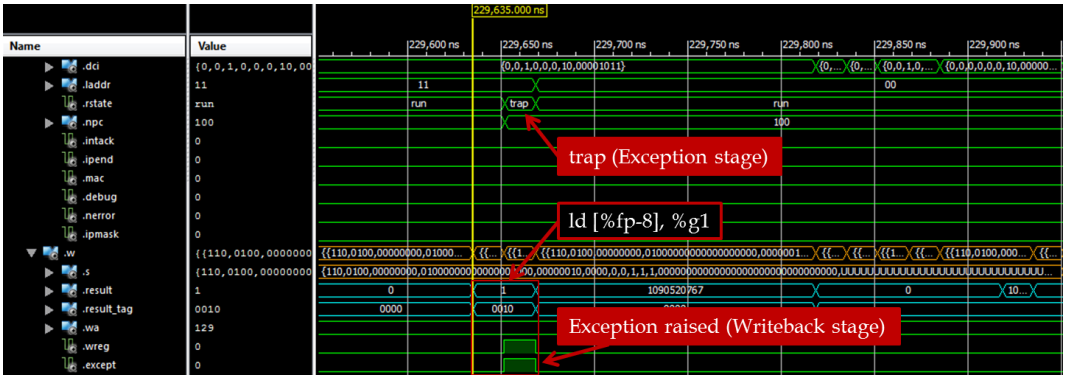

FIGURE A.6: Simulation screenshot in the Writeback Stage showing the security Exception raised when tainted data and invalid pointer combined together.

Bibliography

[1] Levy, Elias. Smashing The Stack for Fun and Profit. `http://www.phrack.org/issues/49/14.html#article`, 1996.

[2] Bertrand, Louis. OpenBSD: Fix the Bugs, Secure the System. In *MUSESS '02: McMaster University Software Engineering Symposium*, 2002.

[3] Crispin Cowan, Calton Pu, et al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[4] Hiroaki Etoh. GCC Extension for Protecting Applications from Stack-smashing Attacks. `http://www.research.ibm.com/trl/projects/security/ssp/main.html`, 2000.

[5] Ollie Whitehouse. GS and ASLR in Windows Vista, 2007.

[6] Shuo Chen, Jun Xu, et al. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[7] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, 2002.

[8] Olatunji Ruwase and Monica Lam. A practical dynamic buffer overflow detector. In *Proceedings of the Network and Distributed System Security Symposium*, 2004.

[9] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. Formatguard: automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium, page 15, Berkeley, CA, USA, USENIX Association*, 2001.

[10] Nikolaj Bjrner, Vijay Ganesh, Margus Veanes. An SMT-LIB Format for Sequences and Regular Expressions. In SMT workshop, 2012

[11] Bypassing PaX ASLR protection. `http://www.phrack.org/issues.html?issue=59&id=9`, 2002.

[12] R. Shirey. Internet Security Glossary, Version 2. `https://tools.ietf.org/html/rfc4949`, 2007.

[13] Microsoft. SQL Injection. `https://www.technet.microsoft.com/en-us/library/ms161953(v=SQL.105).aspx`, 2013.

[14] Grossman, Jeremiah, Hansen, Robert, Fogie, Seth, Petkov, Petko D., Rager, Anton. XSS Attacks: Cross Site Scripting Exploits and Defense, 2008.

[15] Paco Hope, Walther Ben. Web Security Testing Cookbook. Sebastopol, CA: O'Reilly Media, Inc., 2008

[16] Cross-site Scripting. Web Application Security Consortium. `http://projects.webappsec.org/Cross-Site-Scripting`, 2005.

[17] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *Computer Security Applications Conference, Annual*, 2005.

[18] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar Iyer. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *Proceedings of the Intl. Conference on Dependable Systems and Networks*, 2005.

[19] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications using Precise Tainting. In *Proceedings of the 20th IFIP Intl. Information Security Conference, Chiba, Japan*, 2005.

[20] Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[21] Nicholas Nethercote. Dynamic Binary Analysis and Instrumentation. PhD thesis, University of Cambridge, 2004.

[22] Fabrice Bellard. QEMU. A fast and portable dynamic translator. In *Proceedings of the 2005 USENIX, Freenix track, Anaheim, CA*, 2005.

[23] Lap Chung Lam and Tzi cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Annual Computer Security Applications Conference, volume 0, pages 463472, Los Alamitos, CA, USA*, 2006.

[24] Stuart Ellis. The Key Ideas of the Ruby Programming Language. `http://www. stuartellis.eu/articles/ruby-language/`, 2009.

[25] Feng Qin, Cheng Wang, Zhenmin Li, Ho-Seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the 39th International Symposium on Microarchitecture (MICRO), Orlando, FL*, 2006.

[26] James Newsome, Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th NDSS, San Diego, CA*, 2005.

[27] G. Edward Suh, Jaewook Lee, Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Boston, MA*, 2004.

[28] Michael Dalton, Hari Kannan, Christos Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA), San Diego, CA*, 2007.

[29] Hari Kannan. Ordering Decoupled Metadata Accesses in Multiprocessors. In *Proceedings of the 42nd International Conference on Microarchitecture (MICRO), New York City, NY*, 2009.

[30] InfoSec Reading Room, SANS Institute. Worms as Attack Vectors: Theory, Threats, and Defenses, 2003.

[31] J. R. Crandall, F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, MICRO 37, (Washington, DC, USA), pp. 221232, IEEE Computer Society, IEEE Computer Society*, 2004.

[32] G. Venkataramani, I. Doudalis, Y. Solihin, M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on, pp. 173184, IEEE*, 2008.

[33] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, MICRO 37, (Washington, DC, USA), pp. 243254, IEEE Computer Society*, 2004.

[34] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, T. Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the 14th*

international conference on Architectural support for programming languages and operating systems, ASPLOS 09, (New York, NY, USA), pp. 109120, ACM, 2009.

[35] H. Chen, X. Wu, L. Yuan, B. Zang, P. Yew, F. Chong. From speculation to security: Practical and efficient information flow tracking using speculative hardware. In *Proceedings of the ACM SIGARCH Computer Architecture News, vol. 36, no. 3, pp. 401412*, 2008.

[36] M. Ozsoy, D. Ponomarev, N. Abu-Ghazaleh, T. Suri. SIFT: A Low-Overhead Dynamic Information Flow Tracking Architecture for SMT Processors. In *Proceedings of the 8th ACM International Conference on Computing Frontiers, CF 11, (New York, NY, USA), pp. 37:137:11, ACM*, 2011.

[37] J. Andersson, J. Gaisler, R. Weigand. Next Generation Multipurpose Microprocessor. In *Proceedings of Data Systems in Aerospace 2010 (DASIA2010)*, 2010.

[38] Gaisler Research. Jiri Gaisler. LEON3 GR-XC3S-1500 Template Design, 2006.

[39] Cygwin Shell for GRLIB usage on Windows, *http://www.cygwin.com/*.

[40] Michael Dalton, Hari Kannan, Christos Kozyrakis. Real-world buffer overflow protection for userspace and kernelspace. In *Proceedings of the 17th Annual USENIX Security Symposium, pages 395410*, 2008.

[41] Michael Dalton, Hari Kannan, Christos Kozyrakis, Tainting is Not Pointless. In *ACM SIGOPS Operating Systems Review, Volume 44 Issue 2, Pages 88-92*, 2010.

[42] J. C. Martinez Santos and Y. Fei, Leveraging speculative architectures for run-time program validation, ACM Trans. on Embedded Computing Systems, vol. 13, no.1, Aug. 2013.

[43] Symantec Internet Security Threat Report, Volume X: Trends for January 06 - June 06, September 2006.

[44] Center for Strategic and International Studies, Net Losses: Estimating the Global Cost of Cybercrime - McAfee, June 2014.