THE DESIGN AND IMPLEMENTATION OF DYNAMIC INFORMATION FLOW TRACKING SYSTEMS FOR SOFTWARE SECURITY

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Michael Dalton
November 2009

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christoforos Kozyrakis, , Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Monica Lam,

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

David Mazieres,

Approved for the Stanford University Committee on Graduate Studies.

Patricia Gumport, Vice Provost Graduate Education

Abstract

Computer security is in a crisis. Attackers are exploiting an ever-increasing range of software vulnerabilities in critical public and private sector computer systems for massive financial gain [128]. Our existing defenses, such as stack canaries or web application firewalls, often suffer from compatibility issues or are easily evaded by a skilled attacker. Ideally, security defenses should be safe, practical (compatible with current systems), flexible, and fast.

Recent research has established Dynamic Information Flow Tracking (DIFT) [28, 85, 76] as a promising platform for detecting a wide range of security attacks. The idea behind DIFT is to tag (taint) untrusted data and track its propagation at byte or word-granularity through the system to prevent security attacks. However, current software DIFT solutions are neither practical nor fast, while current hardware solutions are neither flexible nor safe. Furthermore, many DIFT policies such as bounds check recognition buffer overflow policies, have unacceptable false positives and negatives in real-world applications.

This dissertation addresses these gaps in existing research by presenting novel DIFT platforms and policies. We show that well-designed DIFT platforms and policies can comprehensively prevent the major server-side security vulnerabilities with little to no performance overhead and without requiring application source code access or debugging information. We describe novel DIFT policies for comprehensively preventing software vulnerabilities. We also present novel hardware and software DIFT platforms for executing these policies. We then demonstrate the effectiveness of our policies and platforms by preventing a wide range of real-world software vulnerabilities, from operating system buffer overflows in the Linux kernel to authentication bypass in PHP web applications. Unlike prior security techniques, DIFT can be fast, safe, practical, and flexible.

We present Raksha, the first flexible hardware DIFT platform, which provides flexibility and safety while maintaining the practicality and performance benefits of traditional hardware DIFT designs. Raksha supports flexible, hardware-enforced DIFT policies using software-controlled tag policy registers. This design allows the best of both worlds, supporting flexible, safe DIFT policies much like a pure software DIFT implementation, while providing the performance and practical legacy code compatibility of a traditional hardware DIFT design. Raksha is also the first DIFT platform to prevent high-level vulnerabilities on unmodified binaries. We demonstrate the Raksha design using an FPGA-based prototype system, and prevent a wide range of attacks on unmodified application binaries.

We use Raksha to develop a novel DIFT policy for robustly preventing buffer overflows on real-world code. Prior DIFT policies for buffer overflow prevention are unsafe and impractical, due to unacceptable false positives and negatives in real-world applications such as GCC and gzip. Furthermore, no buffer overflow protection policy has been successfully applied to the most trusted and privileged software layer – the operating system. Our policy is the first comprehensive DIFT policy for buffer overflow prevention to support large, real-world applications and even the operating system without observed real-world false positives. We demonstrate our buffer overflow policy using the Raksha prototype, and prevent buffer overflows in both userspace applications and the Linux kernel.

We also developed Nemesis, a DIFT-aware PHP interpreter, which was the first system for comprehensively preventing authentication and authorization bypass attacks in web applications. Nemesis uses a novel application of DIFT to automatically infer when a web application has correctly and safely authenticated a web client. We demonstrate the effectiveness of Nemesis by preventing authentication and authorization bypass vulnerabilities in real-world PHP web applications.

Acknowledgements

I would like to thank my family and friends in High Point, Atlanta, Stanford and elsewhere for their support. I would also like to thank my talented colleagues at Stanford Computer Science and Electrical Engineering for many insightful discussions. A special thanks to my friend and colleague Hari Kannan, who has worked with me since my first day of graduate school and is the co-author of all of my Raksha-related work.

During my years of research at Stanford, I have also had the good fortune to interact with excellent partners in industry. I am grateful to Jiri Gaisler, Richard Pender, and everyone at Gaisler Research for their numerous hours of support and help working with the LEON3 processor.

I also also like to express my heartfelt thanks for those who mentored me and encouraged my research career. My parents supported my interest in computer science from an early age, going so far as to enroll me in introductory programming classes while I was in middle school. They also played an important role in my decision to transfer to Stanford as an undergraduate. I cannot thank them enough for their unwavering support and dedication.

My professors at Emory played an invaluable role in my early collegiate career. In particular, I am grateful to Jeanette Allen, Phil Hutto, Vaidy Sunderam, and Ken Mandelberg for encouraging my interest in computer science research. As a Stanford undergraduate, I was fortunate to work under Monica Lam doing research as part of the CURIS program. I thank Monica for mentoring me as a researcher, and for encouraging me to pursue my doctoral studies at Stanford after completing my undergraduate degree.

Christos Kozyrakis has played a crucial role in my research career and has been an excellent advisor. I would like to thank Christos for his dedication to his students and countless hours of mentoring. I especially appreciate his support and trust when I decided to pursue my doctoral research in Dynamic Information Flow Tracking for software security, a field that initially was outside of his core research focus. I would also like to thank Christos, Monica, David Mazières, Dawson Engler, and Martin Fischer for taking the time to be on my committee, and for many fascinating research discussions. I am grateful to Dan Boneh and John Mitchell for numerous exciting and interesting computer security discussions.

Finally, I would like to thank the Aqua Teen Hunger Force.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

It is widely recognized that computer security is a critical problem with far-reaching financial and social implications [95]. Despite significant development efforts, existing security tools do not provide reliable protection against an ever-increasing set of attacks, worms, and viruses that target vulnerabilities in deployed software. Apart from memory corruption bugs such as buffer overflows, attackers are now focusing on high-level exploits such as SQL injection, command injection, authentication bypass, cross-site scripting and directory traversal [45, 123]. Worms that target multiple vulnerabilities in an orchestrated manner are also increasingly common [10, 123]. Cybercrime costs the U.S. economy tens of billions of dollars annually [129], and is growing at a staggering pace [128]. Hence, research on system security and attack prevention is very timely.

The root of the computer security dilemma is that existing protection mechanisms do not exhibit many of the desired characteristics for security techniques: *safety*: they should provide defense against vulnerabilities with no false positives or false negatives; *flexibility*: they should adapt to cover evolving threats; *practicality*: they should work with real-world code and software models (legacy binaries, dynamic code generation, or even operating system code) without specific assumptions about compilers or libraries; and finally *speed*: they should have small impact on application performance.

Recent research has established *Dynamic Information Flow Tracking (DIFT)* [28, 85] as a promising platform for detecting a wide range of security attacks. The idea behind DIFT is to tag (taint) untrusted data and track its propagation through the system. DIFT associates a tag with every word of memory in the system. Any new data derived from untrusted data is also tagged. If tainted data is used in a potentially unsafe manner, such as executing a tagged SQL command or dereferencing a tagged pointer, a security exception is raised.

Current DIFT solutions fail to meet many of our ideal security policy requirements. State of the art DIFT policies exist to comprehensively prevent vulnerabilities such as SQL injection, but several important vulnerability types have no acceptable DIFT solution. Existing DIFT buffer overflow policies are neither *safe* nor *practical*, and contain serious false positives and negatives in many real-world applications. There are also no DIFT policies that prevent web authentication and authorization vulnerabilities, a critical and prevalent flaw in modern web applications.

Current DIFT platforms also fail to meet our ideal design criteria. Software DIFT platforms are neither *fast* nor *practical* when protecting legacy binaries as they result in high performance overheads and restrict applications to a single core. Hardware DIFT platforms are fast and practical, but support only a single, fixed policy (not *flexible* or *safe*). An ideal DIFT platform should satisfy all of these criteria, combining the best of both software and hardware approaches.

This dissertation addresses these gaps in existing research by presenting novel DIFT platforms and policies. We show that robust software and hardware DIFT platforms comprehensively prevent major server-side security vulnerabilities, from buffer overflows to cross-site scripting, with little to no performance overhead and without requiring application source code access or debugging information. Unlike prior defenses in computer security, DIFT can be *fast*, *safe*, *practical*, and *flexible*.

1.1 Contributions

This dissertation explores the potential of DIFT by developing practical DIFT platforms, and then using these platforms to design robust, comprehensive DIFT policies to prevent security attacks on real-world applications. We focus on input validation vulnerabilities such as cross-site scripting, buffer overflows, authentication bypass, and SQL injection. Input validation attacks are software attacks that occur because a non-malicious, but vulnerable application did not correctly validate untrusted user input. Other aspects of computer security such as malware analysis, cryptography, DRM, information leaks, and other security topics are outside the scope of this work.

We developed the first *flexible* hardware DIFT platform, Raksha, for preventing security vulnerabilities in unmodified applications and the operating system. Prior hardware DIFT platforms are *fast* and *practical*, but support only a single, fixed memory corruption policy. Fixed policies in hardware are *inflexible* and *unsafe*, as they cannot adapt to attacker innovations or unexpected application behavior. For example, preventing a SQL injection attack using DIFT requires parsing a SQL query using a database-specific grammar, which is not an operation that should be performed in hardware.

In contrast, software DIFT platforms are *flexible* and *safe*, but are *slow* and *impractical*. Software DIFT results in significant performance overhead, and does not fully support multithreading or self-modifying code. Raksha combines the best of both worlds by providing a flexible hardware DIFT platform with software-managed policies. This design meets all of our criteria: flexible, fast, safe, and practical.

When designing Raksha, we extended the real-world SPARC V8 ISA to transparently support tag propagation and checks during instruction execution. We also added instructions to manipulate tag state, and provided tag policy registers to support software-controlled, hardware-enforced DIFT policy specification. User-level exceptions are also supported to allow for low-overhead security exceptions, and allow Raksha to apply DIFT policies to the operating system. Raksha's design allows complex operations, such as parsing a SQL query, to be deferred to software security handlers while common tag propagation and check operations are performed directly in hardware under the control of the software-managed tag policy registers.

We implemented an FPGA-based prototype of Raksha using an open source SPARC V8 CPU [58]. Using our prototype, we evaluated DIFT policies and successfully prevented low-level buffer overflow and format string vulnerabilities, as well as high-level web vulnerabilities such as command injection, directory traversal, cross-site scripting, and SQL injection. All experiments were performed on unmodified application binaries, with no debugging information. Observed performance overhead on the SPEC CPU2000 benchmarks was minimal.

Using Raksha, we developed the first *safe* and *practical* buffer overflow policy, comprehensively preventing buffer overflows in large real-world applications and even the operating system kernel without any observed false positives. Our policy protects both code and data pointers, even for operating system code. This policy is the first DIFT buffer overflow policy to have no observed false positives on large real-world applications, as well as the first to protect an operating system kernel. We evaluated our policy using real-world applications such as gcc, perl, and apache, as well as the Linux kernel itself. Our experimental results confirm that we detect both code and data pointer overwrites, including off-by-one overwrites, in both user applications and the Linux kernel.

We also developed Nemesis, a DIFT-aware PHP interpreter, which was the *first* system for comprehensively preventing authentication and authorization bypass attacks. Nemesis uses a novel application of DIFT to *infer* when a web application has correctly and safely authenticated a web client. This approach does not require any knowledge of the application's authentication framework, other than the name of the resource (e.g., database table and column names) that stores user authentication credentials.

Using Nemesis, we prevented both authentication and authorization bypass attacks on legacy PHP applications without requiring the existing authentication and access control frameworks to be rewritten. Furthermore, no discernible performance overhead was observed when executing common web server benchmarks, and even CPU-intensive microbenchmarks demonstrated minimal overhead.

1.2 Organization

Chapter 2 presents modern security vulnerabilities and existing defensive countermeasures. This chapter describes each major input validation vulnerability and any existing non-DIFT defenses. Chapter 3 provides an overview of DIFT and presents related DIFT research. Raksha is described in Chapter 4, and evaluated by preventing a wide range of low and high-level software vulnerabilities on real-world unmodified binaries. Our novel DIFT policy for buffer overflow prevention is described in Chapter 5, and is implemented using Raksha to protect userspace applications and the Linux kernel. We present Nemesis, our PHP-based DIFT system for preventing web authentication and authorization bypass vulnerabilities, in Chapter 6. Chapter 7 presents strategies and guidelines for DIFT system design, from policies to runtime environments, based on our experience building Raksha and Nemesis. Finally, Chapter 8 presents conclusions and future work.

Chapter 2

Background & Motivation

In this chapter we present an overview of input validation vulnerabilities in modern software. For each vulnerability, we describe the vulnerability itself, which secure programming techniques can be used to prevent the vulnerability, and what defensive countermeasures exist to prevent attacks on vulnerable applications. This section describes 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 authentication bypass attacks in modern web applications. We omit from this discussion any security techniques based on Dynamic Information Flow Tracking as this material is presented in subsequent chapters.

We evaluate existing defensive countermeasures according to four key metrics:

- Safety - How well a technique resists being circumvented or 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. Practical

```
char buf[1024];

for (int i = 0; i < input_len; i++)
    buf[i] = user_input[i];
```

Figure 2.1: C code showing a sample buffer overflow vulnerability. User input of unbounded length is copied into a fixed-length 1024-byte buffer.

designs should even protect the operating system as it is the most privileged software code in the system, when applicable.

2.1 Buffer Overflows

Buffer overflow attacks are a critical threat to modern software security, even though they have been prevalent for over 25 years. A buffer overflow occurs when a program reads or writes to memory addresses outside of the bounds of an array or other data structure. An example of a buffer overflow vulnerability is presented in Figure 2.1. In this figure, user input is copied into a fixed-length, 1024-byte buffer. If the user input is greater than 1024 bytes in length, then it will overwrite information past the bounds of the buffer, potentially copying user input into security-critical memory such as the stack return address.

Buffer overflow vulnerabilities can result in complete compromise of the vulnerable application, allowing the attacker to execute arbitrary code with the privileges of the application. Typically, attackers use buffer overflow vulnerabilities to write past the end of an array, overwriting security-critical data structures such as the stack return address [81], heap metadata [42], or dynamic linking information [36].

2.1.1 Vulnerability Description

Buffer overflows occur in languages that are weakly typed and do not perform automatic bounds checking. The most widespread and prevalent languages that are vulnerable to buffer overflows are C and C++. These are also the same languages used to build much of today's high-performance, critical systems software. Applications ranging from the IIS and Apache web servers, to the Secure Shell login and authentication system, and even the Windows and Linux operating system kernels are written using C and C++. All of these applications have had critical, remotely exploitable buffer overflow vulnerabilities [70, 31, 65, 133, 60]. Many highly damaging and prominent worms such as Nimda [79] use buffer overflow attacks to infect host systems.

Exploitation of buffer overflows usually results in overwriting some security critical data with untrusted input by writing past the end of an array. Usually this security critical metadata is a code pointer, although attackers have also successfully compromised applications by overwriting data pointers, or even non-pointer data [13]. Commonly, exploitation leads to complete control over the vulnerable application and arbitrary code execution.

Application developers prevent buffer overflow attacks by *bounds checking* values before using them to index into arrays or other data structures. For example, before using any value as an array index, the application developer should perform bounds check comparisons which ensure that the value is not negative, and is less than the number of entries in the array. Failure to perform a bounds check on untrusted input before using it as an array index results in a buffer overflow vulnerability. A buffer overflow vulnerability also occurs if the application developer performs an incorrect bounds check, such as forgetting to check that a signed value is not negative before using it as an array index [115], or checking that a value is less than or equal to the number of entries in an array [35].

2.1.2 Countermeasures

As one of the oldest and most critical security attacks, buffer overflow attacks have received extensive scrutiny from academia and industry. However, despite this prolonged study, these attacks remain a pervasive threat the safety of our software systems. Modern countermeasures can be divided into the following categories: canary words, non-executable data pages, address space layout randomization, and bounds checking compilers.

Canary words

Canary words are used to prevent buffer overflows by placing random values before security-critical data such as stack return addresses or heap management information [19]. When a buffer overflow occurs, the canary word will be overwritten along with the security-critical data it protects. Before using any canary-protected data, application code always checks to ensure that the canary word is unmodified. If the canary has been changed, the application aborts with a security error.

Canary words have been implemented in practice using a security-aware compiler. The most popular Linux [33] and Windows [135] 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. Legacy code, especially code written in assembly, may behave incorrectly when data structure layouts are modified. Canary words are also not *safe* because they only protect certain designated security critical metadata. For example, stack-based canaries protect the return address, but do not protect security-critical local variables on the stack such as function pointers. Heap, BSS, and global data pointers are also unprotected. Attackers can still exploit buffer overflow vulnerabilities by overwriting data or code not protected by canaries, or by using format string vulnerabilities or other information leak attack vectors to leak canary values.

Non-executable Data Page Protection

Non-executable data page protection prevents a common form of buffer overflow attack by extending hardware to support per-page executable protection and requiring all writable data pages to be labeled non-executable. Many common forms of buffer overflow attacks inject code into the application, an approach that requires data pages to be labeled executable. This protection mechanism stops such attacks, as it prevents any writable page from containing executable code.

This technique 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 virtual machine interpreters such as the Java Virtual Machine. Furthermore, this approach is not *safe* at all, and in fact can be completely evaded by attackers. Non-executable data pages only prevent buffer overflow attacks that inject code, but buffer overflow exploits can be crafted that do not require code injection capabilities.

Novel forms of buffer overflow attacks have been invented that overwrite only data or data pointers [13], evading this protection mechanism completely. Furthermore, attackers can transfer control to existing application code rather than injecting their own code and still gain complete control of the vulnerable application using a technique known as return-into-libc attacks [71]. Finally, recent academic research has shown that an advanced technique similar to return-into-libc known as *return-oriented programming* results in arbitrary code execution in practice without requiring the attacker to inject a single byte of executable code [113]. These results largely undermine non-executable data page protection.

Address Space Layout Randomization

Address Space Layout Randomization (ASLR) is a recently proposed security technique that prevents buffer overflows by randomizing the base address of every region of memory (executables, libraries, stack, heap, etc.) within the virtual address space of the application. Buffer overflow attacks that overwrite pointers will be much more difficult to exploit because memory addresses will be randomized. This technique is deployed on modern Linux [84] and Windows [135] systems.

However, ASLR is not completely *safe* and can be bypassed in many practical situations. Any attack that overwrites non-pointer data will bypass ASLR [13]. Furthermore, on 32-bit systems ASLR randomizes an insufficient number of bits [112] to prevent reliable exploitation in practice. ASLR relies on the attacker's lack of knowledge of randomized base addresses, and can be completely defeated by vulnerabilities that leak pointer values to the attacker, which can occur during format string attacks [7, 34].

The most damaging flaw in ASLR's protection is that on little-endian systems such as the Intel x86, attackers can often reliably defeat ASLR by overwriting only the least significant bytes of a pointer [7]. ASLR only randomizes at page granularity, and thus the page offset contained in the least significant 12 bits of a pointer value remains unchanged even with full ASLR enabled. Attackers overwriting only the least significant two bytes of a pointer have a one in sixteen chance of succeeding, as long as the payload is within 65536 bytes of the pointer's initial value. On Windows systems, for backwards compatibility reasons, the least significant 16 bits of a pointer value must remain unchanged when using ASLR. Attackers overwriting the least significant two bytes of a pointer in a Windows environment will thus succeed every time, as ASLR is only applied to the most significant two bytes. This attack is not feasible on big-endian architectures.

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. Furthermore, many legacy applications make assumptions about where objects are laid out in memory, which are violated when memory regions are mapped at random addresses. ASLR can effectively protect an application only if all of the application libraries and executables are randomized by ASLR. A recent exploit for Adobe Flash was able to bypass ASLR because one of Adobe's libraries was not compatible with ASLR, and thus ASLR was disabled for the entire application [29].

Real-world buffer overflow attacks have been developed that successfully exploit Windows Vista systems protected even by the combination of canary words, address space layout randomization, and non-executable data [34]. Transparent, backwards compatible, highly-performant buffer overflow protection remains an open challenge.

Bounds Checking Compilers

Bounds checking compilers provide automatic bounds checking for legacy C applications. This technique modifies each pointer to keep track of the upper and lower bounds of its referent object. All pointer dereferences are then instrumented to ensure no out-of-bounds or illegal memory address is accessed [69].

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 [69, 107], 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.

Backwards compatibility is a serious challenge for bounds checking compilers. The C language was designed without the notion of object bounds, and retrofitting this concept onto existing C applications cannot be done in a transparent manner. For example, C

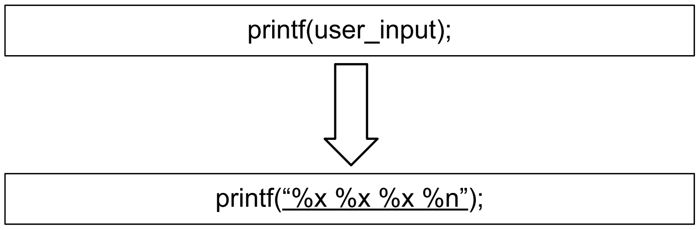


Figure 2.2: Sample C code vulnerable to format string attacks, and the resulting format string when a user supplies the underlined, malicious input.

programs may safely read out of bounds as long as the read does not cross a page boundary. This is done in high-performance library routines for scanning data, which may read in aligned chunks and discard any values read out of bounds. Buffer overflows may also occur undetected by a bounds checking compiler if the application uses its own memory allocator, as is done in large server programs such as Apache.

Furthermore, bounds checking compilers must compile all of an application and its libraries to provide complete protection. Even then, routines written in assembly, which often include critical functions for copying data such as `memcpy()`, cannot be instrumented. As a consequence, manual wrappers must be written for all routines for which C source code is not available, or that cannot be recompiled with a bounds checking compiler. This is an impractical and non-scalable approach that does not reflect modern production environments, where source code for many components is not available. Requiring all system components, from applications to libraries, to be recompiled is not practical, nor is requiring end users or developers to maintain an up-to-date set of wrappers for all uninstrumented functions.

2.2 Format String Attacks

Format string attacks are the most recently discovered form of memory corruption attack. A format string attack allows attackers to read or overwrite arbitrary memory locations in weakly typed languages with variable argument functions. This form of attack occurs when user input is used as a format string to variable argument functions such as the `printf()` family of functions in C.

An example of a format string attack is shown in Figure 2.2. In this example, user input is passed directly to the `printf` library function. The `printf()` style functions take a format string specifier as the first argument, and use this specifier to determine how many words of memory to read and write on the stack. If the untrusted input contains format string specifiers such as %x, the printf function may read or write to arbitrary, attacker-controlled memory addresses.

2.2.1 Vulnerability Description

Format string attacks occur when untrusted input is used as a format string specifier. In weakly typed languages such as C and C++, this allows attackers to effectively read or write arbitrary stack locations by inputting the appropriate format string specifiers. Format string specifiers such as %x print the next word on the stack as an integer, decimal, string, or other type. The %n specifier writes the number of characters printed so far to the memory address specified by the next word on the stack. In Figure 2.2, the malicious format string ”%x %x %n” will cause printf to print the top 2 words on the stack as 32-bit hexadecimal numbers, and then write the number of characters printed so far to the memory address specified by the third word from the top of the stack.

A format string exploit allows attackers to write an arbitrary value to an any memory location, resulting in complete control of the vulnerable application. Application developers prevent format string vulnerabilities by only using trusted application data as a format string. User input must only be used as an argument to a `printf`-style function, never as the format string itself.

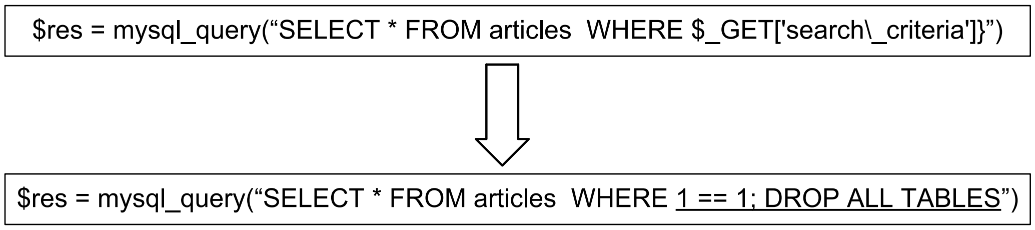
2.2.2 Countermeasures

Only the %n format string specifier writes to a memory address. Thus, vendors may choose to ship C libraries without support for the %n format string specifier, reducing the threat of format string vulnerabilities from arbitrary code execution to information leaks. Another approach is to interpose on all library calls to the printf family of functions and forbid the %n specifier. These approaches are not *practical*, as they break backwards compatibility with existing programs that legitimately use %n. 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 [7].

Academic researchers have also proposed compiling programs with special C preprocessor macros which count the number of arguments passed to the `printf` family of functions by the application, and dynamically verify that the format string supplied to printf does not access more than the application-supplied number of arguments [18]. This approach is not completely *safe*, as it only verifies that format strings do not access more than the application-supplied number of arguments. Attackers could supply malicious format string specifiers to arbitrarily read and write any application-supplied arguments.

Furthermore, these tools must be provided with a list of `printf`-style functions to be protected, and do not protect unknown functions. Any `printf` calls that have a va_list argument (such as `vfprintf`), or any calls made through a function pointer, are also not protected. This approach is also not entirely *practical* as it requires applications to be recompiled. Manually written wrappers must be specified to protect functions for which no source code is available that may invoke a `printf` style function.

The most recent academic proposal for preventing these attacks uses static analysis to determine a whitelist of safe memory addresses that can be accessed by calls to the `printf` family of functions [101]. Static analysis is used to determine which elements on the whitelist should be writable. This approach has better attack coverage than [18], as it handles functions with va_list arguments. However, this proposal is not *practical* as it requires source code access for static analysis and recompilation of the application. Furthermore, a list of all `printf` style functions must be supplied, and wrappers must be written for any functions for which source code is not available that may call a `printf` style function. As a consequence, this approach is not entirely *safe* without significant manual effort.



```
$res = mysql_query("SELECT * FROM articles  WHERE $_GET['search\_criteria']}")
```

```
$res = mysql_query("SELECT * FROM articles  WHERE 1 == 1; DROP ALL TABLES")
```

Figure 2.3: Sample PHP code vulnerable to SQL injection, and the resulting query when a user supplies the underlined, malicious input.

2.3 SQL Injection

SQL injection is a prevalent and highly damaging security vulnerability affecting modern web applications. This attack occurs when user input is used to construct a SQL query without performing adequate validation and filtering. SQL queries are often formed by concatenating user input with SQL commands, forming a single string that is sent to the database. If user input contains SQL command characters that are not filtered, the user may be able to inject arbitrary SQL statements. This vulnerability can occur in all languages, including high-level, strongly typed languages such as Python or PHP. Figure 2.3 provides an example of a SQL injection vulnerability.

2.3.1 Vulnerability Description

SQL injection attacks occur when applications use untrusted input in a SQL query without correctly filtering the untrusted input for SQL operators or tokens. When the user input is combined with the SQL query from the application, the untrusted input can actually change the parse tree of the SQL query by adding the appropriate tokens. In Figure 2.3, an attacker uses SQL Injection to insert a SQL command that deletes all tables from the database.

SQL injection attacks often result in arbitrary SQL query execution with the database privileges of the vulnerable web application. This can allow attackers to arbitrarily read, modify, or delete database entries. Some databases, such as Microsoft SQL Server, also support command execution, allowing a SQL injection attack to perform arbitrary code execution with the privileges of the database user.

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. If an application does not filter user input before it is used in a SQL query (or if the filter is incorrectly applied), then a SQL injection vulnerability occurs.

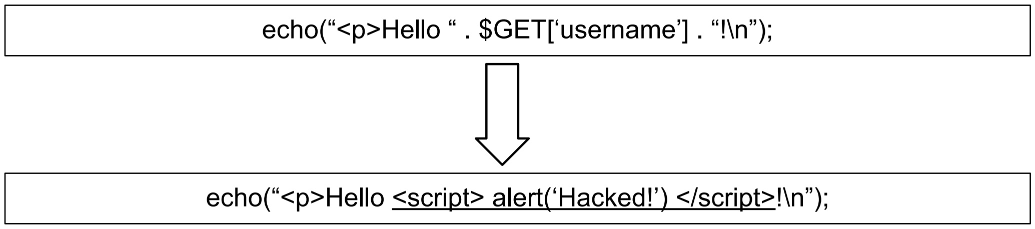
2.3.2 Countermeasures

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. Without the ability to distinguish which bytes of the query came from the untrusted user and which bytes came from the trusted application, security tools resort to heuristics to guess which queries are malicious.

Web Application Firewalls

Web application firewalls (WAFs) are an appliance, plugin, or filter that monitors Internet traffic, including HTTP and database traffic, to detect security attacks such as SQL injection. These devices detect attacks through passive monitoring, using heuristics to determine when a SQL injection attack is underway by examining HTTP parameters and database queries for potentially malicious or anomalous content.

However, WAFs are not *safe*, as they do not have enough information to make accurate decisions regarding SQL injection. Without the knowledge of which bytes a in SQL query are actually derived from user input, web application firewall vendors resort to a blacklisting approach that heuristically flags known malicious or extremely anomalous SQL queries. This allows for false positives and negatives, because the heuristic function does not have sufficient information to accurately detect SQL injection attacks in the general case. Legitimate application queries may result in false positives if the heuristic believes them to be malicious, while SQL injection attacks that do not match any of the malicious query heuristics will be allowed through.



```
echo("<p>Hello " . $GET['username'] . "!\n");
```

```
echo("<p>Hello <script> alert('Hacked!') </script>!\n");
```

Figure 2.4: Sample PHP code vulnerable to cross-site scripting and the resulting HTML output when a user supplies the underlined, malicious input.

Database ACLs

The impact of SQL injection vulnerabilities can be mitigated by restricting the database privileges of web applications. While this does not prevent the attacker from obtaining complete access to all web application database tables, it does prevent the attacker from further accessing any database resources that should not be accessed by the web application. This approach is not *safe* as it not comprehensive. It only mitigates damage.

2.4 Cross-site Scripting

Cross-site scripting is an extremely prevalent web security vulnerability that allows an attacker to inject arbitrary HTML and JavaScript into a web page. This vulnerability occurs when untrusted input is included in an HTTP response without appropriate validation or filtering. Cross-site scripting attacks affects all languages. An example of a cross-site scripting exploit is given in Figure 2.4.

2.4.1 Vulnerability Description

Cross-site scripting vulnerabilities occur when untrusted data from URL parameters, database tables, or other sources is included in an HTTP response without filtering or validation. In Figure 2.4, a URL parameter is included in HTML output without any filtering. A user can set the URL parameter to contain malicious JavaScript, which will be returned as part of the HTML output and executed by the client's web browser in the origin of the vulnerable web site.

An attacker can exploit this vulnerability by sending the victim a link to the vulnerable site with a malicious payload encoded in the URL parameters. When the victim clicks on the URL, the response sent back by the web server will include the attacker's malicious HTML and JavaScript.

Cross-site scripting allows attackers to insert arbitrary HTML and JavaScript into the HTTP response, which will then be executed by the victim client's web browser in the domain of the vulnerable web site. Cross-site scripting attacks can be used to portscan local area networks behind a firewall [43], steal user cookies, or perform arbitrary HTTP transactions. The malicoius payload executes with the privileges of the client in the origin of the vulnerable domain, which even allows current login sessions to be hijacked.

2.4.2 Countermeasures

Cross-site scripting attacks are the most common form of web application vulnerability [17], and unsurprisingly are very difficult to reliably detect. Security vendors cannot tell which bytes of an HTML document come from safe, trusted application, and which bytes come from untrusted sources. Thus, vendors rely on heuristics to guess when an HTTP response contains malicious HTML or JavaScript. Compounding this problem is the state of the modern web browser compatibility, where each browser has its own idiosyncrasies that change across each version, and the HTML standard is loosely adhered to at best. Existing cross-site scripting defenses can be divided into two categories: web application firewalls and browser-based defenses.

Web application firewalls are discussed in Section 2.3.2. Using a WAF to protect against cross-site scripting has the same drawbacks as using a WAF to prevent SQL injection attacks: WAFs ultimately rely on heuristics. Given the loose, ill-defined nature of HTML and how cleverly attackers can exploit browser and even browser version-specific behavior, cross-site scripting represents a particularly difficult environment for heuristic-based approaches [105].

A similar approach taken browsers such as Internet Explorer 8 [104], which uses heuristics to detect when user input passed to the web application is reflected back in an HTTP response in an unsafe manner. This approach has similar drawbacks to a WAF, and the IE

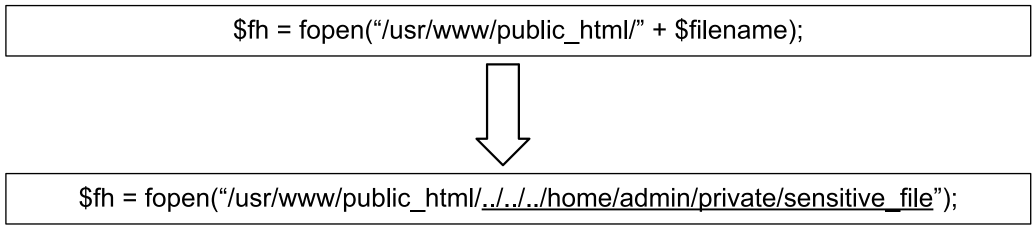

$fh = fopen("/usr/www/public_html/" + $filename);
$fh = fopen("/usr/www/public_html/../../../home/admin/private/sensitive_file");

Figure 2.5: Sample PHP code vulnerable to directory traversal, and the resulting filename when a user supplies the underlined, malicious input.

team has stated that complete attack prevention is not a goal. Both WAFs and browser-based Cross-site scripting protection rely on heuristics, and ultimately are not *safe*. False positives are also possible in both approaches, as heuristics may flag legitimate output as malicious.

2.5 Directory Traversal

Directory traversal attacks are a high-impact security vulnerability that allow attackers to read or write arbitrary files. This vulnerability occurs when untrusted input is used in the filename argument to a filesystem library function such as `open()` without appropriate validation or filtering. This vulnerability affects all languages. An example of a directory traversal attack is given in Figure 2.5

2.5.1 Vulnerability Description

Directory traversal attacks occur when untrusted input is used to construct the filename for a file open library call. If the application does not restrict the untrusted input appropriately, attackers may be able to trick the application into reading or writing attacker-chosen files. This provides the attacker with arbitrary file read or write access with the privileges of the vulnerable application. In Figure 2.5, the vulnerable application appends untrusted input to a filename without filter. When the open called succeeds the file "/home/admin/private/sensitive_file" will be returned for reading or writing.

Application developers prevent directory traversal attacks by filtering user input for filename path separators or other metacharacters, or by restricting filenames containing user input to a whitelisted set of directories and files. If application developers forget to apply the appropriate filter or security check before opening a file containing untrusted input in the filename, then a directory traversal vulnerability occurs.

2.5.2 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.

System Call-based Anomaly Detection

Academic researchers have proposed intrusion detection systems which interpose on system calls to monitor the files opened and executed by an application [44]. In this approach, an operating system module records all system calls and their arguments during training runs of a program with benign input. This information is then used to detect anomalous system calls in production. Anomalous system calls are rejected and assumed to be malicious attacks.

This approach is not *safe*, as it allows for both false positives and negatives due to a lack of sufficient information. Without the ability to detect which bytes of the program name are derived from user input, this technique cannot reliably distinguish between legitimate access to a security-critical file, and a directory traversal attack. Furthermore, any unexplored paths that are only exercised in production may easily result in false positives because this technique requires training to tune its heuristics. Researchers have shown that practical attacks may easily bypass anomaly-based system call intrusion detection systems by mimicking legitimate system call traffic [82].

Per-application ACLs

Another way to limit the damage caused by directory traversal is to restrict the files each application can access using Access Control Lists. All major operating systems today include access control frameworks that provide fine-grained, per-application rules for accessing files and other resources. Examples of such systems include SELinux [111], AppArmor, and Solaris Role Based Access Control. However, this approach only prevents applications from accessing unauthorized files. A successful directory traversal attack will still allow the attacker to access any file that the application is authorized to read or write. Thus, this defense is not *safe* as it not comprehensive – it only mitigates damage.

2.6 Command Injection

Command injection attacks are a high-impact security vulnerability that allows attackers to execute arbitrary programs. This vulnerability occurs when untrusted input is used as an argument to a command execution function such as `execve` without appropriate validation or filtering. This vulnerability affects all languages.

2.6.1 Vulnerability Description

Command injection attacks occur when untrusted input is used when constructing the filename and arguments of a program to be executed. If the application does not restrict the untrusted input appropriately, attackers may be able to trick the vulnerable application into executing arbitrary programs. This provides the attacker with arbitrary program execution capabilities with the privileges of the vulnerable application.

Application developers prevent command injection vulnerabilities by writing filters that restrict user input to a whitelist of safe values. 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. If application developers forget to apply the appropriate filter before executing a command statement containing untrusted input, or if the filter is incorrect, then a command injection vulnerability occurs.

2.6.2 Countermeasures

Current methods for addressing command injection attacks are the same as those used to protect against directory traversal, described in Section 2.5.2. Command injection allows attackers to control what files are executed, while directory traversal allows attackers to control what files are read or written.

2.7 Authentication & Authorization Bypass

Authentication & authorization bypass are prominent web application vulnerabilities that result in unauthorized access to file or database resources. Authentication vulnerabilities occur when the web application authenticates users without requiring valid credentials (such as a password). This often results in attackers obtaining access to the web application as the most privileged user, usually the web application administrator. Authorization bypass vulnerabilities occur when a web application user can access an unauthorized file or database entry due to a missing or incorrect access control check.

2.7.1 Vulnerability Description

Authentication bypass vulnerabilities occur when the web application authentication framework improperly validates authentication credentials. This can occur due to improper trust relationships, such as failing to validate cookies or URL parameters. For example, a vulnerable web application may store the username of the current login session in a cookie in plaintext. A malicious user could then edit their cookie, and change the username to ”admin”, and then would be treated as the admin user by the web application. Whenever a web application user authenticates a user, all authentication parameters and credentials must be thoroughly validated. Any credentials stored in an untrusted location, such as a cookie, should be encrypted with a private, server-side key.

Authorization bypass attacks occur when a web application improperly performs access control checks before performing operations on a protected resource. For example, if a particular database table should only be accessed by the web application administrator, then each access to that table must be preceded by a check to ensure that the current web application user is an administrator. Missing an access control check, or performing an access control check improperly, results in an authorization bypass vulnerability.

2.7.2 Countermeasures

Each web application typically creates its own authorization and authentication framework. Furthermore, there is no ”web application ACL” that maps web application users to the resources they may access. Instead, the access control rules for web application users are implicitly defined by the hundreds of security checks interspersed throughout the web application. These access control checks verify the privileges of the current web application user before allowing access to a restricted file or database entity.

Due to the ad-hoc, application-specific nature of authentication and access control, there are no major countermeasures in use today by industry to prevent authentication and authorization bypass attacks. Authorization and authentication bypass attacks result in unauthorized access to resources. However, these resources are the very same resources that can be legitimately accessed by the web application when acting on behalf of a privileged web application user. Without knowledge of the internal authentication framework and authorization rules used by the application, an outside source cannot distinguish between an authorization or authentication bypass attack and legitimate access to privileged resources.

Preventing web authentication and authorization vulnerabilities is a relatively new area of research. The academic work in this area of which we are aware is the CLAMP research project [83]. CLAMP prevents authorization vulnerabilities in web applications by migrating the user authentication module of a web application into a separate, trusted virtual machine (VM). Each new login session forks a new, untrusted session VM and forwards any authentication credentials to the authentication VM. Authorization vulnerabilities are prevented by a trusted query restricter VM which interposes on all session VM database accesses, examining queries to enforce the appropriate ACLs using the username supplied by the authentication VM. CLAMP does not prevent authentication bypass attacks, and it is not *fast* as it must fork a new VM for each connection. CLAMP was only able to fork two VMs per second in benchmarks, unacceptably slow for real-world servers that must handle thousands of connections per second.

2.8 Conclusion

In this chapter we surveyed the most critical computer security vulnerabilities and their countermeasures. To completely prevent each of these bugs, programmers must insert the appropriate security checks (bounds checking for buffer overflows, access control checks for authorization, SQL escaping for SQL queries, etc) before using untrusted input in a security-critical operation. These attacks occur across many different layers of abstraction, from buffer overflows in operating system device drivers to SQL injection vulnerabilities in PHP web applications.

However, preventing each of these attacks places the same burden on application developers – requiring them to insert appropriate security checks throughout the application, never missing or incorrectly performing a single check. This is a fundamentally unsafe approach as it relies on programmers to be infallible. Furthermore, failure to insert the appropriate check will not be discovered in normal QA testing because missed security checks do not affect the functionality or correctness of the application, only its safety when faced with a malicious adversary.

Existing countermeasures fail to meet the four requirements outlined in Chapter 1. Most protection mechanisms are not *safe*, as is the case with the NX-bit for buffer overflows, or web application firewalls for common web vulnerabilities. Other defenses are not transparent or practical, such as Address Space Layout Randomization. There are even attacks for which no effective countermeasures currently exist at all in industry, such as authorization and authentication bypass attacks.

Furthermore, many of the most prevalent defenses used in industry such as Web application firewalls or intrusion detection systems, rely on anomaly-based heuristics, or heuristics containing a blacklist of malicious behavior. Heuristics are an unsafe approach because they do not have sufficient information to decide whether a filename, SQL query, or execute statement is malicious. Heuristics cannot identify which parts of the filename or query are derived from untrusted sources, and which parts are derived from trusted resources within the application. Instead, heuristics detect anomalous activity or attempt to flag clearly malicious activity.

Consider the cross-site scripting exploit in Figure 2.4. It is impossible to distinguish reliably between HTML and JavaScript output by the application and malicious HTML or JavaScript content injected by the user without knowing which bytes in the output are derived from untrusted sources. Any HTML or JavaScript flagged as malicious by a heuristic filter could have been legitimately generated by the application. Only taint information can reliably disambiguate between these two cases.

To defeat heuristics, attackers must only massage their payload until it appears to be benign to the heuristic filters. This is made easier by the fact that false positives cost users money by denying legitimate paying customers access to resources, and thus any commercially successful heuristic must minimize false positives by using reasonably permissive heuristics. Given the massive financial incentives involved in cybercrime, attackers will find the holes in these permissive heuristics, craft successful attacks, and exploit systems. The vendor will then try to build more complex heuristics that make the current round of attacks more difficult, and the cycle repeats indefinitely. There is no decoupling of heuristic mechanisms from attacks as each heuristic is constructed a single ad-hoc blacklist or filter. Thus, when a heuristic is broken, it likely that an entirely new heuristic must be developed, requiring significant manual effort. Heuristics are not an effective protection technique – they are an endless war against a foe with essentially infinite resources, where the defender is always one step behind.

The state of the art in industry is an ad-hoc collection of per-vulnerability heuristics and tools that fail to provide comprehensive protection against a single form of attack, much less the full breadth of attacks faced by modern software developers [34, 29]. None of the techniques described is *flexible*, as all existing solutions are per-vulnerability or even per-attack. Fortunately, recent research in Dynamic Information Flow Tracking (DIFT) systems has shown great promise towards developing a systematic approach to comprehensively prevent input validation security attacks.

Chapter 3

Dynamic Information Flow Tracking

Dynamic Information Flow Tracking (DIFT) is a promising technique for preventing security attacks. DIFT tracks the flow of untrusted information at runtime, commonly by associating a tag bit with each byte of memory and each register. This tag bit is set for any untrusted information entering the system. Program instructions propagate tag bits from source operands to destination operands at runtime. 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.

DIFT provide a clear separation between mechanism and policy. DIFT platforms provide the basic infrastructure for performing tag checks and propagation on applications. DIFT policies specify the necessary tag check and propagation rules to prevent a class of software vulnerabilities, and are executed by a DIFT platform.

This chapter provides a complete overview of DIFT. Section 3.1 provides a thorough overview of DIFT itself and its applications, while Section 3.2 discusses how DIFT systems are implemented. Section 3.3 discusses the potential DIFT and summarizes the state of the art in DIFT research as well as the challenges that remain in helping DIFT to reach its full potential as a security technique.

3.1 Overview

Dynamic Information Flow Tracking is a technique for tracking and restricting the flow of information when executing programs in a *runtime environment* [76, 121, 12, 40]. A runtime environment consists of memory, storage or I/O device *resources*, and a program interpreter. Runtime environments may be implemented in software, such as the Java Virtual Machine, or hardware, such as an Intel x86 computer. Programs are represented as sequences of instructions. The runtime environment interprets program instructions to allow the program access to memory and storage resources.

A DIFT platform associates tags with memory and resources, and uses these tags to track the flow of information throughout the system. The DIFT platform is controlled by a *DIFT policy*, which specifies the *tag sources*, *tag sinks*, *tag propagation rules*, and *tag check rules* for the runtime environment. Tag sources are resources that produce specially tagged output. If an instruction loads from a tag source, the destination operand of the load instruction has its tag bit set. Tag sinks are security-sensitive operations that require security checks to be performed on the tags of their operands before the operation can be executed. For example, an execute program function might require that the name of the program be untainted to prevent untrusted users from running arbitrary commands. The tag check rules specify the invariants and checks that must be enforced before a tag sink operation is executed. Tag propagation rules specify how tags should be propagated from source operands to destination operands when executing instructions.

In a DIFT-aware runtime environment, programs are unmodified. Tags are managed by the runtime environment and are completely transparent to the program. As the program executes, any attempt to read from a tag source produces specially tagged output. During runtime execution of instructions, tags propagate from source operands to destination operands as defined by the tag propagation policy, allowing data derived from a tag source to be tracked precisely at runtime. Security checks are performed before each tag sink according to the tag check rules. DIFT policies restrict the flow of information from tag sources to tag sinks using tag check rules and track the flow of information during runtime execution using tag propagation rules.

Security exploits occur because applications process untrusted input. At some point, this untrusted input must be 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, results in an input validation security vulnerability. This description applies just as well to high-level vulnerabilities such as cross-site scripting as it does to low-level vulnerabilities like buffer overflows. DIFT is a powerful and comprehensive technique because it is an acknowledgment of the crucial role that information flow plays in all vulnerabilities. By tracking and restricting the flow of untrusted information, DIFT policies can prevent user input from being used unsafely in a security-sensitive manner.

3.2 Implementation

To implement DIFT in a runtime environment, the platform designer must:

- Extend all memory and resources with support for tags
- Instrument loads from tag sources to initialize the tag of the destination operand
- Wrap the execution of all relevant instructions to perform tag propagation
- Insert security checks before each tag sink

The exact manner by which DIFT is implemented varies from system to system, but all DIFT implementations should fulfill the above requirements. How these requirements are implemented determines the performance, safety, and flexibility of the underlying system.

The security and safety of DIFT depends on the precise tracking of information flow. If a DIFT implementation misses potential information flow paths, it may have false negatives, allowing security attacks to circumvent DIFT. However, this occurs only if the application software uses methods that the DIFT implementation does not adequately support, and these methods allow untrusted data to exploit an existing software vulnerability. The critical difference between preventing input validation attacks with DIFT in comparison to other techniques is that effective information flow tracking depends on the code written by the (presumably non-malicious) application developer. Conventional security defenses such as intrusion detection systems or web application firewalls base their safety on the examination of malicious, untrusted attacker input. As attackers should not supply application code [1], DIFT is far less likely to be evaded by a malicious attacker.

When designing a DIFT policy to protect applications, the policy designer must use all available information and determine what constitutes information flow within the application. If a DIFT system is too conservative, it may incorrectly taint the output of operations that are not truly propagating untrusted information. This can result in false positives, as trusted information is incorrectly tainted. Similarly, missing a potential information flow could result in false negatives.

Designing policies for DIFT systems can be an art, and a balance must be struck to ensure that real-world information flow is properly represented, while false positives are avoided [75, 25]. Fortunately, DIFT policies have found great in practice success using relatively simple check and propagation rules, without resorting to a list of dozens of ad-hoc exceptions or heuristics. In practice, 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.

In most DIFT policies designed to prevent input validation attacks [12, 100, 76], tag propagation occurs during all data movement instructions such as add, multiply, or load/store at byte granularity, but no propagation occurs due to control operations such as branch instructions. This is done to prevent false positives because programs rarely copy data using branch conditions. If DIFT policies propagated on tainted branch conditions without restriction the entire address space would quickly become tainted. No known input validation attacks have occurred due to control flow propagation. Furthermore, input validation protection assumes a non-malicious, but vulnerable application. Thus, the application itself is not attempting to subvert DIFT, and DIFT policies may track only the reasonable forms of information flow.

[1]Code injection attacks can be comprehensively prevented by the buffer overflow protection policy described in Chapter 5

3.3 State of the Art

DIFT is a versatile and powerful technique, and academic researchers have explored the application of DIFT to system security in many different environments. However, many problems remain unsolved, and the potential of DIFT has yet to be fully realized. In this section we first discuss the potential for DIFT. We then describe the state of the art in DIFT policies and platforms, comparing current research to an ideal DIFT system. DIFT platforms are discussed from the lowest level (hardware) to the highest language (programming languages). We close with a discussion of other applications of DIFT, describing how DIFT can be used to solve security attacks that are not related to input validation.

3.3.1 Potential

DIFT is a promising and uniquely powerful security technique because it is the first, and to our knowledge only, technique with the potential to meet all of the qualities of an ideal security defense described in Chapter 2.

DIFT can be *safe* because it can comprehensively protect against an attack by preventing untrusted data from performing harmful operations. For example, DIFT can provide complete protection against injection attacks such as SQL injection and command injection [119]. This is because DIFT tracks the flow of untrusted information precisely and can determine which bytes of a filename or SQL query are derived from untrusted input. Unlike many popular defensive techniques used today, such as intrusion detection systems or web application firewalls, DIFT does not rely on heuristics. When using DIFT, untrusted information can always be unambiguously identified by its tag bit, preventing attackers from evading DIFT policies by cleverly encoding their malicious payloads to appear to be benign data. Ideally, DIFT policies should be comprehensive and robust, without real-world false negatives.

Furthermore, DIFT can be a *flexible* technique because researchers have developed DIFT policies to protect against attacks ranging from high-level command injection attacks [119] to low-level buffer overflow attacks [12]. No other security technique has been applied to such a broad range of attacks. Ideally, there should be a DIFT policy to protect against every major class of input validation attack.

DIFT can be *practical* because it does not depend on knowledge of the semantics of the application internals or program design. DIFT only tracks and restricts the flow of information at runtime during program execution. This design allows almost all DIFT systems to work on unmodified application binaries or bytecode, without requiring any source code access or debugging information [121, 40]. Ideally, DIFT policies for preventing input validation attacks should run on unmodified applications with no additional support or debugging information, and have no real-world false positives.

Finally, DIFT platforms can be *fast*, and many high-performance DIFT systems have been implemented in hardware [121, 12] and software [100, 77, 40]. Ideally, DIFT policies should have negligible performance overhead, and impose no scaling or performance restrictions on the application.

3.3.2 Policies

Academic researchers have had success in creating DIFT policies to prevent many kinds of input validation attacks. *Practical* and *safe* policies exist for SQL injection, command injection, cross-site scripting, format string, and directory traversal attacks [119, 40, 68, 76].

However, challenges remain. Existing DIFT buffer overflow policies are *unsafe* and *impractical*, and have significant false positives and negatives in real-world applications. Buffer overflows are the oldest security vulnerability, and are still a critical threat to modern computer systems. A successful buffer overflow exploit results in arbitrary code execution, often immediately gaining the attack complete control over the target system.

Furthermore, there are high-impact vulnerabilities that have yet to be addressed by a DIFT policy. In particular, web authentication and authorization vulnerabilities have no corresponding DIFT policy. These attacks are particularly dangerous as authentication vulnerabilities often allow the malicious attacker to perform arbitrary web application operations with full administrator privileges, resulting in a complete compromise of the vulnerable web application.

Finally, the operating system is a crucial system component and the most trusted software in most computing environments. There are no DIFT policies for addressing operating system security issues, such as operating system-level buffer overflows or user/kernel pointer dereferences.

3.3.3 Hardware Platforms

Researchers have proposed many hardware DIFT systems [121, 12]. In these systems, the runtime environment is the CPU. Memory is represented by CPU caches and physical RAM, while resources map directly to I/O devices. DIFT is implemented by extending all registers, memory, and CPU caches with tag bits. Tag propagation is performed inside the CPU in parallel with instruction execution.

Existing hardware systems are *fast*, as tag storage and tag propagation are provided by hardware. Hardware DIFT designs may also extend the memory coherence protocols to provide safe, low-overhead tag operations even in multithreaded programs where different threads may concurrently update the data and tags of a memory location. This prevents DIFT from inhibiting application scalable in the presence of multithreading and multicore.

However, existing hardware approaches use a single, fixed policy for preventing buffer overflows. These designs are extremely *inflexible*, supporting only a single policy hard-coded into the hardware itself. This is a significant drawback, as DIFT can solve a wide range of security vulnerabilities. Justifying the cost of hardware DIFT is very difficult if the hardware platform cannot address a wide range of security flaws. The current buffer overflow policy in use by existing hardware DIFT implementations is *impractical* and *unsafe* due to real-world false positives and negatives, as discussed in Chapter 5.

3.3.4 Dynamic Binary Translation Platforms

A practical hardware DIFT implementation would require substantial high-risk investments by major hardware vendors. Hardware DIFT designs are also not as flexible as software, which can be made infinitely malleable. As a consequence of these drawbacks, academic researchers have investigated the user of Dynamic Binary Translation (DBT) technology to implement DIFT [94, 100, 76, 72] in software. In a DBT-based DIFT implementation, applications (or the entire system) are run entirely within a dynamic binary translator. The DBT dynamically inserts instructions for performing DIFT operations during binary translation.

Dynamic Binary Translators may be applied to a single application [72, 51, 6], or to the entire system [5]. In the former case, registers, virtual memory, and I/O system calls must be managed by the DIFT infrastructure, whereas in the latter DIFT manages registers, physical memory, and I/O storage devices. In either case, DBT-based designs have been used to prevent many of the same kinds of attacks as hardware-based approaches, as they operate at the same layer of abstraction.

However, DBT-based approaches are *slow*, with performance overheads ranging from 3x [100] to 37x [76]. Without hardware support, DBT-based DIFT requires multithreaded applications to run one thread at a time [73] in a serialized fashion to prevent race conditions when updating the data and tags of shared memory. This significantly limits application scalability. Recent research into hybrid DIFT systems has shown that extra hardware support can allow for multithreaded applications within DBTs [15, 73], but this requires hardware modifications to existing systems.

3.3.5 Programming Language Platforms

Many high-level vulnerabilities may be difficult to express in terms of low-level concepts such as memory addresses or registers. Hardware and dynamic binary translation implementations of DIFT are too low-level to easily express these policies. For example, SQL injection in the Java Virtual Machine occurs when specific database query execution methods are called with untrusted SQL commands. However, the address of these methods varies at runtime due to Java's Just-In-Time (JIT) compiler, which may recompile methods at runtime to apply various (possibly speculative) optimizations. DIFT policies for Java are more naturally expressed within the JVM by providing the class and method names of SQL execution methods, which can easily be resolved by any Java code running within the JVM.

Language DIFT implementations add DIFT capabilities to a language interpreter or runtime. Researchers have proposed DIFT implementations for many languages, such as

C [68], PHP [78], Java [40]. Additionally, DIFT concepts are already used in limited situations by many existing interpreted languages, such as the taint mode found in Perl [86] and Ruby [124].

In a language DIFT implementation, the runtime environment is the language interpreter. From a DIFT perspective, memory consists of language variables, which are extended with tags to track taint. Programs consist of source statements in the language (or in the case of a virtual machine such as the JVM, bytecode instructions for the language virtual machine). Resources are specified by listing all methods in the system library that may interact with files or I/O devices.

Language DIFT systems are *flexible* and have been used with great success to provide protection from high-level vulnerabilities [119, 78, 68, 40, 26] with minimal performance overhead. Researchers have implemented DIFT systems by modifying the interpreters of modern web languages such as PHP to prevent a variety of web input validation bugs such as SQL injection, directory traversal, cross-site scripting, command injection, and authentication bypass.

Language DIFT can be *fast* as the interpreter may optimize tag storage, checks, and propagation using high-level information. However, this approach cannot address particular kinds of vulnerabilities (such as low-level buffer overflows or operating system vulnerabilities), rendering it *unsafe* against certain attacks. Furthermore, this approach is *impractical* if the user wants to defend against vulnerabilities that occur in a wide variety of languages, as this technique protects only a single language from attack.

3.3.6 Other DIFT Applications

DIFT has also been used to prevent security vulnerabilities that are not related to input validation. These additional applications of DIFT have very different threat models as they assume the application itself is malicious. Input validation protection assumes vulnerable, but non-malicious applications that are processing untrusted, malicious input.

Users are often attacked not only by exploits for vulnerable software applications, but also by malicious software or viruses. However, from the operating system's perspective, a vulnerable application compromised by an attacker and a malicious virus or backdoored application are identical – each attempts to execute system calls to circumvent security. Consequently, OS researchers have investigated DIFT-based to prevent potentially untrusted applications from disclosing, leaking, or modifying sensitive data without authorization [139, 54].

This technique, known as Dynamic Information Flow Control (DIFC), applies the DIFT concept to operating system processes. In this design, programs are treated as sequences of system calls. The operating system associates a tag (or *label*) with each process denoting the tags of any information it has been exposed to. For example, a process that reads untrusted information from the network will be labeled untrusted, while a process that reads the password file will be labeled sensitive. The origin of a process may also affect its tag, so that an executable downloaded from the Internet is labeled untrusted at startup. Tags are combined using set union semantics.

In a DIFC system, network connections and file I/O serve as tag sources while interprocess communication results in tag propagation. Tag checks occur before any operation that may result in the transfer of sensitive information out of the system (such as via the network), or any operation that may modify a critical system resource. Sample DIFC policies include preventing an application that has read sensitive information (e.g., financial data) from communicating over the Internet, or preventing any application exposed to untrusted input from performing unauthorized modifications to critical system files.

Researchers have also explored applying DIFT to hardware at the gate level [125] to fully prevent information leaks, even solving the notoriously difficult problem of covert storage and covert timing channels. In this design, hardware state such as registers and memory serve as storage. Programs consist of (possibly concurrent) sequences of operations on logic gates such as AND gates, OR gates, and multiplexors. Information leaks are prevented by tracking information flow at the lowest possible level – the logic gates that make up modern computer hardware. Timing attacks are prevented by using execution leases, which lease the CPU and other resources to a program for a fixed period of time. This design is significantly more fine-grained than all other DIFT platforms, as it tracks information flow at the individual gate level rather than at the ISA instruction or program operation granularity. However, this approach is also considerably more costly to implement.

3.4 Conclusion

Dynamic Information Flow Tracking (DIFT) is a versatile, promising technique that can be used to prevent a wide range of software attacks, and can be implemented at many different layers of abstraction. Researchers have shown great progress in solving many of today's most critical vulnerabilities using DIFT.

However, much work remains to be done. Some vulnerabilities such as buffer overflows, have no practical, safe DIFT policy. Security vulnerabilities such as operating system security flaws and web authentication vulnerabilities do not yet have a DIFT policy solution at all.

DIFT implementations themselves also have significant drawbacks. Software approaches are either limited to a single language or rely on dynamic binary translation, and thus have unacceptable performance overheads. Hardware DIFT platforms are fast, but support only a single, fixed buffer overflow policy which has been shown to have serious false positives and negatives in practice.

Chapter 4

Raksha: A Flexible Hardware Platform for DIFT

This chapter presents the design and implementation of *Raksha*, a flexible hardware platform for dynamic information flow tracking. Raksha is the first hardware DIFT architecture to offer flexible, programmable security policies, the first to provide whole system security, protecting even the operating system from malicious attacks, and the first (and to our knowledge only) to be evaluated using an FPGA-based prototype.

4.1 Motivation

Existing research has demonstrated the potential of DIFT, and the benefits and weaknesses of hardware and software DIFT implementations. Software implementations typically rely on dynamic binary translation, allowing for very flexible policies that do not require changes to existing hardware at the expense of performance and compatibility with legacy code techniques such as multithreading and self-modifying code. Hardware provides excellent performance and complete compatibility with legacy code, but provides only brittle, hard-coded DIFT policies.

We make the case for a flexible DIFT architecture that allows the best of both hardware and software techniques. Specifically, we argue that hardware should provide a few key mechanisms on which software builds in order to create efficient, flexible systems that protect against a wide range of attacks. This approach allows us to combine the best of traditional hardware and software DIFT systems.

4.1.1 The Case for Hardware DIFT

Hardware is an appropriate and natural layer of abstraction for dynamic information flow tracking. Many DIFT policies are applied to binary executables, or even to the entire system (userspace and OS). This is often beneficial for attack prevention because all languages are eventually translated to assembly instructions, and thus binary-level DIFT policies can protect any language, from C to PHP. When protecting binaries or the entire system, hardware possesses unique advantages over a software-only approach.

Hardware provides for low-overhead, whole-system DIFT implementations. Software DIFT approaches often result in significant slowdown. Even when only protecting a single application (which does not require DIFT support in the OS), software DIFT overheads range from 3x [100] to 37x [76]. Hardware approaches allow for negligible overhead, even when protecting all applications and the operating system. This is because hardware can perform DIFT operations in parallel with normal instruction execution.

Often, DIFT policies such as buffer overflow protection may be applied to an entire system – all applications and the operating system. Whole-system DIFT is slow when implemented in software, requiring all devices, the MMU, the OS, and all applications to be virtualized. Existing whole-system software dynamic binary translators such as QEMU [5] have significantly higher performance overhead than application-level DBTs. For example, QEMU's slowdown ranges from 5x to 20x without any DIFT support whatsoever [98]. Hardware can apply DIFT policies to all applications and the operating system without the complexity and performance overhead of whole-system dynamic binary translation.

Hardware also has unique correctness and safety benefits for multithreaded applications. In a software DIFT implementation, updates to a word of memory are no longer atomic as they are broken into two non-contiguous store instructions: one to update the data, and the other to update the metadata (DIFT tag). Multithreaded software DIFT implementations may read stale or incorrect tag values if another thread has executed only one of the two stores, resulting in false positives or negatives. Furthermore, splitting a memory word update into two instructions violates the memory consistency models of most architectures, including the Intel x86, which guarantee the atomicity of aligned memory stores. Thus, software DIFT implementations may break legitimate high-performance code. These safety issues could be mostly addressed in software if the underlying ISA provided an atomic double-compare-and-swap instruction, but the performance impact of replacing every load and store with a double-compare-and-swap would be prohibitively expensive. Furthermore, there would still be no way to provide tag safety guarantees for applications that used the atomic double-compare-and-swap instruction as a DIFT system would need to perform a quadruple-compare-and-swap to simultaneously update the two words of data and the two tags.

To safely address these issues, software DIFT approaches forgo support for multiple threads or processors entirely [100, 5], or restrict applications so that only a single thread is executed at a time [72]. This significantly degrades application scalability and performance. Hardware DIFT platforms can modify the memory coherency and consistency logic to ensure that tag updates are always atomic [15, 48, 131].

4.1.2 The Case for Software

Software DIFT has a number of compelling advantages over existing hardware DIFT approaches. Existing software DIFT platforms [76, 100, 20] support arbitrarily flexible DIFT policies, and could even allow for multiple concurrently active DIFT policies. In contrast, existing hardware platforms [121, 12] support only a single hardcoded policy. Software DIFT policies can be easily updated for application-specific issues or to handle new attacks or vulnerabilities, while hardware policies are immutably fixed into silicon. Software DIFT platforms also require no hardware modifications, and thus can work on existing systems today.

4.2 DIFT Design Requirements

There is a clear need for a DIFT platform that provides the best of both worlds, combining the strengths of hardware and software DIFT approaches. An ideal DIFT design would combine the performance and legacy code compatibility of hardware with the flexibility and malleability of a software DIFT implementation. In this discussion we discuss the critical elements of DIFT design: tag storage, tag policy execution, tag memory model, and tag exceptions. For each of these issues, we describe the issue, and discuss the strengths and weakness of both hardware and software solutions.

4.2.1 Tag Management

A DIFT implementation must provide support for tagged registers and memory. High-speed tag support is critical, as tag checks and propagation may occur during the execution of most application instructions. Even a simple instruction may result in multiple tag reads and at least one tag write. Applications may also desire flexible tag formats, both to support multiple policies and to support a single policy that requires more than one tag bit.

Software-managed tags must be allocated from existing DRAM or registers, and are managed by additional software instructions inserted by a dynamic binary translator. Software tags thus compete with the application for scarce register and memory resources. Furthermore, the instructions inserted to manage these tags result in significant performance degradation at runtime. However, software designs are flexible and may support variable-length or enormous tags [11].

Hardware DIFT implementations implement tags entirely in hardware, with little or no runtime overhead. Traditional implementations may extend each register, cache line, and DRAM word directly with additional tag bits [24, 121, 12]. Register tags have also been implemented on a coprocessor [49], or on another core entirely [11]. To reduce the cost of hardware DIFT, researchers have proposed implementing hardware memory tags in a multi-granular fashion, with both per-page and per-word tags [121, 131]. This approach takes advantage of the extremely high spatial locality commonly observed in DIFT tags when performing common analyses such as taint tracking. Most hardware DIFT implementations use a single tag bit per byte or word.

An ideal solution would provide the performance of managing tags in hardware, while also providing a sufficiently flexible tag format to protect against all of the vulnerabilities described in Chapter 2. Fortunately, we have found that these vulnerabilities can be prevented with a tag policy that requires only one or two tag bits. As described in Section 5.4.4, a hardware DIFT architecture with support for four independently managed tag bits can simultaneously protect against all of these vulnerabilities. Fixed-length four bit tags are small enough to be practically achieved in hardware without the significant performance and design complexity costs associated with huge variable-length tags, while still providing sufficient flexibility to prevent the major server-side vulnerabilities.

While there are non-security applications such as lockset-based race detection [11, 109] that still require enormous tags, these analyses are not used to prevent security vulnerabilities in production servers. Even these applications may be supported by some hardware-managed tag designs. Recent research has also shown that with sufficiently large fixed-length hardware-managed tags, it is possible to support variable-length or enormous tags by adapting techniques from Read-Copy-Update (RCU) algorithms in the Linux kernel [48].

4.3 Tag Policy Execution

Tag policies consist of tag check and tag propagation rules. Tag propagation defines how tag values flow from source operands to destination operands when an instruction is executed. For example, a tag propagation rule may specify that an addition instruction propagates the union of its source operand tags to the destination operand tag. Tag checks restrict the operations that may be performed on tagged data, such as by forbidding tainted code from being executed.

Software DIFT implementations execute tag policies by inserting tag propagate and check instructions into the translated application code at runtime. This results in significant runtime overhead, but allows for arbitrary tag check and propagate policies. Software DIFT policies may be easily updated, can protect against application-specific vulnerabilities, or even adapt to compatibility issues with particular applications.

Existing hardware solutions execute tag policies entirely in hardware, and have little or no performance overhead. However, the designs fix the entire tag policy into hardware, rendering it immutable. Most hardware implementations perform tag checks and propagation in the CPU pipeline in parallel with instruction execution [121, 24]. Other hardware proposals proposals execute a policy on a coprocessor [49], or on another core in a multi-core system [11]. While hardware policy execution may have little or no overhead, existing solutions are fixed and brittle, protecting only against buffer overflow attacks [121, 12, 20].

A hardcoded policy is not sufficient for a robust security system. DIFT can prevent a wide range of vulnerabilities, and many of these vulnerabilities have no easy hardcoded solution. High-level attacks such as web vulnerabilities require tag management policies that are significantly different from memory corruption attacks. To prevent SQL injection attacks, for example, we must verify that any query passed to the SQL server does not contain tagged command characters. Unlike the rules for memory corruption that untaint tags on certain validation instructions, the SQL injection rules have no simple, single-instruction validation policy. The tag check rules are different as well. For SQL injection, we raise an exception to intercept the call to the SQL query execute function so that the SQL query can be checked for tainted command characters. SQL query checks constitute a complex, high-level operation that can be done only in software as they depend on the SQL grammar of the database server. Section 4.7.5 further describes DIFT SQL injection protection.

An ideal DIFT implementation would provide both speed and flexibility. Speed requires a hardware-based approach, but existing hardware designs are inflexible. A practical DIFT hardware design should have support for flexible and programmable policies that are executed by hardware. Hardware designers should provide a high-speed policy execution mechanism, while software should specify the policies themselves. Software must have fine-grain control over tag propagation and check rules to address an evolving set of attacks and to support the intricacies of real-world software. This allows for more robust policies that prevent a wider range of attacks, and also amortizes the cost of the initial hardware investment over a diverse range of software-controlled policies.

4.4 Tag Memory Model

As dicussed in Section 4.1.1, multithreaded applications present a unique challenge for DIFT systems. This is because in a DIFT system, updates to a word of data may be broken into two operations: the update to the data itself, and the update to its corresponding tag.

If the tag and data update do not occur in a single atomic action, a race condition is introduced, compromising the correctness of the DIFT system. Software DIFT implementations resolve this issue by executing only a single thread at a time. Given the prevalence of multicore architectures, this severely restricts application performance and scalability. These drawbacks will only worsen over time as CPU manufacturers increase the number of cores in their designs. Hardware support is necessary to ensure tag coherency and consistency, so that when data is written, any updates to that data's tag occur in a single atomic action along with the data update [131, 15, 48].

An ideal DIFT system must provide a tag memory model with atomic tag and data updates, which requires hardware modifications in most cases. This can be achieved by tracking data updates to ensure that tag updates occur in the same order [48, 131], or by leveraging existing concurrency primitives such as transactional memory [15]. Without hardware support, applications must choose between performance or correctness and safety. All of these characteristics must be present for it to be practical to enable DIFT in production systems.

4.5 Tag Exceptions

Existing hardware DIFT architectures assume that hardware alone can fully identify unsafe uses of tagged data. Hence, tag exceptions simply trap into the OS and terminate the application. The exception overhead is not significant. Software DIFT architectures take a more flexible approach, and may support complicated analyses with exception handlers that can be dynamically defined by plugins [16].

Looking forward, it is more realistic to expect that a hybrid approach will be taken where DIFT hardware will play a key role in identifying potential threats for which further software analysis is needed to detect an actual exploit. For example, when preventing SQL injection the hardware should track tags of user input as the database query is constructed. It will be up to software to determine if the query string contains malicious, tainted command characters, or tainted, yet harmless data. Similarly, a memory corruption policy may use software to correctly determine the code patterns that constitute input data validation.

Enforcing security policies partially in software places an emphasis on the overhead of security exceptions. OS traps cost hundreds to thousands of cycles. Hence, even infrequent security exceptions can have a large impact on application performance. System operators should not have to choose between security and performance.

Furthermore, existing DIFT systems cannot protect the OS code because the OS already runs at the highest privilege level. Hence, it is difficult to protect the security exception handler from a potentially compromised OS component. We view this as a significant shortcoming of DIFT architectures given that a successful attack against the OS can compromise the whole system. Remotely exploitable kernel vulnerabilities occur even in high-security operating systems such as OpenBSD [80], or the popular Windows operating system [133]. Moreover, OS vulnerabilities often take weeks for vendors to patch after they are publicized.

Rather than result in expensive OS traps that require privilege level changes, we believe an ideal DIFT system should support *user-level exceptions*, which transfer control to an exception handler in the same address space at the same privilege level. Tag exceptions should be as expensive as a function call as they result in an unconditional control flow transfer within the same address space without incurring the cost of a privilege level transition. By keeping the overhead of exceptions low, more security functionality can be performed flexibly in software rather than hardcoded into a hardware policy.

User-level exceptions require a mechanism to protect the handler from other code running in the same address space and privilege level. The same mechanism can protect the security handler from other OS components. A portion of the OS must still be trusted, including the security handlers themselves and the code that manages them. Still, we can check large portions of the OS, including the device drivers that are common targets of security attacks [122, 14].

4.6 Raksha Overview

We propose a novel DIFT hardware architecture, Raksha, which extends existing processors with flexible support for dynamic information flow tracking. In this section, we discuss

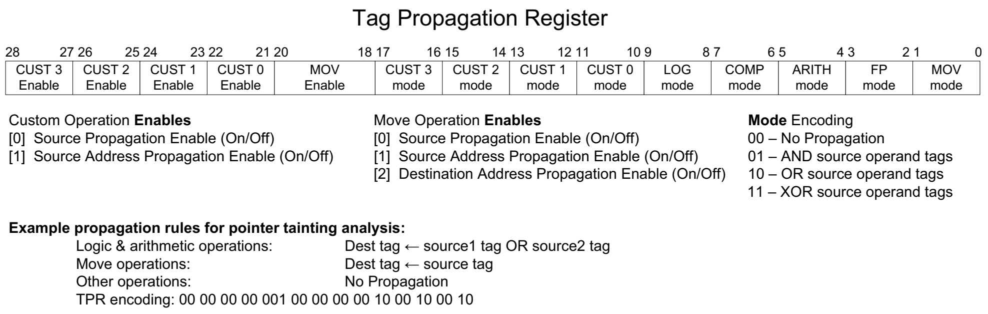

Tag Propagation Register
28 27 26 25 24 23 22 21 20 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
CUST 3 Enable
CUST 2 Enable
CUST 1 Enable
CUST 0 Enable
MOV Enable
CUST 3 mode
CUST 2 mode
CUST 1 mode
CUST 0 mode
LOG mode
COMP mode
ARITH mode
FP mode
MOV mode
Custom Operation **Enables**
[0] Source Propagation Enable (On/Off)
[1] Source Address Propagation Enable (On/Off)
Move Operation **Enables**
[0] Source Propagation Enable (On/Off)
[1] Source Address Propagation Enable (On/Off)
[2] Destination Address Propagation Enable (On/Off)
Mode Encoding
00 – No Propagation
01 – AND source operand tags
10 – OR source operand tags
11 – XOR source operand tags
Example propagation rules for pointer tainting analysis:
Logic & arithmetic operations: Dest tag ← source1 tag OR source2 tag
Move operations: Dest tag ← source tag
Other operations: No Propagation
TPR encoding: 00 00 00 00 001 00 00 00 00 10 00 10 00 10

Figure 4.1: The format of the Tag Propagation Register (TPR). There are 4 TPRs, one per active security policy.

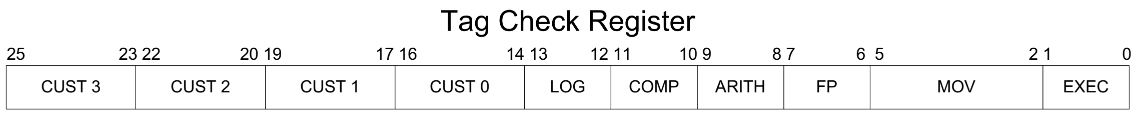
the four key characteristics of Raksha's design: hardware tag management, flexible security policies, multiple active policies, and low-overhead security exceptions. While the implementation discussion focuses on SPARC, Raksha's design principles are applicable to other, non-RISC architectures such as the Intel x86.

4.6.1 Hardware Tag Management

Raksha is a traditional in-core hardware DIFT design, and follows the general model of previous hardware DIFT systems [121, 20, 12]. All storage locations, including registers, caches, and main memory, are extended by tag bits. We chose this implementation strategy to minimize the complexity of our hardware design. Raksha also could be used in more aggressive coprocessor-based designs [49], or in designs with multi-granular, page-level DRAM tags [121]. Raksha supports 4-bit tags per 32-bit register or word of memory.

4.6.2 Flexible Hardware DIFT Policies

Raksha performs tag propagation and checks transparently for all instructions. The exact rules for tag propagation and checks are specified by a set of *tag propagation registers (TPR)* and *tag check registers (TCR)*. There is one TCR/TPR pair for each of the four security policies supported by Raksha. Figures 4.1 and 4.2 present the format for the two registers as well as an example configuration for a buffer overflow protection policy. Software configures the TCR and TPR registers to control the DIFT policy enforced by hardware.



Predefined Operation **Enables**
[0] Source Check Enable (On/Off)
[1] Destination Check Enable (On/Off)

Custom Operation **Enables**
[0] Source 1 Check Enable (On/Off)
[1] Source 2 Check Enable (On/Off)
[2] Destination Check Enable (On/Off)

Execute Operation **Enables**
[0] PC Check Enable (On/Off)
[1] Instruction Check Enable (On/Off)

Move Operation E**nables**
[0] Source Check Enable (On/Off)
[1] Source Address Check Enable (On/Off)
[2] Destination Address Check Enable (On/Off)
[3] Destination Check Enable (On/Off)

Example check rules for pointer tainting analysis:

Execute operations (PC, Instruction):	On
Comparison operations (Sources only):	On
Move operations (Source & Dest addresses):	On
Custom operation 0:	On (for AND instruction, sources only)
Other operations:	Off

TCR encoding: 000 000 000 011 00 01 00 00 0110 11

Figure 4.2: The format of the Tag Check Register (TCR). There are 4 TCRs, one per active security policy.

To balance flexibility and compactness, TPRs and TCRs specify rules at the granularity of *primitive operation classes*. The classes are *floating point*, *move*, *integer arithmetic*, *comparisons*, and *logical*. The move class includes register-to-register moves, loads, stores, and jumps (move to program counter). To track information flow with high precision, we do not assign each ISA instruction to a single class. Instead, each instruction is decomposed into one or more primitive operations according to its semantics. For example, the `subcc` SPARC instruction is decomposed into two operations, a subtraction (arithmetic class) and a comparison that sets a condition code. As the instruction is executed, we apply the tag rules for both arithmetic and comparison operations. This approach is particularly important for ISAs that include CISC-style instructions, such as the x86. It also reflects a basic design principle of Raksha: information flow analysis tracks basic data operations, regardless of how these operations are packaged into ISA instructions.

Previous DIFT systems define tag policies at the granularity of ISA instructions, which creates several opportunities for false positives and false negatives. CISC architectures such as the Intel x86 support thousands of instructions, many of which are equivalent from a DIFT perspective. Some instructions are also extremely complicated, such as the string copy instruction. Primitive operation decomposition allow us to address both of these cases, so that we can succinctly specify policy rules for very similar instructions while decomposing complex instructions into a sequence of DIFT operations. Simple or complex, all ISA instructions are broken down into a sequence of primitive operations as determined by the instruction semantics.

To handle corner cases such as clearing a register with an `xor` instruction, TPRs and TCRs can also specify rules for up to four *custom operations*. Custom operations allow instruction-specific propagation and check rules. As an instruction is decoded, we compare its opcode to the opcode defined by software in the four custom operation registers. If the opcode matches that of a custom operation, we use the corresponding custom rules for propagation and checks instead of the the generic rules for the instruction's primitive operation(s).

As shown in Figure 4.1, each TPR uses a series of two-bit fields to describe the propagation rule for each primitive class and custom operation (bits 0 to 17). Each field indicates if there is propagation from source to destination tags and if multiple source tags are combined using logical AND, OR, or XOR. Bits 18 to 26 contain fields that provide source operand selection for tag propagation for move and custom operations. For move operations, we can propagate tags from the source, source address, and destination address operands. The load instruction `ld [r2], r1`, for example, considers register `r2` as the source address, and the memory location referenced by `r2` as the source.

As shown in Figure 4.2, each TCR uses a series of fields that specify which operands of a primitive operation class or custom operation should be checked for a tag exception. If a check is enabled and the tag bit of the corresponding operand is set, a security exception is raised. For most operation classes, there are three operands to consider but for moves (loads and stores) we must also consider source and destination addresses. Each TCR includes an additional operation class named *execute* for detecting security attacks on code or code pointers. This class specifies the rule for tag checks on instruction fetches. We can choose to raise a security exception if the fetched instruction is tagged or if the program counter is tagged. The former occurs when executing tainted code, such as when an attacker injects malicious code into a process. The latter can happen when a jump instruction propagates an input tag to the program counter, such as when a return instruction is executed and the stack return address has been overwritten with malicious data.

4.6.3 Multiple Active Security Policies

Raksha supports multiple active security policies. Specifically, Raksha supports per-word 4-bit tags, and each tag is independently controlled by a pair of tag check and propagate registers. Policies are encoded by writing the appropriate values to the check and propagation registers that control a particular tag bit. Chapter 5 discusses the only policy that spans multiple tag bits, which is a buffer overflow policy that performs a tag check on two tag bits and raises a tag exception if a particular combination of bits is set. In all other cases, tag checks and propagation occur separately for each of the four tag bits.

As indicated by the popularity of ECC codes, 4 extra bits per 32-bit word is an acceptable overhead for additional reliability. The choice of four tag bits per word was motivated by the number of security policies used to protect against a diverse set of attacks with the Raksha prototype (see Section 4.9). Even if future experiments show that a different number of active policies is needed, the basic mechanisms described in this paper will apply.

4.6.4 User-level Security Exceptions

A security exception occurs when a TCR-controlled tag check fails for the current instruction. Security exceptions are *precise* in Raksha. When the exception occurs, the offending instruction is not committed. Instead, exception information is saved to a special set of registers for subsequent processing (PC, failing operand, which tag policies failed, etc) and control is transferred to the tag exception security handler.

Raksha supports user-level handling of security exceptions. Hence, the exception overhead is similar to that of a function call rather than the overhead of a full OS trap. Two hardware mechanisms are necessary to support user-level exception handling. First, the processor has an additional *trusted mode* that is orthogonal to the conventional user and kernel mode privilege levels. Software can directly access the tags or the policy configuration registers only when trusted mode is enabled [1]. Raksha provides extra instructions to access this additional state when in trusted mode. Tag propagation and checks are also disabled when in trusted mode. Secondly, a hardware register is added to provide the address

[1]Conventional code running outside the trusted mode can implicitly operate on tags via tag propagation but any explicit access is forbidden.

for a *predefined security handler* to be invoked on a tag exception. When a tag exception is raised, the processor automatically switches to the trusted mode and transfers control to the security handler, but remains in the same user/kernel mode and the same address space.

The predefined address for the exception handler is available in a special register that can be updated only while in trusted mode. At the beginning of each program, the exception handler address is initialized by trusted code in the operating system before control is passed to the application. The application cannot change the exception handler address because it runs in untrusted mode.

The exception handler can include arbitrary software that processes the security exception. It may summarily terminate the compromised application or simply clean up and ignore the exception. It may also perform a complex analysis to determine whether the exception is a false positive, or may try to address the security issue without terminating the code. The handler overhead depends on the complexity of the processing performed by the handler code.

Since the exception handler and applications run at the same privilege level and in the same address space, there is a need for a mechanism that protects the handler code and data from a compromised application. Unlike the handler, user code runs only in untrusted mode and is forbidden from using the additional instructions that manipulate tag registers or directly access tags themselves. Still, a malicious application could overwrite the code or data belonging to the handler. To prevent this, we use one of the four security policies to sandbox the handler's data and code. This policy is described fully in Section 4.7.3. We set the sandbox tag bit for every memory location used by the security handler, code or data. The TCR is configured so that any instruction fetch or data load/store to locations with this tag bit set will generate an exception. This sandboxing approach provides efficient protection without requiring different privilege levels. Hence, it can also be used to protect the trusted portion of the OS from the untrusted portion. We can also re-use the sandboxing policy to implement the function call or system call interposition functionality required to detect certain high-level attacks.

4.6.5 Design Discussion

Raksha defines tag bits per 32-bit word instead of per byte. We find the overhead of per-byte tags unnecessary in many cases. Many policies, such as buffer overflow protection, protect 32-bit aligned values such as pointers. Considering the way compilers allocate variables, it is unlikely that two variables with dramatically different security characteristics will be packed into a single word. The one exception we found to this rule so far is that some applications construct strings by concatenating untrusted and trusted information. Infrequently, this results in a word with both trusted and untrusted bytes. These infrequently occurring cases could be handled in software using low-overhead exceptions, rather than increasing tag storage overhead by four-fold.

To ensure that subword accesses do not introduce false negatives, we check the tag bit for the whole word even if a subset is read. For tag propagation on subword writes, we use a control register to allow software to select a method for merging the existing tag with the new one (*and*, *or*, *overwrite*, or *preserve*). As always, it is best for hardware to use a conservative policy and rely on software analysis within the exception handler to filter out the rare false positives due to subword accesses. We would use the same approach to implement Raksha on ISAs that support unaligned accesses that span multiple words. Alternatively, the security handler could manage fine-grained, byte-level tags purely in software, although evaluating the performance overhead of this approach is outside of the scope of this work.

Raksha can be combined with any base instruction set. For a given ISA, we decompose each instruction into its primitive operations and apply the proper check and propagate rules. This is a powerful mechanism that can cover both RISC and CISC architectures. For simple instructions, hardware can perform the decomposition during instruction decoding. For the most complex CISC instructions, it is best to perform the decomposition using a micro-coding approach, as is often done for instruction decoding purposes on the Intel x86 and other CISC ISCAs. Raksha can handle instruction sets with condition code registers or other special registers by properly tagging these registers in the same manner as general purpose registers.

Policy	Tag Initialization	Propagation Rule	Check Rule
Simple Tainting	Untrusted input	Move (source only) Integer Arithmetic Logical	–
Pointer Tainting	Untrusted input	Move (source only) Integer Arithmetic Logical	Move (address) Comparison (source) Program Counter Instruction AND Custom op (source)
System Call Interposition	Trap base register	–	Move (source)
Function Call Interposition	Function entry point	–	Tagged instruction
Fault Isolation	Sandboxed memory	–	Tagged instruction Move (source) Move (destination)

Table 4.1: The tag initialization, propagation, and check rules for the security policies used by Raksha. The propagation rules identify the operation classes that propagate tags. The check rules specify the operation classes that raise exceptions on tagged operands. When necessary, we identify the specific operands involved in propagation or checking.

The operating system can interrupt and switch out an application that is currently in a security handler. As the OS saves/restores the process context, it also saves the trusted mode status. It must also save and restore the tag registers introduced by Raksha as if they were user-level registers. When the application resumes, its security handler will continue.

Like most other DIFT architectures, Raksha does not track implicit information flow since it would cause a large number of false positives. Implicit information flow is of particular concern to high-assurance systems that must prevent malicious code from exfiltrating data. Raksha's focus is preventing malicious attacks from compromising vulnerable, but non-malicious applications. Security exploits typically rely only on tainted code or data that is explicitly propagated through the system, and do not use implicit information flow to propagate exploit payloads.

4.7 Policies

Raksha's flexible design allows for a wide variety of DIFT policies. By utilizing dynamic information flow tracking, Raksha can prevent security attacks comprehensively.

In this section, we describe the DIFT policies presented in Table 4.1. Each of these policies can be implemented on Raksha, and used to prevent security attacks. For each policy, we describe the policy itself and discuss how the policy prevents security attacks on unmodified binaries.

4.7.1 Pointer Tainting

The pointer tainting policy protects against memory corruption attacks, similar to the hard-coded protection provided by previous DIFT systems [121, 20, 12]. This policy prohibits tagged information from being used as a load address, store address, jump address, or instruction. A standard propagation policy is used, propagating on all common operations.

The buffer overflow policy recognizes and untaints values when a recognized bounds check operation occurs. We invoke a software security handler to identify bounds checks for comparison or logical AND instructions. A bounds check occurs when tainted information is compared to untainted information, or an AND is performed with tainted information and an untainted value that is a power of two minus one. The latter case is commonly used as an optimized form of bounds checking when accessing an array with a power-of-two size.

If a bounds check is detected, we clear the associated tags. Hence, untrusted data can be used as an array index only if it has been properly validated by a bounds check. Previous DIFT architectures simply hardcoded the policy that any comparison between a tagged and an untagged operand validates the tagged operand. This avoids the overhead of invoking the software handler through an OS trap but can lead to both false negatives and false positives.

Unlike our other policies discussed in this chapter, we encountered real-world false positives when applying this policy to applications. The false positives are discussed in Sections 4.9.3 and 5.1. These issues motivated our novel buffer overflow prevention policy described in Chapter 5, which has no observed real-world false positives.

4.7.2 Simple Tainting

The simple tainting policy is used in conjunction with other policies (system call and function call interposition) to prevent security attacks. This policy tracks the flow of untrusted information, with no required tag checks and no recognized untainting or validation operations. Other policies then use this taint flow information to make policy-specific decisions when a security-critical system call or function call occurs. This policy does not actually result in the entire address space becoming tainted over time as overwriting a tainted variable with untainted information will still clear the affected variable's tag.

4.7.3 Sandboxing

To protect the security handler from malicious attacks, we use a fault-isolation tag policy that implements sandboxing. The handler code and data are tagged with a sandboxing bit, and a rule is specified that generates an exception if sandboxed memory is accessed outside of trusted mode. Untrusted code is forbidden from loading, storing, or executing from any register or memory location with the sandboxing tag bit set. This policy ensures handler integrity even during a memory corruption attack on the application. The sandboxing tag bit can be re-used to support the system call and function call interposition policies.

4.7.4 System Call Interposition

The system call interposition policy used by Raksha restricts accesses to system calls by setting the sandboxing bit on the Trap Base Register, which is used to execute all system calls. Any attempt to read the trap base register by executing a system call instruction results in a tag exception, allowing the security monitor to ensure that the system call is not malicious. This policy may use the same sandboxing bit as the sandboxing policy described in Section 4.7.3. Alternatively, a software implementation could modify the operating system to perform system call interposition, as described in [37, 38].

To evaluate the safety of a system call, the security monitor must examine the system call arguments and their associated taint information. Thus, system call interposition requires the simple tainting policy described in Section 4.7.2 to supply taint information for the system call arguments.

This policy can be used to prevent a wide range of attacks. A security monitor that checks for tainted '..' characters in a filename when performing filesystem operations such as $open$ will prevent directory traversal attacks. Similarly, the security monitor can ensure that the program name argument to the $execve$ system call is untainted to prevent untrusted input from resulting in a command injection attack.

Cross-site scripting attacks can be prevented by monitoring the system calls used by web servers to send HTTP replies. If a web server sends a reply back to the client (using the $sendmsg$ or another related system call), the security monitor can examine the reply before allowing the system call to execute. The security monitor can scan the HTML output to ensure that there are no tainted bytes that contain malicious $< script >$ tags or other unsafe HTML tags.

4.7.5 Function Call Interposition

The function call interposition policy used by Raksha sets the sandboxing bit at the entry point for all functions that must be interposed on by the security monitor. This sandboxing bit may be the same bit used by the sandboxing policy described in Section 4.7.3. Function call interposition policies must examine the taint values of the function arguments and thus rely on on the simple tainting policy described in Section 4.7.2 for effective taint tracking.

When a monitored function is called, a security exception occurs because the first instruction of function has a sandboxing bit set. The security monitor then examines the function arguments on the stack, as well as their associated taint information, and decides whether a security attack is occurring. If an attack is not underway, the security monitor returns to the entry point of the monitored function using a special variant of the tag monitor return instruction. This instruction suppresses tag exceptions for the instruction that is the target of the tag return instruction, allowing the entry point of the sandboxed function to be executed without resulting in an infinite loop.

This policy can be used to prevent a number of common library-level attacks. Format string vulnerabilities can be prevented by interposing on all library calls to the *printf()* family - *vfprintf*, *fprintf*, {emphprintf, etc. The security monitor will scan the printf format string, ensuring that no format string specifier has its taint bit set. This prevents all format string attacks, as format string attacks occur when user input is used as a format string specifier to a printf-style function.

SQL injection attacks can also be prevented using this policy. SQL vendors often supply client libraries for accessing SQL databases, such as libpq for PostgreSQL or libmysqlcient for MySQL. By interposing on the library calls used to execute SQL queries, we can examine the queries and their taint information before allowing a query to be sent to the database server. SQL injection is defined as an attack occurring when user input influences the parse tree of a SQL query [119]. The security monitor can ensure that tainted information in a SQL query does not change the parse tree of the query before allowing the query to be sent to the database, thus preventing SQL injection attacks comprehensively.

4.7.6 Policy Configuration

We can have all the analyses in Table 4.3 concurrently active using 3 of the 4 tag bits available in Raksha: one for string tainting, one for pointer tainting, and one for sandboxing and function/system call interposition. The fourth tag bit is used in the advanced buffer overflow policy discussed in Chapter 5. The four tag bits supported by Raksha allow us to enable all evaluated DIFT security policies, providing comprehensive protection against low-level and high-level vulnerabilities.

4.8 The Raksha Prototype System

To evaluate Raksha, we developed a prototype system based on the SPARC architecture. Previous DIFT systems used a functional model like Bochs to evaluate security issues and a separate performance model like Simplescalar to evaluate overhead issues with user-only code [121, 20, 12]. Instead, we use a single prototype to provide both functional and performance analysis. Hence, we can get a performance measurement for *any* real-world

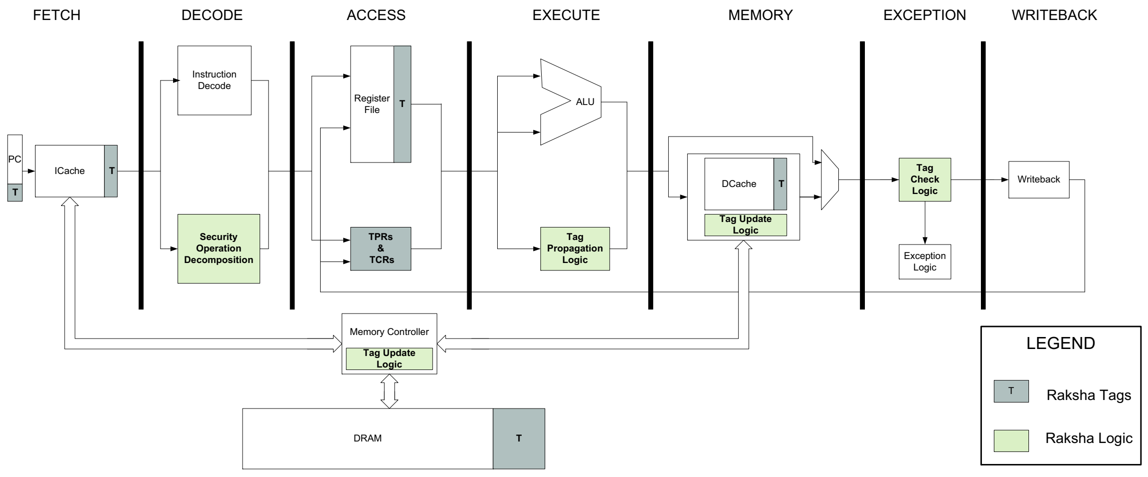


Figure 4.3: The Raksha CPU pipeline.

application that we study for security purposes. Moreover, we can use a single platform to evaluate performance and security issues related to the operating system and the interaction between multiple processes (e.g., a web server and a database).

The Raksha prototype is based on the Leon SPARC V8 processor, a 32-bit open-source synthesizable core developed by Gaisler Research [58]. We modified Leon to include the security features of Raksha and mapped the design onto an FPGA board. The resulting system is a full-featured SPARC Linux workstation. To the best of our knowledge, Raksha is the first and only hardware DIFT platform to have an FPGA-based prototype implementation.

The Leon fully implements the SPARC V8 standard [117]. The SPARC ISA is similar to other RISC ISAs such as 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. The idiosyncrasies of the SPARC ISA did not affect our DIFT policies, as we decomposed all instructions into a sequence of primitive operations for the purpose of determining tag check and propagate policies, and ensured that all registers and memory locations were extended with support for tags.

4.8.1 Hardware Implementation

Figure 4.3 shows a simplified diagram of the Raksha hardware, focusing on the processor pipeline. Leon uses a single-issue, 7-stage pipeline. We modified its RTL code to add 4-bit tags to all user-visible registers, as well as cache and memory locations; introduced the configuration and exception registers defined by Raksha; and added the instructions that manipulate special registers or provide direct access to tags in the trusted mode. Overall, we added 9 instructions and 16 registers to the SPARC V8 ISA. We also added support for the low-overhead security exceptions and extended all buses to accommodate tag transfers in parallel with the associated data.

The processor operates on tags as instructions flow through its pipeline as directed by the policy configuration registers (TCRs and TPRs). The Fetch stage checks the program counter tag and the tag of the instruction fetched from the I-cache. The Decode stage decomposes each instruction into its primitive operations and checks if its opcode matches any of the custom operations. The Access stage reads the tags for the instruction operands from the register file, including the destination operand. It also reads the TCRs and TPRs. By the end of this stage, we know the exact tag propagation and check rules to apply for this instruction. Note that the security rules applied for each of the four tag bits are independent of one another. The Execute and Memory stages propagate source tags to the destination tag in accordance with the active policies. The Exception stage performs any necessary tag checks and raises a precise security exception if needed. All state updates (registers, configuration registers, etc.) are performed in the Writeback stage. Pipeline forwarding for the tag bits 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 4 tag bits per 32-bit word. We also reserved a portion of main memory for tag storage and modified the memory controller to properly access both data and tags on cached and uncached requests. This approach introduces a 12.5% overhead in the memory system for tag storage. On a board with support for ECC DRAM, we could use the 4 bits per 32-bit word available to the ECC code to store the Raksha tags. For future versions of the prototype, we plan to implement the multi-granular tag storage approach proposed by

Parameter	**Specification**
Pipeline depth	7 stages
Register windows	8
Instruction cache	8 KB, 2-way set associative
Data cache	32 KB, 2-way set associative
Instruction TLB	8 entries, fully-associative
Data TLB	8 entries, fully-associative
Memory bus width	64 bits
Prototype Board	GR-CPCI-XC2V board
FPGA device	XC2VP6000
Memory	512MB SDRAM DIMM
I/O	100Mb Ethernet MAC
Clock frequency	20 MHz
Block RAM utilization	22% (32 out of 144)
4-input LUT utilization	42% (28,897 out of 67,584)
Total gate count	2,405,334
Gate count increase over base Leon	7.17%

Table 4.2: The architectural and design parameters for the Raksha prototype.

Suh *et al* [121], where tags are allocated on demand for cache lines and memory pages that actually have tagged data. This will significantly reduce tag storage overhead on most workloads, as tags have very high spatial locality in practice.

We synthesized Raksha on the Pender GR-CPCI-XC2V Compact PCI board which contains a Xilinx XC2VP6000 FPGA. Table 4.2 summarizes the basic board and design statistics, including the utilization of the FPGA resources. Since Leon uses a write-through, no-write-allocate data cache, we had to modify its design to perform a read-modify-write access on the tag bits in the case of a write miss. This change and its small impact on application performance would not have been necessary had we started with a write-back cache. There was no other impact on the processor performance, as tags are processed in parallel and independently from the data in all pipeline stages. A more complete discussion of Raksha's hardware implementation can be found in [47].

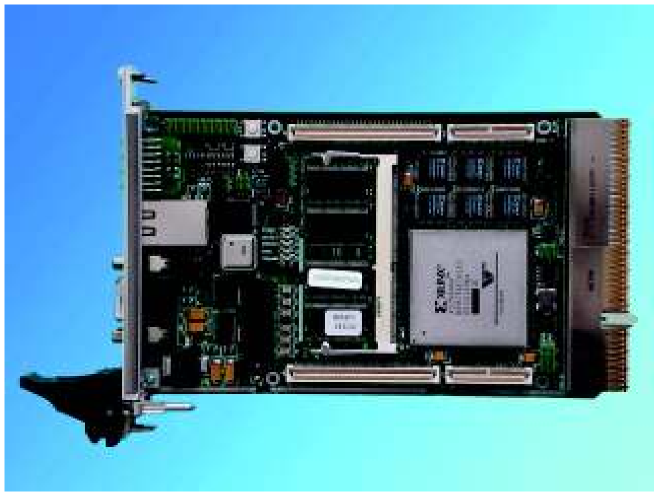


Figure 4.4: The GR-CPCI-XC2V board used for the prototype Raksha system.

Security features are trustworthy only if they have been thoroughly validated. Similar to other ISA extensions, the Raksha security mechanisms define a relatively narrow hardware interface that can be validated using a collection of directed and randomly generated test cases that stress individual instructions and combinations of instructions, modes, and system states. The random test generator creates arbitrary SPARC programs with randomly generated tag policies. Periodically, test programs enable trusted mode and verify that any registers or memory locations modified since the last checkpoint have the expected tag and data values. The expected values are generated by a simple functional-only model of Raksha for SPARC written in C. If the validation fails, the test case halts with an error. The test case generator supports almost all SPARC V8 instructions. We have run tens of thousands of test cases and millions of instructions on the actual FPGA prototype and on the simulated RTL using a cluster of thirty machines.

4.8.2 Software Implementation

The Raksha prototype provides a full-fledged custom Linux distribution derived from Cross-Compiled Linux From Scratch [22]. The distribution is based on Linux kernel 2.6.11, GCC 4.0.2 and GNU C Library 2.3.6. It includes 120 software packages. Our distribution can bootstrap itself from source code and run unmodified enterprise applications such as

Apache, PostgreSQL, and OpenSSH. We created our own Linux distribution because no modern Linux distributions exist for a SPARC V8 processor with no floating point unit.

We have modified the Linux kernel to provide support for Raksha's security features. We ensure that the additional registers are saved and restored properly on context switches, system calls, and interrupts. Register tags must also be saved on signal delivery and SPARC register window overflows/underflows. Tags are properly copied when inter-process communication occurs, such as through pipes or when passing program arguments/environment variables to `execve`. Our FPGA prototype has no hard drive, so we boot Linux from a NFS root filesystem or via ATA over Ethernet [1].

Security handlers are implemented as shared libraries preloaded by the dynamic linker. The OS ensures that all memory tags are initialized to zero when pages are allocated and that all processes start in trusted mode with register tags cleared. The security handler initializes the policy configuration registers and initializes any non-zero register or memory tags before disabling the trusted mode and transferring control to the application. For best performance, the basic code for invoking and returning from a security handler have been written directly in SPARC assembly. The code for any additional software analyses invoked by the security handler can be written in any programming language.

Most security analyses require that tags be properly initialized or set when receiving data from input channels. We have implemented tag initialization within the security handler using the system call interposition tag policy discussed in Section 4.9. For example, a SQL injection analysis may wish to tag all data from the network. The reference handler would use system call interposition on the `recv`, `recvfrom`, and `read` system calls to intercept these system calls, and taint all data returned by them.

4.9 Evaluation

To evaluate the capabilities of Raksha's security features, we attempted a wide range of attacks on unmodified SPARC binaries for vulnerable real-world applications. Raksha successfully detected both high-level attacks and memory corruption exploits on these programs. We also evaluated the performance of Raksha, measuring the performance overhead of various tag policies on the SPEC CPU2000 benchmarks. This section presents our

Program	**Lang.**	**Attack**	**Analysis**	**Detected Vulnerability**
gzip	C	Directory traversal	Simple tainting System call	`Open` file with tainted absolute path
tar	C	Directory traversal	Simple tainting System call	`Open` file with tainted absolute path
Wabbit	PHP	Directory traversal	Simple tainting System call	`Open` file with tainted path outside web root
Scry	PHP	Cross-site scripting	Simple tainting System call	Tainted HTML output includes $< script >$
PhpSysInfo	PHP	Cross-site scripting	Simple tainting System call	Tainted HTML output includes $< script >$
htdig	C++	Cross-site scripting	Simple tainting System call	Tainted HTML output includes $< script >$
OpenSSH	C	Command injection	Simple tainting System call	`execve` tainted filename
ProFTPD	C	SQL injection	Simple tainting Function call	Unescaped tainted SQL query
traceroute	C	Double free	Pointer tainting	Use tainted data pointer
polymorph	C	Buffer overflow	Pointer tainting	Use tainted code pointer
SUS	C	Format string bug	Simple tainting Function call	`Syslog` tainted format string specifier
WU-FTPD	C	Format string bug	Simple tainting Function call	`Vfprintf` tainted format string specifier

Table 4.3: The security experiments performed with the Raksha prototype.

experimental results for the security and performance benchmarks. We also discuss the lessons learned in preventing high-level and low-level attacks using dynamic information flow tracking techniques.

4.9.1 Security Evaluation

Table 4.3 summarizes the security experiments we performed. They include attacks on basic system utilities (tar, gzip, polymorph, sus), network utilities (traceroute, openssh), servers (proftpd, wu-ftpd), Web applications (Scry, Wabbit, PhpSysInfo), and search engine software (htdig). A wide range of applications, programming languages, and vulnerability categories were chosen to demonstrate the flexibility and security of Raksha's DIFT policies. For each experiment, we list the programming language of the application, the type of attack, the DIFT analyses used for the detection, and the actual vulnerability detected by Raksha. All vulnerabilities in our experiments are real-world security flaws discovered in commonly used open source software.

Unlike previous DIFT architectures, Raksha does not have a fixed security policy. The four supported policies can be set to detect a wide range of attacks. Hence, Raksha can be programmed to detect high-level attacks like SQL injection, command injection, cross-site scripting, and directory traversals, as well as conventional memory corruption and format string attacks. The correct mix of policies can be determined on a per-application basis by the system operator. For example, a Web server might select SQL injection and cross-site scripting protection, while an SSH server would probably select pointer tainting and format string protection.

To the best of our knowledge, Raksha is the *first* DIFT architecture to demonstrate detection of high-level attacks on unmodified application binaries. This is a significant result because high-level attacks now account for the majority of software exploits [123]. All prior work on high-level attack detection required access to the application source code or Java bytecode [137, 77, 91, 63]. High-level attacks are particularly challenging because they are language and OS independent. Enforcing type safety cannot protect against these semantic attacks, which makes Java and PHP code as vulnerable as C and C++.

An additional observation from Table 4.3 is that, by tracking information flow at the level of primitive operations, Raksha provides attack detection in a language-independent manner. The same policies can be used regardless of the application's source language. For example, htdig (C++) and PhpSysInfo (PHP) use the same cross-site scripting policy, even though one is written in a low-level, compiled language and the other in a high-level, interpreted language. Raksha can also apply its security policies across multiple collaborating programs that have been written in different programming languages.

	Compare Filter		**AND Filter**		**Combined Filter**	
	Raksha	**OS**	**Raksha**	**OS**	**Raksha**	**OS**
bzip2	2.98x	13.20x	1.19x	1.75x	1.33x	2.80x
crafty	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x
gap	1.12x	1.70x	1.00x	1.01x	1.49x	4.04x
gcc	1.01x	1.04x	1.00x	1.00x	1.00x	1.03x
gzip	1.31x	2.92x	2.39x	7.20x	2.66x	9.97x
mcf	1.00x	1.04x	1.00x	1.00x	1.00x	1.00x
parser	1.04x	1.04x	1.24x	2.28x	1.07x	1.43x
twolf	1.58x	4.19x	1.19x	1.86x	1.85x	4.48x
vpr	1.00x	1.02x	1.00x	1.00x	1.00x	1.00x

Table 4.4: Performance slowdown for the SPEC benchmarks with a pointer tainting analysis that filters false positives by clearing tags for select compare and AND instructions. A slowdown of 1.34x implies that the program runs 34% slower with security checks enabled.

4.9.2 Performance Evaluation

Hardware DIFT systems, including Raksha, perform fine-grain tag propagation and checks transparently as the application executes. Hence, they incur minimal runtime overhead compared to program execution with security checks disabled [121, 20, 12]. The small overhead is due to tag management during program initialization, paging, and I/O events. Nevertheless, such events are rare and involve significantly higher sources of overhead compared to tag manipulation.

We focus our performance evaluation on a feature unique to Raksha - the low-overhead handlers for security exceptions. Raksha supports user-level exception handlers as a mechanism to extend and correct the hardware security analysis. As discussed in Section 4.9, exception overhead is not particularly important in protecting against semantic vulnerabilities. High-level attacks require software intervention only at the boundaries of certain system calls, which are infrequent events that transition to the operating system. On the other hand, fast software handlers can be useful in the protection against memory corruption attacks, by helping identify potential bounds-check operations and managing the tradeoff between false positives and false negatives.

Table 4.4 presents the slowdown experienced by various integer benchmarks from the SPEC2000 suite with a pointer tainting analysis that filters false positives in software by clearing tags for select compare and AND instructions. The comparison filter untaints a tainted operand if it is compared to an untainted operand. The AND filter untaints a tainted operand if it is AND'ed with an untainted power of two minus one, as this technique is commonly used as a bounds check for power of two sized tables. The AND filter requires the use of a custom check policy, specific to the AND instruction. We also attempted to filter both validation cases in a combined analysis.

For every filter case, the left column in Table 4.4 shows the slowdown with Raksha when the software filter utilizes the low-overhead security exception. The right column measures the slowdown when the software filter is invoked through a regular OS exception. OS traps are the mechanism that previous DIFT architectures would use to invoke further software, had they recognized the need for software intervention to properly handle these corner cases.

Table 4.4 indicates that for programs like gcc and crafty, the overhead of software filtering is quite low for both mechanisms, as they rarely use tagged data in comparisons or logical AND instructions. On the other hand, utilities like twolf and bzip2 generate these cases more frequently. Hence, the slowdown is closely related to the overhead of the mechanism used to invoke the software filter. For gzip, Raksha's mechanism limits the overhead of compare filtering to 30%, while OS traps slow down the program by more than 2.9×. The comparison between the two techniques is similar for gzip and parser with the AND instruction filter. There are some pathological cases that run slowly on both systems. For example, bzip2 with the compare filter experiences a 3× slowdown even with user-level exceptions. On the other hand, using OS traps leads to a 13× slowdown! If a user has to choose between a 13× slowdown or program termination due to false positives, she will likely disable DIFT. While Raksha cannot eliminate all performance issues in all cases, it helps reduce the overhead of avoiding false positives and negatives in strong security policies.

Table 4.4 shows that the overhead for the combined filter is sometimes lower than that with one of the individual filters. This is due to the synergistic nature of the two filters. The AND filter may untag an operand that is later used multiple times in compare operations (e.g., by loading a variable from memory during each loop iteration). Another interesting observation is that the filter overheads could be reduced in some cases if, instead of just

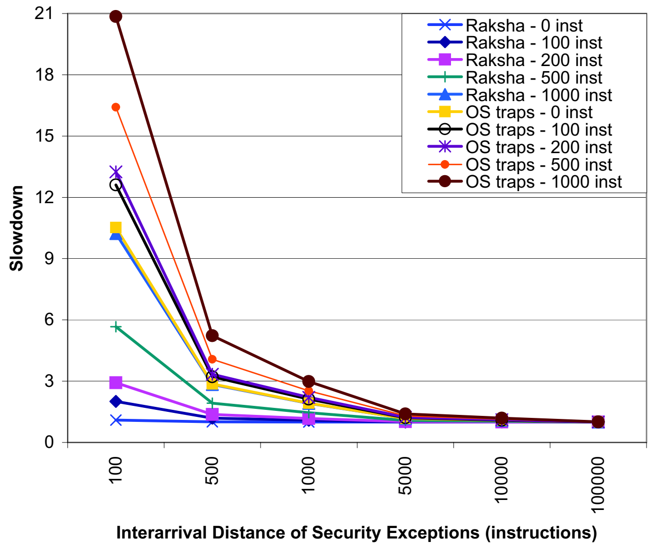


Figure 4.5: The performance degradation for a microbenchmark that invokes a security handler of controlled length every certain number of instructions. All numbers are normalized to a baseline case which has no tag operations.

clearing the register tag, we could also clear the tag for the memory location assigned to the variable (if any). However, current executable binary formats do not have sufficient information to allow for precise alias analysis.

To better understand the tradeoffs between the invocation frequency of software handlers and runtime overhead, we developed a simple microbenchmark. The microbenchmark invokes a security handler every microbenchmark. The microbenchmark invokes a security handler every 100 to 100,000 instructions. The duration of the handler is also controlled to be 0, 200, 500, or 1000 arithmetic instructions. This is in addition to the instructions necessary to invoke and terminate the handler. Figure 4.5 shows that if security exceptions are invoked less frequently than every 5,000 instructions, both user-level and OS-level exception handling are acceptable as their cost is easily amortized. On the other hand, if software is involved as often as every 1,000 or 100 instructions, user-level handlers are critical in maintaining acceptable performance levels. Low-overhead security exceptions allow software to intervene more frequently or perform more work per invocation. For reference, our software filters for the experiments in Table 4.4 require approximately 100 instructions per invocation.

4.9.3 Lessons Learned

Raksha provides a unique platform for the application of DIFT techniques to real-world applications in a modern server environment. This section discusses the successes and challenges we encountered when preventing high-level and low-level attacks using Raksha. In particular, we discuss how DIFT policies were successful in robust prevention of high-level attacks, but challenges remain in effectively addressing buffer overflows due to inherent ambiguities in recognizing bounds checks and other low-level validation operations. The difficulties encountered in buffer overflow detection motivated the research presented in Chapter 5

Lessons from High-Level Attacks

Raksha is the first DIFT system to prevent high-level attacks such as directory traversals on unmodified binaries. Raksha is well-suited to detect such high-level vulnerabilities as they tend to be precisely defined. A SQL query either has, or doesn't have, a tagged command, and Raksha security handlers can easily distinguish between safe and unsafe uses of tainted information for this class of attacks. Application and language routines for validating untrusted information do not have to be separately identified, avoiding any associated false positives and negatives.

Our experiments show that check rules for these high-level attacks must be easily customizable. For example, there is no universally accepted standard for cross-site scripting or SQL filters. A wide variety of filters are necessary to accomodate the diverse set of behaviors in real-world software. Some applications HTML-encode all untrusted input; others allow input to contain a safe, restricted subset of HTML tags; and finally applications such as Bugzilla allow untrusted input to contain a restricted set of SQL commands. Because Raksha provides programmable policies and can be extended through software, it can support such customization.

Most high-level bugs can be caught at the system call layer, which has many advantages. System calls are infrequent, and interposition has minimal overhead for most workloads [37, 96]. The kernel ABI explicitly defines the semantics of each system call. Moreover, system calls provide complete mediation. If we interposed on a higher level routine, applications might evade our protection by calling directly into lower-level functions. Even though all checks are applied at the coarse granularity of system calls, the precise, fine-granularity taint tracking supported by Raksha hardware allows the policy to distinguish a safe (untainted) argument to a system call from an untrusted argument that must be validated.

Prior work has shown that SQL injection can always be safely detected without false positives or negatives, so long as trusted and untrusted data can be distinguished and the SQL grammar is known [119]. Raksha does not provide perfect precision, as tags are tracked at word granularity rather than byte granularity. Strings are one of the few situations in which the same word may contain tainted and untainted bytes. Nevertheless, this has not been a significant enough issue thus far to motivate support for byte-level tags. To ensure that an application performing a byte-by-byte copy over a tainted word actually untaints the word, our string tainting policy uses merge overwrite as the tag merge mode. Our current SQL validation routine is also not as advanced as the algorithm in [119], since we scan for tainted command characters without parsing the SQL grammar. This will be addressed in future work.

Raksha can only protect against high-level vulnerabilities whose security-critical events can be expressed using function and system call interposition. This describes most interpreted languages, such as PHP or Python. However, in a Java program with a pure Java JDBC driver, it would be very difficult for Raksha to intercept calls to the execute query method of the JDBC driver. Java bytecode is a high-level binary format, and does not specify memory addresses for any of the methods associated with a class. Thus, Raksha would not know the location of the entry point for the JDBC driver's execute query method. Furthermore, as the Java Virtual Machine performs JIT compilation at runtime, methods may be recompiled at runtime and moved to different memory addresses during program execution. In this situation, it would be better to apply DIFT techniques within the Java Virtual Machine [40].

Translation and lookup tables remain the most significant problem for web vulnerability detection using DIFT systems [23, 75]. Our string tainting policy correctly propagates during string manipulation, copying, concatenating, etc. However, web applications may translate input from one encoding to another by indexing bytes of an untrusted string into a lookup table (e.g., convert to uppercase characters). Common glibc string conversion functions such as `atoi()` and `sprintf()` also use lookup tables. Currently, the string tainting policy will not propagate tags across such tables. If a tainted string is converted using a lookup table, then the tag bits of the resulting string will be cleared without an actual validation. Enabling move source address propagation in our string tainting policy would allow us to track tags correctly across lookup tables. However, it would also result in frequent false positives for PHP-based web applications as much of the address space becomes tainted. This is because the string propagation rules provide no mechanism for untainting a pointer except for overwriting the pointer with an untainted word. Despite these concerns, our string tainting policy did not prevent us from detecting all attacks in our experiments without any false positives. We are currently investigating better rules to address the issue of translation and lookup tables. Recent work by other DIFT researchers has taken steps to address this crucial issue [75].

Lessons from Low-Level Attacks

Hardware and software DIFT architectures have very little information available when protecting against memory corruption vulnerabilities. Unmodified binaries do not provide bounds information and do not explicitly identify when a pointer has been validated via some sort of bounds check. Hence, DIFT systems must detect on their own when a tagged pointer should be considered safe.

Detecting low-level validation patterns is particularly difficult. Not every bounds check is a comparison and not every comparison is a bounds check. Raksha can mitigate some of the ambiguity by using flexible security policies and perform further processing in the security handlers. To detect the case where an AND instruction is used as a bounds check, we can use a custom operation to specify a unique policy for the AND instruction. If the AND has a tagged source operand, a security exception is raised. The handler untaints the first source operand if the second source operand is untagged and is a power of 2 minus one. Previous DIFT architectures [12, 121] would not correctly identify this behavior, and would often terminate the safe program.

Unfortunately, we have also encountered other cases that cannot be resolved with hardware or software DIFT alone. Several frequently used functions in the GNU C library include tagged pointer dereferences that do not require a bounds check of any sort to be considered safe. For example, all the character conversion and classification functions (`toupper()`, `tolower()`, etc.) use 256-entry tables that can be safely indexed with tagged bytes. This is safe because the table has exactly 256 entries. If the table had 255 entries or fewer, a buffer overflow could result. However, hardware has no bounds information, and cannot reliably disambiguate this case. The significant false positive issues encountered when applying this conventional DIFT buffer overflow policy to real-world programs led to the novel buffer overflow policy described in Chapter 5.

It is important to note that we observed no false positives or negatives for the code pointer buffer overflow protection provided by checking jump address and instruction tags. Only the data pointer protection provided by load and store address checks resulted in false positive or negative issues.

Chapter 5

Userspace & Kernelspace Buffer Overflows

Despite decades of research, buffer overflow attacks remain a critical threat to system security. The successful exploitation of a buffer overflow typically results in arbitrary code execution with the privilege level of the vulnerable application. This chapter describes the deficiencies of current DIFT-based buffer overflow approaches and presents a novel DIFT analysis for reliably preventing buffer overflows in both userspace and the operating system. This analysis is evaluated on Linux userspace and kernelspace using the Raksha hardware DIFT platform.

5.1 Background & Motivation

Buffer overflows have been a pervasive, crucial threat to the security of modern systems since they were employed in the Morris Internet worm [116] in 1988. Existing defenses have proven insufficient to eliminate the threat of buffer overflows. Real-world exploits such as [29] and [34] demonstrated that a seasoned attacker can bypass even the combination of all existing real-world defenses — ASLR, stack canaries, and non-executable pages. Furthermore, existing protection mechanisms often compromise performance, break compatibility with legacy applications, or require recompilation. Further discussion and critiques of existing non-DIFT buffer overflow protection mechanisms can be found in Section

```
static char uppertbl[256] = ...
#define TO_UPPER(x) (uppertbl[(unsigned char)(x)])
```

Figure 5.1: C macro that converts single byte characters to uppercase using an array. The array has 256 entries and thus this macro is safe to use even on untrusted input. No bounds check is required due to the unsigned char typecast.

2.1.2. Due to these flaws in modern buffer overflow protection systems, researchers have investigated the use of dynamic information flow tracking techniques to prevent buffer overflows in unmodified binaries. This section summarizes the state of the art in buffer overflow prevention using Dynamic Information Flow Tracking, and presents the shortcomings of currently available approaches.

5.1.1 DIFT Policies for Buffer Overflow Prevention

DIFT is a powerful technique for preventing security attacks, and has several advantages over existing buffer overflow prevention techniques. DIFT analyses can be applied to unmodified binaries. Using hardware support, DIFT has negligible overhead and works correctly with all types of legacy applications, even those with multithreading and self-modifying code [24, 15]. DIFT can potentially provide a solution to the buffer overflow problem that protects all pointers (code and data), has no false positives, requires no source code access, and works with unmodified legacy binaries and even the operating system. Previous non-DIFT hardware approaches protect only the stack return address [127, 57] or prevent code injection with non-executable pages.

There are two major existing policies for buffer overflow protection using DIFT: *bounds-check recognition (BR)* and *pointer injection (PI)*. The approaches differ in tag propagation rules, the conditions that indicate an attack, and whether tagged input can ever be validated by application code.

Bounds Check Recognition: Most DIFT systems, including the policy used by Raksha in Chapter 4, use a BR policy to prevent buffer overflow attacks [20, 12, 24, 100]. This technique forbids 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 and 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. Further discussion of this technique can be found in Section 4.7.1

Unfortunately, the BR policy leads to significant *false negatives* [24, 50]. Not all comparisons are bounds checks. For example, the glibc `strtok()` function compares each input character against a class of allowed characters, and stores matches in an output buffer. DIFT interprets these comparisons as bounds checks, and thus the output buffer is always untainted, even if the input to `strtok()` was tainted. This can lead to false negatives. For example, this issue caused a real-world false negative when we performed a stack overflow exploit on atphttpd [2].

However, the most critical flaw of BR-based policies is an unacceptable number of *false positives* with commonly used software. Any scheme for 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.

Many common glibc functions, such as `tolower()`, `toupper()`, and various character classification functions (`isalpha()`, `isalnum()`, etc.) index an untrusted byte into a 256 entry table. This is completely safe, and requires no bounds check. Figure 5.1 demonstrates an uppercase conversion macro that uses this approach. BR policies fail to recognize this input validation case because the bounds of the table are not known in a stripped binary. Hence, false positives occur during common system operations such as compiling files with gcc and compressing data with gzip. In practice, false positives occur only for data pointer protection. No false positive has been reported on x86 Linux systems so long as only control pointers are protected [20, 24]. Unfortunately, control pointer protection alone has been shown to be insufficient [13].

Pointer Injection: Recent work [50] has proposed a pointer injection (PI) policy for buffer overflow protection using DIFT. Rather than recognize bounds checks, PI enforces a different invariant: untrusted information should never directly supply a pointer value. Instead, tainted information must always be combined with a legitimate pointer from the application before it can be dereferenced. Applications frequently add an untrusted index to a legitimate base address pointer from the application's address space. On the other hand, 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.

To prevent buffer overflows, a PI policy uses two tag bits per memory location: one to identify tainted data (T bit) and the other to identify pointers (P bit). As in other DIFT analyses, the taint bit is set for all untrusted information, and propagated during data movement, arithmetic, and logical instructions. However, PI provides no method for untainting data, nor does it rely on any bounds check recognition. The P 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-2 boundary. Security attacks are detected if a tainted pointer is dereferenced and the P bit is not set. The primary advantage of PI is that it does not rely on bounds check recognition, thus avoiding the false positive and negative issues that plague BR-based policies.

The disadvantage of the PI policy is that it requires legitimate application pointers to be identified. For dynamically allocated memory, this can be accomplished by setting the P bit of any pointer returned by a memory-allocating system call such as `mmap` or `brk`. However, no such solution has been presented for pointers to statically allocated memory regions. The original proposal requires that each `add` or `sub` instruction determines if one of its untainted operands points into any valid virtual address range [50] . If so, the destination operand has its P bit set, even if the source operand does not. To support such functionality, the hardware would need to traverse the entire page table or some other variable length data-structure that summarizes the allocated portions of the virtual address space for every add or subtract instruction in the program. The complexity and runtime overhead of such hardware is far beyond what is acceptable in modern systems. Furthermore, while promising, the PI policy has not been evaluated on a wide range of large applications, as the original proposal was limited to simulation studies with performance benchmarks.

DIFT has never been used to provide buffer overflow protection for the operating system code itself. The OS code is as vulnerable to buffer overflows as user code and several such attacks, both local and remote, have been documented [133, 60, 80]. Moreover, the

Operation	**Example**	**Taint Propagation**	**Pointer Propagation**
Load	ld `r1`+imm, `r2`	T[`r2`] = T[M[`r1`+imm]]	P[`r2`] = P[M[`r1`+imm]]
Store	st `r2`, `r1`+imm	M[`r1`+imm] = T[`r2`]	P[M[`r1`+imm]] = P[`r2`]
Add/Subtract/Or	add `r1`, `r2`, `r3`	T[`r3`] = T[`r1`] $\vee$ T[`r2`]	P[`r3`] = P[`r1`] $\vee$ P[`r2`]
And	and `r1`, `r2`, `r3`	T[`r3`] = T[`r1`] $\vee$ T[`r2`]	P[`r3`] = P[`r1`] $\oplus$ P[`r2`]
All other ALU	xor `r1`,`r2`,`r3`	T[`r3`] = T[`r2`] $\vee$ T[`r1`]	P[`r3`] = 0
Sethi	sethi imm, `r1`	T[`r1`] = 0	P[`r1`] = P[insn]
Jump	jmpl `r1`+imm, `r2`	T[`r2`] = 0	P[`r2`] = 1

Table 5.1: The DIFT propagation rules for the taint and pointer bit. T[x] and P[x] refer to the taint (T) or pointer (P) tag bits respectively for memory location, register, or instruction x.

complexity of the OS code represents a good benchmark for the robustness of a security policy, especially with respect to false positives. OS code contains many complexities that are not encountered in userspace applications, such as interacting with memory-mapped I/O and page tables.

5.2 BOF Protection for Userspace

To provide comprehensive protection against buffer overflows for userspace applications, we use DIFT with a pointer injection (PI) policy. In contrast to previous work [50], our PI policy has no false positives on large Unix applications, provides reliable identification of pointers to statically allocated memory, and requires simple hardware support well within the capabilities of proposed DIFT architectures such as Raksha.

5.2.1 Rules for DIFT Propagation & Checks

Tables 5.1 and 5.2 present the DIFT rules for tag propagation and checks for buffer overflow prevention. The rules are intended to be as conservative as possible while still avoiding false positives. Since our policy is based on pointer injection, we use two tag bits per word of memory and hardware register. 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 (register or 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. In other words, we allow a program to combine a valid pointer with an untrusted index, but not to use an untrusted pointer directly.

Our propagation rules for the P bit (Table 5.1) are derived from pointer operations used in real code. Any operation that could reasonably result in a valid pointer should propagate the P bit. For example, we propagate the P bit for data movement instructions such as `load` and `store`, since copying a pointer should copy the P bit as well. The `and` instruction is often used to align pointers. To model this behavior, the `and` propagation rule sets the P bit of the destination register if one source operand is a pointer, and the other is a non-pointer. Section 5.2.5 discusses a more conservative `and` propagation policy that results in runtime performance overhead.

The P bit propagation rule for addition and subtraction instructions is more permissive than the policy used in [50], due to false positives encountered in legitimate code of several applications. The P bit is propagated if either operand is a pointer because we encountered real-world situations where two pointers are added together. For example, the glibc function `_itoa_word()` is used to convert integers to strings. When given a pointer argument, it indexes bits of the pointer into an array of decimal characters on SPARC systems, effectively adding two pointers together.

Moreover, we have found that the `call` and `jmpl` instructions, which write the program counter (PC) into a general-purpose register, must always set the P bit of their destination register. This is because assembly routines such as glibc `memcpy()` on SPARC contain optimized versions of Duff's device that use the PC as a pointer [30]. In `memcpy()`, a `call` instruction reads PC into a register and adds to it the (possibly tainted) copy length argument. The resulting value is used to jump into the middle of a large block of copy statements. Unless the `call` and `jmpl` set the destination P bit, this behavior would cause a false positive. Similar logic can be found in the `memcmp()` function in glibc for x86 systems.

Finally, we must propagate the P bit for instructions that may initialize a pointer to a valid address in statically allocated memory. The only instruction used to initialize a pointer to statically allocated memory is `sethi`. The `sethi` instruction sets the most significant

Operation	**Example**	**Security Check**
Load	ld `r1`+imm, `r2`	T[`r1`] $\wedge \neg$ P[`r1`]
Store	st `r2`, `r1`+imm	T[`r1`] $\wedge \neg$ P[`r1`]
Jump	jmpl `r1`+imm, `r2`	T[`r1`] $\wedge \neg$ P[`r1`]
Instruction fetch	-	T[insn]

Table 5.2: The DIFT check rules for BOF detection. `rx` means register x. A security exception is raised if the condition in the rightmost column is true.

22 bits of a register to the value of its immediate operand and clears the least significant 10 bits. If the analysis described in Section 5.2.2 determines that a `sethi` instruction is a pointer initialization statement, then the P bit for this instruction is set at process startup. We propagate the P bit of the `sethi` instruction to its destination register at runtime. A subsequent `or` instruction may be used to initialize the least significant 10 bits of a pointer, and thus must also propagate the P bit of its source operands.

The remaining ALU operations such as multiply or shift should not be performed on pointers. These operations clear the P bit of their destination operand. If a program marshals or encodes pointers in some way, such as when migrating shared state to another process [97], a more liberal pointer propagation ruleset similar to our rules for taint propagation rules may be necessary.

5.2.2 Pointer Identification

The PI-based policy depends on accurate identification of legitimate pointers in the application code in order to initialize the P bit for these memory locations. When a pointer is assigned a value derived from an existing pointer, tag propagation will ensure that the P bit is set appropriately. The P 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. We 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 allocated memory.

```
int x,y;
int * p = &x + 0x80000000;      // symbol + any constant OK
int * p = &x;                   // symbol + no offset OK
int * p = (int) &x + (int) &y;  // cannot add two symbols
int * p = (int) &x × 4;         // cannot multiply a symbol
int * p = (int) &x ⊕ -1;        // cannot xor a symbol
```

Figure 5.2: C code showing valid and invalid references to statically allocated memory. Variables x, y, and p are global variables.

Pointers to Dynamically Allocated Memory

To allocate memory at runtime, user code must use a system call. On a Linux SPARC system, there are five memory allocation system calls: `mmap`, `mmap2`, `brk`, `mremap`, and `shmat`. All pointers to dynamically allocated memory are derived from the return values of these system calls. We modified the Linux kernel to set the P bit of the return value for any successful memory allocation system call. This allows all dynamic root pointer assignments to be identified without false positives or negatives. Furthermore, we also set the P bit of the stack pointer register at process startup.

Pointers to Statically Allocated Memory

All static root pointer assignments are contained in the data and code sections of an object file. The data section contains pointers initialized to statically allocated memory addresses. The code section contains instructions used to initialize pointers to statically allocated memory at runtime. To initialize the P bit for static root pointer assignments, we must scan all data and code segments of the executable and any shared libraries at startup.

When the program source code is compiled to a relocatable object file, all references to statically allocated memory are placed in the relocation table. Each relocation table entry stores the location of the memory reference, the reference type, the symbol referred to, and an optional symbol offset. For example, a pointer in the data segment initialized to *&x + 4* would have a relocation entry with type data, symbol *x*, and offset 4. When the linker creates a final executable or library image from a group of object files, the relocation table in each object file is traversed and any reference to statically allocated memory is updated if the symbol to which it refers has been relocated to a new address.

Algorithm 1 Pseudocode for identifying static root pointer assignments in SPARC ELF binaries.

```
procedure CheckStaticCode(ElfObject o, Word * w)
    if $*w$ is a sethi instruction then
        $x \leftarrow$ extract_cst22($*w$)  ▷ extract 22 bit constant from sethi, clear least significant 10 bits
        if $x >= o$.obj_start and $x < o$.obj_end then
            set_p_bit($w$)
        end if
    end if
end procedure

procedure CheckStaticData(ElfObject o, Word * w)
    if $*w >= o$.obj_start and $*w < o$.obj_end then
        set_p_bit($w$)
    end if
end procedure

procedure InitStaticPointer(ElfObject o)
    for all segment $s$ in $o$ do
        for all word $w$ in segment $s$ do
            if $s$ is executable then
                CheckStaticCode($o$, $w$)
            end if
            CheckStaticData($o$, $w$)          ▷ Executable sections may contain read-only data
        end for
    end for
end procedure
```

With access to full relocation tables, static root pointer assignments can be identified without false positives or negatives. Conceptually, we set the P bit for each instruction or data word whose relocation table entry is a reference to a symbol in statically allocated memory. However, in practice full relocation tables are not available in executables or shared libraries. Hence, we must conservatively identify statically allocated memory references without access to relocation tables. Fortunately, the restrictions placed on references to statically allocated memory by the object file format allow us to detect such references by scanning the code and data segments, even without a relocation table. The only instructions or data that can refer to statically allocated memory are those that conform to an existing relocation entry format.

Like all modern Unix systems, our prototype uses the ELF object file format [108]. Statically allocated memory references in data segments are 32-bit constants that are relocated using the R_SPARC_32 relocation entry type. Statically allocated memory references in code segments are created using a pair of SPARC instructions, `sethi` and `or`. A pair of instructions is required to construct a 32-bit immediate because SPARC instructions have a fixed 32-bit width. The `sethi` instruction initializes the most significant 22 bits of a word to an immediate value, while the `or` instruction is used to initialize the least significant 10 bits (if needed). These instructions use the R_SPARC_HI22 and R_SPARC_LO10 relocation entry types, respectively.

Even without relocation tables, we know that statically allocated memory references in the code segment are specified using a `sethi` instruction containing the most significant 22 bits of the address, and any statically allocated memory references in the data segment must be valid 32-bit addresses. However, even this knowledge would not be useful if the memory address references could be encoded in an arbitrarily complex manner, such as referring to an address in statically allocated memory shifted right by four or an address that has been logically negated. Scanning code and data segments for all possible encodings would be extremely difficult and would likely lead to many false positives and negatives. Fortunately, this situation does not occur in practice, as all major object file formats (ELF [108], a.out, PE [67], and Mach-O) restrict references to statically allocated memory to a single valid symbol in the current executable or library plus a constant offset. Figure 5.2 presents a few C code examples demonstrating this restriction.

Algorithm 1 summarizes our scheme initializing the P bit for static root pointer assignments without relocation tables. We 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 P bit for any matches. To recognize root pointer assignments in code, we 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 set the P bit of the instruction. Unlike the x86, the SPARC has fixed-length instructions, allowing for easy disassembly of all code regions.

Modern object file formats do not allow executables or libraries to contain direct references to another object file's symbols, so we need to compare possible pointer values against only the current object file's start and end addresses, rather than the start and end addresses of all executable and libraries in the process address space. This algorithm is executed *once* for the executable at startup and once for each shared library when it is initialized by the dynamic linker. As shown in Section 5.2.5, the runtime overhead of the initialization is negligible.

In contrast with our scheme, pointer identification in the original proposal for a PI-based policy is impractical. The scheme in [50] attempts to dynamically detect pointers by checking if the operands of any instructions used for pointer arithmetic can be valid pointers to the memory regions currently used by the program. This requires scanning the page tables for every add or subtract instruction, which is prohibitively expensive.

5.2.3 Discussion

False positives and negatives due to P bit initialization: Without access to the relocation tables, our scheme for root pointer identification could lead to false positives or negatives in our security analysis. If an integer in the data segment has a value that happens to correspond to a valid memory address in the current executable or shared library, its P bit will be set even though it is not a pointer. This misclassification can cause a false negative in our buffer overflow detection. A false positive in the buffer overflow protection is also possible, although we have not observed one in practice thus far. All references to statically allocated memory are restricted by the object file format to a single symbol plus a constant offset. Our analysis will fail to identify a pointer only if this offset is large enough to cause the *symbol+offset* sum to refer to an address outside of the current executable object. Such a pointer would be outside the bounds of any valid memory region in the executable and would cause a segmentation fault if dereferenced.

DIFT tags at word granularity: Unlike prior work [50], we use per-word tags (P and T bits) rather than per-byte tags. Our policy targets pointer corruption, and modern ABIs require pointers to be naturally aligned, 32-bit values, even on the x86 [108]. Hence, we can reduce the memory overhead of DIFT from eight bits per word to two bits per word.

As explained in Section 4.6.5, we must specify how to handle partial word writes during byte or halfword stores. These writes only update part of a memory word and must combine the new tag of the value being written to memory with the old tag of the destination memory word. The combined value is then used to update the tag of the destination memory word. For taint tracking (T bit), we OR the new T bit with the old one in memory, since we want to track taint as conservatively as possible. Writing a tainted byte will taint the entire word of memory, and writing an untainted byte to a tainted word will not untaint the word. For pointer tracking (P bit), we must balance protection and false positive avoidance. We want to allow a valid pointer to be copied byte-per-byte into a word of memory that previously held an integer and still retain the P bit. However, if an attacker overwrites a single byte of a pointer [35], that pointer should lose its P bit. To satisfy these requirements, byte and halfword store instructions always set the destination memory word's P bit to that of the new value being written, ignoring the old P bit of the destination word.

Caching P Bit initialization: For performance reasons, it is unwise to always scan all memory regions of the executable and any shared libraries at startup to initialize the P bit. P bit initialization results can be cached, as the pointer status of an instruction or word of data at startup is always the same. The executable or library can be scanned once, and a special ELF section containing a list of root pointer assignments can be appended to the executable or library file. At startup, the security monitor could read this ELF section, initializing the P bit for all specified addresses without further scanning.

5.2.4 Portability to Other Systems

We believe that our approach is portable to other architectures and operating systems, including the Intel x86. The propagation and check rules reflect how pointers are used in practice and for the most part are architecture neutral. However, the pointer initialization rules must be ported when moving to a new platform. Identifying dynamic root pointer assignments is OS-dependent, but requires only modest effort. All we require is a list of system calls that dynamically allocate memory. This can be obtained even for closed source operating systems such as Windows.

Identifying static root pointer assignments depends on both the architecture and the object file format. Our analysis for static pointer initializations within data segments should work on all modern platforms. This analysis assumes that initialized pointers within the data segment are word-sized, naturally aligned variables whose value corresponds to a valid memory address within the executable. This assumption holds for all modern object file formats, including the object file formats used on x86 Windows and Linux systems [67, 108]. These object file formats require all static data pointer references to be word-sized and naturally aligned.

Identifying static root pointer assignments in code segments may require a slightly different algorithm from the one presented in this paper, depending on the ISA. Porting to other RISC systems should not be difficult, as all RISC architectures use fixed-length instructions and provide an equivalent to `sethi`. For instance, MIPS uses the load-upper-immediate instruction to set the high 16 bits of a register to a constant. Hence, we just need to adjust Algorithm 1 to target these instructions.

However, CISC architectures such as the x86 require a slightly different approach because they support variable-length instructions. In a CISC ISA such as the Intel x86, static root pointer assignments are performed using an instruction such as `movl` that initializes a register to a full 32-bit constant. However, precisely disassembling a code segment with variable-length instructions is undecidable. To avoid the need for precise disassembly, we can conservatively identify potential instructions that contain a reference to statically allocated memory.

A conservative analysis to perform P bit initialization on CISC architectures would first scan the entire code segment for valid references to statically allocated memory. A valid 32-bit memory reference may begin at any byte in the code segment, as a variable-length ISA places no alignment restrictions on instructions. For each valid memory reference, we scan backwards to determine if any of the bytes preceding the address can form a valid instruction. This may require scanning a small number of bytes up to the maximum length of an ISA instruction. Disassembly may also reveal multiple candidate instructions for a single valid address. We examine each candidate instruction and conservatively set

Program	Vulnerability	Attack Detected
polymorph [93]	Stack overflow	Overwrite frame pointer, return address
atphttpd [2]	Stack overflow	Overwrite frame pointer, return address
sendmail [64]	BSS overflow	Overwrite application data pointer
traceroute [110]	Double free	Overwrite heap metadata pointer
nullhttpd [74]	Heap overflow	Overwrite heap metadata pointer

Table 5.3: The security experiments for BOF detection in userpace.

the P bit if any candidate instruction may initialize a register to the valid address. This allows us to conservatively identify all static root pointer assignments, even without precise disassembly, and would allow our analyses to run on CISC architectures such as the Intel x86.

5.2.5 Evaluation of Userspace Protection

We evaluated the security and performance of our buffer overflow protection on a wide range of user programs. This section presents our security and performance results. We demonstrate that our novel buffer overflow protection comprehensively prevents userspace buffer overflow vulnerabilities without real-world false positives and has little performance overhead.

Userspace Security

To evaluate our security scheme, we implemented our DIFT policy for buffer overflow prevention on the Raksha system described in Chapter 4. For this research, we updated Raksha's software infrastructure. To provide a realistic, thorough test environment for buffer overflow prevention, we must protect every application in a real Linux distribution from buffer overflow attacks, from init to halt.

The software infrastructure used in Chapter 4 relied on preloaded shared libraries to handle tag initialization and policy management. This is unsatisfactory as shared libraries cannot be run until the dynamic linker is already initialized, which would mean that our buffer overflow code would not be the first code executing in the process. Furthermore, statically linked binaries or binaries not linked against libc could not be protected by DIFT.

Finally, the previous Raksha software architecture used a custom Linux distribution based on Cross-Compiled Linux from Scratch, rather than a modern and widely-used Linux distribution. Basing our software infrastructure on Gentoo allows us easy access to the thousands of well-tested software packages in Gentoo's repository.

For our new buffer overflow research, we migrated to a slightly modified version of the popular Gentoo Linux distribution [39], allowing access to Gentoo's sizable package management system for testing and validating our prototype. We extended a Linux 2.6.21.1 kernel to set the P bit for pointers returned by memory allocation system calls and to initialize taint bits. Policy configuration registers and register tags are saved and restored during traps and interrupts.

Rather than rely on preloaded shared libraries, we modified the operating system to taint the environment variables and program arguments when a process is created, and also taint any data read from the filesystem or network. The only exception is reading executable files owned by root or a trusted user. The dynamic linker requires root-owned libraries and executables to be untainted, as it loads pointers and executes code from these files.

We use a novel approach to ensure that our buffer overflow policy is applied to every single instruction executed in userspace. Our security monitor initializes the P bit of each library or executable in the user's address space and handles security exceptions. The monitor was compiled as a statically linked executable. The kernel loads the monitor into the address space of every process, including `init`. When a process begins execution, the kernel first transfers control to the monitor, which performs P bit initialization on the application binary. The monitor then sets up the policy configuration registers with the buffer overflow prevention policy, disables trusted mode, and transfers control to the real application entry point. The real application entry is provided by the kernel using additional auxiliary vectors [126] placed on the process's stack at startup. This approach does not rely on shared library preloading at all, and works even for statically linked binaries or binaries that do not link against libc.

The dynamic linker was slightly modified to call back to the security monitor each time a new library is loaded, so that P bit initialization can be performed on the new library. All application and library instructions in all userspace programs run with buffer overflow protection, from the first instruction at the entry point of the executable (or dynamic linker) to

Program	PI (normal)	PI (`and` emulation)
164.gzip	1.002x	1.320x
175.vpr	1.001x	1.000x
176.gcc	1.000x	1.065x
181.mcf	1.000x	1.010x
186.crafty	1.000x	1.000x
197.parser	1.000x	2.230x
254.gap	1.000x	2.590x
255.vortex	1.000x	1.130x
256.bzip2	1.000x	1.050x
300.twolf	1.000x	1.010x

Table 5.4: Normalized execution time after the introduction of the PI-based buffer overflow protection policy. The execution time without the security policy is 1.0. Execution time higher than 1.0 represents performance degradation.

the exit system call terminating the process. No userspace applications or libraries, excluding the dynamic linker, were modified to support DIFT analysis. Furthermore, all binaries in our experiments are stripped, and contain no debugging information or relocation tables.

The security of our system was evaluated by attempting to exploit a wide range of buffer overflows on vulnerable, unmodified applications. The results are presented in Table 5.3. We successfully prevented both control and data pointer overwrites on the stack, heap, and BSS. In the case of polymorph, we also tried to corrupt a single byte or a halfword of the frame pointer instead of the whole word. Our policy detected the attack correctly as we do track partial pointer overwrites (see Section 5.2.3).

To test for false positives, we ran a large number of real-world workloads including compiling the Apache web server, booting the Gentoo Linux distribution, and running Unix binaries such as perl, GCC, make, sed, awk, and ntp. No false positives were encountered, despite our conservative tainting policy.

Userspace Performance

To evaluate the performance overhead of our policy, we ran 10 integer benchmarks from the SPECcpu2000 suite. Table 5.4 (column titled "PI (normal)") shows the overall runtime overhead introduced by our security scheme, assuming no caching of the P bit initialization. The runtime overhead is negligible ($<$0.1%) and solely due to the initialization of the P bit.

The propagation and check of tag bits is performed in hardware at runtime and has no performance overhead [24]. No software exception handling is required unless a buffer overflow is detected.

We also evaluated the more restrictive P bit propagation rule for `and` instructions from [50]. The P bit of the destination operand is set only if the P bit of the source operands differ, and the non-pointer operand has its sign bit set. The rationale for this is that a pointer will be aligned by masking it with a negative value, such as masking against -4 to force word alignment. If the user is attempting to extract a byte from the pointer – an operation which does not create a valid pointer, the sign bit of the mask will be cleared.

This more conservative rule requires any `and` instruction with a pointer argument to raise a security exception, as the data-dependent tag propagation rule is too expensive to support in hardware. The security exception handler performs this propagation in software for `and` instructions with valid pointer operands. While we encountered no false positives with this rule, performance overheads of up to 160% were observed for some SPECcpu2000 benchmarks (see rightmost column in Table 5.4). We believe this stricter `and` propagation policy provides a minor improvement in security and does not justify the increase in runtime overhead.

5.3 Extending BOF Protection to Kernelspace

The OS kernel presents unique challenges for buffer overflow prevention. Unlike userspace, the kernel shares its address space with many untrusted processes, and may be entered and exited via traps. Hardcoded constant addresses are used to specify the beginning and end of kernel memory maps and heaps. The kernel may also legitimately dereference untrusted pointers in certain cases. Moreover, the security requirements for the kernel are higher as compromising the kernel is equivalent to compromising all applications and user accounts.

In this section, we extend our userspace buffer overflow protection to the OS kernel. We evaluate our approach by using the PI-based policy to prevent buffer overflows in the Linux kernel. In comparison to prior work [21], we do not require the operating system to be ported to a new CPU architecture, protect the entire OS codebase with no real-world false positives or errors, support self-modifying code, and have low runtime performance overhead. We also provide the first comprehensive runtime detection of user-kernel pointer dereference attacks.

5.3.1 Entering and Exiting Kernelspace

The tag propagation and check rules described in Tables 5.1 and 5.2 for userspace protection are also used with the kernel. The kernelspace policy differs only in the P and T bit initialization and the rules used for handling security exceptions due to tainted pointer dereferences.

Nevertheless, the system may at some point use different security policies for user and kernel code. To ensure that the proper policy is applied to all code executing within the operating system, we take advantage of the fact that the only way to enter the kernel is via a trap, and the only way to exit is by executing a return from trap instruction. When a trap is received, trusted mode is enabled by hardware and the current policy configuration registers are saved to the kernel stack by the trap handler. The policy configuration registers are then re-initialized to the kernelspace buffer overflow policy and trusted mode is disabled. Any subsequent code, such as the actual trap handling code, will now execute with kernel BOF protection enabled. When returning from the trap, the configuration registers for the interrupted user process must be restored.

The only kernel instructions that do not execute with buffer overflow protection enabled are the instructions that save and restore configuration registers during trap entry and exit, a few trivial trap handlers written in assembly which do not access memory at all, and the fast path of the SPARC register window overflow/underflow handler. We do not protect these handlers because they do not use a runtime stack and do not access kernel memory unsafely. Enabling and disabling protection when entering and exiting such handlers could adversely affect system performance without improving security.

5.3.2 Pointer Identification in the Presence of Hardcoded Addresses

The OS kernel uses the same static root pointer assignment algorithm as userspace. At boot time, the kernel image is scanned for static root pointer assignments by scanning its code and data segments, as described in Section 5.2. However, dynamic root pointer assignments must be handled differently. In userspace applications, dynamically allocated memory is obtained via OS system calls such as `mmap` or `brk`. In the operating system, a variety of memory map regions and heaps are used to dynamically allocate memory. The start and end virtual addresses for these memory regions are specified by hardcoded constants in kernel header files. All dynamically allocated objects are derived from the hardcoded start and end addresses of these dynamic memory regions.

In kernelspace, all dynamic root pointer assignments are contained in the kernel code and data at startup. When loading the kernel at system boot time, we scan the kernel image for references to dynamically allocated memory maps and heaps. All references to dynamically allocated memory must be to addresses within the kernel heap or memory map regions identified by the hardcoded constants. To initialize the P bit for dynamic root pointer assignments, any `sethi` instruction in the code segment or word of data in the data segment that specifies an address within one of the kernel heap or memory map regions will have its P bit set. Propagation will then ensure that any values derived from these pointers at runtime will also be considered valid pointers. The P bit initialization for dynamic root pointer assignments and the initialization for static root pointer assignments can be combined into a single pass over the code and data segments of the OS kernel image at bootup.

On our Linux SPARC prototype, the only heap or memory map ranges that should be indexed by untrusted information are the `vmalloc` heap and the fixed address, `pkmap`, and `srmmu-nocache` memory map regions. The start and end values for these memory regions can be easily determined by reading the header files of the operating system, such as the `vaddrs` SPARC-dependent header file in Linux. All other memory map and heap regions in the kernel are small private I/O memory map regions whose pointers should never be indexed by untrusted information and thus do not need to be identified during P bit initialization to prevent false positives.

Kernel heaps and memory map regions have an inclusive lower bound, but exclusive upper bound. However, we encountered situations where the kernel would compute valid addresses relative to the upper bound. In this situation, a register is initialized to the upper bound of a memory region. A subsequent instruction subtracts a non-zero value from the register, forming a valid address within the region. To allow for this behavior, we treat a `sethi` constant as a valid pointer if its value is greater than or equal to the lower bound of a memory region and less than or equal to the upper bound of a memory region, rather than strictly less than the upper bound. This issue was never encountered in userspace.

5.3.3 Untrusted Pointer Dereferences

Unlike userspace code, there are situations where the kernel may legitimately dereference an untrusted pointer. Many OS system calls take untrusted pointers from userspace as an argument. For example, the second argument to the `write` system call is a pointer to a user buffer.

Only special routines such as `copy_to_user()` in Linux or `copyin()` in BSD may safely dereference a userspace pointer. These routines typically perform a simple bounds check to ensure that the user pointer does not point into the kernel's virtual address range. The untrusted pointer can then safely be dereferenced without compromising the integrity of the OS kernel. If the kernel does not perform this access check before dereferencing a user pointer, the resulting security vulnerability allows an attacker to read or write arbitrary kernel addresses, resulting in a full system compromise.

We must allow legitimate dereferences of tainted pointers in the kernel, while still preventing pointer corruption from buffer overflows and detecting unsafe user pointer dereferences. Fortunately, the design of modern operating systems allows us to distinguish between legitimate and illegitimate tainted pointer dereferences. In the Linux kernel and other modern UNIX systems, the only memory accesses that should cause an MMU fault are accesses to user memory. For example, an MMU fault can occur if the user passed an invalid memory address to the kernel or specified an address whose contents had been paged to disk. The kernel must distinguish between MMU faults due to load/stores to user memory and MMU faults due to bugs in the OS kernel. For this purpose, Linux maintains a list of all kernel instructions that can access user memory and recovery routines that handle faults for these instructions. This list is kept in the special ELF section `__ex_table` in the Linux kernel image. When an MMU fault occurs, the kernel searches `__ex_table` for the faulting instruction's address. If a match is found, the appropriate recovery routine is called. Otherwise, an operating system bug has occurred and the kernel panics.

We modified our security handler so that in the event of a security exception due to a load or store to an untrusted pointer, the memory access is allowed if the program counter (PC) of the faulting instruction is found in the `__ex_table` section and the load/store address does not point into kernelspace. Requiring tainted pointers to specify userspace addresses prevents user/kernel pointer dereference attacks. Additionally, any attempt to overwrite a kernel pointer using a buffer overflow attack will be detected because instructions that access the corrupted pointer will not be found in the `__ex_table` section.

5.3.4 Portability to Other Systems

We believe this approach is portable to other architectures and operating systems. To perform P bit initialization for a new operating system, we would need to know the start and end addresses of any memory regions or heaps that would be indexed by untrusted information. Alternatively, if such information was unavailable, we could consider any value within the kernel's virtual address space to be a possible heap or memory map pointer when identifying dynamic root pointer assignments at system bootup.

Our assumption that MMU faults within the kernel occur only when accessing user addresses also holds for FreeBSD, NetBSD, OpenBSD, and OpenSolaris. Rather than maintaining a list of instructions that access user memory, these operating systems keep a special MMU fault recovery function pointer in the Process Control Block (PCB) of the current task. This pointer is only non-NULL when executing routines that may access user memory, such as `copyin()`. If we implemented our buffer overflow protection for these operating systems, a tainted load or store would be allowed only if the MMU fault pointer in the PCB of the current process was non-NULL and the load or store address did not point into kernelspace.

5.3.5 Evaluation of Kernelspace Protection

To evaluate our buffer overflow protection scheme with OS code, we enabled our PI policy for the Linux kernel. The SPARC BIOS was extended to initialize the P bit for the OS kernel at startup. After P bit initialization, the BIOS initializes the policy configuration registers, disables trusted mode, and transfers control to the startup entry point of the

Module Targeted	Vulnerability	Attack Detected
quotactl system call [138]	User/kernel pointer	Tainted pointer to kernelspace
i2o driver [138]	User/kernel pointer	Tainted pointer to kernelspace
sendmsg system call [132, 3]	Heap overflow Stack Overflow	Overwrite heap metadata pointer Overwrite local data pointer
moxa driver [118]	BSS Overflow	Overwrite BSS data pointer
cm4040 driver [103]	Heap Overflow	Overwrite heap metadata pointer

Table 5.5: The security experiments for BOF detection in kernelspace.

OS kernel. The operating system then begins execution with buffer overflow protection enabled.

When running the kernel, we considered any data received from the network or disk to be tainted. Any data copied from userspace was also considered tainted, as were any system call arguments from a userspace system call trap. As specified in Section 5.2.5, we also save/restore policy registers and register tags during traps. The above modifications were the only changes made to the kernel. All other code, even optimized assembly copy routines, context switching code, and bootstrapping code at startup, were left unchanged and ran with buffer overflow protection enabled. Overall, our extensions added 1774 lines to the kernel and deleted 94 lines, mostly in architecture-dependent assembly files. Our extensions include 732 lines of code for the security monitor, written in assembly.

To evaluate the security of our approach, we exploited real-world user/kernel pointer dereference and buffer overflow vulnerabilities in the Linux kernel. Our results are summarized in Table 5.5. The sendmsg vulnerability allows an attacker to choose between overwriting a heap buffer or stack buffer. Our kernel security policy was able to prevent all exploit attempts. For device driver vulnerabilities, if a device was not present on our FPGA-based prototype system, we simulated sufficient device responses to reach the vulnerable section of code and perform our exploit.

We evaluated the issue of false positives by running the kernel with our security policy enabled under a number of system call-intensive workloads. We compiled large applications from source, booted Gentoo Linux, performed logins via OpenSSH, and served web pages with Apache. Despite our conservative tainting policy, we encountered only one issue, which initially seemed to be a false positive. However, we have established it to be a bug and potential security vulnerability in the current Linux kernel on SPARC32 and have notified the Linux kernel developers. This issue occurred during the `__bzero()` routine, which dereferenced a tainted pointer whose address was not found in the `__ex_table` section. As user pointers may be passed to `__bzero()`, all memory operations in `__bzero()` should be in `__ex_table`. Nevertheless, a solitary block of store instructions did not have an entry. A malicious user could potentially exploit this bug to cause a local denial-of-service attack, as any MMU faults caused by these stores would cause a kernel panic. After fixing this bug by adding the appropriate entry to `__ex_table`, no further false positives were encountered in our system.

Performance overhead is negligible for most workloads. However, applications that are dominated by copy operations between userspace and kernelspace may suffer noticeable slowdown, up to 100% in the worst case scenario of a file copy program. This is due to runtime processing of tainted user pointer dereferences, which require the security exception handler to verify the tainted pointer address and find the faulting instruction in the `__ex_table` section.

We profiled our system and determined that almost all of our security exceptions came from a single kernel function, `copy_user()`. To eliminate this overhead, we manually inserted security checks at the beginning of `copy_user()` to validate any tainted pointers. After the input is validated by our checks, we disable data pointer checks until the function returns. This change reduced our performance overhead to a negligible amount (<0.1%), even for degenerate cases such as copying files. Safety is preserved, as the initial checks verify that the arguments to this function are safe, and manual inspection of the code confirmed that `copy_user()` would never behave unsafely, so long as its arguments were validated. Our control pointer protection prevents attackers from jumping into the middle of this function. Moreover, while checks are disabled when `copy_user()` is executing, taint propagation is still enabled. Hence, `copy_user()` cannot be used to sanitize untrusted data.

5.4 Comprehensive Protection with Hybrid DIFT Policies

The PI-based policy presented in this paper prevents attackers from corrupting any code or data pointers. However, false negatives do exist, and limited forms of memory corruption attacks may bypass our protection. This should not be surprising, as our policy focuses on a specific class of attacks (pointer overwrites) and operates on unmodified binaries without source code access. In this section, we discuss security policies that can be used to mitigate these weaknesses.

False negatives can occur if the attacker overwrites non-pointer data without overwriting a pointer [13]. This is a limited form of attack, as the attacker must use a buffer overflow to corrupt non-pointer data without corrupting any pointers. The application must then use the corrupt data in a security-sensitive manner, such as an array index or a flag determining if a user is authenticated. The only form of non-pointer overwrite our PI policy detects is code overwrites, as tainted instruction execution is forbidden. Non-pointer data overwrites are not detected by our PI policy and must be detected by a separate, complementary buffer overflow protection policy.

5.4.1 Preventing Pointer Offset Overwrites

The most frequent way that non-pointers are used in a security-sensitive manner is when an integer is used as an array index. If an attacker can corrupt an array index, the next access to the array using the corrupt offset will be attacker-controlled. This indirectly allows the attacker to control a pointer value. For example, if the attacker wants to access a memory address `y` and can overwrite an index into array `x`, then the attacker should overwrite the index with the value `y-x`. The next access to `x` using the corrupt index will then access `y` instead.

Our PI policy does not prevent this attack because no pointer was overwritten. We cannot place restrictions on array indices or other types of offsets without bounds information or bounds check recognition. Without source code access or application-specific knowledge, it is difficult to formulate general rules to protect non-pointers without false positives. If source code is available, the compiler may be able to automatically identify security-critical data, such as array offsets and authentication flags, that should never be tainted [9]. DIFT could protect these whitelisted variables from being overwritten with tainted data.

A recently proposed form of ASLR [52] can be used to protect against pointer offset overwrites. This novel ASLR technique randomizes the relative offsets between variables by permuting the order of variables and functions *within* a memory region. This approach would probabilistically prevent all data and code pointer offset overwrites, as the attacker would be unable to reliably determine the offset between any two variables or functions. However, randomizing relative offsets requires access to full relocation tables and may not be backwards compatible with programs that use hardcoded addresses or make assumptions about the memory layout. The remainder of this section discusses additional DIFT policies to prevent non-pointer data overwrites without the disadvantages of ASLR.

5.4.2 Protecting Offsets for Control Pointers

To the best of our knowledge, only a handful of reported vulnerabilities allow control pointer offsets to be overwritten [136, 66]. This is most likely due to the relative infrequency of large arrays of function pointers in real-world code. A buffer overflow is far more likely to directly corrupt a pointer before overwriting an index into an array of function pointers.

Nevertheless, DIFT platforms can provide control pointer offset protection by combining our PI-based policy with a restricted form of BR-based protection. If BR-based protection is only used to protect control pointers, then the false positive issues described in Section 5.1.1 do not occur in practice [20]. To verify this, we implemented a control pointer-only BR policy and applied the policy to userspace and kernelspace. This policy did not result in any false positives, and prevented buffer overflow attacks on control pointers. Our policy classified `and` instructions and all comparisons as bounds checks. This policy is identical to the one described in Section 4.7.1, except `and` instructions automatically untaint rather than result in a tag exception, and tag checks for Move source/destination addresses are omitted.

The BR policy has false negatives in different situations than the PI policy. Hence the two policies are complementary. If control-pointer-only BR protection and PI protection are used concurrently, then a false negative would have to occur in both policies for a control pointer offset attack to succeed. The attacker would have to find a vulnerability that allowed a control pointer offset to be corrupted without corrupting a pointer. The application would then have to operate on the malicious offset using a comparison instruction or an `and` instruction that was not a real bounds check before using the untrusted data in a memory corruption exploit. We believe this is very unlikely to occur in practice. As we have observed no false positives in either of these policies, even in kernelspace, we believe these policies should be run concurrently for additional protection.

5.4.3 Protecting Offsets for Data Pointers

Unfortunately, the BR policy cannot be applied to data pointer offsets due to the severe false positive issues discussed in Section 5.1.1. However, specific situations may allow for DIFT-based protection of non-pointer data. For example, Red Zone heap protection prevents heap buffer overflows by placing a canary or special DIFT tag at the beginning of each heap chunk [102, 99]. This prevents heap buffer overflows from overwriting the next chunk on the heap and also protects critical heap metadata such as heap object sizes.

Red Zone protection can be implemented by using DIFT to tag heap metadata with a sandboxing bit. Access to memory with the sandboxing bit set is forbidden, but sandboxing checks are temporarily disabled when `malloc()` is invoked. A modified `malloc()` is necessary to maintain the sandboxing bit, setting it for newly created heap metadata and clearing it when a heap metadata block is freed. The DIFT Red Zone heap protection could be run concurrently with PI protection, providing enhanced protection for non-pointer data on the heap. This policy is identical to the sandboxing policy described in Section 4.7.3, with tag initialization controlled by the modified `malloc()` implementation.

We implemented a version of Red Zone protection that forbids heap metadata from being overwritten, but allows out-of-bounds reads. We then applied this policy to both glibc `malloc()` in userspace and the Linux slab allocator in kernelspace. No false positives were encountered during any of our stress tests, and we verified that all of our heap exploits from our userspace and kernelspace security experiments were detected by the Red Zone policy.

Tag Bit	**Polices Supported**
Taint Bit	All
Pointer Bit	PI-based Buffer Overflow
Sandbox Bit	Command Injection, Directory Traversal SQL Injection, Format String Attacks Cross-Site Scripting, Red-Zone Bounds Check
BR Taint Bit	BR Control Pointer Buffer Overflow

Table 5.6: Raksha's four tag bits, and how they are used to support DIFT policies that prevent high level and low-level security vulnerabilities. The sandboxing bit is used for system call and function call interposition as well as to protect the security monitor.

5.4.4 Beyond Pointer Corruption

Not all memory corruption attacks rely on pointer or pointer offset corruption. For example, some classes of format string attacks use only untainted pointers and integers [23]. While these attacks are rare, we should still strive to prevent them. The format string policy described in Section 4.7.5 could be used to provide comprehensive protection against format string attacks. The policy uses the same taint information as our PI buffer overflow protection, and thus does not require an extra tag bit. All calls to the `printf()` family of functions are interposed on by the security monitor, which verifies that the format string does not contain tainted format string specifiers such as %n.

For the most effective memory corruption protection for unmodified binaries, DIFT platforms such as Raksha should concurrently enable PI protection, control-pointer-only BR protection, format string protection, and Red Zone heap protection. This would prevent pointer and control pointer offset corruption and provide complete protection against format string and heap buffer overflow attacks.

We can support all these policies concurrently using the four tag bits provided by the Raksha hardware. The P bit and T bit are used for buffer overflow protection and the T bit is also used to track tainted data for format string protection and all other function call or system call interposition-based policies. The sandboxing bit, which prevents stores or code execution from tagged memory locations, is used to protect heap metadata for Red Zone bounds checking, to interpose on calls to the `printf()` functions, and to protect the security monitor (see Section 4.7.3). We can also use the sandboxing bit to support the system call policies described in Chapter 4, such as command injection protection and cross-site scripting protection. Finally, the fourth tag bit is used for control-pointer-only BR protection.

Using this policy configuration, we can support all of the policies described in Chapter 4 as well as the enhanced buffer overflow policy described in this chapter. This combination of policies allows our DIFT platform to address today's most common security threats: buffer overflows, format string attacks, cross-site scripting, directory traversal, command injection, and SQL injection, all using a single DIFT platform operating on unmodified binaries. Table 5.6 describes how all of these policies are mapped to the four tag bits supported by Raksha.

Chapter 6

Web Authentication & Authorization

This chapter presents *Nemesis*,[1] a DIFT-based solution to web authentication and authorization bypass attacks. Nemesis protects web applications by *automatically inferring* when user authentication is performed in web applications using DIFT, without relying on the safety or correctness of the existing code. Nemesis can then use this information to *automatically enforce* access control rules and ensure that only authorized web application users can access resources such as files or databases. We can also use the authentication information to improve the precision of other security analyses, such as DIFT-based SQL injection protection, to reduce their false positive rate.

In this chapter, we present the design for Nemesis, describe a prototype implementation built by modifying the PHP interpreter, and evaluate our design by protecting real-world PHP applications from authentication and authorization attacks. Nemesis is a new interpreter-based DIFT platform implemented completely in software, and does not require a hardware DIFT implementation such as Raksha.

6.1 Web Application Security Architecture

A key problem underlying many security vulnerabilities is that web application code executes with full privileges while handling requests on behalf of users that only have limited

[1]Nemesis is the Greek goddess of divine indignation and retribution, who punishes excessive pride, evil deeds, undeserved happiness, and the absence of moderation.

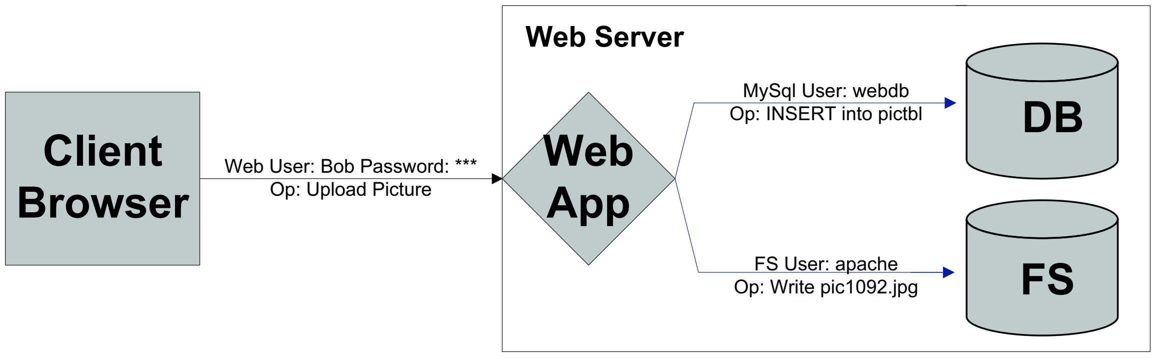


Figure 6.1: The security architecture of typical web applications. Here, user Bob uploads a picture to a web application, which in turn inserts data into a database and creates a file. The user annotation above each arrow indicates the credentials or privileges used to issue each operation or request.

privileges, violating the principle of least privilege [53]. Figure 6.1 provides a simplified view of the security architecture of typical web applications today. As can be seen from the figure, the web application is performing file and database operations on behalf of users using its own credentials, and if attackers can trick the application into performing the wrong operation, they can subvert the application's security. Web application security can thus be viewed as an instance of the confused deputy problem [41]. The remainder of this section discusses this architecture and its security ramifications in more detail.

6.1.1 Authentication Overview

When clients first connect to a typical web application, they supply an application-specific username and password. The web application then performs an authentication check, ensuring that the username and password are valid. Once a user's credentials have been validated, the web application creates a login session for the user. This allows the user to access the web application without having to log in each time a new page is accessed. Login sessions are created either by placing authentication information directly into a cookie that is returned to the user, or by storing authentication information in a session file stored on the server and returning a cookie to the user containing a random, unique session identifier. Thus, a user request is deemed to be authenticated if the request includes a cookie with valid authentication information or session identifier, or if it directly includes a valid username and password.

Once the application establishes a login session for a user, it allows the user to issue requests, such as posting comments on a blog, which might insert a row into a database table, or uploading a picture, which might require a file to be written on the server. However, there is a *semantic gap* between the user authentication mechanism implemented by the web application, and the access control or authorization mechanism implemented by the lower layers, such as a SQL database or the file system. The lower layers in the system usually have no notion of application-level users; instead, database and file operations are usually performed with the privileges and credentials of the web application itself.

Consider the example shown in Figure 6.1, where the web application writes the file uploaded by user Bob to the local file system and inserts a row into the database to keep track of the file. The file system is not aware of any authentication performed by the web application or web server, and treats all operations as coming from the web application itself (e.g. running as the Apache user in Unix). Since the web application has access to every user's file, it must perform internal checks to ensure that Bob hasn't tricked it into overwriting some other user's file, or otherwise performing an unauthorized operation. Likewise, database operations are performed using a per-web application database username and password provided by the system administrator, which authenticates the web application as user `webdb` to MySQL. Much like the filesystem layer, MySQL has no knowledge of any authentication performed by the web application, interpreting all actions sent by the web application as coming from the highly-privileged `webdb` user.

6.1.2 Authentication & Access Control Attacks

The fragile security architecture in today's web applications leads to two common problems, authentication bypass and access control check vulnerabilities.

Authentication bypass attacks occur when an attacker can fool the application into treating his or her requests as coming from an authenticated user, without having to present that

```
$res = mysql_query("SELECT * FROM articles  WHERE $_GET['search\_criteria']}")
```

Figure 6.2: Sample PHP code that may be vulnerable to SQL injection attacks

user's credentials, such as a password. A typical example of an authentication bypass vulnerability involves storing authentication state in an HTTP cookie without performing any server-side validation to ensure that the client-supplied cookie is valid. For example, many vulnerable web applications store only the username in the client's cookie when creating a new login session. A malicious user can then edit this cookie to change the username to the administrator, obtaining full administrator access. Even this seemingly simple problem affects many applications, including PHP iCalendar [88] and phpFastNews [87], both of which are discussed in more detail in the evaluation section.

Access control check vulnerabilities occur when an access check is missing or incorrectly performed in the application code, allowing an attacker to execute server-side operations that she might not be otherwise authorized to perform. For example, a web application may be compromised by an invalid access control check if an administrative control panel script does not verify that the web client is authenticated as the admin user. A malicious user can then use this script to reset other passwords, or even perform arbitrary SQL queries, depending on the contents of the script. These problems have been found in numerous applications, such as PhpStat [89].

Authentication and access control attacks often result in the same unfettered file and database access as traditional input validation vulnerabilities such as SQL injection and directory traversal. However, authentication and access control bugs are more difficult to detect, because their logic is application-specific, and they do not follow simple patterns that can be detected by simple analysis tools.

6.1.3 Other Web Application Attacks

Authentication and access control also play an important, but less direct role, in SQL injection [120], command injection, and directory traversal attacks. For example, the PHP code in Figure 6.2 places user-supplied search parameters into a SQL query without performing any sanitization checks. This can result in a SQL injection vulnerability; a malicious user could exploit it to execute arbitrary SQL statements on the database. The general approach to addressing these attacks is to validate all user input before it is used in any filesystem or database operations, and to disallow users from directly supplying SQL statements. These checks occur throughout the application, and any missing check can lead to a SQL injection or directory traversal vulnerability.

However, these kinds of attacks are effective only because the filesystem and database layers perform all operations with the privilege level of the web application rather than the current authenticated webapp user. If the filesystem and database access of a webapp user were restricted only to the resources that the user should legitimately access, input validation attacks would not be effective as malicious users would not be able not leverage these attacks to access unauthorized resources.

Furthermore, privileged users such as site administrators are often allowed to perform operations that could be interpreted as SQL injection, command injection, or directory traversal attacks. For example, popular PHP web applications such as DeluxeBB and phpMyAdmin allow administrators to execute arbitrary SQL commands. Alternatively, code in Figure 6.2 could be safe, as long as only administrative users are allowed to issue such search queries. This is the very definition of a SQL injection attack. However, these SQL injection vulnerabilities can only be exploited if the application fails to check that the user is authenticated as the administrator before issuing the SQL query. Thus, to properly judge whether a SQL injection attack is occurring, the security system must know which user is currently authenticated.

6.2 Authentication Inference

Web applications often have buggy implementations of authentication and access control, and no two applications have the exact same authentication framework. Rather than try to mandate the use of any particular authentication system, Nemesis prevents authentication and access control vulnerabilities by automatically inferring when a user has been safely authenticated, and then using this authentication information to automatically enforce access control rules on web application users. An overview of Nemesis and how it integrates

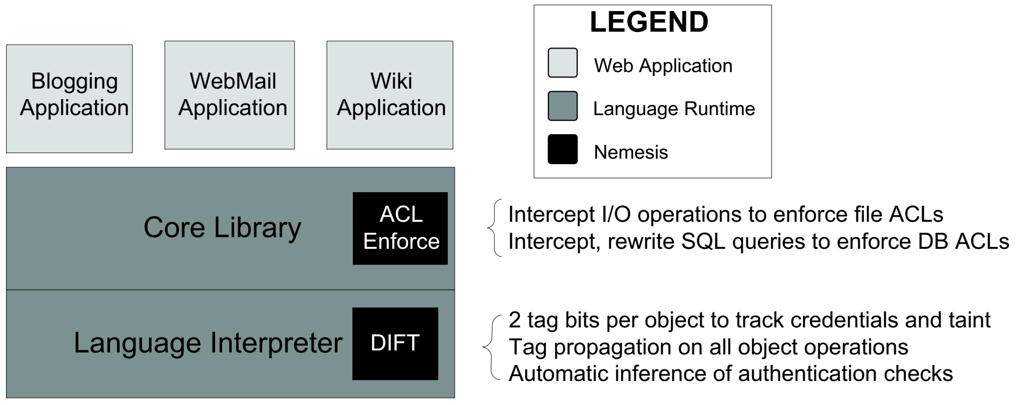

LEGEND
Web Application
Language Runtime
Nemesis
Blogging Application
WebMail Application
Wiki Application
Core Library
ACL Enforce
Language Interpreter
DIFT
Intercept I/O operations to enforce file ACLs
Intercept, rewrite SQL queries to enforce DB ACLs
2 tag bits per object to track credentials and taint
Tag propagation on all object operations
Automatic inference of authentication checks

Figure 6.3: Overview of Nemesis system architecture

into a web application software stack is presented in Figure 6.3. In this section, we describe how Nemesis performs *authentication inference*.

6.2.1 Shadow Authentication Overview

To prevent authentication bypass attacks, Nemesis must infer when authentication has occurred without depending on the correctness of application authentication systems., which are often buggy or vulnerable. To this end, Nemesis constructs a *shadow authentication system* that works alongside the application's existing authentication framework. In order to infer when user authentication has safely and correctly occurred, Nemesis requires the application developer to provide one annotation—namely, where the application stores user names and their known-good passwords (e.g. in a database table), or what external function it invokes to authenticate users (e.g. using LDAP or OpenID). Aside from this annotation, Nemesis is agnostic to the specific hash function or algorithm used to validate user-supplied credentials.

To determine when a user successfully authenticates, Nemesis uses DIFT. In particular, Nemesis keeps track of two tag bits for each data item in the application—a "credential" taint bit, indicating whether the data item represents a known-good password or other credential, and a "user input" taint bit, indicating whether the data item was supplied by the user as part of the HTTP request. User input includes all values supplied by the untrusted client, such as HTTP request headers, cookies, POST bodies, and URL parameters. Taint bits can be stored either per object (e.g., string), or per byte (e.g., string characters), depending on the needed level of precision and performance.

Nemesis must also track the flow of authentication credentials and user input during runtime code execution. Much like other DIFT systems [78, 40, 68], this is done by performing taint propagation in the language interpreter. Nemesis propagates both taint bits at runtime for all data operations, such as variable assignment, load, store, arithmetic, and string concatenation. The propagation rule we enforce is *union*: a destination operand's taint bit is set if it was set in any of the source operands. Since Nemesis is concerned with inferring authentication rather than addressing covert channels, implicit taint propagation across control flow is not considered. The remainder of this section describes how Nemesis uses these two taint bits to infer when successful authentication has occurred.

6.2.2 Creating a New Login Session

Web applications commonly authenticate a new user session by retrieving a username and password from a storage location (typically a database) and comparing these credentials to user input. Other applications may use a dedicated login server such as LDAP or Kerberos, and instead defer all authentication to the login server by invoking a special third-party library authentication function. We must infer and detect both authentication types.

As mentioned, Nemesis requires the programmer to specify where the application stores user credentials for authentication. Typical applications store password hashes in a database table, in which case the programmer should specify the name of this table and the column names containing the user names and passwords. For applications that defer authentication to an external login server, the programmer must provide Nemesis with the name of the authentication function (such as ldap_login), as well as the function arguments that represent the username and password, and the value that is returned by the function if authentication succeeds. In either case, the shadow authentication system uses this information to determine when the web application has safely authenticated a user.

Direct Password Authentication

When an application performs authentication via direct password comparisons, the application must read the username and password from an authentication storage location, and compare them to the user-supplied authentication credentials. Whenever the authentication storage location is read, our shadow authentication system records the username read as the current user under authentication, and sets the "credential" taint bit for the password string. In most web applications, a client can only authenticate as a single user at any given time. If an application allows clients to authenticate as multiple users at the same time, Nemesis would have to be extended to keep track of multiple candidate usernames, as well as multiple "credential" taint bits on all data items. However, we are not aware of a situation in which this occurs in practice.

When data tagged as "user input" is compared to data tagged as "credentials" using string equality or inequality operators, we assume that the application is checking whether a user-supplied password matches the one stored in the local password database. If the two strings are found to be equal, Nemesis records the web client as authenticated for the candidate username. We believe this is an accurate heuristic, because known-good credentials are the only objects in the system with the "credential" taint bit set, and only user input has the "user input" taint bit set. This technique even works when usernames and passwords are supplied via URL parameters (such as "magic URLs" which perform automatic logins in HotCRP) because all values supplied by clients, including URL parameters, are tagged as user input.

Tag bits are propagated across all common operations, allowing Nemesis to support standard password techniques such as cryptographic hashes and salting. Hashing is supported because cryptographic hash functions consist of operations such as array access and arithmetic computations, all of which propagate tag bits from inputs to outputs. Similarly, salting is supported because prepending a salt to a user-supplied password is done via string concatenation, an operation that propagates tag bits from source operands to the destination operand.

This approach allows us to infer user authentication by detecting when a user input string is compared and found to be equal to a password. This avoids any internal knowledge of the application, requiring only that the system administrator correctly specify the storage location of usernames and passwords. A web client will only be authenticated by our shadow authentication system if they know the password, because authentication occurs only when a user-supplied value is equal to a known password. Thus, our approach does not suffer from authentication vulnerabilities, such as allowing a user to log in if a magic URL parameter or cookie value is set.

Deferred Authentication to a Login Server

We use similar logic to detect authentication when using a login server. The web client is assumed to be authenticated if the third-party authentication function is called with a username and password marked as “user input”, and the function returns success. In this case, Nemesis sets the authenticated user to the username passed to this function. Nemesis checks to see if the username and password passed to this function are tainted in order to distinguish between credentials supplied by the web client and credentials supplied internally by the application. For example, phpMyAdmin uses MySQL’s built-in authentication code both to authenticate web clients, and to authenticate itself to the database for internal database queries [92]. Credentials used internally by the application should not be treated as the client’s credentials, and Nemesis ensures this by only accepting credentials which came from the web client. Applications that use single sign-on systems such as OpenID must use deferred authentication, as the third-party authentication server (e.g., OpenID Provider) performs the actual user authentication.

6.2.3 Resuming a Previous Login Session

As described in Section 6.1.1, web applications create login sessions by recording pertinent authentication information in cookies. This allows users to authenticate once, and then access the web application without having to authenticate each time a new page is loaded. Applications often write their own custom session management frameworks, and session management code is responsible for many authentication bypass vulnerabilities.

Fortunately, Nemesis does not require any per-application customization for session management. Instead, we use an entirely separate session management framework. When

Nemesis infers that user authentication has occurred (as described earlier in this section), a new cookie is created to record the shadow authentication credentials of the current web client. We do not interpret or attempt to validate any other cookies stored and used by the web application for session management. For all intents and purposes, session management in the web application and Nemesis are orthogonal. We refer to the cookie used for Nemesis session management as the *shadow cookie*. When Nemesis is presented with a valid shadow cookie, the current shadow authenticated user is set to the username specified in the cookie.

Shadow authentication cookies contain the shadow authenticated username of the current web user and an HMAC of the username computed using a private key kept on the server. The user cannot edit or change their shadow authentication cookie because the username HMAC will no longer match the username itself, and the user does not have the key used to compute the HMAC. This cookie is returned to the user, and stored along with any other authentication cookies created by the web application.

Our shadow authentication system detects a user safely resuming a prior login session if a valid shadow cookie is presented. The shadow authentication cookie is verified by recomputing the cookie HMAC based on the username from the cookie. If the recomputed HMAC and the HMAC from the cookie are identical, the user is successfully authenticated by our shadow authentication system. Nemesis distinguishes between shadow cookies from multiple applications running on the same server by using a different HMAC key for each application, and including a hash derived from the application's HMAC key in the name of the cookie.

In practice, when a user resumes a login session, the web application will validate the user's cookies and session file, and then authorize the user to access a privileged resource. When the privileged resource access is attempted, Nemesis will examine the user's shadow authentication credentials and search for valid shadow cookies. If a valid shadow cookie is found and verified to be safe, the user's shadow authentication credentials are updated. Nemesis then performs an access control check on the shadow authentication credentials using the web application access access control list (ACL).

6.2.4 Registering a New User

The last way a user may authenticate is to register as a new user. Nemesis infers that new user registration has occurred when a user is inserted into the authentication credential storage location. In practice, this is usually a SQL INSERT statement modifying the user authentication database table. The inserted username must be tainted as "user input", to ensure that this new user addition is occurring on behalf of the web client, and not because the web application needed to add a user for internal usage.

Once the username has been extracted and verified as tainted, the web client is then treated as authenticated for that username, and the appropriate session files and shadow authentication cookies are created. For the common case of a database table, this requires us to parse the SQL query, and determine if the query is an INSERT into the user table or not. If so, we extract the username field from the SQL statement.

6.2.5 Authentication Bypass Attacks

Shadow authentication information is only updated when the web client supplies valid user credentials, such as a password for a web application user, or when a valid shadow cookie is presented. During authentication bypass attacks, malicious users are authenticated by the web application without supplying valid credentials. Thus, when one of these attacks occurs, the web application will incorrectly authenticate the malicious web client, but shadow authentication information will not be updated.

While we could detect authentication bypass attacks by trying to discern when shadow authentication information differs from the authenticated state in the web application, this would depend on internal knowledge of each web application's code base. Authentication frameworks are often complex, and each web application typically creates its own framework, possibly spreading the current authentication information among multiple variables and complex data structures.

Instead, we note that the goal of any authentication bypass attack is to use the ill-gotten authentication to obtain unauthorized access to resources. These are exactly the resources that the current shadow authenticated user is not permitted to access. As explained in the next section, we can prevent authentication bypass attacks by detecting when the current shadow authenticated user tries to obtain unauthorized access to a system resource such as a file, directory, or database table. An authentication bypass attack is useless unless the attacker can leverage his stolen authentication credentials to access an unauthorized resource.

6.3 Authorization Enforcement

Both authentication and access control bypass vulnerabilities allow an attacker to perform operations that she would not be otherwise authorized to perform. The previous section described how Nemesis constructs a shadow authentication system to keep track of user authentication information despite application-level bugs. However, the shadow authentication system alone is not enough to prevent these attacks. This section describes how Nemesis mitigates the attacks by connecting its shadow authentication system with an access control system protecting the web application's database and file system.

To control which operations any given web user is allowed to perform, Nemesis allows the application developer to supply ACLs for files, directories, and database objects. Nemesis extends the core system library so that each database or file operation performs an ACL check. The ACL check ensures that the current shadow authenticated user is permitted by the web application ACL to execute the operation. This enforcement prevents access control bypass attacks. An attacker exploiting a missing or invalid access control check to perform a privileged operation will be foiled when Nemesis enforces the supplied ACL. This also mitigates authentication bypass attacks—even if an attacker can bypass the application's authentication system (e.g., due to a missing check in the application code), Nemesis will automatically perform ACL checks against the username provided by the shadow authentication system, which is not subject to authentication bypass attacks.

6.3.1 Access Control

In any web application, the authentication framework plays a critical role in access control decisions. There are often numerous, complex rules determining which resources (such as files, directories, or database tables, rows, or fields) can be accessed by a particular user. However, existing web applications do not have explicit, codified access control rules. Rather, each application has its own authentication system, and access control checks are interspersed throughout the application.

For example, many web applications have a privileged script used to manage the users of the web application. This script must only be accessed by the web application administrator, as it will likely contain logic to change the password of an arbitrary user and perform other privileged operations. To restrict access appropriately, the beginning of the script will contain an access control check to ensure that unauthorized users cannot access script functionality. This is actually an example of the policy, "only the administrator may access the admin.php script", or to rephrase such a policy in terms of the resources it affects, "only the administrator may modify the user table in the database". This policy is often never explicitly stated within the web application, and must instead be inferred from the authorization checks in the web application. Nemesis requires the developer or system administrator to explicitly provide an access control list based on knowledge of the application. Our prototype system and evaluation suggests that, in practice, this requires little programmer effort while providing significant security benefits. Note that a single developer or administrator needs to specify access control rules. Based on these rules, Nemesis will provide security checks for all application users.

File Access

Nemesis allows developers to restrict file or directory access to a particular shadow authenticated user. For example, a news application may only allow the administrator to update the news spool file. We can also restrict the set of valid operations that can be performed: read, write, or append. For directories, read permission is equivalent to listing the contents of the directory, while write permission allows files and subdirectories to be created. File access checks happen before any attempt to open a file or directory. These ACLs could be expressed by listing the files and access modes permitted for each user.

SQL Database Access

Nemesis allows web applications to restrict access to SQL tables. Access control rules specify the user, name of the SQL database table, and the type of access (INSERT, SELECT, DELETE, or UPDATE). For each SQL query, Nemesis must determine what tables will be accessed by the query, and whether the ACLs permit the user to perform the desired operation on those tables.

In addition to table-level access control, Nemesis also allows restricting access to individual rows in a SQL table, since applications often store data belonging to different users in the same table.

An ACL for a SQL row works by restricting a given SQL table to just those rows that should be accessible to the current user, much like a *view* in SQL terminology. Specifically, the ACL maps SQL table names and access types to an SQL predicate expression involving column names and values that constrain the kinds of rows the current user can access, where the values can be either fixed constants, or the current username from the shadow authentication system, evaluated at runtime. For example, a programmer can ensure that a user can only access their own profile by confining SQL queries on the profile table to those whose *user* column matches the current shadow username.

SELECT ACLs restrict the values returned by a SELECT SQL statement. DELETE and UPDATE query ACLs restrict the values modified by an UPDATE or DELETE statement, respectively. To enforce ACLs for these statements, Nemesis must rewrite the database query to append the field names and values from the ACL to the WHERE condition clause of the query. For example, a query to retrieve a user's private messages might be "SELECT * FROM messages WHERE recipient=$current_user", where $current_user is supplied by the application's authentication system. If attackers could fool the application's authentication system into setting $current_user to the name of a different user, they might be able to retrieve that user's messages.

Using Nemesis, the programmer can specify an ACL that only allows SELECTing rows whose sender or recipient column matches the current shadow user. As a result, if user Bob issues the query, Nemesis will transform it into the query "SELECT * FROM messages

WHERE recipient=$current_user AND (sender=Bob or recipient=Bob)". This mitigates the damage caused by any authentication bypass attacks.

Finally, INSERT statements do not read or modify existing rows in the database. Thus, access control for INSERT statements is governed solely by the table access control rules described earlier. However, sometimes developers may want to set a particular field to the current shadow authenticated user when a row is inserted into a table. Nemesis accomplishes this by rewriting the INSERT query to replace the value of the designated field with the current shadow authenticated user (or to add an additional field assignment if the designated field was not initialized by the INSERT statement).

Modifying INSERT queries has a number of real-world uses. Many database tables include a field which stores the username of the user who inserted the field. The administrator can choose to replace the value of this field with the shadow authenticated username, so that authentication flaws do not allow users to spoof the owner of a particular row in the database. For example, in the PHP forum application DeluxeBB, we can override the author name field in the table of database posts with the shadow authenticated username. This prevents malicious clients from spoofing the author when posting messages, which can occur if an authentication flaw allows attackers to authenticate as arbitrary users.

6.3.2 Enhancing Access Control with DIFT

Web applications often perform actions which are not authorized for the currently authenticated user. For example, in the PHP image gallery Linpha, users may inform the web application that they have lost their password. At this point, the web client is unauthenticated (as they have no valid password), but the web application changes the user's password to a random value, and e-mails the new password to the user's e-mail account. While one user should not generally be allowed to change the password of a different user, doing so is safe in this case because the application generates a fresh password not known to the requesting user, and only sends it via email to the owner's address.

One heuristic that helps us distinguish these two cases in practice is the taint status of the newly-supplied password. Clearly it would be inadvisable to allow an unauthenticated user to supply the new password for a different user's account, and such password values would have the “user input” taint under Nemesis. At the same time, our experience suggests that internally-generated passwords, which do not have the “user input” taint, correspond to password reset operations, and would be safe to allow.

To support this heuristic, we add one final parameter to all of the above access control rules: taint status. An ACL entry may specify, in addition to its other parameters, taint restrictions for the file contents or database query. For example, an ACL for Linpha allows the application to update the password field of the user table regardless of the authentication status, as long as the query is untainted. If the query is tainted, however, the ACL only allows updates to the row corresponding to the currently authenticated user.

6.3.3 Protecting Authentication Credentials

Additionally, there is one security rule that does not easily fit into our access control model, yet can be protected via DIFT. When a web client is authenticating to the web application, the application must read user credentials such as a password and use those credentials to authenticate the client. However, unauthenticated clients do not have permission to see passwords. A safe web application will ensure that these values are never leaked to the client. To prevent an information leak bug in the web application from resulting in password disclosure, Nemesis forbids any object that has the authentication credential DIFT tag bit set from being returned in any HTTP response. In our prototype, this rule has resulted in no false positives in practice. Nevertheless, we can easily modify this rule to allow passwords for a particular user to be returned in a HTTP response once the client is authenticated for that user. For example, this situation could arise if a secure e-mail service used the user’s password to decrypt e-mails, causing any displayed emails to be tagged with the password bit.

6.4 Prototype Implementation

We have implemented a proof-of-concept prototype of Nemesis by modifying the PHP interpreter. PHP is one of the most popular languages for web application development. However, the overall approach is not tied to PHP by design, and could be implemented for any other popular web application programming language. Our prototype is based on an existing DIFT PHP tainting project [130]. We extend this work to support authentication inference and authorization enforcement. We do not need Raksha's support for this prototype because Nemesis prevents only high-level PHP web application vulnerabilities, and thus can be implemented entirely within the PHP interpreter. Raksha would still be necessary if we were protecting against vulnerabilities outside of the PHP web application, such as buffer overflows in the OS or the PHP interpreter itself.

6.4.1 Tag Management

PHP is a dynamically typed language. Internally, all values in PHP are instances of a single type of structure known as a *zval*, which is stored as a tagged union. Integers, booleans, and strings are all instances of the *zval* struct. Aggregate data types such as arrays serve as hash tables mapping index values to *zval*s. Symbol tables are hash tables mapping variable names to *zval*s.

Our prototype stores taint information at the granularity of a *zval* object, which can be implemented without storage overhead in the PHP interpreter. Due to alignment restrictions enforced by GCC, the *zval* structure has a few unused bits, which is sufficient for us to store the two taint bits required by Nemesis.

By keeping track of taint at the object level, Nemesis assumes that the application will not combine different kinds of tagged credentials in the same object (e.g. by concatenating passwords from two different users together, or combining untrusted and authentication-based input into a single string). While we have found this assumption to hold in all encountered applications, a byte granularity tainting approach could be used to avoid this limitation if needed, and prior work has shown it practical to implement byte-level tainting in PHP [78]. When multiple objects are combined in our prototype, the result's taint bits are the union of the taint bits on all inputs. This works well for combining tainted and untainted data, such as concatenating an untainted salt with a tainted password (with the result being tainted), but can produce imprecise results when concatenating objects with two different classes of taint.

User input and password taint is propagated across all standard PHP operations, such as variable assignment, arithmetic, and string concatenation. Any value with password taint is forbidden from being returned to the user via echo, printf, or other output statements.

6.4.2 Tag Initialization

Any input from URL parameters (GET, PUT, etc), as well as from any cookies, is automatically tainted with the 'user input' taint bit. Currently, password taint initialization is done by manually inserting the taint initialization function call as soon as the password enters the system (e.g., from a database) as we have not yet implemented a full policy language for automated credential tainting. For a few of our experiments in Section 6.5 (phpFastNews, PHP iCalendar, Bilboblog), the admin password was stored in a configuration PHP script that was included by the application scripts at runtime. In this case, we instrumented the configuration script to set the password bit of the admin password variable in the script.

When the password taint is initialized, we also set a global variable to store the candidate username associated with the password, to keep track of the current username being authenticated. If authentication succeeds, the shadow authentication system uses this candidate username to set the global variable which stores the shadow authenticated user, as well as to initialize the shadow cookie. If a client starts authenticating a second time as a different user, the candidate username is reset to the new value, but the authenticated username is not affected until authentication succeeds.

Additionally, due to an implementation artifact in the PHP $setcookie()$ function, we also record shadow authentication in the PHP built-in session when appropriate. PHP forbids new cookies to be added to the HTTP response once the application has placed part of the HTML body in the response output buffer. In an application that uses PHP sessions, the cookie only stores the session ID and all authentication information is stored in session files on the server. These applications may output part of the HTML body before authentication is complete. We correctly handle this case by storing shadow authentication credentials in the server session file if the application has begun a PHP session. When validating and recovering shadow cookies for authentication purposes, we also check the session file associated with the current user for shadow authentication credentials. This approach relies on PHP safely storing session files. This is a reasonable assumption, as PHP session files are stored locally on the server in a temporary directory.

6.4.3 Authentication Checks

When checking the authentication status of a user, we first check the global variable that indicates the current shadow authenticated user. This variable is set if the user has just begun a new session and has been directly authenticated via password comparison or deferred authentication to a login server. If this variable is not set, we check to see if shadow authentication information is stored in the current session file (if any). Finally, we check to determine if the user has presented a shadow authentication cookie, and if so, we validate the cookie and extract the authentication credentials. If none of these checks succeeds, the user is treated as unauthenticated.

6.4.4 Password Comparison Authentication Inference

Authentication inference for password comparisons is performed by modifying the PHP interpreter's string comparison equality and inequality operators. When one of these string comparisons executes, we perform a check to see if the two string operands were determined to be equal. If the strings were equal, we then check their tags, and if one string has only the authentication credential tag bit set, and the other string has only the user input tag bit set, then we determine that a successful shadow authentication has occurred. In all of our experiments, only PhpMyAdmin used a form of authentication which did not rely on password string comparison, and our handling of this case is discussed in Section 6.5.

6.4.5 Access Control Checks

We perform access control checks for files by checking the current authenticated user against a list of accessible files (and file modes) on each file access. Similarly, we restrict SQL queries by checking if the currently authenticated user is authorized to access the table, and by appending additional WHERE clause predicates to scope the effect of the query to rows allowed for the current user.

Due to time constraints, we manually inserted these checks into applications based on the ACL needed by the application. ACLs that placed constraints on field values of a database row required simple query modifications to test if the field value met the constraints in the ACL.

In a full-fledged design, the SQL queries should be parsed, analyzed for the appropriate information, and rewritten if needed to enforce additional security guarantees (e.g., restrict rows modified to be only those created by the current authenticated user). Depending on the database used, query rewriting may also be partially or totally implemented using database views and triggers [83, 114].

6.4.6 SQL Injection

Shadow authentication is necessary to prevent authentication bypass attacks and enforce our ACL rules. However, it can also be used to prevent false positives in DIFT SQL injection protection analyses. The most robust form of SQL injection protection [120] forbids tainted keywords or operators, and enforces the rule that tainted data may never change the parse tree of a query.

Our current approach does not support byte granularity taint, and thus we must approximate this analysis. We introduce a third taint bit in the *zval* which we use to denote user input that may be interpreted as a SQL keyword or operator. We scan all user input at startup (GET, POST, COOKIE superglobal arrays, etc) and set this bit only for those user input values that contain a SQL keyword or operator. SQL quoting functions, such as `mysql_real_escape_string()`, clear this tag bit. Any attempt to execute a SQL query with the unsafe SQL tag bit set is reported as a SQL injection attack.

We use this SQL injection policy to confirm that DIFT SQL Injection has false positives on real-world web applications. This is because DIFT treats all user input as untrusted, but some web applications allow privileged users such as the admin to submit full SQL queries. DIFT treats admin-submitted SQL queries as a SQL injection attack becaues it does not have any notion of users or privileges, treating all information either as totally trusted or totally untrusted. As discussed in Section 6.5, we eliminate all encountered false positives using authentication policies which restrict SQL injection protection to users that are not

Program	Lines of code in application	Lines of code for authentication inference	Number of ACL checks	Lines of code for ACL checks	Vulnerabilities prevented
PHP iCalendar	13500	3	8	22	Authentication bypass
phpStat (IM Stats)	12700	3	10	17	Missing access check
Bilboblog	2000	3	4	11	Invalid access check
phpFastNews	500	5	2	17	Authentication bypass
Linpha Image Gallery	50000	15	17	49	Authentication bypass
DeluxeBB Web Forum	22000	6	82	143	Missing access check

Table 6.1: Applications used to evaluate Nemesis.

shadow authenticated as the admin user. We have confirmed that all of these false positives are due to a lack of authentication information, and not due to any approximations made in our SQL injection protection implementation.

6.5 Experimental Results

To validate Nemesis, we used our prototype to protect a wide range of vulnerable real-world PHP applications from authentication and access control bypass attacks. A summary of the applications and their vulnerabilities is given in Table 6.1, along with the lines of code that were added or modified in order to protect them.

For each application, we had to specify where the application stores its username and password database, or which function it invokes to authenticate users. This step is quite simple for all applications, and the "authentication inference" column indicates the amount of code we had to add to each application to specify the table used to store known-good passwords, and to taint the passwords with the "credential" taint bit.

We also specified ACLs on files and database tables to protect them from unauthorized accesses; the number of access control rules for each application is shown in the table. As explained in Section 6.4, we currently enforce ACLs via explicitly inserted checks, which slightly increases the lines of code needed to implement the check (also shown in the table). As we develop a full MySQL parser and query rewriter, we expect the lines of code needed for these checks to drop further.

We validated our rules by using each web application extensively to ensure there are no false positives, and then verified that our rules prevented real-world attacks previously discovered in the applications. We also verified that our shadow authentication information is able to prevent false positives in DIFT SQL injection analyses for both the DeluxeBB and phpMyAdmin applications. These were the only applications that allowed administrators or other privileged users to directly submit SQL queries.

6.5.1 PHP iCalendar

PHP iCalendar is a PHP web application for presenting calendar information to users. The webapp administrator is authenticated using a configuration file which stores the admin username and password. Our ACL for PHP iCalendar allows users read access to various template files, language files, and all of the calendars. In addition, caches containing parsed calendars can be read or written by any user. The admin user is able to write, create, and delete calendar files, as well as read any uploaded calendars from the uploads directory. We added 8 authorization checks to enforce our ACL for PHP iCalendar.

An authentication bypass vulnerability occurs in PHP iCalendar because a script in the admin subdirectory incorrectly validates a login cookie when resuming a session [88]. This vulnerability allows a malicious user to forge a cookie that will cause her to be authenticated as the admin user.

Using Nemesis, when an attacker attempts to exploit the authentication bypass attack, she will find that her shadow authentication username is not affected by the attack. This is because shadow authentication uses its own secure form of cookie authentication, and stores its credentials separately from the rest of the web application. When the attacker attempts to use the admin scripts to perform any actions requiring admin access, such as deleting a calendar, a security violation is reported because the shadow authentication username will not be ’admin’, and the ACL will prevent that username from performing administrative operations.

6.5.2 Phpstat

Phpstat is an application for presenting a database of IM statistics to users, such as summaries and logs of their IM conversations. Phpstat stores its authentication credentials in a database table.

The access control list for PhpStat allows users to read and write various cache files, as well as read the statistics database tables. Users may also read profile information about any other user, but the value of the password field may never be sent back to the Web client. The administrative user is also allowed to create users by inserting into or updating the users table, as well as all of the various statistics tables. We added 10 authorization checks to enforce our ACL for PhpStat.

A security vulnerability exists in PhpStat because an installation script will reset the administrator password if a particular URL parameter is given. This behavior occurs without any access control checks, allowing any user to reset the admin password to a user-specified value [89]. Successful exploitation grants the attacker full administrative privileges to the Phpstat. Using Nemesis, when this attack occurs, the attacker will not be shadow authenticated as the admin, and any attempts to execute a SQL query that changes the password of the administrator are denied by our ACL rules. Only users shadow authenticated as the admin may change passwords.

6.5.3 Bilboblog

Bilboblog is a simple PHP blogging application that authenticates its administrator using a username and password from a configuration file.

Our ACL for bilboblog permits all users to read and write blog caching directories, and read any posted articles from the article database table. Only the administrator is allowed to modify or insert new entries into the articles database table. Bilboblog has an invalid access control check vulnerability because one of its scripts, if directly accessed, uses uninitialized variables to authenticate the admin user [4]. We added 4 access control checks to enforce our ACL for bilboblog.

In PHP, if the register_globals option is set, uninitialized variables may be initialized at startup by user-supplied URL parameters [90]. This allows a malicious user to supply the administrator username and password against which the login will be authenticated. The attacker may simply choose a username and password, access the login script with these credentials encoded as URL parameters, and then input the same username and password when prompted by the script. This attack grants the attacker full administrative access to Bilboblog.

This kind of attack does not affect shadow authentication. A user is shadow authenticated only if their input is compared against a valid password. This attack instead compares user input against a URL parameter. Passwords read from the configuration file have the password bit set, but URL parameters do not. Thus, no shadow authentication occurs when this attack succeeds. If an attacker exploits this vulnerability on a system protected by our prototype, she will find herself unable to perform any privileged actions as the admin user. Any attempt to update, delete, or modify an article will be prevented by our prototype, as the current user will not be shadow authenticated as the administrator.

6.5.4 phpFastNews

PhpFastNews is a PHP application for displaying news stories. It performs authentication via a configuration file with username and password information. This web application displays a news spool to users. Our ACL for phpFastNews allows users to read the news spool, and restricts write access to the administrator. We added 2 access control checks to enforce our ACL for phpFastNews.

An authentication bypass vulnerability occurs in phpFastNews due to insecure cookie validation [87], much like in PHP iCalendar. If a particular cookie value is set, the user is automatically authenticated as the administrator without supplying the administrator's password. The attacker need only forge the appropriate cookie and full admin access is granted.

When an authentication bypass attack occurs, our prototype will prevent any attempt to perform administrator-restricted actions such as updating the news spool because the current user is not shadow authenticated as the admin.

6.5.5 Linpha

Linpha is a PHP web gallery application, used to display directories of images to web users. It authenticates its users via a database table.

Our ACL for Linpha allows users to read files from the images directory, read and write files in the temporary and cache directories, and insert entries into the thumbnails table. Users may also read from the various settings, group, and configuration tables. The administrator may update or insert into the users table, as well as the settings, groups, and categories tables. Dealing with access by non-admin users to the user table is the most complex part of the Linpha ACL, and is our first example of a database row ACL. Any user may read from the user table, with the usual restriction that passwords may never be output to the Web client via echo, print, or related commands.

Users may also update entries in the user table. Updating the password field must be restricted so that a malicious user cannot update the other passwords. This safety restriction can be enforced by ensuring that only user table rows that have a username field equal to the current shadow authenticated user can be modified. The exception to this rule is when the new password is untainted. This can occur only when the web application has internally generated the new user password without using user input. We allow these queries even when they affect the password of a user that is not the current shadow authenticated user because they are used for lost password recovery.

In Linpha, users may lose their password, in which case Linpha resets their password to an internally generated value, and e-mails this password to the user. This causes an arbitrary user's password to be changed on the behalf of a user who isn't even authenticated. However, we can distinguish this safe and reasonable behavior from an attack by a user attempting to change another user's password by examining the taint of the new password value in the SQL query. Thus, we allow non-admin users to update the password field of the user table if the password query is untainted, or if the username of the modified row is equal to the current shadow authenticated user. Overall, we added 17 authorization checks to enforce all of our ACLs for Linpha.

Linpha also has an authentication bypass vulnerability because one of its scripts has a SQL injection vulnerability in the SQL query used to validate login information from user cookies [62]. Successful exploitation of this vulnerability grants the attacker full administrative access to Linpha. For this experiment, we disabled SQL injection protection provided by the tainting framework we used to implement the Nemesis prototype [130], to allow the user to submit a malicious SQL query in order to bypass authentication entirely.

Using Nemesis, once a user has exploited this authentication bypass vulnerability, she may access the various administration scripts. However, any attempt to actually use these scripts to perform activities that are reserved for the admin user will fail, because the current shadow authenticated user will not be set to admin, and our ACLs will correspondingly deny any admin-restricted actions.

6.5.6 DeluxeBB

DeluxeBB is a PHP web forum application that supports a wide range of features, such as an administration console, multiple forums, and private message communication between forum users. Authentication is performed using a table from a MySQL database.

DeluxeBB has the most intricate ACL of any application in our experiments. All users in DeluxeBB may read and write files in the attachment directory, and the admin user may also write to system log files. Non-admin users in DeluxeBB may read the various configuration and settings tables. Admin users can also write these tables, as well as perform unrestricted modifications to the user table. DeluxeBB treats user table updates and lost password modifications in the same manner as Linpha, and we use the equivalent ACL to protect the user table from non-admin modifications and updates.

DeluxeBB allows unauthenticated users to register via a form, and thus unauthenticated users are allowed to perform inserts into the user table. As described in Section 6.2.4, inserting a user into the user table results in shadow authentication with the credentials of the inserted user.

The novel and interesting part of the ACLs for DeluxeBB are the treatment of posts, thread creation, and private messages. All inserts into the post, thread creation, or private message tables are rewritten to use the shadow authenticated user as the value for the author field (or the sender field, in the case of a private message). The only exception is when a query is totally untainted. For example, when a new user registers, a welcome message is sent from a fake system mailer user. As this query is totally untainted, we allow it to be inserted into the private message table, despite the fact that the identity of the sender is clearly forged. We added fields to the post and thread tables to store the username of the current shadow authenticated user, as these tables did not directly store the author's username. We then explicitly instrumented all SQL INSERT statements into these tables to append this information accordingly.

Any user may read from the thread or post databases. However, our ACL rules further constrain reads from the private message database. A row may only be read from the private message database if the ’from’ or ’to’ fields of the row are equal to the current shadow authenticated user. We manually instrumented all SELECT queries from the private message table to add this condition to the WHERE clause of the query. In total, we modified 16 SQL queries to enforce both our private message protection and our INSERT rules to prevent spoofing messages, threads, and posts. We also inserted 82 authorization checks to enforce the rest of the ACL.

A vulnerability exists in the private message script of this application [27]. This script incorrectly validates cookies, missing a vital authentication check. This allows an attacker to forge a cookie and be treated as an arbitrary web application user by the script. Successful exploitation of this vulnerability gives an attacker the ability to access any user’s private messages.

Using Nemesis, when this attack is exploited, the attacker can fool the private message script into thinking he is an arbitrary user due to a missing access control check. The shadow authentication for the attack still has the last safe, correct authenticated username, and is not affected by the attack. Thus, the attacker is unable to access any unauthorized messages, because our ACL rules only allow a user to retrieve messages from the private message table when the sender or recipient field of the message is equal to the current shadow authenticated user. Similarly, the attacker cannot abuse the private message script to forge messages, as our ACLs constrain any messages inserted into the private message table to have the sender field set to the current shadow authenticated username.

DeluxeBB allows admin users to execute arbitrary SQL queries. We verified that this results in false positives in existing DIFT SQL injection protection analyses. After adding an ACL allowing SQL injection for web clients shadow authenticated as the admin user, all SQL injection false positives were eliminated.

6.5.7 PhpMyAdmin

PhpMyAdmin is a popular web application used to remotely administer and manage MySQL databases. This application does not build its own authentication system; instead, it checks usernames and passwords against MySQL's own user database. A web client is validated only if the underlying MySQL database accepts their username and password.

We treat the MySQL database connection function as a third-party authentication function as detailed in Section 6.2.2. We instrumented the call to the MySQL database connection function to perform shadow authentication, authenticating a user if the username and password supplied to the database are both tainted, and if the login was successful.

The ACL for phpMyAdmin is very different from other web applications, as phpMyAdmin is intended to provide an authenticated user with unrestricted access to the underlying database. The only ACL we include is a rule allowing authenticated users to submit full SQL database queries. We implemented this by modifying our SQL injection protection policy defined in Section 6.4.6 to treat tainted SQL operators in user input as unsafe only when the current user was unauthenticated. Without this policy, any attempt to submit a query as an authenticated user results in a false positive in the DIFT SQL injection protection policy. We confirmed that adding this ACL removes all observed SQL injection false positives, while still preventing unauthenticated users from submitting SQL queries.

6.5.8 Performance

We also conducted performance tests on our prototype implementation, measuring overhead against an unmodified version of PHP. We used the bench.php microbenchmark distributed with PHP, where our overhead was 2.9% compared to unmodified PHP. This is on par with prior results reported by object-level PHP tainting projects [130]. However, bench.php is a microbenchmark which performs CPU-intensive operations. Web applications often are network or I/O-bound, reducing the real-world performance impact of our information flow tracking and access checks.

To measure performance overhead of our prototype for web applications, we used the Rice University Bidding System (RUBiS) [106]. RUBiS is a web application benchmark suite, and has a PHP implementation with approximately 2,100 lines of code. Our ACL for

RUBiS prevents users from impersonating another user when placing bids, purchasing an item, or posting a bid comment. Three access checks were added to enforce this ACL. We compared latency and throughput for our prototype and an unmodified PHP, and found no discernible performance overhead.

Chapter 7

Designing Systems for DIFT

Academic researchers have proposed a number of hardware [24, 121, 12, 131, 49] and software [40, 77, 26] DIFT implementations. However, little has been published on how to design DIFT systems. In this chapter we describe methods for designing DIFT systems, exploring design options and their their associated tradeoffs.

7.1 DIFT Design Overview

The goal of a DIFT software security system is to prevent security vulnerabilities in unmodified software applications. A DIFT system designer must balance important design criteria such as backwards compatibility, performance overhead, supported languages, and cost.

In our experience, designing a DIFT system consists of the following steps:

1. *Create a Thread Model* - Determine which applications are being protected and which vulnerabilities must be addressed

2. *Develop DIFT Policies* - For each attack DIFT must prevent, develop a DIFT policy which avoids real-world false positives and minimizes false negatives.

3. *Design DIFT System Architecture* - Design a DIFT architecture to support your policies, balancing safety, flexibility, cost, and other factors.

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.

The next three sections focus on the three-step process of designing a DIFT system, providing practical advice and guidance from our experience building Raksha and Nemesis. We end this chapter with a final section discussing this problem from a different angle: how language designers can make DIFT (and other forms of dynamic analysis) much easier to implement.

7.2 Threat Model

At a high level, DIFT systems are designed to prevent security attacks on unmodified applications. However, when designing a DIFT system, a precise and unambiguous threat model must be developed to precisely enumerate which security vulnerabilities are to be addressed. Deciding which security problems to address has a significant effect on the design of the DIFT policies and system architecture.

The DIFT system designer must determine *which* programs are being protected, and *what* vulnerabilities the protection must defend against. When considering programs, there are many important issues to consider. Is the DIFT system intended to protect all applications, or a subset of available applications which have common behavior or APIs (e.g., JDBC or Servlets)? Should the operating system be protected as well? Are all protected applications written in the same language? It may be more economically feasible to support the most popular languages on the market and implement DIFT purely in software, than to support all languages by implementing DIFT in hardware or using a dynamic binary translator.

This decision will likely be influenced by the intended purpose is for the DIFT system. For example, designing a DIFT system that is intended for high-performance production use against a set of common web vulnerabilities in an enterprise web application programming language will require different tradeoffs than a system intended for research purposes that may need to quickly evaluate a wide range of policies and vulnerability types in arbitrary languages. In the former case, the developer will build a less flexible, high-performance system optimized for a single target language. It is likely that there will be a fixed set of highly optimized policies, and that the the system will likely integrate tightly with the language interpreter or runtime, and may even be implemented in hardware. In the latter case, a researcher will likely prefer flexibility to absolute performance, and will be better served by a design that is language neutral or at least easy to port between the most interesting and common languages. Policies may be loaded at runtime via plugins, and the design may be loosely coupled to the initial target language or languages.

Often the choice of which applications are protected will have significant influence on the vulnerabilities considered. For example, if all applications are written in type-safe languages such as Java and PHP, there will be little need for a buffer overflow prevention policy. Due to cost and time requirements, it may be prudent to begin by supporting the most critical vulnerabilities faced by the protected programs, such as SQL injection and cross-site scripting in web applications, rather than attempting to address every vulnerability in the first release of the DIFT system.

7.3 DIFT Policies

Once the applications and vulnerabilities have been identified, DIFT policies should be formulated to protect applications from security attacks that exploit the vulnerabilities. Typically this will involve considering each vulnerability in turn, although it may be the case that the designer discovers a single policy that can be re-used to prevent multiple vulnerabilities.

If a vulnerability has already been explored by other researchers in a similar environment, the bulk of the policy design work may already be complete. Academic researchers have proposed DIFT policies for common web vulnerabilities [119, 68, 77] such as SQL injection, cross-site scripting, and directory. Chapter 5 provides a comprehensive DIFT policy for buffer overflow prevention.

When examining existing policies, the runtime environment in which the policy is applied is of critical importance. Policies applied to x86 assembly may differ from policies applied to PHP due to differences in the runtime environment and its supported operations. Therefore, if a DIFT system design is being considered for a new language or computer architecture, there may be corner cases not encountered in prior work. In this case, designers can use prior work as a starting point, but a thorough review must be performed to ensure that the new environment does not negate or modify assumptions made in previous work. Academic researchers have proposed DIFT architectures for the Intel x86 [11, 131, 100, 76], SPARC (this dissertation), Java [40], PHP [77], and C [55].

If the vulnerability is sufficiently novel or if existing DIFT solutions do not meet the system requirements (compatibility, performance, etc.), then the system designer must develop a novel DIFT protection policy. The rest of this section details the process by which DIFT policies are designed, providing tips and techniques we have used when during policy development.

7.3.1 Define the Vulnerability Using DIFT

The first step to preventing an attack is to define the vulnerability itself, specifying the definition in terms of information flow if possible. Often, descriptions of security vulnerabilities used in practice are loose and ad-hoc. In some situations, a precise definition specified in terms of information flow immediately results in a DIFT policy, and policy design is almost complete. For example, the SQL injection definition in [119] describes SQL injection as an attack occurring when untrusted input influences the parse tree of a SQL query. That definition immediately lends itself to a DIFT-based solution: intercept all calls to SQL query functions and ensure that tainted data in the SQL query string does not affect the parse tree of the query.

Not all vulnerabilities have a simple or easy definition in terms of information flow. If a vulnerability seems particularly difficult to define, solving a complementary or related problem may lead to the correct solution. For example, when preventing authentication bypass vulnerabilities with Nemesis, we do not use DIFT to prevent the authentication bypass vulnerability itself. Instead, we consider the problem's complement: rather than identifying authentication bypass, we use DIFT to infer when authentication has been performed safely. Then we can detect authentication bypass attacks by applying ACLs to the current Nemesis-authenticated user. The ACL checks effectively prevent the privilege escalation caused by authentication bypass attacks, rendering such attacks useless.

7.3.2 Examine Applications & Vulnerabilities

A better understanding of vulnerabilities and potential DIFT solutions may be obtained by studying applications, particularly vulnerable portions of real-world applications. Determine the *taint sources*, *taint sinks*, and *taint propagators* in the application that allowed untrusted input to reach and exploit the vulnerable code. Ensure that DIFT rules can be developed to trace this information flow, and begin to identify security invariants (such as 'no untrusted SQL commands in a query') that can be enforced using DIFT to prevent attacks.

It is also useful to identify rules that are enforced by the runtime environment. The OS system call interface, Application Binary Interface, language specification, bytecode format, executable format, or other specifications may enforce invariants that are crucial for preventing a particular vulnerability, particularly when protecting unmodified applications with no debugging information. For example, our buffer overflow policy in Chapter 5 relies on the Linux system call interface as well as the relocation table entry formats used by the ELF object file specification. Such specification-based rules can be of crucial importance if they cannot be broken (or at least are not broken in practice, even in high-performance legacy code).

7.3.3 Examine Exploits & Existing Defenses

It is important to study how attackers compromise vulnerable applications, and determine the assumptions their exploits rely on. The policy designer must distinguish between assumptions that can be easily changed by an intelligent attacker and assumptions that are fundamentally required for the attacker to successfully exploit a given vulnerability. Existing defenses and their successes and failures in preventing attacks may also provide useful information.

For example, our buffer overflow policy in Chapter 5 is powerful and comprehensive because buffer overflow attacks rely on overwriting pointer values. This claim is validated by studying buffer overflow attacks, both simple and complex exploits, and by examining the memory layouts of real-world vulnerable applications. We also benefited from previous research on the safety of a prior non-DIFT buffer overflow protection mechanism, Address Space Layout Randomization (ASLR). ASLR provides a limited form of pointer overwrite protection by randomizing the base addresses of memory regions (and thus of pointer values). By studying the existing successful attacks against ASLR as well as the successes ASLR has had in preventing real-world attacks, we gained a better understanding of the effectiveness of pointer overwrite protection. For example, all prevalent ASLR bypass techniques centered on defeating the randomization rather than overwriting non-pointer data, indicating that pointer overwrites were crucial to common buffer overflow exploits.

7.3.4 Determine Tag Format

Once the set of policies is understood, the tag format itself must be designed. In the simplest case, policies may require only the ability to track untrusted information flow and interpose on method calls, requiring only a single taint bit.

Tags may be fixed-length or variable-length, and controlled by a single policy or split among multiple policies, where each policy controls a slice of the tag. Most policies preventing server-side input validation vulnerabilities require only a single tag bit as they specify rules and constraints based on taint tracking. A few policies, such as Pointer Injection, may require two tag bits as they track multiple forms of information flow.

Server-side vulnerability prevention usually has a fixed number of information flow types, and thus gains significant performance benefits by using fixed-length tags. When preventing vulnerabilities in production, performance outweighs any of the debugging or introspection benefits of using huge, variable-length tags. With the set of policies and the threat model already determined, a fixed tag length to support all required policies can be computed.

In the common case, combinations of tag policies can be supported by using a small, fixed number of tag bits (typically 1-4) supporting the minimum number of policies required to prevent all of the vulnerabilities in the threat model. Policies may be able to share a tag bit if they have identical check and propagation rules, or if the rules are very similar and the two policies can be distinguished at runtime (i.e. if they have different taint sinks). Expensive variable-length tags may be required if non-security policies such as lockset-based race detection [109, 11] must be supported, or if the system is designed with research in mind and flexibility is more important than performance.

7.3.5 Determine Tag Granularity

With a tag format determined, the designer should next determine the granularity of DIFT tags. Tags can implemented at many different granularities, and determining the appropriate granularity often has a significant effect on the performance and cost of the overall DIFT system. Tags may be implemented at a per-byte, per-word, per-object, or mult-granularity basis.

The finest granularity tags are byte-level tags, where a tag is associated with each byte. This will be necessary in DIFT systems where vulnerability prevention may depend upon the tag state of characters or byte-level types. This approach should be used only if bytes within the same word could have different tag values in real-world applications. If this is not true, then word granularity (or coarser) tags should suffice. For example, cross-site scripting detection will likely require byte-level taint tracking. This is because HTML documents are often stored as an array of characters and manipulated on a per-character basis during program execution. There is no guarantee that bytes within a word will have the same tag because user input and application HTML are often concatenated together at arbitrary places.

DIFT developers may also use word granularity tags, where the tag is associated with a naturally aligned word rather than a byte. This granularity should be used protecting pointers or other word-sized data The major object file format specifications [126, 67] guarantee that word-sized data will be naturally aligned. The buffer overflow policy described in Chapter 5 is an example of a DIFT policy that requires only word-granularity tags. While more space-efficient than byte granularity tags, word granularity tags introduce the difficulty of dealing with sub-word writes. The policy designer must determine if writing a tainted byte to an untainted word will result in a tainted word, or vice versa. However, if the policies can work effectively with word-granularity protection, tag storage may be decreased from one tag per byte to one tag per word.

Recently, researchers performing DIFT on Java have used object-level tags [40]. This granularity requires all applications to be written in an object-oriented, type-safe language. In this tag storage design, tags are associated with objects. The tags themselves may not necessarily be a single bit. For example, a variable-length tag could be used to store the taint status of every character in a String object, allowing for byte-level taint precision (but only for the String object, rather than all bytes).

Object-level tainting can be very fast and results in much lower storage overhead than byte or word-granularity tags. However, this approach may result in significant safety compromises in some languages when compared with byte or word-granularity tags. The reason for this is that any tag propagation that occurs outside of the monitored objects will *not* be tracked, which can cause this approach to miss malicious security attacks. For example, the object-level taint tracking system described in [40] would lose track of taint information if the user copied a String object character-by-character, or converted a String to a character array before a security-sensitive operation. This is because in Java, characters are not objects, and array objects cannot be instrumented or modified easily as they are implemented natively by the JVM. If you choose an object granularity tainting approach, be sure that your tag sources, tag propagators, and tag sinks *all* use objects with which you associate tags, and that application developers do not rely on primitives (e.g., characters) that you cannot instrument.

Finally, multi-granularity tag support is possible [121]. Tags have been observed to have very significant spatial locality. To reduce tag storage requirements, researchers have stored tags at multiple granularities, starting first with very coarse, page-granularity tags and then allocating successively finer tags as needed until byte granularity is reached [121, 72]. A multi-granularity scheme can afford for the default tag size to be very coarse, as the uncommon case of byte-granularity tags is addressed by allocating fine-grained tags on an on-demand basis.

7.3.6 Determine Check & Propagate Rules

The next step in producing a DIFT policy is to determine the precise tag check and propagate rules. DIFT policies are a balance struck between *safety* and *compatibility*. A safe policy provides comprehensive protection against security attacks, while a compatible policy does not erroneously report legitimate traffic as a security attack. In other words, a balance must be struck between false positives and false negatives. An ideal analysis would have neither, but in practice some vulnerabilities such as buffer overflow protection on unmodified binaries do not have a known perfect solution using DIFT or any other technique.

The definition of information flow itself is also ambiguous [75], and represents a balancing act between false positives and false negatives. All major DIFT systems treat direct data flow (e.g., assignment, arithmetic expressions, and so forth) as information flow. However, the treatment of load/store addresses (does loading from a tainted address produce tainted output) and branch conditions depends upon the implementation. Tracking information flow via branch conditions results in a significant number of false positives and is not done in practice by any DIFT system that prevents input validation attacks. Similarly, most systems today do not fully propagate load/store address taint, although there are legitimate cases where this should be done to prevent input validation attacks in certain situations.

False positives often result in an accidental denial-of-service attack on the protected application, preventing legitimate users from accessing the application. This may be significantly more costly to a corporation than a minor risk of false negatives. It is the role of the designer to determine how to balance the two. False negative risk in DIFT is mitigated because the precision of DIFT systems depends on the (presumably non-malicious) application developer's code, not on the malicious input itself. The current state of the art DIFT policies have both no real-world false positives, and few false negatives. Thus, the correct answer when faced with a choice between false positives and false negatives may be to spend additional time researching to see if a novel policy can be discovered, rather than to significantly compromise safety or compatibility.

7.3.7 Ignore Validation Operations

We do not recommend creating DIFT policies to recognize and enforce application validation operations. Many initial DIFT policies identified validation operations that should be applied to user input, and then threw a security exception if user input reached a *taint sink* without first passing through a validation function. For example, a SQL injection policy might determine the safe quoting functions provided by the vendor (such as *mysql_real_escape_string()* in PHP), and require all user input to be passed to a SQL quoting function before being used in a SQL query. While this may seem like a natural method for building DIFT policies, this kind of policy design is fundamentally flawed for a number of reasons.

In our experience, many applications write their own extensive set of validation routines, even when well-tested, third-party code is already available. This behavior will result in false positives, since the DIFT system is unaware of the application-specific validation operations.

Furthermore, for performance reasons, application developers often identify corner cases where some or all validation checks can be elided. This will again result in false positives, as the DIFT system may not have the application-specific knowledge necessary to determine when a validation check may be safely elided. Section 5.1.1 describes a corner case where a bounds check is not required which foiled all previous attempts at developing a robust buffer overflow protection policy. Our buffer overflow solution described in Chapter 5 resolved this issue by using a novel approach that did not rely on bounds check recognition.

Even in the case where validation operations are explicitly defined (either via a specification or an API call invoked by the user whenever validation occurs), cannot be elided, and are used everywhere in safe applications, DIFT policies that enforce validation functions are still an unattractive solution. This is because they will throw a security exception whenever a *vulnerability* is detected, not when an *attack* is detected. Use of benign input that is not validated will result in the same security exception as malicious input. In both malicious and non-malicious cases, untrusted data reaches a taint sink without passing through a validation function. This kind of DIFT policy effectively turns security vulnerabilities, including those that are not being actively exploited, into *denial of service attacks*.

DIFT policies should be based around security invariants that prevent *attacks*, not vulnerabilities. Vulnerable code accessed with non-malicious parameters should be safely allowed to continue (ideally, detected so that a warning can be sent to the developers). Only malicious attacks against vulnerable code should result in a security exception.

7.3.8 DIFT Is Not Just Taint Tracking

Dynamic Information Flow Tracking is a versatile tool, and should not be relegated to only defeating straightforward input validation attacks. Traditionally, DIFT is used for taint tracking, monitoring the flow of untrusted information during program execution. However, DIFT can be used to track other forms of information that can help a DIFT policy prevent security attacks. This is especially useful if a powerful security invariant depends on multiple kinds of information flow.

For example, previous buffer overflow DIFT policies had far too many false positives. Our solution presented in Chapter 5 abandoned all attempts to recognize the myriad forms of bounds check operations, and instead prevented attackers from injecting pointer values. This required the policy to distinguish between legitimate application pointers and pointers injected by the attacker. To track legitimate pointers, we added a second tag bit to track pointers and used DIFT to track those pointers legitimately derived from the application. Our solution used two kinds of information flow, simultaneously tracking both the flow of untrusted information and that of legitimate application pointers.

7.4 Define DIFT System Architecture

Once the set of policies have been determined, a DIFT system architecture must be designed to support these policies. When designing a DIFT architecture, the designer must balance a number of conflicting goals. These include the original four goals of a security protocol described in Chapter 1: *flexible*, *fast*, *practical*, and *safe*. Additionally, we add one addition goal: *cheap*, reflecting the desire to minimize cost.

In the remainder of this section, we will provide an overview of the major DIFT implementation types: compiler, bytecode rewriting, metaprogramming, modified language interpreter, dynamic binary translation, and hardware. We will also discuss how the selection of protected applications and DIFT policies influence the choice of DIFT implementation. This section closes with a discussion of tag coherency and consistency, and how this is affected by the memory model of the DIFT system architecture.

7.4.1 Compiler DIFT

Academic researchers have proposed implementing DIFT by modifying existing compilers [55]. In this approach, compilers transparently add DIFT operations to an application during the compilation process, emitting a DIFT-aware executable. This approach allows the compiler to use standard optimization techniques to minimize the overhead of tracking information flow. Existing research has focused on the C programming language, and observed performance overheads have been minimal. While these prototypes implemented only a few fixed policies, it would be relatively easy to add support for arbitrary policies, providing a solution that is *flexible*, *cheap*, and *fast*.

However, this approach is not *practical* as it requires source code access to all protected applications. Developers must opt-in to DIFT protection by recompiling their applications and shipping binaries with DIFT enabled. This leaves the developers with two unattractive options: ship with DIFT enabled for all binaries, even for customers that do not want DIFT, or ship two versions of the program, one with DIFT and one without.

It is unlikely that developers will find this to be an acceptable tradeoff for economic reasons unless their products are marketed towards an extremely security-savvy customer base. Furthermore, this approach requires the *entire* system to be recompiled with DIFT support to accurately track taint for complete information flow tracking safety. This is a serious, fundamental drawback.

Any code not compiled by the DIFT compiler, such as code written in another programming language, assembly code, code from third-party libraries or system libraries, or dynamically generated code, will have *no* support for DIFT, and no tag checks or propagation.

Existing researchers require all such functions to be annotated with manually-generated information flow summaries. We do not believe this to be an acceptable solution. There are billions of lines of legacy code, and not protecting this code (which doubtless will interact with tag propagation and checks) will allow for many false negatives. Developers are very unlikely to go back and annotate entire codebases, especially for third-party libraries. For this reason, compiler-based DIFT approaches are also not *safe*. Even with full annotations, data structures not visible to the compiler, such as the Global Offset Table, cannot be effectively protected. Due to lack of safety and practicality, we do not believe compiler-based DIFT will be an acceptable solution in production environments. Even with manual annotations for all assembly and full source code access, this approach should not be applied to multithreaded applications (or an operating system) due to race conditions on tag updates as described in Section 4.1.1.

7.4.2 Bytecode Rewriting DIFT

For languages such as Java which compile source code to a modifiable bytecode format, researchers have provided DIFT solutions that implement information flow tracking by statically or dynamically rewriting the application bytecode [40, 63]. In this approach, classes used as tag sources, sinks, or propagators are modified by inserting bytecode instructions to perform the relevant DIFT operations.

Bytecode rewriting DIFT requires the instrumented language has 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 such as Java and Microsoft .NET meet this requirement.

Existing bytecode rewriting DIFT prototypes 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 is *flexible*, *cheap*, and *practical*. It is *safe* if all relevant classes associated with DIFT operations are instrumented. The only known performance overhead results are from academic prototypes that perform bytecode rewriting on Java applications for object-granularity tags by modifying String-related classes [40, 63]. The overhead in this case is extremely low.

However, these object-granularity prototypes are not *safe* as they do not track information flow across characters or character arrays. 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 (see Section 3.3.4) as DBT-based DIFT approaches perform very similar operations on low-level ISA instructions but additionally incur the overhead of dynamic binary translation.

This technique should not be extended to x86 binaries. Static rewriting of x86 object files is not recommended. Even static disassembly of x86 binaries is undecidable [59] because the x86 allows branches into the middle of an instruction due its variable-length instruction width. Furthermore, such an approach could not handle self-modifying code or dynamically generated code. Implementing DIFT in software on unmodified binaries is best done using dynamic binary translation, as described in Section 3.3.4.

7.4.3 Metaprogramming DIFT

DIFT may also be implemented in certain high-level languages by using metaprogramming APIs to perform runtime modification (or ”monkeypatching”) of all tag sources, sinks, and propagators [32]. In this approach, the high-level language allows programs to modify existing classes at runtime by redefining or wrapping existing methods, or by adding additional fields.

The promise of metaprogramming is that it allows DIFT to be implemented using constructs already provided by the language. A full DIFT environment can be created without modifying the interpreter, investing in hardware, or performing bytecode rewriting. Metaprogramming allows 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. Then the original applications runs, and the modified classes serve as the runtime DIFT environment. This approach is theoretically the *cheapest*, as it is the least complex way to implement DIFT. It is also *practical* and *flexible*, and is *safe* if all sources, sinks, or propagators can be modified. However, there are currently no empirical evaluations of this approach, or available research prototypes, and thus the performance overheads are unknown.

This approach is applicable *only* for languages that allow runtime modification of all relevant taint sources, sinks, and propagators via metaprogramming APIs. For example, Python will not fit this requirement for most reasonable DIFT policies because system classes such as String cannot have their methods redefined or modified.

The latest versions of Java supports a restricted, partial implementation of metaprogramming via JVMTI [46]. JVMTI allows redefinition of existing methods at runtime via bytecode rewriting. To add methods or fields, developers must use static bytecode rewriting on disk or JVMTI's load-time bytecode rewriting hooks for class files. JVMTI cannot modify primitive types or array classes.

To our knowledge, the only mainstream language that fully supports metaprogramming is Ruby. Ruby is an object-oriented, type-safe language. 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 adding fields and wrapping or redefining existing methods. Thus, DIFT may be implemented by redefining methods in the classes used as tag sources, sinks and propagators. 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. Ruby is the only mainstream language that may be used to implement DIFT entirely via metaprogramming. No other mainstream language has such extensive support for reflection and metaprogramming.

7.4.4 Interpreter DIFT

Another method for implementing DIFT is to modify the language interpreter itself. This implementation strategy supports DIFT by adding the appropriate functionality to the interpreter, ensuring that information flow tracking occurs for all tag sources, sinks, and propagators. Having direct access to the interpreter allows for DIFT-specific optimizations to be placed into the interpreter itself, most likely in native code, providing potentially better performance than interpreter-independent techniques such as metaprogramming or bytecode rewriting.

Unlike metaprogramming or bytecode rewriting, this approach is tied to a particular interpreter, and porting modifications between interpreters is likely to be expensive. However, most languages have a single canonical or reference interpreter, such as CPython for Python or HotSpot for Java, where DIFT developers could focus their efforts.

Modifying the interpreter allows for *flexible*, *practical*, and *safe* DIFT policies. The *cost* depends on the complexity of the interpreter, and the scope of the changes required to support DIFT. For example, academic researchers have added support for DIFT to the PHP interpreter [77] without difficulty, but performing similar changes in a JIT-compilation interpreter such as HotSpot would be significantly more difficult. Generally, interpreters which do not perform JIT compilation or dynamic code generation are usually reasonably *cheap* to modify and are *fast*, as DIFT support adds relatively little overhead. No studies have been done on the performance overhead of adding DIFT to a JIT compilation interpreter, but overheads can be expected to be smaller than implementing DIFT using dynamic binary translation (see Section 7.4.5.

However, this approach comes with maintenance costs. Unless the developers of the interpreter merge support for DIFT, providing a DIFT-aware interpreter requires the designer to maintain a fork of the interpreter. This can be 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.

7.4.5 Dynamic Binary Translation DIFT

Metaprogramming, bytecode rewriting, and interpreter-based DIFT only protect and instrument software written in a single language. In contrast, Dynamic binary translation (DBT) techniques can be used to apply DIFT to unmodified binaries [100, 72] or even the entire system (including the OS kernel) [5, 94].

In a DBT-based DIFT approach, dynamic binary translation is used to insert tag propagation and check instructions into application code at runtime. Each register and memory location is extended with a tag, and the DBT modifies the instruction stream to insert a DIFT propagation and check instructions. For example, after an add instruction, a logical OR instruction would be inserted to propagate taint from the source operand tags to the destination operand tag.

This approach is *flexible*, as it allows arbitrarily complex policies to be applied to binaries or even the operating system. However, this flexibility is limited by the amount of information present in binaries. For example, in a JIT-compilation environment such as the Java Virtual Machine (JVM), the address of methods is unknown as the JVM determines this information at runtime and may even change the address of a method during program execution when performing optimizations. As a consequence, normal SQL injection protection policies which perform safety checks on any SQL execute query method cannot be used because there is too little information available in the executable binary. In such a case, it may be beneficial to perform DIFT using bytecode rewriting or by modifying the interpreter directly so that high-level concepts such as objects and classes may be utilized by DIFT policies. Alternatively, the interpreter could be modified to provide the DIFT implementation with additional information so that even a hardware or DBT-based DIFT implementation could map classes and methods to memory addresses.

This approach is also *safe*, as any instruction can be instrumented with DIFT support. Developing a DBT is a complex, time-consuming process consuming man-years of time [134, 6, 100]. However, adding DIFT support to an existing DBT such as Valgrind or QEMU has been done in reasonable time frames with only a handful of graduate students [94, 76]. Adding DIFT to a well-designed DBT is thus *cheap*, but building a DBT from scratch is not.

For systems requiring support for multiple cores or with significant concurrency needs, this approach is *not practical*. As explained in Section 4.1.1, DBT-based DIFT solutions cannot safely permit concurrent updates to memory locations by multiple threads of execution, and risk false positives and negatives when this situation occurs. To avoid these correctness issues, DBT solutions either restrict the application to a single core [5] or allow only one thread to run at a time [72].

This approach is also *not fast*, resulting in observed overheads ranging from 3x [100] to 37x [76]. In general, overheads for DBT-based approaches that protect the entire system [94] remain significantly higher than approaches which perform dynamic binary translation on a single application [100]. This is not unexpected, as system level dynamic binary translators without support for DIFT [5] have much higher performance overhead than comparable application-level DBTs [6, 51, 100, 134].

7.4.6 Hardware DIFT

Hardware DIFT provides a number of attractive advantages over a pure software approach. Many proposed DIFT systems have been implemented as hardware prototypes [24, 121, 12, 131, 11], including Raksha. Hardware DIFT systems are implemented by tracking information flow across hardware resources such as registers and memory. Tag propagation and tag checks are often performed entirely or almost entirely in hardware. This allows DIFT policies to be enforced on unmodified application binaries, or the entire system (including the OS kernel). Of all the DIFT architecture designs, hardware is the *fastest*, and many implementations have zero or near-zero overhead for DIFT.

Hardware is *safe*, and is much more *practical* than its software equivalent, DBT-based DIFT. Unlike DBT-based DIFT, hardware DIFT supports self-modifying code, dynamically generated code, multi-threading, multiple cores, and other advanced code operations. Section 4.1.1 provides further discussion concerning the practicality advantages of hardware-based DIFT.

The major drawback to hardware DIFT is that it is *expensive*. Hardware is significantly more expensive to develop, manufacture, and distribute than software. Designs are fixed into silicon, and upgrading units in the field with hot fixes or patches is not possible. The high cost of hardware development, especially using the latest manufacturing processes, makes a DIFT hardware design unlikely in the short term.

Hardware can be somewhat *flexible* with careful design. Research prototypes such as Raksha (see Chapter 4) and Flexitaint [131] have shown that hardware can be developed to support a wide variety of policies. However, no hardware approach is as malleable or flexible as software. Some policies are likely too complex and dynamic to be placed into hardware. Such policies may benefit from a hardware-assisted approach which accelerates software implementations of tag checks and propagation. For example, the Log-Based Architecture (LBA) [11] project performs tag propagation in software on a separate core using hardware assistance. LBA supports policies such as lockset-based race detection, where propagation may involve inserting objects into a data structure such as a hash table. This form of tag propagation would be too complex to place into hardware.

The challenge for hardware DIFT is to prove that the prohibitive cost of implementation is justified economically. The DIFT system designer must find the right set of hardware primitives that support enough DIFT policies to justify the cost of the implementation. The potential benefit of a hardware design is near-zero performance overhead when protecting unmodified binaries or the entire system with DIFT, and support for scalable multi-threaded applications.

7.4.7 Coherency & Consistency

No matter which DIFT system architecture is selected, as long as the architecture allows for multiple concurrent threads of execution the designer must still address tag coherency and consistency. When one thread updates shared data, the update to the data and its associated tag must be a single atomic action, or other threads could read incorrect taint or data values. Reading an incorrect taint value could result in false positives, terminating legitimate applications due to a security exception, or false negatives, allowing attackers to bypass DIFT.

There are two ways in which a tag-data update may have a race condition. The first method occurs when application actions which were previously considered atomic, such as storing to a naturally aligned word of memory or executing an atomic compare-and-swap instruction, are no longer atomic because both the tag and data must be updated in two separate instructions. The second method is via an existing race condition in the application.

To prevent false positives, the first case must be fully addressed. If reporting a security exception due to a race condition in the application is acceptable, then the second case does not need to be addressed for false positive protection. However, fully preventing false negatives requires addressing *both* cases, as attackers may exploit existing application race conditions between data and tag updates to bypass DIFT. This is no idle threat, as attackers have used race conditions to perform tasks as complicated as exploiting an OS kernel buffer overflow.

The DIFT designer must choose the tag coherence and consistency model that balances performance, safety, and correctness. Due to implementation costs, it may be acceptable just to guarantee that applications without race conditions will have correct tag values, but that any race conditions have a low-probability of reading or writing stale, incorrect tag values. In the remainder of this section, we discuss the major tag coherency and consistency policies.

Forced Single Threading

The current solution to tag coherency and consistency employed by DBT-based DIFT is to prevent race conditions entirely by forbidding multiple concurrent threads of execution. This can be accomplished by supporting only a single physical processor core [100, 5], or by running a single thread at a time [72].

This approach works even for unmodified binaries and legacy code. Unsafe race conditions are guaranteed to produce safe tag-data updates, providing correctness and safety to legacy applications. However, there are significant performance drawbacks. As discussed in Section 4.1.1, systems today have multiple cores, and the number of cores will only increase over time. Restricting applications to a single core causes significant performance degradation and restricts scalability. These drawbacks will only become more costly as future systems add additional cores.

Object DIFT

One method for solving this problem in high-level languages with object-granularity tags is to perform taint propagation inside object methods. This approach requires all relevant objects to be immutable or lock-protected. For example, in the existing Java DIFT environments [40, 63], taint is propagated across String-related methods. The String class is immutable, so any attempt to modify a String creates a new String. As long as taint is stored within the String class and taint propagation occurs inside the String modification methods, this approach is thread-safe. Race conditions on immutable objects are impossible, as any modification to the object produces a new object, rather than resulting in modifications to the existing object.

Similar guarantees can be made for objects that are automatically lock-protected on method entry, such as StringBuffers in Java. These objects ensure that a per-object mutex lock is acquired before performing any modifications to the object. No data can be updated without holding this lock, and the application does not need to explicitly acquire the lock as it is acquired automatically upon method entry. As long as all taint propagation occurs within the scope of the object lock, thread safety is assured and race conditions are not possible.

A third class of object is mutable but is not lock-protected, such as StringBuilders in Java. When accessing this type of object, the caller must lock the object if it is used in a multi-threaded environment. The benefit of this approach is that callers in a single threaded environment do not pay the overhead of locking. Performing DIFT inside such an object will work correctly and safely so long as the caller follows the locking protocol, as all DIFT operations will occur with the lock held. Unlike the other object types, this design allows for applications race conditions if the object is accessed without acquiring the lock. These race conditions could cause stale or incorrect tag values to be read. The DIFT system designer must determine if this is an acceptable tradeoff, as this type of object allows DIFT false positives and false negatives in applications with race condition bugs. If this is not acceptable, an alternative coherency and consistency method must be used (or the offending object class changed).

Shared Nothing

Alternatively, languages can take a ”shared-nothing” approach and forbid thread-shared data entirely in user applications. This is the approach taken by PHP, which supports multiple threads serving different HTTP requests, but does not allow data to be shared between threads. Languages based purely on message passing, such as Erlang, also avoid thread-shared state. Shared-nothing designs eliminate the possibility of tag-data update race conditions because application threads cannot share data (or tags).

This approach provides excellent performance and safety, but it restricts the flexibility of application designs. A language with a shared-nothing memory model may still be successful, as evidenced by the widespread popularity of PHP, and the success of Erlang in the telecommunications industry. However, a shared-nothing model cannot be retrofitted into an existing language with support for thread-shared data structures without breaking compatibility with virtually every application.

Hardware

Finally, explicit hardware support can be provided to extend the memory coherency and consistency protocols with support for tags [131, 48]. In this design, the coherency and consistency protocols themselves are responsible for ensuring that tag values are always up-to-date. Although this approach is the most expensive, it is the only approach that works with unmodified binaries or low-level languages while still allowing for multiple concurrent threads of execution. Care must be taken to ensure that the performance impact on non-DIFT applications is minimal, and that the hardware support is sufficiently flexible to allow for many different kinds of metadata.

The approach described in [48] presents a method for supporting tag consistency and coherency for even arbitrarily-sized tags in hardware. In this design, hardware provides direct support for up to a word of tag per word of data. For larger metadata, DIFT policies use the tag to store a word-sized pointer to the large metadata structure. Tag propagation, likely performed entirely or almost entirely in software, will then ensure consistency by using the Read-Copy-Update (RCU) [61] synchronization algorithms to access and update the pointer stored in the memory tag.

7.4.8 Guidelines

This section has discussed the cost, flexibility, and performance tradeoffs of the various DIFT system architecture designs. Another important design consideration is the influence of the DIFT policies and threat model. Some DIFT system architectures are better than others at preventing particular classes of vulnerabilities or supporting certain kinds of DIFT

DIFT System	**Low-Level Vuln**		**High-Level Vuln**		
	Assembly	C	C	Interpreted	JIT
Compiler DIFT	✘	✔	✔	?	✘
Bytecode DIFT	✘	✘	✘	✔	✔
Metaprog. DIFT	✘	✘	✘	✔	✔
Interpreter DIFT	✘	✘	✘	✔	✔
DBT DIFT	✔	✔	✔	?	?
HW DIFT	✔	✔	✔	?	?

Table 7.1: The DIFT system architectures and their applicability to various vulnerability types. Low-level vulnerabilities include all memory corruption attacks, specifically buffer overflows, user/kernel pointer dereferences, and format string vulnerabilities. High-level vulnerabilities include all API-level injection attacks, specifically SQL injection, cross-site scripting, directory traversal, authentication and authorization bypass, and command injection. The type of language (raw assembly, C, type-safe interpreted language, or type-safe JIT compiled language) is also specified where relevant.

policies. For example, a DIFT system built by modifying the PHP interpreter will be of little use in detecting buffer overflows.

In general, DIFT systems implemented a particular layer of abstraction prevent vulnerabilities in all of the layers above them. A DIFT system built by modifying the JVM will prevent Java application vulnerabilities, while a DIFT system built in hardware will be able to address assembly or ISA-instruction level vulnerabilities.

However, building DIFT systems at the lowest possible level of abstraction does not always result in a better design. When you lower the layer of abstraction, such as by choosing a dynamic binary translator-based design over a JVM-based design, you lose information about the applications you are protecting. The JVM has much more information about a Java class than a dynamic binary translator, which cannot even resolve method names to memory addresses. In the worst case, your design may not have sufficient information about the target application to properly support the desired set of DIFT policies. This deficiency can be mitigated 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.

For example, if a hardware DIFT system is able to protect PHP applications against SQL injection because the PHP SQL libraries are implemented as C extensions. However, if the PHP interpreter created raw sockets and the SQL libraries manipulated those sockets entirely in PHP, it would be difficult for hardware to interpose on calls to SQL library methods without the ability to map PHP function names to memory addresses at runtime. A modified PHP interpreter cold provide this information to a low-level DIFT system architecture, such as a DBT or hardware DIFT design.

Table 7.1 describes the applicability of DIFT system architectures to particular vulnerabilities. A question mark is used to indicate cases that are situation-dependent (i.e. may work on PHP but not necessarily on all language interpreters), may require additional information, or would need a trusted higher-level component to communicate information to the DIFT system. For example, a hardware DIFT environment could prevent SQL vulnerabilities in Java applications if a modified trusted JVM provided the DIFT system with up-to-date mappings of java method names to memory addresses. However, without this information, the DIFT system could not support Java (or other JIT-compiled applications).

7.5 Implications for Language Design

The design of DIFT systems is in its infancy and no mainstream languages of any kind, from ISAs implemented at the hardware level to high-level web programming languages such as Ruby, have been designed with DIFT or other forms dynamic analysis in mind.

In designing and implementing DIFT systems, we have identified key changes that language designers can make to allow for easier dynamic analysis of languages, both for DIFT and other runtime analyses. In this section, we describe these changes, presenting the benefits for DIFT and discussing any associated disadvantages.

7.5.1 Coherency and Consistency

Tag coherency and consistency are a significant challenge for DIFT system designers. This can be made significantly easier by providing a language memory model that makes it easier to guarantee coherency and consistency.

The easiest model is a shared-nothing approach, as adopted by Erlang and PHP, where coherency and consistency concerns are not present. Alternatively one could imagine a hybrid approach: a language where all thread-shared objects were immutable or automatically locked on access, and all other objects were shared-nothing. This design would allow DIFT system architects to safely use object-granularity tags for thread-shared data by explicitly identifying the classes that may be thread-shared, and guaranteeing safe, race-free semantics for these classes.

7.5.2 High-level Bytecode

Bytecode rewriting DIFT is a high-performance, relatively easy way to implement DIFT systems. However, the design of many languages and language interpreters preclude this approach from being used. However, there are many design requirements that must be met for bytecode rewriting DIFT to be applicable.

High-level languages should translate all code, either at compile-time or load-time, into a well-specified bytecode format, as is used in the JVM or Microsoft CLR. An instrumentation system, such as JVMTI [46] in Java should be present so that instrumentation agents can interpose on all bytecode before it is executed, including the bytecode of core system classes such as String or Integer. The bytecode format must also allow for arbitrary modifications, such as adding fields to a class, or inserting bytecode instructions into existing methods.

All classes should be specified in bytecode. This includes classes that have native-compiled methods implemented in languages such as C or C++. DIFT is implemented by adding taint tracking operations before or after a source, sink, or propagation operation occurs. Even native methods can be instrumented by renaming the native method. The instrumented method is then defined as a wrapper that calls the renamed native method with the appropriate arguments and performs DIFT operations before or after the renamed method is called.

All of these requirements must be met for bytecode rewriting DIFT solutions to be viable. For example, Java meets all of these requirements [1], and a number of bytecode

[1]With the exception of arrays and primitive types, which cannot be instrumented via bytecode rewriting

rewriting DIFT systems for Java are available [40, 63]. However, Python does not meet any of these requirements, and thus software DIFT implementations for Python must modify the interpreter. Given an environment that meets the requirements outlined in this section, even low-level C code could potentially be translated into LLVM bytecode [56] and instrumented using bytecode-rewriting DIFT.

7.5.3 Precise, Well-Defined Interfaces

Any interface (system call, method, function) used as a taint sink, source, or propagator should have a precise and unambiguous definition. In a high-level language, the types used in the interface method arguments and return value should be modifiable via code rewriting or metaprogramming, so that a DIFT implementation may easily use object-granularity tags without resorting to the more costly DIFT approaches, such as interpreter rewriting or dynamic binary translation.

For example, current Java DIFT prototypes [40, 63] use object-granularity tags because the taint sources and sinks in Java SQL Injection policies are the Servlet and JDBC methods, which use Strings. The String class can be instrumented using bytecode rewriting, allowing for an object-granularity DIFT solution. If these interfaces used character arrays, object-granularity tags could not be used because the character array class has no associated bytecode and is implemented fully in native code by the JVM.

If multiple vendors may provide security-critical functionality, the standard library should provide a common interface adhered to by all vendors. For example, all SQL drivers used in Java implement the JDBC interfaces. Common interfaces allow DIFT implementations to precisely identify tag sources and sinks by instrumenting all implementations of the relevant interface, rather than forcing DIFT vendors to enumerate all possible vendors implementing a given interface.

7.5.4 Metaprogramming APIs

Metaprogramming is a new potential avenue for implementing DIFT. However, it is only a possible implementation strategy if *all* taint sources, sinks, and propagators can be instrumented using the metaprogramming API. In a high-level language with support for dynamic class modification via metaprogramming, the language designer can support metaprogramming DIFT by permitting modifications to any method or operator. Python fails this requirement, despite its robust metaprogramming support. This is because Python forbids runtime modification of system classes such as String. In contrast, Ruby allows all classes and operations, even integer addition, to be instrumented via metaprogramming. The language designer must weigh the benefit of allowing for dynamic analyses such as DIFT to be implemented via metaprogramming against the performance overhead of supporting metaprogramming for all operations and methods.

7.5.5 DIFT APIs

A language that is DIFT-aware can allow DIFT implementations much more introspection into the language and its applications by providing DIFT APIs. A DIFT API would allow the language interpreter or application to supply additional runtime information to the DIFT system, such as taint sources, taint sinks, or even specify entire DIFT policies by providing check and propagation rules.

This enables application developers to create policies that prevent application-specific vulnerabilities. Furthermore, if the language interpreter can use this API to provide information to the DIFT system, then low-level DIFT system architectures can be used to prevent high-level vulnerabilities. As shown in Table 7.1, low-level DIFT system architectures such as hardware or DBT-based DIFT platforms may require additional knowledge or information to prevent high-level vulnerabilities in interpreted or JIT compiled languages.

For example, in a language that will be JIT compiled, the language designer should include APIs so that the runtime engine can provide the DIFT system architecture with up to date memory addresses for any relevant classes or methods, such as the location of the database SQL query execution method. The benefit of this design is that the DIFT system can simultaneously protect the application from high-level vulnerabilities and the interpreter from low-level vulnerabilities. Low-level memory corruption policies do not require additional information from the interpreter and thus do not rely on the integrity of the interpreter, allowing the DIFT system to protect the interpreter itself from this class of attacks. The interpreter supplies information only for policies protecting against high-level vulnerabilities in the application. The information supplied by the interpreter effectively bridges the semantic gap between the high-level language and the lower-level DIFT system architecture.

7.6 Testing & Validation

DIFT systems are different from most conventional software or hardware projects. DIFT tracks the flow of information through an unmodified application, and thus may be involved in every aspect and layer of abstraction in the software stack. It is important to robustly test DIFT designs to prevent false positives, negatives, and outright bugs and crashes from occurring in practice. Given the difficulty of implementing DIFT systems and the complexity of DIFT's input (arbitrary programs), a high priority must be placed on testing and validation. We have had great success using randomized testing.

In randomized testing, randomly generated program fragments are created by a test generator that also precomputes the expected tag and data values after each generated program operation. Periodically, the generated program fragment will pause its execution, checkpoint itself, and compares the tag and data values of all registers and memory with the expected values. Any unexpected values indicate a bug or error. Furthermore, each tag exception checks to see if the exception was unexpected or if any prior expected exceptions did not occur, and if so an error is reported. Generated programs should be deterministic so that they can replayed by developers to identify the root cause of the bug. All sources of non-determinism, such as thread interleavings or timers, must be made deterministic by modifying the environment, using virtual machine replay capabilities, or some other means.

With sufficiently many machines generating reasonably large program fragments, a good portion of the design space can be explored, giving significantly greater test coverage than manual testing. Care must be taken to ensure that the random test case generator can emit any potentially valid program fragment, including support for mock I/O and all other operations. If you do not support test case generation of a particular operation or library call, then you are very likely to miss bugs. This is particularly important for hardware or DBT-based DIFT systems as real-world ISAs are very complex. The most painful, longest lasting, and difficult to debug issue in Raksha was a bug caused by a race condition in our memory controller that occurred only during DMA transfers. Our simulator did not have support for modeling DMA. Thus, this issue was not caught by our random test case generator which otherwise would have identified the bug automatically in minutes or hours. Instead, the debugging process was done entirely by hand over a number of weeks.

It may also be useful to examine recent promising research in symbolic testing [8], which may provide better path coverage than generating random program fragments. You may also wish to bias your randomly generated fragments to make very unlikely corner cases (evicting cache lines, etc) occur more frequently.

Chapter 8

Conclusions & Future Work

In this chaper we present our conclusions and describe future work in Dynamic Information Flow Tracking research.

8.1 Conclusions

Dynamic Information Flow Tracking (DIFT) is a powerful and flexible technique for comprehensively preventing the most critical software security threats. This thesis demonstrated that DIFT can be used to prevent a wide range of security attacks on unmodified applications and even the operating system, with little to no performance overhead.

We developed a flexible hardware DIFT platform, Raksha, for preventing security vulnerabilities in unmodified applications and the operating system. Raksha allows the flexible specification of DIFT policies using software-managed tag policy registers. We implemented an FPGA-based prototype of Raksha using an open source SPARC V8 CPU. Using our prototype, we evaluated DIFT policies and successfully prevented low-level buffer overflow and format string vulnerabilities, as well as high-level web vulnerabilities such as command injection, directory traversal, cross-site scripting, and SQL injection. All experiments were performed on unmodified application binaries, with no debugging information. Observed performance overhead on the SPEC CPU2000 benchmarks was minimal.

Using Raksha, we developed a novel DIFT policy for preventing buffer overflow attacks. Unlike prior policies, our buffer overflow protection resulted in no real-world false positives, even in real-world applications such as Apache, Bash, Perl, and Sendmail. This policy was also used to protect the Linux kernel, providing the first comprehensive kernelspace buffer overflow protection.

We also developed Nemesis, a DIFT-aware PHP interpreter, which was the first system for comprehensively preventing authentication and authorization bypass attacks. Using Nemesis, we prevented both authentication and authorization bypass attacks on legacy PHP applications without requiring the existing authentication and access control frameworks to be rewritten. Furthermore, no discernible performance overhead was observed when executing common web server benchmarks, and even CPU-intensive micobenchmarks demonstrated minimal overhead.

8.2 Future Work

Several key challenges remain in bringing the potential of DIFT to fruition. Real-world adoption of DIFT has yet to occur on any wide scale, and many practical breakthroughs are required to bring DIFT from the research lab to a production environment. More study is needed to determine what policies scale to truly enterprise environments, and the amount of on-site configuration that will be necessary. DIFT is relatively unknown outside of academia, and would benefit greatly from having its principles and policies tested on large-scale enterprise backend workloads in real-world environments.

The most likely avenue for promoting real-world DIFT usage in the near-term will be modifying language interpreters for popular languages such as Python and PHP to support DIFT. As of yet, no mainstream language interpreter has merged patches for supporting DIFT, although a patch by Wietse Venema to add DIFT support to PHP is under consideration by the PHP core team [130].

Very little research has been done into the easy specification of DIFT policies, or presenting administrators with an easy to understand visualization of the information flow in their systems. Similarly, little research has been done into the implications of adding DIFT support to an existing language. Many open questions remain, such as the easiest way to present DIFT APIs to policy developers, and how to integrate DIFT fully with existing language semantics while preserving backwards compatibility with legacy applications. Creating an accessible language implementation of DIFT with very low overhead and easy configuration and information flow visualization would be an excellent first step in promoting the widespread adoption of DIFT techniques.

More work could also be done to address the limitations of DIFT. Information flow is often ambiguous. User input may only partially determine an output or may be used to select from a handful of predefined choices rather than completely determining the output of an operation. Current DIFT input validation systems do not account for this behavior, as information flow is usually encoded using a single bit (trusted or untrusted), and thus it is assumed that user input either completely determines an output or does not influence the output at all.

For example, DIFT systems for input validation do not track implicit information flow due to control dependencies. Tracking this information would result in a tremendous increase in false positives, as most user input used in a branch condition or control flow operation does not result in direct data copying or other security-relevant operations. However, this can result in false negatives when control flow information is used in a security-sensitive manner. Similarly, most DIFT policies today ignore address propagation, and do not propagate taint if a load/store address itself is tainted. This can seriously diminish the effectiveness of DIFT, as applications using translation tables to remap user input from one encoding format to another. If a DIFT application cannot track this information flow, then false negatives may result. Recent research has made some progress in addressing these issues by more precisely defining and quantifying information flow [75].

Bibliography

[1] AoE (ATA over Ethernet). `http://www.coraid.com/documents/AoEr8.txt`, 2004.

[2] ATPHTTPD Buffer Overflow Exploit Code. `http://www.securiteam.com/exploits/6B00K003GY.html`, 2001.

[3] Attacking the Core: Kernel Exploiting Notes. `http://phrack.org/issues.html?issue=64\&id=6`, 2007.

[4] BilboBlog admin/index.php Authentication Bypass Vulnerability. `http://www.securityfocus.com/bid/30225`, 2008.

[5] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX '05: Proceedings of the USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[6] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* PhD thesis, Massachusetts Institute of Technology, September 2004.

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

[8] Cristian Cadar, Paul Twohey, Vijay Ganesh, and Dawson Engler. EXE: A System for Automatically Generating Inputs of Death Using Symbolic Execution. In *In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS*, 2006.

[9] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th conference on Operating Systems Design and Implementation*, 2006.

[10] CERT Coordination Center. Overview of attack trends. http://www.cert.org/archive/pdf/attack_trends.pdf, 2002.

[11] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 377–388, Washington, DC, USA, 2008. IEEE Computer Society.

[12] Shuo Chen, Jun Xu, et al. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *Proceedings of the Intl. Conference on Dependable Systems and Networks*, 2005.

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

[14] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 73–88, New York, NY, USA, 2001. ACM.

[15] JaeWoong Chung, Michael Dalton, et al. Thread-Safe Dynamic Binary Translation using Transactional Memory. In *Proceedings of the 14th Intl. Symposium on High-Performance Computer Architecture*, 2008.

[16] J. Clause, W. Li, and A. Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2007)*, pages 196–206, London, UK, July 2007.

[17] Web Application Security Consortium. Web application security statistics. `http://www.webappsec.org/projects/statistics/`, 2007.

[18] 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*, pages 15–15, Berkeley, CA, USA, 2001. USENIX Association.

[19] 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.

[20] Jedidiah R. Crandall and Frederic T. Chong. MINOS: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th Intl. Symposium on Microarchitecture*, 2004.

[21] John Criswell, Andrew Lenharth, et al. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of the 21st Symposium on Operating System Principles*, October 2007.

[22] Cross-Compiled Linux From Scratch. `http://cross-lfs.org`.

[23] M. Dalton, H. Kannan, and C. Kozyrakis. Deconstructing Hardware Architectures for Security. In *the 5th Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2006.

[24] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Intl. Symposium on Computer Architecture*, 2007.

[25] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Real-world buffer overflow protection for userspace and kernelspace. In *Proceedings of the 17th Annual USENIX Security Symposium*, pages 395–410, 2008.

[26] Michael Dalton, Nickolai Zeldovich, and Christos Kozyrakis. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *Proceedings of the 18th Annual USENIX Security Symposium*, 2009.

[27] DeluxeBB PM.PHP Unauthorized Access Vulnerability. `http://www.securityfocus.com/bid/19418`, 2006.

[28] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *ACM Communications*, 20(7), 1977.

[29] M. Dowd. Application-Specific Attacks: Leveraging the ActionScript Virtual Machine. `http://documents.iss.net/whitepapers/IBM_X-Force_WP_final.pdf`, 2008.

[30] Duff's Device. `http://www.lysator.liu.se/c/duffs-device.html`, 1983.

[31] eEye Digital Security. Microsoft Internet Information Services Remote Buffer Overflow . `http://eeye.com/html/Research/Advisories/AD20010618.html`, 2001.

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

[33] Hiroaki Etoh. GCC Extension for Protecting Applications from Stack-smashing Attacks. `http://www.trl.ibm.com/projects/security/ssp/`.

[34] Peter Ferrie. ANI-hilate This Week. In *Virus Bulletin*, March 2007.

[35] The frame pointer overwrite. `http://www.phrack.org/issues.html?issue=55\&id=8`, 1999.

[36] Fr'ed'eric Perriot and Peter Szor. An Analysis of the Slapper Worm Exploit. `http://www.symantec.com/avcenter/reference/analysis.slapper.worm.pdf`, 2003.

[37] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proceedings of the 11th Network and Distributed Systems Security Symposium*, San Diego, CA, February 2004.

[38] Tal Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proceedings of the Symposium on Network and Distributed Systems Security*, February 2003.

[39] Gentoo Linux. `http://www.gentoo.org`.

[40] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. *Computer Security Applications Conference, Annual*, 0:303–311, 2005.

[41] Norm Hardy. The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Operating System Review*, 1988.

[42] w00w00 on Heap Overflows. `http://www.w00w00.org/files/articles/heaptut.txt`, 1999.

[43] Billy Hoffman and John Terrill. The Little Hybrid Worm That Could, 2007.

[44] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[45] Imperva Inc., How Safe is it Out There: Zeroing in on the vulnerabilities of application security. `http://www.imperva.com/company/news/2004-feb-02.html`, 2004.

[46] Jvm tool interface version 1.1. `http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html`, 2006.

[47] Hari Kannan. *The Design and Implementation of Hardware Systems for Information Flow Tracking*. PhD thesis, Stanford University, 2009.

[48] Hari Kannan. Metadata consistency in multiprocessor systems. In *Proceedings of the 42nd Intl. Symposium on Microarchitecture*, 2009.

[49] Hari Kannan, Michael Dalton, and Christos Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *Proceedings of the 39th Intl. Conference on Dependable Systems and Networks*, 2009.

[50] Satoshi Katsunuma, Hiroyuki Kuriyta, et al. Base Address Recognition with Data Flow Tracking for Injection Attack Detection. In *Proceedings of the 12th Pacific Rim Intl. Symposium on Dependable Computing*, 2006.

[51] Chi keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005.

[52] Chongkyung Kil, Jinsuk Jun, et al. Address Space Layout Permutation: Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of 22nd Applied Computer Security Applications Conference*, 2006.

[53] Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazires, Robert Morris, Michelle Osborne, Steve VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, Santa Fe, NM, June 2005.

[54] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information Flow Control for Standard OS Abstractions. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 321–334, New York, NY, USA, 2007. ACM.

[55] 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 463–472, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[56] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[57] Ruby Lee, David Karig, et al. Enlisting Hardware Architecture to Thwart Malicious Code Injection. In *Proceedings of the Intl. Conference on Security in Pervasive Computing*, 2003.

[58] LEON3 SPARC Processor. `http://www.gaisler.com`.

[59] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *In ACM Conference on Computer and Communications Security (CCS)*, pages 290–299. ACM Press, 2003.

[60] Linux Kernel Remote Buffer Overflow Vulnerabilities. `http://secwatch.org/advisories/1013445/`, 2006.

[61] Read-copy update (rcu). `http://www.rdrop.com/users/paulmck/RCU/`, 2009.

[62] Linpha User Authentication Bypass Vulnerability. `http://secunia.com/advisories/12189`, 2004.

[63] Benjamin Livshits, Michael Martin, and Monica S. Lam. SecuriFly: Runtime Protection and Recovery from Web Application Vulnerabilities. Technical report, Stanford University, September 2006.

[64] M. Dowd. Sendmail Header Processing Buffer Overflow Vulnerability. `http://www.securityfocus.com/bid/6991`.

[65] Mark Dowd. OpenSSH Challenge-Response Buffer Overflow Vulnerabilities. `http://www.securityfocus.com/bid/5093`, 2002.

[66] Microsoft Excel Array Index Error Remote Code Execution. `http://lists.virus.org/bugtraq-0607/msg00145.html`.

[67] Microsoft. *Microsoft Portable Executable and Common Object File Format Specification*, 2006.

[68] Susanta Nanda, Lap-Chung Lam, and Tzicker Chiueh. Dynamic multi-process information flow tracking for web application security. In *Proceedings of the 8th International Conference on Middleware*, pages 1–20, New York, NY, USA, 2007. ACM.

[69] 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.

[70] Neel Mehta and Mark Litchfield. Apache Chunked-Encoding Memory Corruption Vulnerability. `http://www.securityfocus.com/bid/5033`, 2002.

[71] nergal. The advanced return-into-lib(c) exploits: PaX case study. In *Phrack Magazine*, 2001. Issue 58, Article 4.

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

[73] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual Execution Environments*, pages 65–74, New York, NY, USA, 2007. ACM.

[74] Netric Security Team. Null HTTPd Remote Heap Overflow Vulnerability. `http://www.securityfocus.com/bid/5774`, 2002.

[75] James Newsome, Stephen McCamant, and Dawn Song. Measuring channel capacity to distinguish undue influence. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 73–85, New York, NY, USA, 2009. ACM.

[76] James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[77] A. Nguyen-Tuong, S. Guarnieri, et al. Automatically Hardening Web Applications using Precise Tainting. In *Proceedings of the 20th IFIP Intl. Information Security Conference*, 2005.

[78] Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *In 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[79] Nimda worm. `http://www.cert.org/advisories/CA-2001-26.html`.

[80] OpenBSD IPv6 mbuf Remote Kernel Buffer Overflow. `http://www.securityfocus.com/archive/1/462728/30/0/threaded`, 2007.

[81] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

[82] Chetan Parampalli, R. Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *ASIACCS '08: Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*, pages 156–167, New York, NY, USA, 2008. ACM.

[83] Bryan Parno, Jonathan M. McCune, Dan Wendlandt, David G. Andersen, and Adrian Perrig. CLAMP: Practical Prevention of Large-Scale Data Leaks. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, May 2009.

[84] The PaX project. `http://pax.grsecurity.net`.

[85] Perl taint mode. `http://www.perl.com`.

[86] Perl Security. `http://perldoc.perl.org/perlsec.html`.

[87] phpFastNews Cookie Authentication Bypass Vulnerability. `http://www.securityfocus.com/bid/31811`, 2008.

[88] PHP iCalendar Cookie Authentication Bypass Vulnerability. `http://www.securityfocus.com/bid/31320`, 2008.

[89] Php Stat Vulnerability Discovery. `http://www.soulblack.com.ar/repo/papers/advisory/PhpStat_advisory.txt`, 2005.

[90] PHP: Using Register Globals. `http://us2.php.net/register_globals`.

[91] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Proceedings of the Recent Advances in Intrusion Detection Symposium*, Seattle, WA, September 2005.

[92] PhpMyAdmin control user. `http://wiki.cihar.com/pma/controluser`.

[93] Polymorph Filename Buffer Overflow Vulnerability. `http://www.securityfocus.com/bid/7663`, 2003.

[94] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *SIGOPS Operating Systems Review*, 40(4):15–27, 2006.

[95] President's Information Technology Advisory Committee. CyberSecurity: A Crisis of Prioritization. `http://www.nitrd.gov/pitac/reports/20050301\_cybersecurity/cybersecurit%y.pdf`, February 2005.

[96] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[97] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[98] Finally user-friendly virtualization for Linux. `http://www.linuxinsight.com/finally-user-friendly-virtualization-for-li%nux.html`, 2006.

[99] Feng Qin, Shan Lu, and Yuanyuan Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Proceedings of the 11th Intl. Symposium on High-Performance Computer Architecture*, 2005.

[100] Feng Qin, Cheng Wang, et al. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the 39th the Intl. Symposium on Microarchitecture*, 2006.

[101] Michael F. Ringenburg and Dan Grossman. Preventing format-string attacks via automatic and efficient dynamic checking. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 354–363, New York, NY, USA, 2005. ACM.

[102] William Robertson, Christopher Kruegel, et al. Run-time Detection of Heap-based Overflows. In *Proceedings of the 17th Large Installation System Administration Conference*, 2003.

[103] Daniel Roethlisberger. Omnikey Cardman 4040 Linux Drivers Buffer Overflow. `http://www.securiteam.com/unixfocus/5CP0D0AKUA.html`, 2007.

[104] David Ross. IE8 Security Part IV: The XSS Filter. `http://blogs.msdn.com/ie/archive/2008/07/01/ie8-security-part-iv-the-xss-filter.aspx`, 2008.

[105] XSS (Cross Site Scripting) Cheat Sheet. `http://ha.ckers.org/xss.html`.

[106] Rice University Bidding System. `http://rubis.objectweb.org`, 2009.

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

[108] Santa Cruz Operation. *System V Application Binary Interface, 4th ed.*, 1997.

[109] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15, 1997.

[110] Pekka Savola. LBNL Traceroute Heap Corruption Vulnerability. `http://www.securityfocus.com/bid/1739`, 2000.

[111] Security-enhanced linux. `http://www.nsa.gov/research/selinux/index.shtml`.

[112] H. Shacham, M. Page, et al. On the Effectiveness of Address Space Randomization. In *Proceedings of the 11th ACM Conference on Computer Security*, 2004.

[113] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security*, pages 552–561, New York, NY, USA, 2007. ACM.

[114] Shariq Rizvi Shariq, Alberto Mendelzon, S. Sudarshan, and Prasan Roy. Extending Query Rewriting Techniques for Fine-Grained Access Control. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2004.

[115] Sinan "noir" Eren. Smashing the kernel stack for fun and profit. `http://www.phrack.org/issues.html?issue=60\&id=6`, 2002.

[116] Eugene H. Spafford. The internet worm program: An analysis. Technical Report Purdue Technical Report CSD-TR-823, Purdue University, West Lafayette, IN 47907-2004, 1988.

[117] SPARC International. *The SPARC Architecture Manual Version 8*, 1990.

[118] Brad Spengler. Linux Kernel Advisories. `http://lwn.net/Articles/118251/`, 2005.

[119] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the 33rd Symposium on Principles of Programming Languages*, 2006.

[120] Zhendong Su and Gary Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the 33rd Annual Symposium on Principles of Programming Languages*, pages 372–382, Charleston, SC, January 2006. ACM Press New York, NY, USA.

[121] G. Edward Suh, Jae W. Lee, et al. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[122] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Computer Systems*, 23(1), 2005.

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

[124] David Thomas and Andrew Hunt. *Programming Ruby: the pragmatic programmer's guide*. The Pragmatic Programmers, LLC., Raleigh, NC, USA, 2 edition, August 2005.

[125] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 109–120, New York, NY, USA, 2009. ACM.

[126] Tool Interface Standard (TIS) Committee. *Executable and Linking Format (ELF) specification*, 1995.

[127] N. Tuck, B. Calder, and G. Varghese. Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow, 2004.

[128] US Federal Bureau of Investigation. 2005 FBI computer crime survey, January 2006. `http://www.digitalriver.com/v2.0-img/operations/naievigi/site/media/pdf%/FBIccs2005.pdf`.

[129] US Government Accountability Office. Cybercrime: Public and private entities face challenges in addressing cyber threats, June 2007. Report number GAO-07-705.

[130] Wietse Venema. Taint support for php. `http://wiki.php.net/rfc/taint`, 2008.

[131] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *HPCA*, pages 173–184, 2008.

[132] Alexander Viro. Linux Kernel Sendmsg() Local Buffer Overflow Vulnerability. `http://www.securityfocus.com/bid/14785`, 2005.

[133] Microsoft Windows TCP/IP IGMP MLD Remote Buffer Overflow Vulnerability. `http://www.securityfocus.com/bid/27100`, 2008.

[134] Cheng Wang, Shiliang Hu, Ho-Seop Kim, Sreekumar R. Nair, Mauricio Breternitz Jr., Zhiwei Ying, and Youfeng Wu. Stardbt: An efficient multi-platform dynamic binary translation system. In Lynn Choi, Yunheung Paek, and Sangyeun Cho, editors, *Asia-Pacific Computer Systems Architecture Conference*, volume 4697 of *Lecture Notes in Computer Science*, pages 4–15. Springer, 2007.

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

[136] Fang Xing and Bernhard Mueller. Macromedia Flash Player Array Index Overflow. `http://securityvulns.com/Fnews426.html`.

[137] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[138] Junfeng Yang. Potential Dereference of User Pointer Errors. `http://lwn.net/Articles/26819/`, 2003.

[139] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.