Program Transformation Techniques for Automatic Runtime Detection of Software Exploits

A DISSERTATION PRESENTED

BY

WEI XU

TO

THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

STONY BROOK UNIVERSITY

AUGUST 2009

Stony Brook University

The Graduate School

Wei Xu

We, the dissertation committee for the above candidate for the degree of Doctor of Philosophy, hereby recommend acceptance of this dissertation.

R. Sekar - Dissertation Advisor
Professor of Computer Science

Scott Stoller - Chairperson of Defense
Associate Professor of Computer Science

Rob Johnson
Assistant Professor of Computer Science

Junfeng Yang
Assistant Professor of Computer Science
Columbia University

This dissertation is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Program Transformation Techniques for Automatic Runtime Detection of Software Exploits

by

Wei Xu

Doctor of Philosophy

in

Computer Science

Stony Brook University

2009

The size and complexity of modern software poses a great challenge in the context of securing them against cyber attacks. This factor has motivated research in the development of automated techniques and tools for software vulnerability and exploit detection. These researches fall into two basic categories: one targeted at software developers, while another at end-users and system administrators. Because software vulnerabilities continue to increase despite the efforts taken by developers, in this dissertation, we focus on the latter approach to prevent software vulnerabilities from being exploited in successful attacks. Our research significantly expands the classes of attacks that can be addressed using this approach.

In the first part of our research, we develop a technique that provides comprehensive protection against buffer overflows and other attacks that are based on the lack of memory safety in the C programming language. Unlike previous techniques that were targeted at thwarting the steps involved in typical memory error exploits, our approach gets to the root cause of these attacks, namely, memory errors, and prevents them.

In the second part of our research, we significantly extend the scope of our work to address a new generation of attacks that have emerged in the past several years. This class of attacks, which include SQL injection, command injection, cross-site scripting, path traversal and format string attacks, arise due to input validation errors in applications. As a result of these errors, attackers can cause a vulnerable program to execute operations that can compromise its security. Our technique provides the first systematic solution to this class of attacks by tracking the origin of data within programs, and applying policies that can accurately distinguish between benign uses of untrusted data from attacks.

Together, our techniques are applicable to vulnerabilities that account for over 75% of advisories from US-CERT, and about two-thirds of the vulnerabilities reported by CVE, the industry-standard vulnerability dictionary and the main source of US NIST's National Vulnerability Database (NVD).

Contents

List of Figures **x**

Acknowledgments **xiii**

1 Introduction **1**

1.1 Software Vulnerabilities 1

1.1.1 Memory Error Vulnerabilities 2

1.1.2 Injection Vulnerabilities 3

1.2 Preventive Approaches in Software Development 5

1.3 Our Approach 7

1.3.1 Runtime Defenses Against Memory Error Vulnerabilities 8

1.3.2 Runtime Defenses Against Injection Vulnerabilities 9

1.4 Research Contributions 10

1.5 Dissertation Organization 12

I Memory Safety Enforcement **13**

2 Memory Safety Problem Statement **14**

2.1 Memory Access Errors 14

2.2 Approach Overview 17

2.3 Related Work . . . 20

3 Transformation Framework for Memory Safety 26

3.1 Acceptable Memory Access . . . 26

3.2 Metadata Representation . . . 28

3.3 Detecting Temporal and Spatial Memory Errors . . . 31

3.4 Temporal Capability Management . . . 32

3.5 Transformation of Pointer Statements . . . 33

3.5.1 Pointer creation . . . 35

3.5.2 Pointer assignment . . . 36

3.5.3 Pointer dereference . . . 37

3.6 Transformation of Functions . . . 38

4 Memory Safety with Shadow Metadata Structures 40

4.1 Metadata Representation and Allocation . . . 40

4.2 Capability Store Management . . . 43

4.3 Applying Transformation Rules . . . 44

4.4 Handling Type Casts and Unions . . . 46

4.4.1 Notion of Subtyping . . . 47

4.4.2 Extending Transformation with RTTI . . . 50

4.4.3 Supporting Unions . . . 54

4.5 Optimizations . . . 56

4.5.1 Local Optimizations . . . 56

4.5.2 Possible Global Optimizations . . . 58

5 Memory Safety with Shadow Metadata Memory 61

5.1 Metadata Representation and Management . . . 62

5.2 Handling Arbitrary Type Casts and Pointer Arithmetic . . . 65

5.3 Applying Transformation Rules 67
5.4 Transformation of Functions 70
5.4.1 Supporting Pointer Arguments of Functions 70
5.4.2 Interfacing with Untransformed Functions 70
5.5 Optimizations 72

II Taint-Enhanced Policy Enforcement 74

6 Injection Vulnerability Problem Statement 75
6.1 Injection Attacks and Input Validation Errors 75
6.2 Approach Overview 76
6.3 Related Work 78

7 Motivation for Taint-Enhanced Policies 85
7.1 SQL and Command Injection 85
7.2 Cross-Site Scripting (XSS) 86
7.3 Format String Attacks 87
7.4 Memory Error Exploits 88
7.5 Attacks that “Hijack” Access Privileges 88
7.6 Discussion 89

8 Transformation for Taint Tracking 91
8.1 Runtime Representation of Taint 91
8.2 Transformation for Fine-Grained Taint Tracking 91
8.3 Support for Implicit Information Flow 93
8.4 Optimizations 95
8.4.1 Runtime Optimizations 96
8.4.2 Compile-time Optimizations 97

8.5 Protecting Memory Regions 99

9 Input Marking and Policy Specification 101

9.1 Marking Trusted and Untrusted Inputs 101

9.2 Specifying Taint-Enhanced Policies 102

9.3 Refining Taint-Enhanced Policies 104

9.3.1 Lexical Confinement Policies 105

9.3.2 Syntactic Confinement Policies 106

9.3.3 Other Policy Refinements 107

III Experimental Evaluation 109

10 Experiments: Memory Safety Enforcement 110

10.1 Implementation 110

10.2 Evaluation 111

10.2.1 Effectiveness 111

10.2.2 Performance 111

11 Experiments: Taint-Enhanced Policy Enforcement 118

11.1 Implementation 118

11.1.1 Coping with Untransformed Libraries 118

11.1.2 Injecting Marking and Checking Code 119

11.2 Evaluation 120

11.2.1 Attack Detection 120

11.2.2 False Positives 125

11.2.3 False Negatives 126

11.2.4 Performance 126

12 Conclusions **129**

12.1 Dissertation Summary 129

12.2 Future Research Directions 130

12.2.1 Extending Memory Safety Enforcement 130

12.2.2 Extending Taint-Enhanced Policy Enforcement 131

12.2.3 Supporting Multi-Threaded Programs 131

Bibliography **133**

List of Figures

1.1 Breakdown of CVE COTS software vulnerabilities in 2003-2004. 2

1.2 Example of stack buffer overflow vulnerability and exploit. 2

1.3 SquirrelMail Command Injection [76]. 3

1.4 Number of CVE vulnerabilities by year in 2000-2008 6

1.5 Number of Microsoft critical vulnerability advisories by year in 2004-2008 6

2.1 Examples of temporal memory errors. 15

2.2 Examples of spatial memory errors. 15

3.1 Notations for types, variables and memory blocks. 27

3.2 Basic pointer expressions in C. 28

3.3 Basic pointer statements in C. 28

3.4 Memory safety metadata and helper routines. 29

3.5 Memory safety metadata and guard of pointer expressions. 34

3.6 Transformation of pointer statements. 34

4.1 Defining basic shadow metadata structures for variables. 41

4.2 Representing memory safety metadata with basic shadow metadata structures. 45

4.3 Definitions of memory error checking routines. 46

4.4 Transforming pointer statements with basic shadow metadata structures . . 47

4.5 Definitions of RTTI helper routines `CHECK_RTTI` and `LINK_PTR_ARITH`. . . 51

4.6 Defining RTTI-enhanced shadow metadata structures for variables. 52

4.7 Transforming pointer statements with RTTI-enhanced shadow metadata structures. 53
4.8 Defining RTTI-enhanced shadow metadata structures for unions. 55
4.9 Defining split shadow metadata structures for variables. 57
4.10 Syntax of a simplified C language with STATIC and RTTI pointers. 58
4.11 Type inference rules for identifying STATIC and RTTI pointers. 58
5.1 Generic metadata structure types for shadow metadata memory 63
5.2 Shadow metadata memory layout. 63
5.3 Defining shadow metadata memory for variables. 64
5.4 Definition of helper function `GET_LINK`. 66
5.5 Representing metadata with shadow metadata memory objects. 67
5.6 Initializing block header in shadow metadata memory. 68
5.7 Transformation of pointer assignments and dereferences using shadow metadata memory. 69
8.1 Definition of taint for expressions 92
8.2 Transformation of statements for taint tracking 92
9.1 Illustrative security policies 103
10.1 Transformation performance for Olden/SPECINT benchmarks and UNIX utilities. A ratio 1.22 means that the transformed program was 22% slower/larger than the original. 112
10.2 Performance overheads of checking different memory errors 113
10.3 Effectiveness of optimizations 116
11.1 Attacks used in effectiveness evaluation 120
11.2 Performance overheads of servers. For Apache server, performance is measured in terms of latency and throughput degradation. For other programs, it is measured in terms of overhead in client response time. 125

11.3 Performance overheads of CPU-intensive programs. Performance is measured in terms of CPU time. Overheads in different columns correspond to: (A) No optimizations, (B) Use of local tag variable, (C) B + Use of 2-bit taint value, (D) C + Use of dependency analysis. 125

Acknowledgments

I am grateful to have this opportunity to thank those people who have helped me complete this dissertation.

First and foremost, I thank my advisor, Prof. R. Sekar for his guidance, support, encouragement and great patience in this long journey. His technical contributions to this research are deep, and his comprehensive and constructive comments on this dissertation have helped improve it immensely. I have also learned from him on how to approach challenging and promising research problems. His high standard to strive for perfection in research, writing and presentation has also been a great source of inspiration. I am lucky to have studied under him for my Ph.D. research.

I sincerely thank my committee members: Scott Stoller, Rob Johnson and Junfeng Yang. I have learned a lot from Prof. Scott Stoller's program analysis and security policy classes. His comments on my dissertation have resulted in significant improvements in the sections of taint-enhanced policies, performance optimizations and concurrency issues in multi-threaded programs. Prof. Rob Johnson's witty and insightful comments have covered many details of my dissertation and prompted me to think deeper on these problems and work out the solutions. Prof. Junfeng Yang's comments in my defense are also very valuable and have helped me present the memory safety research work in a clearer way.

I would like to thank other faculty members in Stony Brook who have helped me in these years. Especially, I thank Prof. I.V. Ramakrishnan for his valuable advices on my research and countless help on my administrative matters. It is also a great experience to work as the teaching assistant of Prof. David S. Warren and Prof. Radu Grosu during my first year of Ph.D. study. Their care for students and quality teaching work has set an exemplary model for me.

I was fortunate to work in Secure Systems Lab and have maintained a good working and personal relationship with the passionate and helpful people there. In particular, I thank Daniel C. DuVarney, Sandeep Bhatkar, V. N. Venkatakrishnan and Prem Uppuluri for their great collaboration. I thank Ajay Gupta, Zhenkai Liang, Weiqing Sun, Alok Tongaonkar and Rahul Agarwal for their useful discussions and immense help. I also thank other lab members who together have made our lab such a pleasant working place.

I have met many friends in Stony Brook, and they have made my stay in Stony Brook pleasant and enjoyable. In particular, I thank Hong Liu, Aili Li, Shengying Li, Xin Guan, Feng Qiu and Li Hui for numerous home-cooking dinners that we have shared together. I thank Jun Zheng, Zhunyin Huang and Guanghao Yan for helping me get around many difficulties in daily life during my first years in Stony Brook. I thank Faxing Shen and Jing Luo for having been my long time classmates since my undergraduate study in Peking University and also my roommates in Stony Brook. It is really a comfortable thinking to know that they will always be there when I need help. I also thank Ningning Zhu, Lan Huang and Guizheng Yang for their encouragement in the late and difficult years in my Ph.D. study.

I would like also to give special thanks to my family. I thank my parents and sisters for their endless affection, love and encouragement. They are always my safe harbor in the difficulty times of my life. I thank my wife Chang Zhao for being a dear companion and constant supporter for me for so many years, and my son Alan Yiheng Xu for giving me memorable moments with joy every day. My family has always been my source of strength. Without their support, I could not have finished this research and dissertation.

This research is supported in part by NSF grants CCR-0098154, CCR-0208877 and CCR-0205376 and an ONR grant N000140110967.

Chapter 1

Introduction

1.1 Software Vulnerabilities

Today computer software is pervasive in business and everyday life. Over time, software has become increasingly complex. Flaws in software have widespread impact and financial loss in damage. A study by NIST in 2002 estimates that faulty software costs the US economy about $60 billion dollars annually [66].

A software security vulnerability is a defect within a software system that can cause the software to violate its stated security policy, and can be exploited by an attacker to gain unauthorized access to resources. In the past several years, software vulnerabilities have been the biggest culprit behind cyber attacks. The massively disruptive Internet incidents such as Code Red, Blaster and SQL Slammer are well-known examples of software vulnerability exploitations [6, 9, 8].

Figure 1.1 shows the distribution of commercial, off-the-shelf (COTS) software vulnerabilities in CVE [1], the industry standard vulnerability and exposure name database, in 2003–2004. From this figure, there are two large classes of vulnerabilities that account for about two-thirds of the all these CVE vulnerabilities: memory error vulnerabilities and injection vulnerabilities (including vulnerabilities for format string attacks, SQL injections,

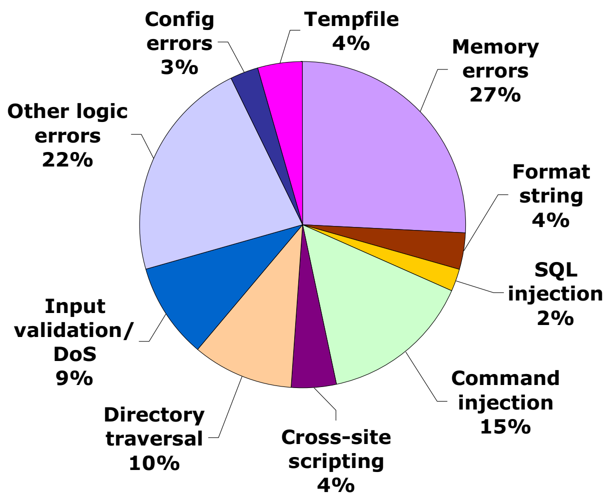

Figure 1.1: Breakdown of CVE COTS software vulnerabilities in 2003-2004.

```
uint *p, *q;
void vulnerable(uint *buf)
{
   uint tinyBuf[10];
   p = tinyBuf, q = buf;
   while (*q) {
      // Bug: No bounds check on p
      *p++ = *q++;
   }
}
```

```
void malicious(void)
{
   void *hugeBuf[64];
   int i;
   for (i = 0; i < 64; i++) {
      hugeBuf[i] = &shellCode;
   }
   vulnerable((uint *)hugeBuf);
}
```

Figure 1.2: Example of stack buffer overflow vulnerability and exploit.

command injections, cross-site scripting and directory traversal). The remaining vulnerabilities are classified as configuration errors, other denial of services or logic errors.

1.1.1 Memory Error Vulnerabilities

The single largest class of vulnerabilities in Figure 1.1 are due to buffer overflows or other memory errors. Memory error vulnerabilities have also dominated the US-CERT vulnerability notes database [5], with more than 80% US-CERT vulnerabilities in 2004–2005 caused by memory errors [19].

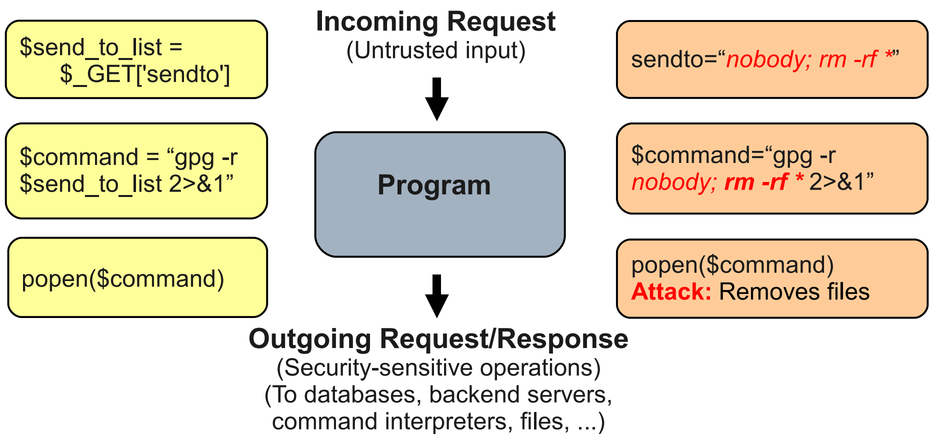

Figure 1.3: SquirrelMail Command Injection [76].

A memory error occurs when the access to an object via a pointer or array expression is beyond the intended lifetime or boundary of the object. Memory errors can be broadly classified into *temporal errors* (such as dangling pointer dereferences and double-free) and *spatial errors* (such as array bounds errors and out-of-bounds pointer dereferences) [13]. Memory error exploits are very versatile, leading to data corruption, arbitrary code execution, resource exhaustion, and so on. Figure 1.2 shows an example of stack buffer overflow vulnerability and an exploit that takes advantage of this vulnerability by calling the `vulnerable` function with a larger-than-expected, malicious argument in order to overwrite the return address on stack with a shell code address.

1.1.2 Injection Vulnerabilities

Injection vulnerabilities, which include SQL injections, command injections, cross-site scripting, path traversal, format string vulnerabilities, and so on, have been rapidly increased in Web applications in recent years. They account for more than one-third of CVE vulnerabilities in 2003-2004 as shown in Figure 1.1. In 2006-2007, injection vulnerabilities has increased to more than 50% of CVE vulnerabilities as reported by [76].

These vulnerabilities also constantly rank at the top of the ten most serious web application vulnerabilities published by OWASP [67].

Figure 1.3 uses shell command injection as an example to show the general context of injection attacks [76]. On the left of the figure shows a PHP program code snippet in SquirrelMail that is vulnerable to shell command injections. An attack that exploits this command injection vulnerability is displayed on the right, with the attacker provided input values in italics.

In general, the attack target of an injection attack is a program that takes requests from untrusted sources and responds accordingly. For each request, the target application determines the operations to perform and the parameters to those operations. The operations are usually security-sensitive and are performed on back-end resources such as database servers, files and command interpreters. To prevent an untrusted user from controlling security-sensitive operations, the target application should validate all untrusted inputs. Due to software bugs, however, such input validations might be incomplete, or even missing from some program paths. An attacker can exploit such input validation vulnerabilities to inject malicious operations or parameters through the input requests. In the above example, for instance, the attacker is able to inject a malicious shell command "`rm -rf *`" into the vulnerable program.

We would like to point out that memory error vulnerabilities could be seen as injection vulnerabilities as well. However, we make this distinction because we can have a more comprehensive and precise defense against exploits of memory error vulnerabilities by ensuring runtime memory safety.

1.2 Preventive Approaches in Software Development

Software developers and vendors have been spending a lot of time and effort, with the aid of commercial and research tools and techniques, to identify and fix these vulnerabilities. A notable example is the Microsoft effort to boost its software security and actively fight against software vulnerabilities [7, 2].

Extensive testing has been established as one of the standard practices to find software bugs and vulnerabilities in the life cycle of software development. Many security-targeted testing techniques such as penetration testing [12] and fuzz testing [58] have been developed and widely used. However, it is generally infeasible to construct test cases and inputs that could exercise every possible code execution path, which leaves significant room for vulnerabilities to escape from the testing phases.

A large body of research has relied on static analysis techniques to discover potential software vulnerabilities, including buffer overflows and memory errors [86, 72, 52, 89, 39, 34], user/kernel pointer bugs [48], format string vulnerabilities [78] and SQL injections and other vulnerabilities in web applications [53, 46, 88]. Static code analysis could, in theory, cover every possible code path in a program and thereby find all potential vulnerabilities statically. In practice, however, one of the main problems of static analysis techniques is the lack of accuracy. This inaccuracy comes from the approximations used in static analyses to abstract inputs, memory aliases, loops, dynamic function dispatches and other aspects of runtime behaviors that cannot precisely be predicted statically. Due to the inaccuracies, static analysis techniques can suffer from a high rate of false positives. They are also incomplete and suffer from false negatives.

In spite of these efforts from developers to get rid of vulnerabilities, new vulnerabilities continue to surface in recently released software as well as in older software. The total number of software vulnerabilities continues to escalate from one year to the next. As shown in Figure 1.4, thousands of newly discovered vulnerabilities are reported in CVE

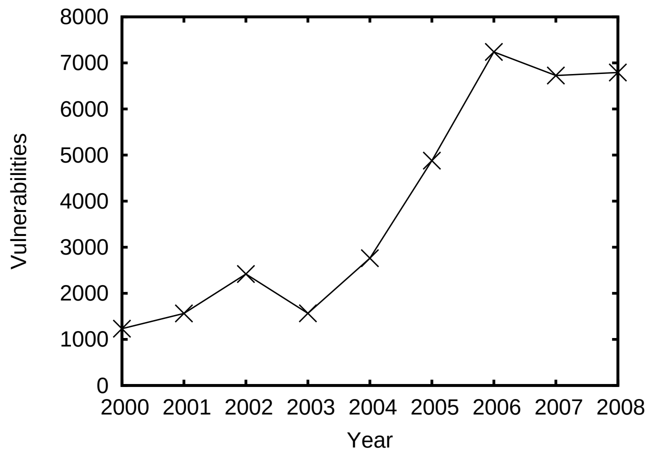

Figure 1.4: Number of CVE vulnerabilities by year in 2000-2008

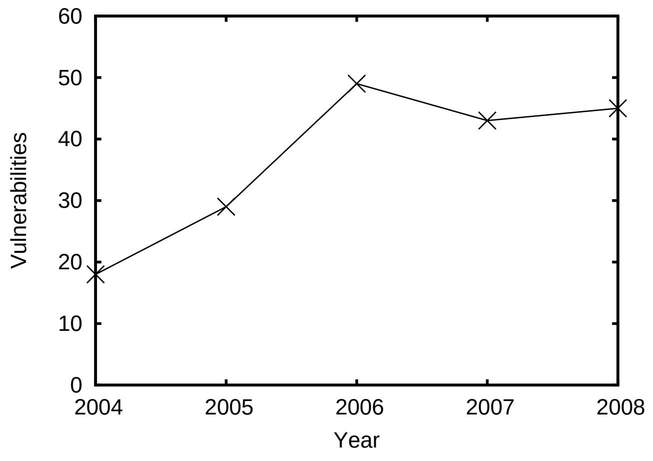

Figure 1.5: Number of Microsoft critical vulnerability advisories by year in 2004-2008

every year. The annual number of CVE vulnerabilities has increased to more than 400% in 2008 compared to 2000. Similarly, a significant number of critical vulnerabilities in Microsoft products [2] continue to be revealed every year as shown in Figure 1.5.

1.3 Our Approach

We recognize that it is very hard for developers to eliminate software vulnerabilities, especially from legacy software. We seek to cope with and remedy these issues differently. Our goal is to develop techniques that be can used by the software operators to achieve a much greater accuracy in detection of common exploits. For this purpose, testing and static analysis techniques are not appropriate.

In light of this, we focus our efforts on providing tools and techniques to automatically detect and prevent exploitation of vulnerabilities at runtime. We look for techniques that can effectively stop exploits of the most common software vulnerabilities, but do not require significant effort of programmers or manual source code modifications.

In this dissertation, we present new source code transformation based techniques that can enable the automated runtime detection and defense of memory errors and injection vulnerabilities, the two largest classes of software vulnerabilities shown in Figure 1.1. Our source-code transformation techniques have been applied to C programs. Even though these techniques are equally applicable for C++ programs as well, they are not implemented for C++ in our work.

We focus our techniques on C/C++ programs because most of today's important server applications (such as the Apache Web server, the Postfix email server and the OpenSSH server), the interpreters of many of the most popular scripting languages (such as Bash and PHP), and a large number of useful utility programs (such as tar and gzip) are still written in the C/C++ language. Techniques developed for C/C++ programs can be directly applied to these applications and thus bring the protection and benefits offered by the techniques to a large fraction of presently important applications. Moreover, vulnerabilities such as memory errors affect C/C++ programs the most because of the weak type system and the ability to use pointers unrestrictedly in C/C++. Consequently, techniques that detect memory errors and exploits are primarily concerned with C/C++ programs.

1.3.1 Runtime Defenses Against Memory Error Vulnerabilities

Attackers can exploit memory error vulnerabilities in a variety of ways to execute arbitrary code, steal sensitive information, alter application logic, corrupt important data, or cause denial of service. The exploitations are usually carried out by taking advantage of the memory error vulnerabilities to overwrite important program control or non-control data structures. The common targets of such overwrites include stack return addresses, function pointers, heap management metadata, security-critical non-control data [25], and so on.

A large number of runtime memory error defense techniques have been developed, which can be categorized into *exploit prevention techniques* and *memory error detection techniques*.

Exploit prevention techniques defeat memory error attacks by disrupting specific steps required in a successful attack. Often, these techniques prevent the corrupted memory from being used to the attacker's advantage. For instances, StackGuard [30] prevents the use of function return addresses corrupted by buffer overflows. PointGuard [29] uses encryption to prevent dereferences of corrupted function and data pointers. Address space randomization techniques such as [18, 19, 90] rely on hiding secrets, such as randomized memory addresses, from the attackers. Techniques such as DieHard [17] and boundless buffers [71] mask and tolerate memory errors by approximating infinite-sized heap or buffers.

Many exploit prevention techniques are only effective against specific types of memory error attacks. More general exploit prevention approaches, such as address space randomization and the infinite-sized heap or buffers techniques, can only provide a probabilistic protection against memory error exploits. As a side effect of these approaches, the behaviors of protected programs can become unpredictable in presence of memory errors. Furthermore, recent researches [77] show that the address space randomization techniques could be defeated by obtaining the secrets through brute-force attacks or information leakage attacks.

Memory error detection techniques can catch memory errors at the time when these errors occur without waiting till the time when the exploits use the corrupted data or pointers. Such techniques have the advantage of not only stopping the exploits, but also identify the exact vulnerabilities to be fixed. However, existing memory error detection techniques suffer from one or more following drawbacks:

- Incomplete coverage of memory error detection;
- Incompatibility with existing C programs due to changed pointer representations;
- Need for significant source code modifications;
- Reliance on garbage collection or other non-standard C memory management;
- Excessive performance overheads.

In this dissertation, we develop source code transformation based techniques to detect all spatial and temporal memory errors at the memory allocation block level in C programs. Compared to the previous approaches, our techniques preserve pointer representations and standard C memory management, provide backwards compatibility with existing C programs, and require no hardware modifications and almost no source code modification. In addition, our techniques promise reasonable performance overheads. All of these make our technique suitable for providing the protection against memory error exploits for numerous widely used applications that are written in C.

1.3.2 Runtime Defenses Against Injection Vulnerabilities

A number of runtime approaches that have been proposed to address specific types of injection attacks. Among them, FormatGuard [28] detects format-string attacks on the `printf` family functions. SQLrand [21] detect SQL injection attacks through SQL instruction-set randomization, while AMNESIA [42] relies on lexical analysis of SQL queries. Although these techniques can be effective against their targeted attacks, they usually cannot be extended to detect other types of injection vulnerability exploits.

Fine-grained dynamic taint analysis has recently been proposed in [83, 24, 64] and used to detect control flow attacks. Approaches such as [65, 70, 81] have also explored the dynamic tainting based techniques to detect SQL injection attacks. However, the above approaches are all limited to specific types of injection attacks as well. Furthermore, the taint tracking in these approaches is either inaccurate [81], or very expensive [64], or requires significant hardware [83, 24] or source code modifications [65, 70].

We identify that the root cause of injection vulnerability exploits is due to untrusted input data being used unsafely in security operations. Based on this key observation, in this dissertation, we develop a unified, dynamic analysis based framework called *taint-enhanced policy enforcement* to detect all the attacks that exploit injection vulnerabilities. In this framework, the propagation of untrusted (tainted) input data is tracked at runtime and the unsafe use of such untrusted data in arguments to security operations is prevented by security policies expressed over both the argument values and their taint information. We also develop the first automated source-code transformation for C programs that enables an efficient fine-grained (byte-level) dynamic taint tracking in transformed programs to serve as the basis of this framework.

1.4 Research Contributions

In this dissertation research, we have made the following contributions:

- We have developed the first backwards-compatible, comprehensive memory error detection approach for C programs which possesses all the following properties:

  - It provides a complete coverage of both spatial and temporal memory errors at the block level.
  - It separates metadata from pointers themselves and preserves the layout of C data structures for backwards-compatibility with existing C programs.
  - It supports type casts and pointer arithmetic that are pervasive in C programs, without the need of source code modifications.
  - It does not rely on garbage collection and retains the explicit memory management model in C.

  Furthermore, our source-to-source transformation has also significantly reduced the performance overheads over previous complete memory error detection approaches.

- We have developed the taint-enhanced policy enforcement framework, which is one of the first unified approaches that apply fine-grained dynamic taint analysis to detect a broad range of attacks that exploit injection vulnerabilities induced by input validation errors. In our framework, security policies are augmented with byte-level taint information of the data to reason about their trustworthiness. By enforcing such policies on security-sensitive operations, we can detect injection attacks such as SQL injection, shell command injection, cross-site scripting, path traversal, format string attacks, and so on.

- We have developed the first efficient, automated, source code based dynamic taint tracking technique for C programs. Our technique is built upon a source-code transformation and provides the byte-level taint tracking at runtime. It serves as the basis of our taint-enhanced policy enforcement framework. Moreover, we have shown that by transforming the PHP interpreter engine, which is written in C, we can also achieve fine-grained taint tracking in PHP web applications.

1.5 Dissertation Organization

This dissertation is structured as follows. In Chapter 2 – Chapter 5 we present the transformation techniques for ensuring memory safety in C programs, which can prevent attacks that exploit memory errors. In Chapter 6 – Chapter 9, we then describe the transformation techniques for achieving taint-enhanced policy enforcement, which can be used to detect a wide range of attacks that exploit injection vulnerabilities due to input validation errors. We provide the experimental results in Chapter 10 and Chapter 11. Finally, we conclude this dissertation and outline the future research work in Chapter 12.

Part I

Memory Safety Enforcement

Chapter 2

Memory Safety Problem Statement

2.1 Memory Access Errors

The C programming language is commonly used for systems programming because of its speed and the precise control it provides over memory allocation. Unfortunately, this control is more than most programmers can fully manage, as evidenced by the profligate frequency of bugs such as memory leaks, dangling pointers, and buffer overruns in commercial C programs. Such memory errors are one of the most common reasons for software failures. Moreover, these errors are hard to track down, and hence contribute significantly to the effort needed for testing and debugging C programs. Even worse, memory errors are the culprit behind most of today's security vulnerabilities in software.

Intuitively, a memory error occurs in C/C++ programs when the access to an object via a pointer or array expression is beyond the intended lifetime or boundary of the object. We call the object being accessed through the dereference of a pointer (or array) as the *referent* of the pointer (or array).

Initially proposed by SafeC [13], memory errors can be broadly classified into temporal errors and spatial errors, as described below:

```
p = (char *)malloc(32);
q = p;
free(p);
*q;                     // Error: Dangling pointer dereference
r = (int *)malloc(8);   // Reuse the above freed memory
*q;                     // Error: Access to reallocated memory
free(q);                // Error: Double free
q = &r;
free(q);                // Error: Invalid free
```

Figure 2.1: Examples of temporal memory errors.

```
p = (char *)malloc(32);
q = p - 1;
*q;                     // Error: Buffer underflow
q = p + 40;
*q;                     // Error: Buffer overflow
q = p + 30;
r = (int *)q;
*r;                     // Error: Buffer overflow
```

Figure 2.2: Examples of spatial memory errors.

- *Temporal memory error*, which occurs when dereferencing a pointer whose referent has outlived its intended lifetime (e.g. the referent has been freed previously). Figure 2.1 shows several examples of temporal memory errors, including dangling pointer dereference, access to reallocated memory, invalid free and double-free. Of particular significance are temporal errors involving a pointer whose referent has been freed and subsequently reallocated to a different, unrelated object. In this case, a dereference of this pointer can read or modify the unrelated object. Moreover, the modification could go undetected until the other object is accessed, which may happen much later in the program's lifetime. This means that the symptom of the error will be spatially and temporally removed from the underlying erroneous pointer operation, making debugging very difficult.

- *Spatial memory error*, which occurs when dereferencing a pointer whose referent has gone beyond its intended spatial boundaries. Common spatial errors include array bounds errors and dereferences of out-of-bounds pointers obtained by pointer arithmetic or type casts. Figure 2.2 shows examples of buffer underflow and overflow spatial memory errors.

The dereferences of uninitialized or NULL pointers are special and can be classified as either temporal memory errors or spatial memory errors.

Attackers can exploit memory error vulnerabilities to execute arbitrary code, steal sensitive information, alter application logic, corrupt important data, or cause denial of service. The common exploitation techniques include stack return address overwriting, function pointer overwriting, heap management metadata overwriting, security-critical non-control data overwriting, and so on.

The distinction between spatial and temporal memory errors is important because much of earlier work [51, 80] has addressed spatial errors, but only provided limited support for the detection of temporal errors. Approaches such as CCured [62] and Cyclone [47] avoid temporal errors by ignoring `free` operations. They rely on a garbage collector (specifically, the Boehm-Demers-Weiser conservative garbage collector [20]) to free unused memory. In addition, local variables accessed via pointers are moved from the stack to the heap to ensure that they are not deallocated on function returns. This garbage collection based technique has helped these approaches avoid the overheads of temporal error checking. However, even though garbage collection has been enjoying its prime time in type-safe languages such as Java, it has not yet found wide acceptance in the type-unsafe C/C++ world. Prior approaches that handle temporal errors, such as [49, 73, 13, 44], have suffered high performance penalties, typically ranging from a few hundred percent to a thousand percent. Even among these approaches, it is common to handle just the first class of temporal errors mentioned above, but not the second class (i.e., dangling pointer to reallocated memory). Techniques such as [49, 73] fall in this category.

2.2 Approach Overview

As discussed in Section 1.3.1, there exist a large number of runtime defense techniques against memory error exploits. However, history has shown that memory corruption attacks are very versatile — as defenses are developed, attackers develop alternative attacking techniques. This is evidenced by the proliferation of memory error exploitation techniques and reported memory error vulnerabilities.

In recognizing this, we focus our research on detecting all memory errors, both spatial and temporal errors, which prevent the root cause of memory error attacks. The ultimate goal of memory error detection, which is also the objective of our work, is to develop an approach which possesses the following essential properties:

- $P1$: The ability to detect all memory errors, including spatial and all kinds of temporal errors.

- $P2$: Backward-compatibility between protected C programs and existing, unprotected libraries.

- $P3$: The ability to handle all C programs without modification. Because of the sheer size of existing C programs, even if only about 5% of the code needs to be rewritten, it will have prohibitively high cost.

- $P4$: Preservation of the explicit memory management model in C. As we discussed in Section 2.1, an ideal approach should not rely on garbage collection to ensure temporal memory safety in C programs.

- $P5$: Acceptable performance overhead.

To detect spatial and temporal errors at runtime, extra information (henceforth called metadata) has to be maintained with each pointer or allocated object. Some previous approaches have used *fat pointers*, which store metadata together with the pointer. Unfortunately, this changes the underlying pointer representation, thereby changing layouts of C-struct's. This creates a number of compatibility problems:

- *Breaks many programs.* Systems programs often make assumptions regarding the size of pointers and the layouts of structures. For instance, they may assume that the size of integers and pointers will be the same. Another typical assumption is that, in a union consisting of an integer and pointer, a value stored as a pointer may be read as an integer. These assumptions are no longer valid when normal pointers are replaced by fat pointers, and hence such programs need to be modified before they will work correctly with the transformation.

- *Compatibility with libraries.* Since fat pointers change structure layouts, compatibility with precompiled libraries is lost. (Such libraries will access fields in structures using offsets based on untransformed layout of structs, and hence will access incorrect fields or maybe the metadata stored within the fat pointers.)

Even though the memory error detection problem has been studied extensively for many years, it still remains as an open problem to develop a practical approach for detecting temporal as well as spatial memory errors. This is because existing memory error detection techniques fail to meet one or more of the desired properties. More specifically, some techniques can only detect a subset of memory errors. For instances, Purify [44] uses red zones to detect out-of-bound pointer references, but it cannot detect erroneous dereferences of an out-of-bound pointer that passes the red zones surrounding its referent and points to a valid adjacent memory block. Libsafe [15] and LibsafePlus [14] detects buffer overflow errors only in standard C library functions by providing a checked version of these functions. There are techniques aiming at a more comprehensive detection of memory errors. Among them, CCured [62] and Cyclone [47] rely on garbage collection and also requires significant source code modifications. SafeC [13] and Patil & Fischer's work [69] do not handle type casts that are pervasive in C programs. SafeC, CCured and Cyclone change the pointer representations as well. Jones and Kelly's bounds-checking C [49] and the extended work CRED [73] maintain the bounds metadata for each allocated object in a splay tree. These approaches have achieved a good backwards-compatibility. But they suffer from excessive performance overheads. More recently, baggy bounds checking [11] follows up Jones and Kelly's work to also look up the bounds information from bounds tables using the memory address value stored in each pointer. It significantly reduces the runtime performance overheads by constraining the memory allocation size and alignment to compact bounds metadata and speed up the metadata lookup. All these three approaches are limited in temporal memory error detection and cannot detect reallocation temporal errors, either. SoftBound [61] is another recent bound checking only approach. It associates bounds metadata with each pointer, rather than with each allocated object, and uses a global data structure (a hashtable or contiguous array) to store metadata of in-memory pointers in order to support arbitrary pointer type casts. SAFECode [33, 32] relies on Pool Allocation in its spatial and temporal memory error detection. Its temporal memory safety protection technique could exhaust the virtual memory space, or risk the incomplete detection of reallocation temporal memory errors.

In contrast, we develop source-to-source transformations to detect all temporal and spatial memory errors in C programs. Our technique detects memory access errors at the level of *memory blocks*, which correspond to the least units of memory allocation, such as a global or local variable, or memory returned by a single invocation of `malloc`. It flags memory errors only at a point where a pointer is dereferenced. Other operations, e.g., the assignment of a pointer from an arbitrary expression, will not cause errors.

Due to the afore-mentioned problems with fat pointers, we use an alternative approach that separates metadata from the pointers, thereby preserving the layout of structures. In addition, our technique can also handle important unique features in C, such as arbitrary type casts, unions, and pointer arithmetic, without modifications to the program source code. Another key feature of our approach is that all spatial and temporal errors are promptly detected, without changing the memory allocation model and using garbage collection.

In short, our approach satisfies the properties of $P1$, $P2$ and $P4$. It also satisfies the property $P3$ with very few exceptions (e.g. replacing certain user-defined memory allocation functions with `malloc`) due to the limitations of our implementation. For the performance property $P5$, our approach has reduced the overheads by at least a factor of two compared to previous approaches (e.g. [13, 68, 49]) that are aimed at detecting spatial as well as temporal memory errors. We would also like to note that it is still acceptable for an approach that can detect all memory errors to have a relatively high overheads. The reason is that what is exploitable has always remained unknown and the best approach is to protect all vulnerabilities, especially when they relate to type and memory errors, which have historically led to many unanticipated effects.

We describe our common transformation framework for detecting memory errors in Chapter 3, and present two applications and extensions of this transformation framework in Chapter 4 (using shadow metadata structures) and Chapter 5 (using shadow metadata memory). The implementation and experimental evaluation of our approach are described in Chapter 10.

2.3 Related Work

Much research effort has been put into the development of techniques for detecting memory access errors in C programs. We discuss works that are most closely related to ours.

Approaches that rely on garbage collection Cyclone [47] is a variant of C that is designed with the express goal of reducing source code changes needed to convert C programs to Cyclone. However, it still makes significant changes to C-language syntax and semantics, e.g., C-style unions are replaced by ML-style unions. In addition, to support separate compilation, type declarations that provide detailed information about pointer types become necessary, and these declarations cannot be automatically generated. As a result, large C-programs require non-trivial effort to port to Cyclone. In contrast, our approach requires almost no changes to existing C-programs.

CCured [62] uses type-inferencing to differentiate pointers into *(definitely) safe* and *(potentially) unsafe* pointers at compile-time, and insert run-time checks to ensure validity of accesses through unsafe pointers. Unsafe pointers are further subdivided into those using pointer arithmetic (sequential pointers) and those using casts (dynamic pointers). The representation of safe pointers is unchanged, but a fat-pointer representation is used for unsafe pointers. This can cause some compatibility problems, as described in Section 2.2. In their more recent work [27], they have separated metadata associated with sequential pointers to gain increased compatibility, especially with external libraries that often take pointers to arrays or character buffers. However, dynamic pointers continue to use fat pointer representation. To further reduce compatibility problems and to improve performance, they have added runtime type information, which can result in fewer pointers being classified as dynamic. However, the maintenance of RTTI is governed by programmer annotations, unlike our approach where it is automatic. In addition, their notion of subtyping is more restrictive than ours, so some programs that require the use of dynamic pointers can still continue to work with our notion of RTTI.

The most important difference between our approach and that of Cyclone or CCured is their reliance on garbage collection. While this change may be acceptable for some programs, garbage collection has not yet be widely accepted in the C/C++ community.

Approaches targeting compatibility with the C-runtime model *Safe-C* inserts runtime checks to detect all memory errors. However, the approach introduces compatibility problems due to the use of fat pointers. These compatibility problems were addressed by Patil and Fischer [69, 68], but their performance overheads are still much higher than ours — by as much as a factor of 2 to 4 times over our approach. Moreover, neither of these approaches support downcasts, thereby significantly restricting the set of programs to which they can successfully be applied.

SoftBound [61] is similar to our spatial memory error detection approach in that it also associates bounds metadata with each pointer and separates the metadata from the pointer itself. SoftBound stores metadata for in-memory pointers in a global hash table or contiguous array, which enables it to support arbitrary pointer type casts. SoftBound can also support pointer bounds narrowing for sub-object bounds error detection, though this may cause backwards-compatibility issues with existing C programs. SoftBound provides a complete coverage for spatial memory errors, but lacks of the ability to detect temporal memory errors. Its runtime overheads for spatial memory error detection is comparable to our approach.

Jones and Kelly [49] use a splay tree to store runtime information about every allocated memory block. Pointer arithmetic and dereferences are checked using this splay tree data structure to ensure the pointer validity. Their work has a very good backwards-compatibility and is applicable to a broad class of C programs, including programs that cast pointers to integers and vice-versa. Unfortunately, the approach incurs heavy performance penalties (more than 10 times slowdown for pointer-intensive code). Moreover, it does not detect dangling pointers to reallocated memory. CRED [73] extends Jones and Kelly's work to introduce a better handling of out-of-bounds pointers and thus improve the compatibility with existing C programs. When checking for the same set of memory errors, the performance of CRED is similar to that of Jones and Kelly's.

Baggy bounds checking [11] also takes the referent object based approach as pioneered by Jones and Kelly. Similarly, baggy bounds checking maintains bounds metadata information for each allocated object and looks up the metadata from bounds tables with the source pointer value in pointer arithmetic expressions. To significantly reduce the runtime overheads, this approach constrains the memory allocation size and alignment in order to compact the bounds metadata representation and speed up the metadata lookups and updates. Baggy bounds checking cannot detect reallocation temporal memory errors.

SAFECode [33, 32] relies on Pool Allocation in its spatial and temporal memory error detection and uses whole-program analysis to perform automatic pool allocation. Its spatial memory error detection technique improves over Jones and Kelly's work by providing a separate splay tree for each pool. It also optimizes the out-of-bounds pointer handling in CRED through the use of hardware memory protection for detecting out-of-bounds pointer dereferences. Its temporal memory error detection technique allocates a new virtual page for each memory allocation, and uses the pool information to alleviate the virtual memory exhaustion problem. However, this technique interacts poorly with long-lived objects and memory reachable from these objects, which could lead to virtual memory address exhaustion in practice. In contrast, our approach uses a unique capability for each new allocation. As we will describe in details in Section 3.3, the uniqueness of capability is determined by both the capability slot address and the capability index value. Because we use 32-bit index values to reuse capability slots and each capability slot is only 4-byte long (compared to 4KB per allocation in SAFECode), the virtual memory exhaustion problem is practically a non-issue in our approach.

Debugging-targeted approaches These approaches typically operate by inserting checks into a program either at the source or binary level, with the goal of detecting certain subclasses of memory errors. These include Kendall's *bcc* [51], *Purify* [44], Steffen's *rtcc* [80], Loginov, et.al.'s work on *runtime type checking* [54], and Haugh and Bishop's STOBO tool [45]. Also in this category is the *CodeCenter* interpretive debugger [50]. Since these techniques are focused on debugging, performance is typically not a serious consideration. For instance, Purify and CodeCenter often have overheads in excess of 1000%. In contrast, our interest is in techniques that can be enabled in production code, so it is important to keep the overheads reasonably low. Another distinction is that our approach is focused on detecting all memory errors, while the above approaches generally target detection of a subset of errors that are commonly encountered. For instance, bcc and rtcc focus mainly on spatial errors. Similarly, Purify does not detect certain spatial errors (specifically, when pointer arithmetic causes a pointer to overshoot past the end of one object into the middle of the next object), or dangling pointers to reallocated memory. On the other hand, some of these approaches do provide better compatibility with external libraries, e.g., Purify can perform memory error checks within libraries that are provided in binary format.

Other approaches There are a number of exploit prevention techniques that utilize code transformation to protect return addresses of activation record [30, 15, 26, 35]. More recently, works have emerged that use address space layout randomization to achieve broad protection against attacks that exploit memory errors [18, 29]. Approaches like DieHard [17] and boundless buffers [71] approximate infinite-sized heap or buffers to mask and tolerate many out-of-bounds memory errors. While these techniques are good in achieving their intended goal, namely, protecting against certain memory error attacks, and doing so with low overheads, they do not provide a comprehensive and deterministic protection against memory errors and their exploits.

There have been many approaches which rely on static source code analysis to detect potential security-related memory errors prior to execution [34, 39, 89, 52, 86, 72, 38], but these are not as powerful as runtime detection techniques and hence not used widely.

Several techniques that combine static analysis with dynamic checking have also emerged. Yong and Horwitz [93] presented an approach to detect invalid memory writes by combining static analysis and a run-time technique similar to Purify [44]. By reducing the size of checked memory regions, they achieved both good run-time performance and improved error detections. Their performance overheads are generally lower than ours. However, this result is achieved by omitting error checks for read operations, which are typically much larger in number than write operations. Moreover, because of its reliance on Purify-like techniques, it cannot detect all memory errors. WIT (white integrity testing [10]) uses points-to analysis to compute the set of objects that can be written by each instruction. It then instruments the program to enforce the write object set at runtime. WIT has a very low runtime overheads. However, it does not detect invalid memory reads. DFI (data flow integrity [23]) computes reaching definitions using static analysis and instruments the program to enforce that the definition reaching each read at runtime is in the set of reaching definition identifiers computed statically. Both WIT and DFI subject to the precision of the static points-to analysis.

Lhee and Chapin [84] developed a technique called type-assisted dynamic buffer overflow detection, in which the compiler is extended to provide information about the sizes of buffers and library calls are intercepted. Their technique detects many errors with fairly low overhead, but focuses only on array-based spatial errors that occur within a pre-specified set of library calls. Dynamically sized automatic arrays (i.e., those allocated by the `alloca` function) are not supported by their approach, either. Avijit, Gupta, and Gupta [14] invented a very similar approach called TIED/LibsafePlus which operates on binary (rather than source) code.

Chapter 3

Transformation Framework for Memory Safety

3.1 Acceptable Memory Access

Our approach provides memory safety to C programs at the granularity of *memory blocks*. A memory block corresponds to the least unit of memory allocation, such as a global or local variable, or memory returned by a single invocation of dynamic memory allocation function such as `malloc`.

All the memory accesses through program variables without involving pointer dereferences or array indexing are guaranteed to be safe because these memory accesses are checked, by the compiler, against the statically declared types and scopes of the variables. A memory access involving pointer dereferences or array indexing, however, could be unsafe. Because array indexing is a special case of pointer dereferences, we define the rules for acceptable pointer-induced memory access below. Any pointer dereferences that violate these rules are memory errors and not acceptable.

- Each pointer is bound to exactly one memory block, which is the *referent block* of

Notation	Comments
τ	A type
x	A variable
n	An integer variable
p, q	A pointer variable
s	A structure or structure pointer variable
$s.d$	Field d of structure s
e	A pointer expression
$R(p)$	The referent of pointer p. We can access it through $*p$.
$R_b(p)$	The memory block where the pointer referent $R(p)$ belongs to.
b	A memory block, which can be either a variable x or a pointer referent block $R_b(p)$.
r	Return value of a function.

Figure 3.1: Notations for types, variables and memory blocks.

the pointer. Uninitialized pointers are bound to a special, always invalid memory block. Only pointer creations and assignments can change the binding of a pointer to its referent block. Pointer arithmetic and type casts do not change this binding. Note that pointer assignments also include the assignments between C structs that contains pointer fields.

- A memory access via a pointer should respect the boundary of the pointer referent block. Any pointer dereferences that cross the pointer referent block boundary are disallowed. Note that it is acceptable for a pointer to have an out-of-bounds value, as long as the pointer is not dereferenced with such a value.

- A memory block deallocation removes the block and invalidates all the bindings to this block. Any further access to the deallocated block by dereferencing a pointer is disallowed, even after the memory bytes have been reallocated to another object.

	Expression	As L-value
[simple pointer]	p	Yes
[field pointer]	$s.d$	Yes
[address-of]	$\&x$	No
[field address-of]	$\&s.d$	No
[pointer arithmetic]	$p+n$	No
[pointer type cast]	$(\tau)x$	No
[pointer dereference]	$*p$	Yes
[struct pointer dereference]	$s{-}{>}d$	Yes

Figure 3.2: Basic pointer expressions in C.

[heap allocation]	$p = \texttt{malloc}(n)$
[heap deallocation]	$\texttt{free}(p)$
[pointer assignment]	$e_1 = e_2$

Figure 3.3: Basic pointer statements in C.

3.2 Metadata Representation

Figure 3.1 shows the notations that we use to represent types, variables and pointer referents in this chapter. In particular, we use p to denote a pointer variable and $s.d$ to denote a field pointer d in a structure s. We define the memory bytes being accessed via $*p$ as the referent of p, denoted as $R(p)$. We also define the memory block to which $R(p)$ belongs as the referent block of the pointer, denoted as $R_b(p)$. Note that $R_b(p)$ can be the same as $R(p)$, but more likely $R_b(p)$ is a superset of $R(p)$. For example, when p points to an element in an array, $R(p)$ is just that element, while $R_b(p)$ is the entire array.

Figure 3.2 lists the basic pointer expressions in the C language, which include simple pointer variable, field pointer, address-of, pointer arithmetic, pointer type casts and pointer dereferences. Figure 3.2 also shows which pointer expressions can be used a l-value in an assignment. Figure 3.3 lists the basic statements involving pointers in C, which include allocation, deallocation and pointer assignments. In a pointer assignment $e_1 = e_2$, e_1 has to be an expression that can be used as a l-value. For example, $*p$ is allowed to be e_1, while

Notation	Comments
$B(b)$	Base address of block b.
$N(b)$	Allocation size of block b.
$C(b)$	Temporal allocation capability of b.
$W(x)$	Base address of metadata for pointer in variable x.
$W(s,d)$	Base address of metadata for field pointer d in structure variable s.
$L(x)$	Compile-time type size of variable x.
$L(s,d)$	Compile-time type size of field d in structure variable s.
$M(p)$	Metadata attributes $\langle M_b, M_{C'}, M_W \rangle$ for pointer p.
	Note that $W(p)$ = &$M(p)$.
$M_b(p)$	Address of referent block metadata attributes $\langle M_b^B, M_b^N, M_b^C \rangle$.
$M_b^B(p)$	Base address of $R_b(p)$. Same as $B(R_b(p))$.
$M_b^N(p)$	Allocation size of $R_b(p)$. Same as $N(R_b(p))$.
$M_b^C(p)$	Temporal allocation capability of $R_b(p)$. Same as $C(R_b(p))$,
	which is updated upon allocation and deallocation of $R_b(p)$.
$M_{C'}(p)$	Cached temporal capability value of $R_b(p)$.
	Updated upon pointer assignment to p.
$M_W(p)$	Base address of metadata for pointer in $R(p)$, i.e., $W(R(p))$.
$M_W(p,d)$	Base address of metadata for field pointer d in $R(p)$, i.e., $W(R(p),d)$.
G_H	Guard routine to detect erroneous heap memory deallocations.
G_T	Guard routine to detect temporal memory errors in pointer dereferences.
G_S	Guard routine to detect spatial memory errors in pointer dereferences.
$G(e)$	Guard code to detect memory errors of pointer dereferences in e.

Figure 3.4: Memory safety metadata and helper routines.

$\&x$ is not. Note that we use a three-address code form to simplify our transformation and description. We would also like to point out that we do not list the array subscript dereference form $p[n]$ in Figure 3.2 because it is only a syntactic sugar and can be represented as $*p_1$ where p_1 has the value of $p + n$.

We define the metadata and helper routines for ensuring memory safety in Figure 3.4. For each memory block and its content, we define the following metadata attributes:

- $B(b)$: The base address of memory block b. When b is a variable x, we can use $\&x$ as the value for $B(x)$.

- $N(b)$: The allocation size of memory block b.

- $C(b)$: The allocated temporal capability of memory block b. An allocated temporal capability is an address in a capability store which stores a capability index value. Section 3.3 provides more details on capability and its index value.

- $W(p)$, $W(s,d)$: $W(p)$ is the base address of metadata for pointer variable p, while $W(s,d)$ is the base address of metadata for field pointer d in structure variable s.

- $L(x)$, $L(s,d)$: $L(x)$ is the compile-time type size of variable x, while $L(s,d)$ is the compile-time type size of field d in structure variable s. We can use `sizeof` to represent L. For instance, $L(x)$ is `sizeof`(x). Note that $L(x)$ has the same size as $N(x)$ when x does not involve any pointer dereferences.

For each pointer p, we maintain a metadata tuple $M = \langle M_b, M_{C'}, M_W \rangle$ as defined below:

- $M_b(p)$: The address of metadata attributes for the referent block $R_b(p)$, which include:
  - $M_b^B(p)$: The base address of the referent block $R_b(p)$, which always has the same value as $B(R_b(p))$.
  - $M_b^N(p)$: The allocation size in bytes of the referent block $R_b(p)$, which always has the same value as $N(R_b(p))$.
  - $M_b^C(p)$: The temporal allocation capability of the referent block $R_b(p)$, which always has the same value as $C(R_b(p))$. $C(R_b(p))$ is set upon the allocation of the referent block and invalidated upon the deallocation of the referent block.

- $M_{C'}(p)$: The cached temporal capability index value of the referent block $R_b(p)$. Initially it is a copy of the value $*C(R_b(p))$. But this value is only propagated along with pointer assignments, and not invalidated when the referent block is deallocated.

- $M_W(p)$,$M_W(p,d)$: $M_W(p)$ and $M_W(p,d)$ are the base address of metadata for embedded pointer or field pointer in $R(p)$.

Note that the metadata attributes $M_b^B(p)$, $M_b^N(p)$ and $M_b^C(p)$ are the properties of the referent block $R_b(p)$, that is, all the pointers that point to the same memory block share the same values of these metadata attributes at any time. $M_W(p,d)$ is the properties of the referent $R(p)$, that is, all the pointers that point to the same memory bytes share the same base address of embedded pointer metadata in $R(p)$. $M_{C'}(p)$ is the property of the pointer p itself, and each pointer is allowed to have a different value. We can refer to all these metadata attributes for p collectively as $M(p) = \langle\langle M_b^B, M_b^N, M_b^C\rangle, M_{C'}, M_W\rangle$.

A key requirement for maintaining the pointer metadata attributes is not to use fat pointers. These metadata attributes need to be separated from the pointers themselves for backwards-compatibility with the sheer size of legacy C programs.

3.3 Detecting Temporal and Spatial Memory Errors

Pointer dereferences need to be checked for memory errors. We use $M_b^B(p)$ and $M_b^N(p)$ to detect spatial memory errors, and $M_b^C(p)$ and $M_{C'}(p)$ to detect temporal memory errors. $M_W(p)$ and $M_W(p,d)$ are used to locate the metadata for an embedded pointer (e.g. $*p$) or a field pointer (e.g. $p{-}{>}d$).

A spatial memory error is reported when dereferencing a pointer p if any portion of the memory range to be dereferenced, $[p, p + L(*p))$, falls outside of the boundary $[M_b^B(p), M_b^B(p) + M_b^N(p))$.

To detect temporal memory errors, we first associate a unique capability with each memory block b. A capability is created from a capability store when b is allocated (on stack or heap), and stored as $C(b)$. When a pointer p to b is created, the allocation capability metadata attribute $M_b^C(p)$ is initialized to $C(b)$. When a memory block is deallocated, due to `free` or function returns, its allocation capability is revoked and marked as invalid. Thus, to detect temporal memory errors, we simply need to check whether $M_b^C(p)$ still points to a valid capability.

The above approach has the benefit that each capability requires just one bit of storage, but suffers from the drawback that this storage cannot be reused. As a result, a program performing a large number of memory allocations and deallocations will eventually run out of memory. To support reuse of capabilities, the above technique is modified as follows. Each capability is given a non-zero integer index value. (A value of zero denotes an invalid capability.) This value is also stored in a second metadata attribute $M_{C'}(p)$. The dereference of pointer p is temporally safe if and only if $M_b^C(p)$ points to a valid capability and $*M_b^C(p)$ has the same value as $M_{C'}(p)$. Now, a capability slot in a capability store can be reused as long as the capability value is changed before each reuse. With a 32-bit capability index, this approach allows capabilities to be reused $2^{32} - 1$ times before they become unusable. Alternatively, we may allow the index to wrap around (skipping over zero), treating the probability of temporal errors being missed due to such wrap-around as negligible.

We can detect the reallocation temporal memory errors in which the deallocated memory bytes are reused for a different object by the time a dangling pointer is dereferenced to attempt to access the previous object. This is because even though $M_b^C(p)$ may point to a valid capability due to the reallocation, it will have a different index value than $M_{C'}(p)$, and thereby fail the temporal validity check.

3.4 Temporal Capability Management

We rely on the uniqueness of allocation capabilities for temporal error detection. An allocation capability should never be assigned or reassigned to different memory blocks, except when the memory blocks have the exactly same lifetime scope. To avoid interpreting values from random memory locations as capabilities, the capabilities have to be allocated from a dedicated memory pool and separated from the memory block data themselves.

Because global variables are allocated when a program starts and have a lifetime throughout the entire program execution, they are always temporally valid. We can share a single always-valid allocation capability (denoted as $\top$) for all global variables.

The capabilities for heap and stack memory blocks need to be managed separately. Each heap memory block has a unique allocation capability, which is allocated from a heap capability pool when a memory block is newly allocated from the heap. When a heap memory block is freed, its allocation capability is marked as invalid and returned back to the heap capability pool.

All the stack variables of a function have the same lifetime scope. More specifically, they are temporally valid during the invocation of the function and become invalid once the function returns. Knowing about this fact, we can assign a single capability for all the stack variables in the same function. This capability, called *stack frame allocation capability*, is allocated from a stack capability pool on a function entry, and is marked as invalid and returned back to the stack capability pool upon the function returns.

3.5 Transformation of Pointer Statements

Figure 3.5 defines the metadata attributes values of pointer expressions and the associated guard code to detect memory errors in the transformation. Figure 3.6 shows the transformation of basic pointer statements in C. The goal of the transformation is two-fold: to maintain metadata for pointer operations as well as to check both temporal and spatial memory errors for pointer dereferences. It is worth noting that depending on the implementation of $M(p)$, some of metadata attribute updates shown in the transformation rules might become no-op and can then be optimized away during the transformation. For example, if we maintain the referent block metadata attributes separately, instead of keeping of a copy of these attributes in $M(p)$ of each pointer bound to the block, we can then

Expression e	Metadata $M(e) = \langle\langle M_b^B, M_b^N, M_b^C\rangle, M_{C'}, M_W\rangle$	Guard $G(e)$
p	$M(p)$	$\emptyset$
$s.d$	$*W(s,d)$	$\emptyset$
$\&x$	$\langle\langle B(x), L(x), C(x)\rangle, *C(x), W(x)\rangle$	$\emptyset$
$\&s.d$	$\langle\langle B(s), L(s), C(s)\rangle, *C(s), W(s,d)\rangle$	$\emptyset$
$p+n$	$M_W(p+n)$	$\emptyset$
$(\tau)p$	$M(p)$	$\emptyset$
$(\tau)n$	$\perp$	$\emptyset$
$*p$	$*M_W(p)$	$G_T(p, M_b^C(p), M_{C'}(p))$; $G_S(p, L(*p), M_b^B(p), M_b^N(p))$;
s->d	$*M_W(s,d)$	$G_T(\&s$->$d, M_b^C(s), M_{C'}(s))$; $G_S(\&s$->$d, L(s$->$d),$ $M_b^B(s), M_b^N(s))$;

Figure 3.5: Memory safety metadata and guard of pointer expressions.

Statement	Transformation
$p = \texttt{malloc}(n)$	$\langle r, M_b^C(r), M_W(r)\rangle = \texttt{malloc}'(n)$; $M_b^B(p) = r$; $M_b^N(p) = n$; $M_b^C(p) = M_b^C(r)$; $M_{C'}(p) = *M_b^C(r)$; $M_W(p) = M_W(r)$; $p = r$
$\texttt{free}(p)$	$G_H(p, M_b^C(p), M_{C'}(p))$; $*M_b^C(p) = \perp$; $\texttt{free}(p)$
$e_1 = e_2$	$G(e_2)$; $G(e_1)$; $M(e_1) = M(e_2)$ $e_1 = e_2$

Figure 3.6: Transformation of pointer statements.

only update the address to the referent block metadata (i.e., $M_b(p)$) in lieu of updating $\langle M_b^B(p), M_b^N(p), M_b^C(p)\rangle$ in the pointer assignment transformation rules.

Next we describe in the transformation of each kind of pointer operations in details. In the description of each pointer operation, we substitute $e_1 = e_2$ with the corresponding representative expression forms from Figure 3.5.

3.5.1 Pointer creation

A new pointer value can be created through the heap allocation statement `malloc` or the address-of (`&`) expression.

For a pointer p created through heap allocation, we first instrument `malloc` to create a new allocation capability for the allocated memory block. We also modify `malloc` to allocate additional storage to hold metadata pertaining to the embedded pointers stored within the allocated memory block. All the embedded pointer metadata attributes are initialized with values to indicate that none of the embedded pointers has a valid referent. The call to `malloc` is extended to return not only the base address r of the allocated memory block, but also the new allocation capability $M_b^C(r)$ and the base address of allocated embedded pointer metadata $M_W(r)$. The metadata of p, $M(p)$, is then initialized properly with the argument and return values of `malloc`.

Note that `malloc` may be used to allocate storage for an array rather than a single object. The size argument of `malloc` is used to determine whether the allocation is for a single object or array. In the latter case, $M(p)$ refers to an array rather than a single metadata structure. Code must be generated to initialize all of the elements of $M(p)$. The embedded pointer metadata within each of the allocated array elements needs to be initialized as well.

When a memory block is being deallocated via `free`(p), we first need to check whether it is a legitimate deallocation of a valid heap memory block. The checking is performed by the guard routine G_H. After that, we simply revoke its allocation capability $M_b^C(p)$ and mark it as invalid, denoted as $\perp$.

When a pointer p is created using the `&` operator on a variable x, i.e., $p = \&x$, we assign the metadata properties of x to $M(p)$. In particular, $M_b^B(p)$ and $M_b^N(p)$ are assigned with the base address of x, $B(x)$ and the allocation size of x decided by its compile-time type, $L(x)$, respectively. $B(x)$ and $L(x)$ can be derived from the compile-time symbol table information about x. $M_b^C(p)$ and $M_{C'}(p)$ are assigned with the allocation capability of x and its index value ($C(x)$ and $*C(x)$), respectively. If x is a global variable, $C(x)$ is the always-valid capability shared by all global variables. If x is a local variable, $C(x)$ is the stack frame capability for the current function. When x has embedded pointers, the base address of the embedded pointer metadata, $W(x)$, is assigned to $M_W(p)$. Note that $W(x)$ and $M_W(p)$ are properties of the referent x. This assignment has the desired effect of sharing the embedded pointer metadata located at $W(x)$, rather than copying them. This is not an optimization, but is a necessary step to ensure that the transformation handles aliasing effects correctly.

The transformation is similar when the pointer p is created by taking the address of a structure field $s.d$. We need to replace $W(x)$ with the version that supports fields: $W(s, d)$, and assign $B(s)$, $N(s)$ and $C(s)$ to the memory block attributes in $M(p)$.

3.5.2 Pointer assignment

Transformation for assignments of one pointer value to another is very simple: When p_1 is assigned to p through $p = p_1$, an assignment $M(p) = M(p_1)$ is introduced in the transformed program to propagate the metadata from p_1 to p. If a field pointer $s.d$ is assigned to p, we then assign the metadata values of the field pointer, $*W(s, d)$, to $M(p)$, the metadata of p.

The transformation for pointer arithmetic is slightly different. For $p = p_1 + n$, we first copy all metadata attribute values from $M(p_1)$ to $M(p)$. In addition, we also update $M_W(p)$ with the value of $M_W(p_1 + n)$. This is necessary because $M_W(p)$ is the base address of the metadata for the embedded pointer $*p$. When p gets a new value $p_1 + n$, $*p$ is also changed to $*(p_1 + n)$, which corresponds to a different pointer from the original $*p$ and thereby needs to associate with a new metadata address $M_W(p_1 + n)$. Just like invalid pointers are permitted in C programs as long as they are not dereferenced, $M_W(p)$ is permitted to contain invalid addresses as well, but is checked before accesses.

There are two cases of pointer assignments with type casts. If a pointer p_1 is cast to a pointer of different type and assigned to p, i.e. $p = (\tau)p_1$, the transformation is then the same as the normal pointer assignment statement, that is, an assignment $M(p) = M(p_1)$ is introduced in the transformed program. The transformation for $p = (\tau)n$, in which an integer variable n is cast to a pointer type and assigned to p, is quite different. In this case, n is not a pointer and has no associated pointer metadata. As a result, we mark $M(p)$, the metadata of p, as invalid (denoted as $\perp$). Any future dereferences of p will fail the guard routines G_T and G_S that detect memory errors and thus be caught.

3.5.3 Pointer dereference

When a pointer p is dereferenced, the transformation inserts runtime checks G_T and G_S before the pointer dereference to prevent temporal and spatial memory errors.

The guard routine G_T detects temporal memory errors. It takes the allocation capability $M_b^C(p)$ and the cached capability $M_{C'}(p)$ as the input arguments and checks whether the allocation capability still agrees with the cached capability. If the referent block $R_b(p)$ is not deallocated and still valid, the allocation capability and the cached capability should have the same value. If the memory of the referent block is deallocated or reallocated for a different memory block, $M_b^C(p)$ will have a different value than $M_{C'}(p)$. In this case, G_T reports a temporal memory error for the dereference of p.

The guard routine G_S detects spatial memory errors. It needs four arguments: the pointer value, the number of bytes to access, the base address and size of the referent block. G_S enforces that the memory to be accessed should completely fall inside of the lower and upper bounds of the referent block. For dereferences in the form of $*p$, we simply use p, $L(*p)$, $M_b^B(p)$ and $M_b^N(p)$ as the required arguments to G_S. For a structure pointer dereference that accesses a field in the form of $s\text{->}d$, we use the field expressions $\&s\text{->}d$ and $L(s\text{->}d)$ as the arguments of the pointer value and the number of bytes to access to G_S. This is important because it allows the program to access the beginning bytes of a structure referent even if the remaining of the structure is beyond the referent block boundary. Such cases could occur as a result of type casts between different types of structure pointers.

3.6 Transformation of Functions

For stack variables in a function, our transformation introduces the management of the stack frame allocation capability for them as described in Section 3.4.

Functions can take pointers as input arguments. Their return values may also contain pointers. Our transformation transforms such functions to pass the metadata of pointer arguments into the function and return the metadata of pointers in the return value back to the caller.

There are two possible ways to pass the additional metadata into and out of a function: modifying the function interface, or using a global argument buffer.

We can modify the interface of a function to introduce additional arguments to pass the metadata pertaining to the original function arguments. If a function returns a pointer or a structure with field pointers, we need to modify the return type to be a structure that contains both the original return value and its associated metadata.

Note that the introduction of additional arguments affects the compatibility with external functions. Unlike fat pointers, however, this incompatibility can be easily fixed in most cases. In particular, if an external function takes objects containing pointer values but does not modify them (or return pointers), then calls to this function are left untransformed. If the function modifies pointer values or returns them, then wrapper functions will need to be created. Or alternatively we can just assume these pointers are always valid. The only drawback is that memory errors caused by these pointers will not be detected.

When using a global argument buffer to pass around the metadata of the original function arguments and return value, we may get a better compatibility with external libraries. But wrapper functions may still be needed for external, untransformed functions in order to achieve more accurate memory error detection. We have studied both argument passing methods in our research.

Chapter 4

Memory Safety with Shadow Metadata Structures

In this chapter, we describe our transformation for memory error detection that maintains all the pointer metadata in shadow structures. We have also presented a version of this transformation in [92].

4.1 Metadata Representation and Allocation

In this transformation, we maintain all the pointer and referent block metadata attributes for a pointer in a metadata structure. More precisely, for each pointer variable `p` in the original program, we introduce a metadata structure variable `p_info`. For array variables, a shadow array is introduced to hold metadata pertaining to the element pointers of the array. Each pointer metadata variable contains the following metadata fields:

- `base`: The base address of the referent block of `p`.
- `size`: The allocation size in bytes of the referent block.
- `blk_cap`: The allocation capability address of the referent block.

```
Original                  Transformed
---------------------------------------------------------------------
                          struct ptrinfo_t {
                            void              *base;
                            u32_t             size;
struct Stu {                capability        *blk_cap;
  int  id;                  capability        saved_cap;
  char *name;               struct ptrinfo_t  *link;
  int  age;               };
} Stu;                    struct Stu_info_t {
                            struct ptrinfo_t  name;
struct Stu s;             };
                          struct Stu        s;
                          struct Stu_info_t s_info;
---------------------------------------------------------------------
                          struct Stu_ptrinfo_t {
                            void              *base;
                            u32_t             size;
                            capability        *blk_cap;
struct Stu *p;              capability        saved_cap;
                            struct Stu_info_t *link;
                          };
                          struct Stu           *p;
                          struct Stu_ptrinfo_t p_info;
---------------------------------------------------------------------
```

Figure 4.1: Defining basic shadow metadata structures for variables.

- `saved_cap`: The cached capability index value of the referent block.
- `link`: The metadata address of the embedded pointers stored within the referent `*p`.

A distinctive feature of this transformation is that the type of `link` is defined to match the metadata type for the pointer referent type. Note that the field `link` is implemented to support $M_W(p)$, or in other words, it is used to retrieve the metadata address of embedded pointers such as `*p`. For a simple scalar pointer, the type of `link` is defined as a pointer to the generic pointer metadata type `ptrinfo_t`. For a pointer that points to a structure type `s`, we first define a shadow structure `s_info_t` for the program structure `s`. For each field pointer `d` within `s`, there is a corresponding field `d` defined in `s_info_t` to store the metadata for the pointer `s.d`. A variable of type `s` will also be accompanied with a metadata variable of type `s_info_t`. The `link` field of the structure pointer metadata type is then defined as a pointer to `s_info_t`. Figure 4.1 shows the transformation examples of introducing metadata variables for pointer and structure variables. Such definitions of the `link` field provide us an easy way to access metadata of embedded pointers and field pointers in the pointer referent. For instance, to get the access to the metadata of `p->name`, we can simply use `p_info.link->name`. However, this does complicate the support for arbitrary type casts in C programs. We discuss these issues in Section 4.4.

To protect pointers dynamically allocated on the heap, we instrument `malloc` to allocate additional storage to hold metadata pertaining to pointers that are to be stored within the allocated block. The type of the recipient pointer `p` for the return value of `malloc` indicates which fields within the allocated block will contain pointers.

Note that `malloc` may be used to allocate storage for an array rather than a single object. The size argument of `malloc` is used to determine whether the allocation is for a single object or array. In the latter case, `p_info` refers to an array rather than a single struct. Code must be generated to initialize all of the elements of `p_info`. In addition, the metadata pertaining to embedded pointers within each of the array elements needs to be initialized as well.

We would like to point out that unlike the previous fat pointer approaches, our transformation allocates metadata separately from their corresponding memory blocks and pointers and does not modify the internal pointer representation. This allows the transformed programs to achieve a much better backwards-compatibility with existing C libraries.

All the pointer metadata, no matter whether it is allocated globally, on stack, or on heap, are initialized in a way to ensure that any dereference or deallocation of the corresponding pointer before its first assignment will raise spatial and temporal memory errors that are caught by our error checking routines G_H, G_T and G_S.

4.2 Capability Store Management

As described in Section 3.3, we use capabilities for temporal memory error detection.

In this transformation, we use a *global capability store* (GCS) to manage capabilities. A capability identifies the lifetime scope of a memory block. It is created when a block is allocated (on stack or heap), and stored in GCS.

When a pointer to a memory block b is created and assigned to a variable `p`, a pointer to b's capability is stored in `p_info.blk_cap`, and its capability index value is copied to `p_info.saved_cap`. When the memory block is freed, the associated capability (in the GCS) is marked invalid. A capability is valid if and only if it points to a valid capability slot in the GCS and its index value is valid. It is necessary to have dedicated memory pools for capabilities and separate them from the program data objects. This can ensure that program data objects will never be incorrectly interpreted as capabilities to compromise the temporal memory error detection.

Similar to the Safe-C approach [13], we define a global capability `FOREVER` that is stored in GCS for all global variables because they are always temporally valid. We also define a special capability `NEVER` that is used for uninitialized memory blocks. For heap and stack allocated objects, their capabilities are held in the heap and stack capability stores (HCS and SCS) respectively. The GCS consists of all the above components.

HCS is an expandable array. Each slot in the array may store a valid capability, or it may be unused. Unused slots in the HCS are organized into a linked list called the *free list*. This means that a used HCS slot contains a capability index, whereas an unused slot contains a pointer to next available slot in the HCS free list. To distinguish between the two types of information, we ensure that the least significant bit (LSB) of valid capability indices is always one, whereas the LSB of pointers in the HCS free list is always zero. (This is because the pointers are aligned on a 32-bit boundary.)

When a memory block is allocated on the heap, a capability for this block is allocated from the head of the HCS free list. When a memory block is freed, its capability is inserted at the beginning of the HCS free list. Since valid capability index values are different from pointers in the free list, the insertion of a capability into the free list immediately causes it to become invalid.

The SCS is allocated separately from HCS since capabilities on the SCS are allocated in a LIFO manner. Note that all of the local variables for a function are allocated and freed together. Hence, we can use a single capability for all of the local variables (as opposed to one per variable). This stack frame allocation capability, `STACK_FRAME_CAP`, is pushed on the SCS on a function entry, and popped off on exit.

We use the second least significant bit of a capability index value to distinguish heap and stack capabilities. This means that there are at most 2^{30} possible capability index values given a 32-bit long capability index.

4.3 Applying Transformation Rules

The metadata attributes and helper functions in Chapter 3 can be defined with the shadow metadata structures as shown in Figure 4.2. We assume that no pointer type casts are involved in these transformations. We will handle pointer type casts in Section 4.4. The definitions of the memory error checking functions, `CHECK_FREE`, `CHECK_TEMPORAL` and `CHECK_SPATIAL`, are given in Figure 4.3. Most of these definitions are straight-forward and self-explanatory. We would like to make a note about `INVALID_PTRINFO_ADDR`. It is the address of a special pointer metadata structure which corresponds to any non-pointer referent. This metadata structure is read-only and initialized with metadata values that indicate the referent is both temporally and spatially invalid.

With all the definitions of metadata attributes and helper functions, we can then apply the transformation rules in Figure 3.6 to transform C programs with the shadow metadata structures and generate an instrumented version of the original program that is capable of

Metadata	Condition	Transformation
$B(x)$		`&x`
$B(*p)$		`p_info.base`
$N(x)$		`sizeof(x)`
$N(*p)$		`p_info.size`
$L(x)$		`sizeof(x)`
$L(*p)$		`sizeof(*p)`
$L(s, d)$		`sizeof(s.d)`
$C(x)$	x is a global variable	`FOREVER`
$C(x)$	x is a local variable of the current function	`STACK_FRAME_CAP`
$C(*p)$		`p_info.blk_cap`
$W(x)$	x is an integer	`INVALID_PTRINFO_ADDR`
$W(x)$	x is a pointer	`&x_info`
$W(s, d)$	d is an integer	`INVALID_PTRINFO_ADDR`
$W(s, d)$	d is a pointer	`&s_info.d`
$W(*p)$		`p_info.link`
$W(p, d)$		`p_info.link->d`
$M(p)$		`p_info`
$M_b^B(p)$		`p_info.base`
$M_b^N(p)$		`p_info.size`
$M_b^C(p)$		`p_info.blk_cap`
$M_{C'}(p)$		`p_info.saved_cap`
$M_W(p)$		`p_info.link`
$M_W(p, d)$		`&p_info.link->d`
$M_W(p + n)$	*p has the same static and runtime type	`p_info.link + n`
G_H		`CHECK_FREE`
G_T		`CHECK_TEMPORAL`
G_S		`CHECK_SPATIAL`

Figure 4.2: Representing memory safety metadata with basic shadow metadata structures.

detect all temporal and spatial memory errors at the memory block level. Figure 4.4 gives the transformation examples of pointer creation, dereference and assignment statements using the basic shadow metadata structures.

To support pointer arguments in functions, we introduce an additional argument for each pointer argument of a function to pass around the metadata information.. The definitions of the metadata arguments follow the same transformation rules for regular stack

```
void CHECK_FREE(const void *p,
                const capability *allocatedCap,
                capability savedCap)
{
   CHECK_TEMPORAL(p, allocatedCap, savedCap);
   if (!IS_HEAP_CAPABILITY(allocatedCap)) {
      Abort("invalid free at %p\n", p);
   }
}

void CHECK_TEMPORAL(const void *p,
                    const capability *allocatedCap,
                    capablity savedCap)
{
   if (!IS_VALID_CAPABILITY(allocatedCap) ||
       *allocatedCap != savedCap) {
      Abort("temporal memory error at %p\n", p);
   }
}

void CHECK_SPATIAL(const void *p, u32_t accessSize,
                   const void *blkBase, u32_t blkSize)
{
   if (p < blkBase ||
       p + accessSize > blkBase + blkSize) {
      Abort("spatial memory error at %p\n", p);
   }
}
```

Figure 4.3: Definitions of memory error checking routines.

variables. In other words, we also create a shadow memory structure for each pointer or structure type in the arguments.

4.4 Handling Type Casts and Unions

The C language allows programs to perform arbitrary casts. However, most casts in typical C programs involve types that have some sort of subtype–supertype relationship [79]. Such casts can be classified into upcasts (cast from subtype to supertype) and downcasts (casting from a supertype into a subtype). As in object-oriented languages, upcasts are always safe. But downcasts are not always safe. and need to be checked at runtime by

```
Original           Transformed
------------------------------------------------------------------------
                   q_info.base      = &p;
                   q_info.size      = sizeof(p);
                   q_info.blk_cap   = STACK_FRAME_CAP;
q = &p;            q_info.saved_cap = *STACK_FRAME_CAP;
                   q_info.link      = &p_info;
                   q = &p;
------------------------------------------------------------------------
                   q_info      = p_info;
q = p + 4;         q_info.link = p_info.link + 4;
                   q = p + 4;
------------------------------------------------------------------------
                   CHECK_TEMPORAL(p, p_info.blk_cap,
                                     p_info.saved_cap);
                   CHECK_SPATIAL(p, sizeof(*p),
q = *p;                            p_info.base, p_info.size);
                   q_info = *p_info.link;
                   q = *p;
------------------------------------------------------------------------
                   CHECK_TEMPORAL(&p->d, p_info.blk_cap,
                                     p_info.saved_cap);
                   CHECK_SPATIAL(&p->d, sizeof(p->d),
q = p->d;                          p_info.base, p_info.size);
                   q_info = p_info.link->d;
                   q = p->d;
------------------------------------------------------------------------
```

Figure 4.4: Transforming pointer statements with basic shadow metadata structures

maintaining *runtime type information (RTTI)* with each object. A downcast is safe if the runtime type of the pointer referent is a subtype of its static type.

4.4.1 Notion of Subtyping

When a type τ_1 is laid out in memory exactly as a prefix of another type τ_2, type τ_2 is called a *physical subtype* of type τ_1. Physical subtyping has been studied for understanding type casts in C programs [79], but it is unnecessarily restrictive for our purposes. To maximize the set of programs that can be handled by our approach, we define a weaker notion of subtyping in this section that is sufficient to ensure the soundness of our memory safety enforcement approach.

Consider a program fragment as given below:

```
typedef struct T {
  char  *f;
} T;
typedef struct T0 {
  u32_t a;
  T     *b, *c;
} T0;
typedef struct T1 {
  u16_t a0, a1;
  u32_t a2;
  T     *b, *c;
} T1;
typedef struct T2 {
  u32_t w0, w1;
  T     *x, *y, *z;
} T2;
```

```
typedef struct T_info_t {
  struct ptrinfo_t f;
} T_info_t;
typedef struct T0_info_t {
  T_ptrinfo_t  b, c;
} T0_info_t;
typedef struct T1_info_t {
  T_ptrinfo_t  b, c;
} T1_info_t;
typedef struct T2_info_t {
  T_ptrinfo_t  x, y, z;
} T2_info_t;
```

```
 1: T0 *p0;
 2: T1 *p1;
 3: T2 *p2 = malloc(...);
 4: ... // code that initializes p2
 5: p = (void *)p2;  // upcast
 6: ...
 7: p0 = (T0 *)p;    // unsafe downcast
 8: ... // code that dereferences p0
 9: q = p0->b;
10:
10: p1 = (T1 *)p;    // safe downcast
11: ... // code that dereferences p1
12: q = p1->b;
```

Let's consider the downcast `p0 = (T0 *)p` at line 7. Note that if `T0` contains no pointers, then this use is always acceptable. Any spatial or temporal error in accessing a non-pointer field of the form `p0->a` will be detected using the metadata in `p0_info`, which would have been copied from `p2_info` via `p_info`. On the other hand, if `T0` contains a pointer, say, in a field named `b` as shown in the code fragment, then we need to make sure that correct metadata information is available for checking the dereference of `q`, which is assigned from `p0->b`. Suppose that we blindly access `p0_info.link->b`. Since `p0_info.link` was copied from `p2_info.link`, this access will be equivalent to `p2_info.link->x`, where `x` is at the same offset in `T2_info` as is `b` in `T0_info`. Similarly, the `p0->b` accesses the same location as `p2->w1` where `b` and `w1` are at the same offset with respect to the base of `T0` and `T2` respectively. Thus, the access `p0_info.link->b` will yield correct results if `p2->w1` is a pointer and `p2_info.link->x` corresponds to the metadata associated with this pointer. Unfortunately, in the case of `T0` and `T2`, this is not true and the downcast at line 7 is unsafe. It is true for `T1` and `T2`, however, because for `T1`, `b` corresponds to `x` in both `T2` and `T2_info`. Consequently, `p1_info.link->b` can yield correct results from `p2_info.link->x` and the downcast at line 10 is also safe.

Note that the types of pointers `p1->b` and `p2->x` need not be the same: if a dereference of `q->f` (which is equivalent to `*(p1->b->f)`) is incorrect, it will be detected when the runtime type of `q_info.link->f` (which is equivalent to `p1_info.link->b.link->f`) is examined.

Based on the above discussion, we define the following notion of subtyping. Suppose that τ_1 contains pointer fields at offsets $p_1, ..., p_n$, where p_i's are in sorted order. Similarly, let $q_1, ..., q_m$ denote all of the offsets of pointers fields within τ_2. Now, τ_2 is a subtype of τ_1 iff $n \leq m$ and $q_i = p_i$ for $1 \leq i \leq n$. In the previous example, `T_2` is a subtype of `T_1`, but not a subtype of `T_0`.

4.4.2 Extending Transformation with RTTI

The following changes are made to the basic transformation to support casting between subtypes and supertypes:

- *Declarations:* The type and pointer metadata variable declarations are modified to incorporate an additional metadata field called `tid` that identifies the actual type of data stored in the referent.

- *Casts:* Note that the metadata variables for all pointers have the same structure: they have the same layout and the same interpretation of the fields of this structure. Thus casts between any two pointer types are allowable, as long as the runtime type associated with the pointer is checked prior to the use of the `link` field of its metadata variable.

- *Dereferencing:* For `p->d`, the transformation is the same as before if `p->d` is a non-pointer. But if it is a pointer, then the metadata structure associated with `p->d` will be invalid if the runtime type check fails, i.e., `p_info.tid` is not a subtype of the

```
Bool CHECK_RTTI(int runtimeTID, int staticTID)
{
   return IS_SUBTYPE(runtimeTID, staticTID);
}

void *LINK_PTR_ARITH(void *p, u32_t accessSize,
                     void *link, int n,
                     int runtimeTID, int staticTID)
{
   int n2, elemSize, infoSize;
   if (link == INVALID_PTRINFO_ADDR) {
      return INVALID_PTRINFO_ADDR;
   }
   elemSize = TYPEINFO[runtimeTID].elemSize;
   infoSize = TYPEINFO[runtimeTID].infoSize;
   if (staticTID == runtimeTID ||
       accessSize == elemSize) {
      return link + n * infoSize;
   }
   else if ((n * accessSize) % elemSize == 0) {
      n2 = (n * accessSize / elemSize) * infoSize;
      return link + n2;
   }
   return INVALID_PTRINFO_ADDR;
}
```

Figure 4.5: Definitions of RTTI helper routines `CHECK_RTTI` and `LINK_PTR_ARITH`.

compile-time type of `*p`. We define the runtime type check routine `CHECK_RTTI` in Figure 4.5.

- *Assignments:* For assignments between pointers, the `tid` field must also be copied on pointer assignments. When pointers are created using `malloc` or `&` operator, the `tid` field needs to be populated. For `malloc`, the type of the pointer is used for this purpose. For the `&` operator, the statically declared type of the object (whose address is being taken) is used.

Pointer arithmetic operations need some care. Consider a pointer arithmetic statement such as `q = p + n`. Let `elemSize` and `infoSize` be the element sizes of runtime type and

```
Original                    Transformed
------------------------------------------------------------------------
                            struct ptrinfo_t {
                              int                tid;
                              void               *base;
struct Stu {                  u32_t              size;
  int  id;                    capability         *blk_cap;
  char *name;                 capability         saved_cap;
  int  age;                   struct ptrinfo_t   *link;
} Stu;                      };
                            struct Stu_info_t {
struct Stu s;                 struct ptrinfo_t   name;
                            };
                            struct Stu        s;
                            struct Stu_info_t s_info;
------------------------------------------------------------------------
                            struct Stu_ptrinfo_t {
                              int                tid;
                              void               *base;
                              u32_t              size;
struct Stu *p;                capability         *blk_cap;
                              capability         saved_cap;
                              struct Stu_info_t  *link;
                            };
                            struct Stu            *p;
                            struct Stu_ptrinfo_t p_info;
------------------------------------------------------------------------
```

Figure 4.6: Defining RTTI-enhanced shadow metadata structures for variables.

metadata type of the object pointed by `p` respectively. (We maintain these sizes information in transformed programs.) Then the link field of `p_info` is also incremented, using

```
q_info.link = (void*)p_info.link + n',
```

where

```
n' = (n * sizeof(*p) / elemSize) * infoSize.
```

This statement increments `link` to point to the pointer metadata variable for the new value of `*q`. Note that this transformation supports programs that advance structure pointers by first casting them into `char*` or `void*`, then adding carefully calculated offsets to them and casting them back to pointers of desired types. To ensure that this will result in a correct link field value, we must ensure that the product of `n * sizeof(*p)` be a multiple of

Original	Transformed

char *p;
q = &p;
	q_info.tid = TID_char_ptr;
	q_info.base = &p;
	q_info.size = sizeof(p);
	q_info.blk_cap = STACK_FRAME_CAP;
	q_info.saved_cap = *STACK_FRAME_CAP;
	q_info.link = &p_info;
	q = &p;

T **p;
q = p + 4;
	q_info = p_info;
	q_info.link = LINK_PTR_ARITH(p, sizeof(*p),
	p_info.link, 4,
	p_info.tid,
	TID_T_ptr);
	q = p + 4;

T **p;
q = *p;
	CHECK_TEMPORAL(p, p_info.blk_cap,
	p_info.saved_cap);
	CHECK_SPATIAL(p, sizeof(*p),
	p_info.base, p_info.size);
	q_info = CHECK_RTTI(p_info.tid, TID_T_ptr) ?
	***p_info.link : *INVALID_PTRINFO_ADDR;**
	q = *p;

T **p;
q = p->d;
	CHECK_TEMPORAL(&p->d, p_info.blk_cap,
	p_info.saved_cap);
	CHECK_SPATIAL(&p->d, sizeof(p->d),
	p_info.base, p_info.size);
	q_info = CHECK_RTTI(p_info.tid, TID_T_ptr) ?
	p_info.link->d : *INVALID_PTRINFO_ADDR;
	q = p->d;

Figure 4.7: Transforming pointer statements with RTTI-enhanced shadow metadata structures.

`elemSize`. If this condition is not satisfied, the link field is marked invalid and hence cannot be used. The above link field pointer arithmetic algorithm is implemented in the helper function `LINK_PTR_ARITH` shown in Figure 4.5. It is worth mentioning that our transformation still allows dereferencing of `q`, but causes problems if pointers contained within `*q` are accessed. Thus, it will be possible to write a program that takes a pointer to an array of structures, casts it into a `char*`, and sequentially accesses the entire array as a sequence of characters. However, problems do arise if pointer arithmetic is used to advance through an array that is embedded within a structure when the array elements contain pointers that are dereferenced.

Figure 4.6 shows the examples of defining RTTI-enhanced metadata variables, and Figure 4.7 shows the pointer statement transformation examples with RTTI-enhanced metadata. All the RTTI-related changes are highlighted in bold.

Our approach performs subtyping checks at the time of pointer dereferencing, rather than at the time of casting. This is important to support programs which perform a cast, but do not use the resulting pointer in incorrect ways.

Even though we can perform the subtyping check as defined in Section 4.4.1 by comparing offsets of pointer fields in the two source types at runtime, this would be inefficient. As an optimization, we precompute the subtype relationship at compile-time and use a simple array lookup at runtime to perform the check. In particular, an integer identifier i is associated with each type τ_i. Then a two-dimensional bit-array `isSubType` is defined, where `isSubType[i][j]` is `1` iff τ_i is a subtype of τ_j. To minimize the size of this array, we consider only those types that are involved in pointer casts. Specifically, a simple program analysis is performed to obtain the list of all types that were involved in a cast. A type id is allocated only for such types. In our benchmark programs, the size of this array was small — always less than 150 in each dimension.

4.4.3 Supporting Unions

Unions in C represent implicit type casts when a union member is stored as one type and accessed as another. Thus, the basic machinery we have developed above will work for dealing with unions. Figure 4.8 shows an example of how a union definition is transformed.

Note that the metadata type for the union does not have tags. This is not a problem, since the `tid` is always the first field of all the structs within `u_info`, and can hence be

Original

```
union U {
  char       *p1;
  struct stu *p2;
} u;
```

Transformed

```
struct ptrinfo_t {
    int                tid;
    void               *base;
    u32_t              size;
    capability         *blk_cap;
    capability         saved_cap;
    struct ptrinfo_t *link;
};
struct Stu_ptrinfo_t {
    int                 tid;
    void                *base;
    u32_t               size;
    capability          *blk_cap;
    capability          saved_cap;
    struct Stu_info_t *link;
};
union U_info_t {
  struct ptrinfo_t     p1;
  struct Stu_ptrinfo_t p2;
};
union U u;
union U_info_t u_info;
```

Figure 4.8: Defining RTTI-enhanced shadow metadata structures for unions.

accessed to determine the type of the pointer currently stored in the union `u`. To ensure that this approach will work correctly, it is necessary to create and maintain the `tid` field even when the union holds a non-pointer. For instance, if an integer field `p3` is added to the above union, then a field `p3_info` will be added to `u_info`. When an integer is stored into the union using the field `p3`, then `p3_info.tid` will be set to indicate that the current content of the union is an integer.

It is important to note that since we do not convert unions into "tagged unions," as is the case in [68] or Cyclone [47], programs that expect to store into `p1` and access this later using the field name `p2` will work as expected. However, any attempt to dereference this pointer will be checked using the runtime type information, so invalid memory accesses will be caught.

4.5 Optimizations

4.5.1 Local Optimizations

The most significant optimization we have performed is to separate metadata into two data structures: a `blkheader_t` that holds the metadata information relating to a memory block, and a `ptrinfo_t` that holds metadata information relating to a specific pointer. A `blkheader_t` variable is introduced for each memory block (variables and heap allocated blocks). We also introduce a metadata attribute `hdr` to `ptrinfo_t` and move the three block metadata attributes into the `blkheader_t` variable. We then define $M_b(p)$ as `p_info.hdr` and apply the transformation rules in Chapter 3 accordingly. Figure 4.9 illustrates this separation, with new changes highlighted in bold.

This separation yields significant savings when there are multiple pointers to the same block, since the `blkheader_t` fields of the block can be shared by all pointers to that block. Moreover, the split reduces the number of fields involved in metadata maintenance operations, and hence improves performance.

Pointer dereferences tend to repeat frequently, and for each such dereference, our transformation generates pointer validity checks. However, `gcc 3`, which was used in our experiments, did not recognize and eliminate such checks when they involve fields in a structure. For the same reason, `gcc` optimizations could not recognize and eliminate metadata updates even when the results of these updates are not used again. In order to exploit the common subexpression and dead variable elimination optimizations of `gcc`, we therefore convert metadata structures into individual variables whenever it is safe to do so. (In particular, we break the metadata structure associated with a local pointer variable when the address of the variable is not taken.) This optimization eliminates several redundant checks and updates at an intra-procedural level. It is quite possible that with the recent significant advances in `gcc 4` development, such optimizations could be taken care of by the compiler itself and needs not to be performed as part of the transformation.

```
Original                     Transformed
-------------------------------------------------------------------------
                             struct blkheader_t {
                               void       *base;
                               u32_t       size;
                               capability *blk_cap;
                             };
struct Stu {                 struct ptrinfo_t {
  int  id;                     int                 tid;
  char *name;                  struct blkheader_t *hdr;
  int  age;                    capability          saved_cap;
};                             struct ptrinfo_t   *link;
                             };
struct Stu s;                struct Stu_info_t {
                               struct ptrinfo_t name;
                             };
                             struct Stu s;
                             struct Stu_info_t  s_info;
                             struct blkheader_t s_hdr;
-------------------------------------------------------------------------
                             struct Stu_ptrinfo_t {
                               int                 tid;
                               struct blkheader_t *hdr;
struct Stu *p;                 capability          saved_cap;
                               struct stu_info_t  *link;
                             };
                             struct Stu *p;
                             struct Stu_ptrinfo_t p_info;
-------------------------------------------------------------------------
```

Figure 4.9: Defining split shadow metadata structures for variables.

Another optimization we have implemented is to eliminate unnecessary operations on the stack capability store (SCS). This elimination is performed in the case of functions which never compute the address of a local variable, and hence have no risk of a local pointer escaping. This optimization primarily benefits function call-intensive programs where the frequently called functions do not contain reference operations that generate a pointer to a local variable.

The effect of these optimizations are discussed in Section 10.2.2. On the average, these optimizations improve performance by about a factor of two.

Types:	τ	::=	$\texttt{int} \mid \tau\ \texttt{ref STATIC} \mid \texttt{RTTI}$
Expressions:	e	::=	$x \mid n \mid e_1\ \texttt{op}\ e_2 \mid (\tau)e \mid e_1 \oplus e_2 \mid *e$
Statements:	c	::=	$\texttt{skip} \mid c_1; c_2 \mid e_1 := e_2$

Figure 4.10: Syntax of a simplified C language with STATIC and RTTI pointers.

Expressions:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1\ \texttt{op}\ e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash e : \tau' \quad \tau' \leq \tau}{\Gamma \vdash (\tau)e : \tau} \qquad \frac{}{\Gamma \vdash (\tau\ \texttt{ref STATIC})0 : \tau\ \texttt{ref STATIC}}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{ref STATIC} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \oplus e_2 : \tau\ \texttt{ref STATIC}} \qquad \frac{\Gamma \vdash e_1 : \texttt{RTTI} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{RTTI}}$$

$$\frac{\Gamma \vdash e : \tau\ \texttt{ref STATIC}}{\Gamma \vdash *e : \tau} \qquad \frac{\Gamma \vdash e : \texttt{RTTI}}{\Gamma \vdash *e : \texttt{RTTI}}$$

Statements:

$$\frac{}{\Gamma \vdash \texttt{skip}} \qquad \frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1; c_2}$$

$$\frac{\Gamma \vdash e : \tau\ \texttt{ref STATIC} \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e := e'} \qquad \frac{\Gamma \vdash e : \texttt{RTTI} \quad \Gamma \vdash e' : \texttt{RTTI}}{\Gamma \vdash e := e'}$$

Convertibility:

$$\frac{}{\tau \leq \tau} \qquad \frac{}{\tau \leq \texttt{int}} \qquad \frac{}{\texttt{int} \leq \texttt{RTTI}}$$

Figure 4.11: Type inference rules for identifying STATIC and RTTI pointers.

4.5.2 Possible Global Optimizations

There are a number of global optimizations which could be done to eliminate checks and metadata, reducing the overhead of our approach from its current level. Even though these optimizations might be effective, we have not implemented them in our research, partly because these global optimizations usually require global program analysis and non-trivial code changes for program annotations, which could sometimes break existing C programs.

The first global optimization is to identify pointers whose type can be statically determined, and replace RTTI with constant type information for these pointers. This can be done using a fairly simple flow-insensitive type inference approach similar to the method used to determine so-called *safe pointers* in CCured [62]. Figure 4.10 shows the syntax of a simplified C language similar to what is defined in [62], but with STATIC and RTTI pointer types. Figure 4.11 shows the type inference rules adapted from those of CCured to identify STATIC and RTTI pointers in C programs. In these rules, RTTI behaves the same as DYNAMIC in CCured. However, the STATIC pointers is the union of both SAFE and SEQ pointers in CCured. This is because safe and sequential pointers are never involved in (non-trivial) casts, and hence their static type will be the same as runtime type, thereby making RTTI information redundant. The results presented in [62] showed that for typical C programs, roughly 90% of pointers could be statically shown to be safe pointers, and similar results are likely for our approach.

A second optimization is to identify pointer operations which can be statically determined to be temporally safe, and remove the temporal checks for these operations. Criteria for statically determining the temporal safety of a pointer access depend on the storage region(s) the pointer refers to. For heap pointers, there must be no path along which the referent could be freed between the previous check and the current dereference location. For stack pointers, there must be no chance that the pointer could have escaped from a previous function call. Static data pointers are inherently temporally safe.

A third optimization is to identify pointer operations which are statically spatially safe, and eliminate bounds checks for these pointers. The spatial safety of a pointer access is determined by examining all paths between the access and all recent checks/allocations. If the value of the current pointer expression is the same as the previous ones, then the access is spatially safe. If the value is increasing (e.g., the result of an increment operation such as `p++`), then the access is *lower-bound safe* but not *upper-bound safe*. Similarly, some accesses might be determined to be *upper-bound safe* but not *lower-bound safe*. This information can be used to eliminate or simplify checks inserted by the transformation.

Once checks have been eliminated, unnecessary maintenance code can be eliminated. Updates to metadata which can never reach a check can be detected using standard data flow techniques and eliminated. Metadata variables/fields which are never used can then be detected and eliminated. For pointers which exhibit partial safety (e.g., lower bound safety), the metadata can be simplified.

Chapter 5

Memory Safety with Shadow Metadata Memory

The shadow metadata structure based transformation as described in Chapter 4 fully satisfies properties (1) and (4) listed in Section 2.2, which detects all block-level temporal and spaital memory errors while preserving the explicit memory management model in C. It also requires almost no source modifications before applying transformation and has achieved a significantly improved backward-compatibility compared to previous approaches that are aimed at detecting both temporal and spatial memory errors. However, the shadow metadata structure approach still has some shortcoming with regard to full backward-compatibility with existing C programs. In particular, the shadow metadata structure based transformation has the following limitations:

- It does not support arbitrary type casts, such as casts of a pointer to a structure of one type into a pointer to structure of an unrelated type (i.e. these two structures do not form a subtype–supertype relationship).
- It does not support arbitrary pointer arithmetic.
- It does not support customized memory allocation functions that are not similar to

`malloc` because it needs to know the type of allocated memory at each allocation site, in order to allocate appropriate memory space for metadata of embedded pointers accordingly.

In this chapter, we present a shadow metadata memory based approach that can overcome the above limitations.

The key difference between the shadow metadata structure approach and the shadow metadata memory approach lies in the way in which the metadata for embedded pointers in a memory block are allocated and accessed.

The shadow structure approach creates a shadow metadata structure for each memory block data type that can host pointers. For each field pointer in the memory block data type, a `ptrinfo_t`-like metadata field is created within the shadow structure. The accesses to the embedded pointer metadata are done via a syntactical field access path starting from the `link` field that points to the shadow structure, for example, `p_info.link->d`. As a result, although all the pointer metadata variables share the same memory layout as `ptrinfo_t`, the actual metadata type is different for pointers of different types due to the different typing of the `link` field.

In contrast, the shadow memory approach uses generic metadata structure types for all metadata. For any memory block that can contain pointers, it simply allocates an array of such generic metadata structures, one for each word in the memory block that might host a pointer. The access to the metadata of an embedded pointer `p` is achieved by computing the address of the embedded pointer metadata from the pointer value and the memory block metadata attributes.

5.1 Metadata Representation and Management

In this approach, we maintain the metadata in two generic data types: `blkheader_t` for metadata attributes of a memory block, and `ptrinfo_t` for metadata attributes of a

```
typedef struct ptrinfo_t ptrinfo_t;

typedef struct blkheader_t {
    void        *base;
    u32_t       size;
    capability  *blk_cap;
    ptrinfo_t   *linkbase;
} blkheader_t;

struct ptrinfo_t {
   blkheader_t  *hdr;
   capability   saved_cap;
};
```

Figure 5.1: Generic metadata structure types for shadow metadata memory

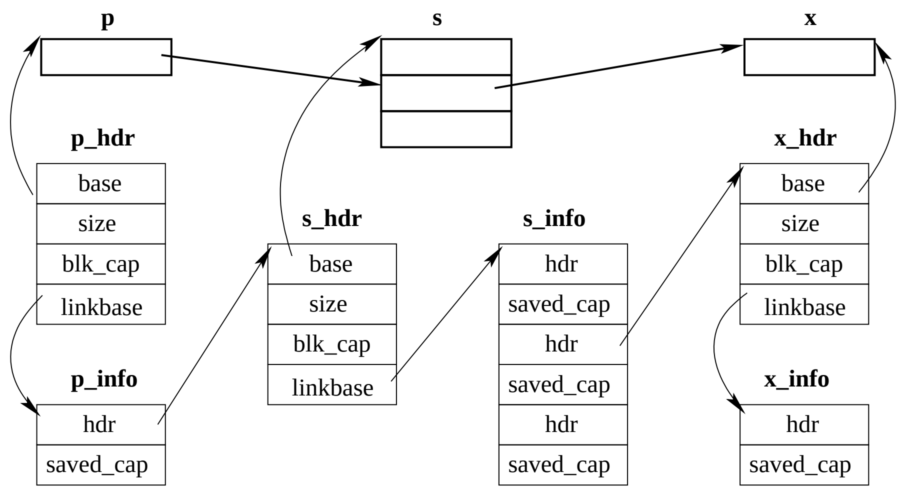

Figure 5.2: Shadow metadata memory layout.

particular pointer. The separation of block metadata fields and pointer metadata fields is similar to what we have presented in the optimization section for the shadow metadata structure approach. However, we use one generic `ptrinfo_t` metadata type for pointers, instead of creating a myriad of such similar types.

A block metadata object is allocated and associated with each memory block allocated

```
Original                  Transformed
------------------------------------------------------------
struct Stu {
  int  id;                ptrinfo_t    s_info[3];
  char *name;             blkheader_t  s_hdr;
  int  age;               struct Stu   s;
};
struct Stu s;
------------------------------------------------------------
                          ptrinfo_t    p_info;
struct Stu *p;            blkheader_t  p_hdr;
                          struct Stu   *p;
------------------------------------------------------------
```

Figure 5.3: Defining shadow metadata memory for variables.

by the program, while a pointer metadata object is allocated and associated with each four-byte long allocated memory (in a 32-bite architecture), regardless of whether or not the memory actually stores a pointer.

Figure 5.1 shows the type definitions of block metadata and pointer metadata types in C. Each block metadata object (of the type `blkheader_t`) stores the following attributes:

- `base`: The base address of the memory block.
- `size`: The allocated size in bytes of the memory block.
- `blk_cap`: The allocation capability of the memory block.
- `linkbase`: The base address of pointer metadata objects allocated for the memory words in the memory block.

Each pointer metadata object (of the type `ptrinfo_t`) stores the following values:

- `hdr`: The address of the block metadata object associated with the referent block.
- `saved_cap`: The cached capability index value of the referent block.

Figure 5.3 illustrates how additional definitions are introduced in the transformed program to hold metadata for each variable defined in the original program. The relationship between the metadata variables and their corresponding program variables is illustrated in Figure 5.2.

Similar to the shadow structure approach, the shadow metadata memory is also separated from the associated memory blocks and pointers to avoid modifying the underlying pointer representation.

Block metadata objects are allocated from three dedicated pools that are accessible by the instrumentation code. The three pools are used to allocate block metadata objects for global variables, stack variables, and heap variables, respectively. The management of these pools mimics the management of their corresponding memory regions for efficient allocation and utilization of the pool resources. For example, the pool for stack variables is also organized as a stack and operates in a LIFO manner; while the pool for heap variables is an expandable array, where unused slots are organized into a linked free list. We can still use separated capability stores, though they can be combined with these block metadata pools.

Pointer metadata objects are allocated in the same data store as their corresponding program variables and have the same lifetime and scope as their program variables as well. For examples, a stack variable has its pointer metadata objects on stack, while a heap block has its pointer metadata objects on heap.

5.2 Handling Arbitrary Type Casts and Pointer Arithmetic

We allocate only one block metadata object for each memory block in a program regardless of the size of the memory block. However, pointer metadata are usually defined as arrays. The number of pointer metadata objects for each memory block is determined by the size of the memory block. For every four bytes in the memory block, we need a pointer

```
ptrinfo_t *GET_LINK(void *p, blkheader_t *hdr)
{
   int off = p - hdr->base;
   ASSERT(off >= 0);
   /* Pointer should be aligned at 32-bit boundary. */
   if (off % 4 != 0) {
      Abort("unaligned pointer value %p\n", p);
   }
   return hdr->linkbase + (off >> 2);
}
```

Figure 5.4: Definition of helper function `GET_LINK`.

metadata object.

Given a pointer metadata object, we can use the `hdr` field to locate the referent block metadata object. Given a block metadata object and an offset within the memory block, we can use pointer arithmetic on `linkbase` to locate the pointer metadata object associated with the pointer stored at that offset. This is because pointer metadata are maintained for every four-byte word (a possible pointer) in a memory block. We define a helper function `GET_LINK` as shown in Figure 5.4 to compute the address of an embedded pointer metadata given the pointer value and its referent block metadata. As a result, the `link` metadata field used in the shadow structure approach can be replaced by `GET_LINK` and thus is no longer needed. We also no longer need to maintain the runtime type information metadata.

This technique helps us to support non-`malloc` like customized memory allocation functions and also overcome the arbitrary type casts and pointer arithmetic limitations of the shadow structure approach. The reason is that we are assured that the pointer metadata for any possible pointer is always allocated and we know how to locate such a pointer metadata no matter what the declared type of the memory block is. However, it could increase the cost to retrieve an embedded pointer metadata because of the added computations.

Metadata	Condition	Transformation
$B(x)$		`&x`
$B(*p)$		`p_info.hdr->base`
$N(x)$		`sizeof(x)`
$N(*p)$		`p_info.hdr->size`
$L(x)$		`sizeof(x)`
$L(*p)$		`sizeof(*p)`
$L(s, d)$		`sizeof(s.d)`
$C(x)$	x is a global variable	`FOREVER`
$C(x)$	x is a local variable of the current function	`STACK_FRAME_CAP`
$C(*p)$		`p_info.hdr->blk_cap`
$W(x)$	x is an integer	`INVALID_PTRINFO_ADDR`
$W(x)$	x is a structure	`x_hdr.linkbase`
$W(s, d)$		`GET_LINK(&s.d, &s_hdr)`
$W(*p)$		`GET_LINK(p, p_info.hdr)`
$W(p, d)$		`GET_LINK(&p->d, p_info.hdr)`
$M_b(p)$		`p_info.hdr`
$M_b^B(p)$		`p_info.hdr->base`
$M_b^N(p)$		`p_info.hdr->size`
$M_b^C(p)$		`p_info.hdr->blk_cap`
$M_{C'}(p)$		`p_info.saved_cap`
$M_W(p)$		`GET_LINK(p, p_info.hdr)`
$M_W(p, d)$		`GET_LINK(&p->d, p_info.hdr)`
$M_W(p+n)$		`GET_LINK(p+n, p_info.hdr)`
G_H		`CHECK_FREE`
G_T		`CHECK_TEMPORAL`
G_S		`CHECK_SPATIAL`

Figure 5.5: Representing metadata with shadow metadata memory objects.

5.3 Applying Transformation Rules

The metadata attributes and helper routines in Chapter 3 are defined with the shadow metadata memory objects in Figure 5.5. We still use `CHECK_FREE`, `CHECK_TEMPORAL` and `CHECK_SPATIAL` defined in Figure 4.3 as the memory error checking routines.

Most of these definitions are similar to the ones defined for the shadow metadata structure based transformation, except that the block metadata attributes need to be read through

```
Original            Transformed
---------------------------------------------------------------
                    ptrinfo_t    s_info[3];
                    blkheader_t  s_hdr;
                    struct Stu   s;
struct Stu s;
                    s_hdr.base     = &s;
                    s_hdr.size     = sizeof(s);
                    s_hdr.blk_cap  = STACK_FRAME_CAP;
                    s_hdr.linkbase = s_info;
---------------------------------------------------------------
```

Figure 5.6: Initializing block header in shadow metadata memory.

`p_info.hdr`. The definition of $W(x)$ and $M_W(p)$ and their variations are changed significantly because we use the `GET_LINK` helper function to replace the type-based `link` field.

With the above definitions of metadata attributes and helper functions, we can then apply the transformation rules in Figure 3.6 to transform C programs using the shadow metadata memory based technique. Note that we only need to update the metadata fields `hdr` and `saved_cap` as defined in `ptrinfo_t` for pointer assignments. We should skip the updates to block metadata attributes because the properties of the memory block itself are not changed as a result of these operations.

Block metadata are only updated at the memory block creation time, which include the allocation of global variables when a program starts up, the allocation of stack variables when entering a function, and the allocation of heap variables through dynamic memory management functions such as `malloc`. After that, block metadata attributes remain stationary until the memory block is deallocated. During the initialization of block metadata, the values of `base` and `size` are initialized from the location and size information pertaining to the memory block itself; the value of `blk_cap` is a freshly created capability for stack and heap variables, or a special `FOREVER` capability for global variables; and the value of `linkbase` is initialized to the base address of the pointer metadata variable for this memory block. This metadata initialization is based on the mapping between program

Original	Transformed

`p = &s;`	`p_info.hdr       = &s_hdr;`
	`p_info.saved_cap = *s_hdr.blk_cap;`
	`p = &s;`

`q = p + 4;`	`q_info = p_info;`
	`q = p + 4;`

`p = s.d;`	`p_info = *GET_LINK(&s.d, &s_hdr);`
	`p = s.d;`

`q = *p;`	`CHECK_TEMPORAL(p, p_info.hdr->blk_cap,`
	`               p_info.saved_cap);`
	`CHECK_SPATIAL(p, sizeof(*p),`
	`              p_info.hdr->base,`
	`              p_info.hdr->size);`
	`q_info = *GET_LINK(p, p_info.hdr);`
	`q = *p;`

Figure 5.7: Transformation of pointer assignments and dereferences using shadow metadata memory.

variables and their pointer metadata variables maintained during the transformation.

Figure 5.6 shows an example of updating block metadata for a stack variable. Figure 5.7 shows the transformation for several pointer assignment and dereference statements.

When the pointer assignments take place through an assignment from a structure to another, a `memcpy` operation is introduced to copy the values of all the pointer metadata pertaining to the source structure to the pointer metadata for the destination structure.

There are no additional treatments required for pointer assignments involving type casts because pointer metadata are now maintained for every memory word that can be used as a pointer and the pointer metadata for a given memory block can be located through its block metadata.

5.4 Transformation of Functions

5.4.1 Supporting Pointer Arguments of Functions

In this transformation, we choose to use a global argument buffer, defined as the type of `metaarg_pipe_t`, to pass pointer metadata of arguments across functions and keep the function prototypes intact. This is useful especially for supporting transformed functions that are called back from untransformed code (e.g. a comparison function provided to the library function `qsort`). Besides, this is also helpful for supporting separate transformation and compilation.

The steps during a life-cycle of a function invocation are described below:

- When calling a function, the pointer metadata of the function input arguments are copied to the input metadata argument pipe.
- When entering a function, the pointer metadata of its arguments are copied from the input metadata argument pipe to its stack temporary variables.
- When exiting from a function, the pointer metadata of its return value are copied to the output metadata argument pipe.
- When returning from a function at the call site, the pointer metadata of the function return value are copied from the output metadata argument pipe to the destination variables.

Only the pointer metadata of the arguments of the most recent called function need to be kept in the current input and output metadata argument pipes.

5.4.2 Interfacing with Untransformed Functions

To achieve the backwards compatibility with existing C programs, our transformation has to accommodate scenarios where transformed functions interface with untransformed variables or functions.

We assume that we know which variables and functions are external and untransformed. This can be done by annotations or by scanning all source files of an application once before the actual transformation.

Because we do not change the function prototypes and the pointer representations, we do not need to write complex wrapper functions for each external, untransformed function which translate the function prototype and pointer representation between transformed and untransformed formats.

However, we still need to handle metadata for memory blocks that are affected by external functions, because external functions may return new memory blocks, or modify existing memory blocks. Below we discuss some possible approaches that attempt to recover as much as possible metadata when handling various cases of external functions that manipulate pointers.

- *Case 1*: An external function returns a pointer to heap memory blocks allocated in the untransformed libraries.

  We can modify the `malloc/free` family functions (e.g. through LD_PRELOAD or user-defined static linked functions) so that the untransformed libraries can also use the modified `malloc/free` functions. In this way, heap memory blocks allocated in the untransformed code can also be tracked. Their associated metadata values are added into a global lookup table, which can then be consulted by the instrumented code.

- *Case 2*: An external function returns a pointer to stack memory blocks allocated in the untransformed libraries.

  In this case, there are no metadata available for these memory blocks or the resultant pointer. We can either assume such pointers are always valid and permit dereferences of them or flag such dereferences as errors, depending on how conservative we want to be.

- *Case 3*: An external function returns a pointer to unmodified memory blocks that are allocated in the transformed code.

  We can remember all the memory blocks that have been passed into an external function. When the function returns, we check at the call site whether the resultant pointer points to memory blocks referred by arguments.

- *Case 4*: An external function modifies memory blocks allocated in the transformed code and returns a pointer to such memory blocks.

  Similar to Case 3. In addition, certain pointer metadata values might be invalidated and become invalid during the checking.

5.5 Optimizations

In this section, we discuss several plausible optimizations to the shadow metadata memory based transformation.

The first obvious venue for optimizations is the maintenance of pointer metadata for non-pointer variables. In our basic transformation, we maintain the pointer metadata for all the variables that are large enough to hold pointer values. However, this approach may be too conservative and cause unnecessary performance overheads. We can take advantage of static analysis techniques to identify the non-pointer variables that may actually hold pointers and only maintain pointer metadata for pointer variables and these non-pointer variables.

Another alternative approach to the above problem is that we only actively maintain pointer metadata for pointers, but not for non-pointers. When a pointer is cast to an integer, its metadata will be saved in a global lookup table indexed by the address range of the pointer referent in a way similar to the Jones & Kelly's approach. When an integer is cast to a pointer, the global table will be queried and if there are metadata associated with the integer value, the metadata will be retrieved and used to update the pointer metadata of the assigned pointer. When a memory block is released, its entry in the global lookup table needs not be removed immediately because its block metadata is already invalidated and this invalidation will be seen when the table entry is accessed. A periodically cleanup of the lookup table is sufficient.

The second optimization is that we can eliminate a block metadata object if the corresponding memory block is not referenced through pointers, i.e., the address of the memory block is not assigned to any pointers. In addition, we can further reduce the number of attributes in pointer metadata for trade-offs between the number of capability slot reuses and the execution time and memory space overheads in common cases. For example, we may use only four bits for `saved_cap`, and store these four bits as the lowest bits in `hdr` because each block metadata object has 16 bytes and the pointer `hdr` is aligned at the 16-byte boundary.

We can also reduce the performance overheads by removing the redundant pointer metadata updates and checks through intra-procedural static analyses. For example, it is easy to see that there is no need to update pointer metadata if the updated pointer still points to the same referent, e.g. `p = p + 1`.

Part II

Taint-Enhanced Policy Enforcement

Chapter 6

Injection Vulnerability Problem Statement

6.1 Injection Attacks and Input Validation Errors

While memory error vulnerabilities in C/C++ applications still remain as a significant threat, the vulnerabilities reported in Web applications have been rapidly increased recently. The majority of these Web vulnerabilities can be classified as *injection vulnerabilities*, including SQL injection, command injection, cross-site scripting, path traversal, and so on. They have become one of the largest classes of CVE vulnerabilities [76] in recent years.

As described in Section 1.1.2, the attack target of an injection attack is a program that takes requests from untrusted sources and carries out the requests with operations that are often security-sensitive. Due to software bugs, the input validations on the untrusted inputs might be incomplete or even missing from some program paths. By exploiting these input validation vulnerabilities, an attacker can then "inject" malicious operation parameters or even operations themselves. In the example shown in Figure 1.3, the attack target is SquirrelMail, a Web email application written in PHP. When a user requests to encrypt an email with PGP, SquirrelMail constructs a `gpg` shell command with the user-provided recipient value (passed to SquirrelMail through the `sendto` HTTP GET variable) as the argument. SquirrelMail then invokes the `popen` function to execute this shell command. A legitimate recipient value should not contain any shell command meta-characters. However, due to an input validation bug in SquirrelMail, an attacker could provide an illegal recipient value to inject any arbitrary shell command that will be executed by `popen` on the Web server.

6.2 Approach Overview

Previous runtime approaches have been limited to address only specific types of injection attacks [28, 21, 42, 70, 65]. In our research, we identify that the root cause of injection vulnerability exploits is due to untrusted (i.e. tainted) input data being used unsafely in security operations. Based on this key observation, we have developed a dynamic taint analysis based approach to detect the attacks that exploit injection vulnerabilities.

Information flow analysis (a.k.a. *taint analysis*) has played a central role in computer security for over three decades [16, 37, 31, 85, 74]. The recent works of [83, 24, 64] pioneered the idea of detecting memory corruption induced control flow attacks on contemporary software through fine-grained dynamic taint analysis.

Our approach, called *taint-enhanced policy enforcement* [91], is also built on the basic idea of using fine-grained taint analysis for attack detection, but expand its scope significantly by showing that the technique can be applied to detect a much wider range of attacks prevalent today, including all the injection attacks. Our approach consists of two essential steps: *fine-grained taint tracking* and *policy enforcement*.

The first step is to track the source (or *taintedness*) of each byte of data that is manipulated by a program during its execution. We develop an automated source-to-source transformation to instrument C programs for enabling efficient fine-grained byte-level runtime taint tracking. Taint originates at input functions, e.g., a `read` or `recv` function used by a server to read network inputs. Input operations that return untrusted inputs are specified using *marking specifications* described in Chapter 9.

The second step is to enforce taint-enhanced security policies that are associated with security operations. These policies are taint-enhanced in that they make use of the runtime taint information to check "unsafe" content in security operation parameters. There are typically a small number of such security operations, e.g., system calls such as `open` and `execve`, library functions such as `vfprintf`, functions to access external modules such as an SQL database, and so on.

The combination of these two steps turns out to be powerful for injection attack detection, and offers the following advantages over previous techniques:

- *Practicality.* The techniques of [83] and [24] rely on hardware-level support for taint-tracking, and hence cannot be applied to today's systems. TaintCheck [64] addresses this drawback, and is applicable to arbitrary COTS binaries. However, due to difficulties associated with static analysis (or transformation) of binaries, their implementation uses techniques based on a form of runtime instruction emulation [63], which causes a significant slowdown, e.g., Apache server response time increases by a factor of 10 while fetching 10KB pages. In contrast, our technique is much faster, increasing the response time by a factor of 1.1.

- *Broad applicability.* Our technique is directly applicable to programs written in C, and several other scripting languages (e.g., PHP, Bash) whose interpreters are implemented in C. Security-critical servers are most frequently implemented in C. In addition, PHP and similar scripting languages are common choices for implementing web applications, and more generally, server-side scripts.

- *Ability to detect a wide range of common attacks, including all types of injection attacks.* By combining expressive security policies with fine-grained taint information, our technique can address a broader range of attacks than previous techniques.

Figure 1.1 shows the distribution of the CVE vulnerabilities reported on COTS software in 2003 and 2004. Our technique is applicable for detecting exploitations of about two-thirds of these vulnerabilities, including buffer overflows, format-string attacks, SQL injection, cross-site scripting, command and shell-code injection, and directory traversal. In contrast, previous approaches typically handled smaller attack classes, e.g., [30, 36, 18, 64, 83, 24] handle buffer overflows, [28] handles format string attacks, and [70, 65] handle SQL injection attacks.

We have successfully applied our technique to several medium to large programs, such as the PHP interpreter (300KLOC+) and `glibc`, the GNU standard C library (about 1MLOC). By leveraging the low-level nature of the C language, our implementation works correctly even in the face of memory errors, type casts, aliasing, and so on. At the same time, by exploiting the high-level nature of C (as compared to binary code), we have developed optimizations that significantly reduce the runtime overheads of fine-grained dynamic taint-tracking.

We establish the need for taint-enhanced policy enforcement with motivating attack examples in Chapter 7. We describe our source-code transformation for fine-grained taint tracking in Chapter 8. Taint-enhanced policy specifications and refinements are provided in Chapter 9. We present The implementation and experimental evaluation of our approach in Chapter 11.

6.3 Related Work

Fine-Grained Taint Analysis. The key distinctions between our work and previous fine-grained taint analysis techniques of [64, 83, 24] were already discussed in the introduction, so we limit our discussion to the more technical points here. As mentioned earlier, [83, 24] rely on hardware support for taint-tracking. [64] is closer to our technique than these two techniques. It has an advantage over ours in that it can operate on arbitrary COTS binaries, whereas we require access to the C source code. This avoids problems such as hand-written assembly code. Their main drawback is performance: on the application Apache that they provide performance numbers on, their overheads are more than 100 times higher than ours. This is because (a) they rely on Valgrind, which in itself introduces more than 40 times overheads as compared to our technique, and (b) they are constrained by having to work on binary code, and without the benefit of static analyses and optimizations that have gone into our work. (Here, we are not only referring to our own analyses and optimizations, but also many of the optimizations implemented in the GCC compiler that we used to compile the transformed programs.)

There are several other technical differences between our work and that of [64]. For instance, they track 32-bits of taint information for each byte of data, whereas we use 2 bits. Another important difference is our support for implicit flows, which are not handled in [64].

The more recent approach from Prateek et al [75] employs a combination of static analysis techniques to optimize the performance of binary instrumentation based fine-grained taint tracking. In contrast, our approach is based on source-code transformation and is able to take advantage of high-level information available in source code and the optimizations implemented in C compilers. Their approach has significantly reduced the performance overheads compared to TaintCheck and reported performance numbers comparable to ours.

Dynamic Taint Based Techniques for Detecting Attacks on Web Applications. Independently and in parallel to our work, [65] and [70] have proposed the idea of using fine-grained taint analysis to detect injection attacks on web applications. The implementations of [65] and [70] are very similar, using hand-transformation of the PHP interpreter to track taint data. However, [70] provides a more detailed formulation and discussion of the problem, so we focus on this work here. They explain that these injection attacks are the result of *ad hoc* serialization of complex data such as SQL queries or shell commands, and develop a detection technique called *context-sensitive string evaluation (CSSE)*, which involves checking the use of tainted data in strings. Our work improves over theirs in several ways. First, by working at the level of the C language, we are able to handle many more applications: most server programs that are written in C, as well as programs written in interpreted languages such as PHP, bash and so on. Second, our formulation of the problem as taint-enhanced policy enforcement is more general, and can be applied to stealthy attacks such as those discussed in Chapter 7 that do not involve serialization problems; and to attacks involving arbitrary types of data rather than being limited to strings. Third, our approach relies on a simple transformation that is shown in Chapter 8, and implemented using 3.6KLOC of code, while their approach relies on manual transformation of a large piece of software that has over 300KLOC. Other technical contributions of our work include (a) the development of a simple policy language for concise specification of taint-enhanced polices, and (b) support for implicit flows that allow us to provide some support for character encodings and translations.

Su et al [81] describe a technique for detecting SQL injection attacks using syntax analysis. Their main focus is on providing a precise and formal characterization of SQL injection attacks. However, their implementation of taint tracking is not very reliable. In particular, they suggest a technique that avoids runtime operations for taint-tracking by "bracketing" each input string with two special symbols that surround untrusted input strings. Assuming that these brackets would be propagated together with input strings, checking for the presence of taint would reduce to checking for the presence of these special symbols. However, this assumption does not hold for programs that extract parts of their input and use them, e.g., a web application may remove non-alphanumeric characters from an input string and use them, and this process would likely discard the bracketing characters. In other cases, a web application may parse a user input into multiple fields, and use each field independently, once again causing the special symbols to be lost.

Sekar has recently proposed *taint inference* as a black box technique to detect injection attacks on Web applications [76]. This approach still uses taint-annotated security policies for attack detection. However, the taint information does not come from the runtime tracking of taint propagation during program execution. Instead, the taint information is inferred by comparing the argument values of security operations with the logged inputs. Taint inference has the nice property of being neutral to the application implementation details, though taint tracking, whenever possible, can always provide more accurate taint information and improve the attack detection accuracy.

Manual Approaches for Correcting Input Validation Errors. Taint analysis targets vulnerabilities that arise due to missing or incorrect input validation code. One can manually review the code, and try to add all the necessary input validation checks. However, the notion of validity is determined by the manner in which the input is used. Thus, one has to trace forward in the program to identify all possible uses of an input in security sensitive operations, which is a very time-consuming and error-prone task. If we try to perform the validation check at the point of use, we face the problem that the notion of validity depends on the data source. For instance, it is perfectly reasonable for an SQL query to contain semicolons if these originated within the program text, but not so if it came from external input. Thus, we have to trace back from security-sensitive operations to identify how its arguments were constructed, once again having to manually examine large number of program paths. This leads to situations where validation checks are left out on some paths, and possibly duplicated on others. Moreover, the validation checks themselves are notoriously hard for programmers to code correctly, and have frequently been the source of vulnerabilities.

Information Flow. Information flow analysis has been researched for a long time [16, 37, 31, 56, 85, 59, 74]. Early research was focused on multi-level security, where fine-grained analysis was not deemed necessary [16]. More recent work has been focused on tracking information flow at variable level, and many interesting research results have been produced. While these techniques are promising for protecting privacy and integrity of sensitive data, as discussed in Chapter 7, the variable-level granularity is insufficient for detecting most attacks discussed in this chapter.

Static Analysis. Static taint analysis techniques have been proposed by many for finding security vulnerabilities, including input validation errors in web applications [53, 46, 88], user/kernel pointer bugs [48], format string bugs [78], and bugs in placement of authorization hooks [94]. The main advantage of static analysis (as compared to runtime techniques) is that all potential vulnerabilities can be found statically, while its drawback is a relative lack of accuracy. In particular, these techniques typically detect *dependencies* rather than vulnerabilities. For instance, [53] will produce a warning whenever untrusted data is used in any manner in an SQL query. This may not be very useful if such a dependency is an integral part of application logic. To solve this problem, the concept of *endorsement* can be used to indicate "safe" dependencies. Typically, this is done by first performing appropriate validation checks on a piece of untrusted data, and then endorsing it to indicate that it is safe to use (i.e., no longer "tainted"). However, programmers are still responsible for determining what is "safe" — as discussed before, there is no easy way for them to do this.

An important difference between our work and static analysis is one of intended audience. Static analysis based tools are typically intended for use by developers, since they need detailed knowledge about program logic to determine where to introduce endorsements, and what validation checks need to be made before endorsement. In contrast, the audience for our tool is a system administrator or an outside security engineer that lacks detailed knowledge of application code.

Other Techniques. SQLrand [21] defeats SQL injection by randomizing the textual representation of SQL commands. A drawback of this approach, as compared to the technique presented in this chapter, is that it requires manual changes to the program so that the program uses the modified representation for SQL commands generated by itself. Our approach was inspired by the effect achieved by SQLrand, namely, that of distinguishing commands generated by the application from those provided by untrusted users.

AMNESIA[42] is another interesting approach for detecting SQL injection attacks. It uses a static analysis of Java programs to compute a finite-state machine model that captures the lexical structure of SQL queries issued by a program. SQL injection attacks cause SQL queries issued by the program to deviate from this model, and hence detected. A key benefit of this approach is that by using static analysis, it can avoid runtime taint-tracking, and is hence much more efficient than our approach. Although this approach has been demonstrated to work well for SQL injections, the conservative nature of its static analysis and its inability to distinguish different sources of inputs can lead to a higher rate of false positives when applied to other types of attacks.

There are several recent interesting work on detecting cross-site scripting (XSS) attacks. Both DSI [60] and Noncespaces [41] requires the cooperations between the web server and the browser in XSS attack detection. In DSI, the server demarcates the inline untrust content in a web page with special, randomized markup primitives. The browser is responsible for tracking the processing of the untrusted content and prevent them from compromising the document structure integrity. In Noncespaces, the web application partitions content on a web page into different trust classes represented as random XML namespace prefixes and also specifies the policies for each trust class, which are enforced in the browser. BluePrint [55] works on a different assumption. It does not put any trust on the browser. Instead, the web application takes control of parsing web pages. It carefully transports and reproduces a blueprint of an untrusted web content that is free of XSS attacks. These approaches represent the most recent advances on XSS attack detection. However, they do not provide a general protection against other types of injection attacks.

Perl has a taint mode [87] that tracks taint information at a coarse granularity – that of variables. In Perl, one has to explicitly untaint data before using it in a security sensitive context. This is usually done after performing appropriate validations. In our approach, due to the flexibility provided by our policy language, we have not faced a need for such explicit untainting. Nevertheless, if a user explicitly wants to trust some input, a primitive can be easily added to support this.

Chapter 7

Motivation for Taint-Enhanced Policies

In this chapter, we present motivating attack examples. We conclude by pointing out the integral role played by taint analysis as well as security policies in detecting these attacks.

7.1 SQL and Command Injection

Figure 1.3 shows an example of common injection attacks, which involve untrusted inputs being used as to construct commands executed by command interpreters (e.g., `bash`) or the argument to `execve` system call.

SQL injection is a common vulnerability in web applications. These server-side applications communicate with a web browser client to collect data, which is subsequently used to construct an SQL query that is sent to a back-end database. Consider the statement (written in PHP) for constructing an SQL query used to look up the price of an item specified by the variable `name`.

```
$cmd = "SELECT price FROM products WHERE name = '" . $name . "'"
```

If the value of `name` is assigned from an HTML form field that is provided by an untrusted user, then an SQL injection is possible. In particular, an attacker can provide the following value for `name`:

```
xyz'; UPDATE products SET price = 0 WHERE name = 'OneCaratDiamondRing
```

With this value for `name`, `cmd` will take the value:

```
                          SELECT ...  WHERE name =
'[xyz'; UPDATE products SET price = 0 WHERE name = 'OneCaratDiamondRing]'
```

Note that semicolons are used to separate SQL commands. Thus, the query constructed by the program will first retrieve the price of some item called `xyz`, and then set the price of another item called `OneCaratDiamondRing` to zero. This attack enables the attacker to purchase this item later for no cost.

Our taint-enhanced policy enforcement approach can detect both SQL and command injections. In this SQL injection example, fine-grained taint analysis will mark every character in the query that is within the box as tainted. Now, a policy that precludes tainted control-characters (such as semicolons and quotes) or commands (such as `UPDATE`) in the SQL query will defeat the above attack. A more refined policy is described in Section 9.3.

7.2 Cross-Site Scripting (XSS)

Consider an example of a bank that provides a "ATM locator" web page that customers can use to find the nearest ATM machine, based on their ZIP code. Typically, the web page contains a form that submits a query to the web site, which looks as follows:

```
http://www.xyzbank.com/findATM?zip=90100
```

If the ZIP code is invalid, the web site typically returns an error message such as:

```
<HTML> ZIP code not found:  90100 </HTML>
```

Note in the above output from the web server, the user-supplied string `90100` is reproduced. This can be used by an attacker to construct an XSS attack as follows. To do this, the attacker may send an HTML email to an unsuspecting user, which contains text such as:

```
To claim your reward, please click <a href="http://www.xyzbank.com/
findATM?zip=<script%20src='http://www.attacker.com/malicious_script.js'>
</script>">here</a>
```

When the user clicks on this link, the request goes to the bank, which returns the following page:

```
<HTML> ZIP code not found: <script
src='http://www.attacker.com/malicious_script.js'></script> </HTML>
```

The victim's browser, on receiving this page, will download and run Javascript code from the attacker's web site. Since the above page was sent from `http://www.xyzbank.com`, this script will have access to sensitive information stored on the victim computer that pertains to the bank, such as cookies. Thus, the above attack will allow cookie information to be stolen. Since cookies are often used to store authentication data, stealing them can allow attackers to perform financial transactions using victim's identity.

Fine-grained taint analysis will mark every character in the zip code value as tainted. Now the above cross-site scripting attack can be prevented by disallowing tainted `script` tags in the web application output.

7.3 Format String Attacks

The `printf` family of functions (which provide formatted printing in C) take a format string as a parameter, followed by zero or more parameters. A common misuse of these functions occurs when untrusted data is provided as the format string, as in the statement "`printf(s)`." If `s` contains an alphanumeric string, then this will not cause a problem, but if an attacker inserts format directives in `s`, then she can control the behavior of `printf`. In the worst case, an attacker can use the "`%n`" format directive, which can be used to overwrite a return address with attacker-provided data, and execute injected binary code.

When fine-grained taint analysis is used, the format directives (such as "`%n`") will be marked as tainted. The above attack can be then prevented by a taint-enhanced policy that disallows tainted format directives in the format string argument to the `printf` family of functions.

7.4 Memory Error Exploits

There are many different types of memory error exploits, such as *stack-smashing, heap-overflows* and *integer overflows*. All of them share the same basic characteristics: they exploit bounds-checking errors to overwrite security-critical data, almost always a code pointer or a data pointer, with attacker-provided data. When fine-grained taint analysis is used, it will mark the overwritten pointer as tainted. Now, this attack can be stopped by a policy that prohibits dereferencing of tainted pointers.

7.5 Attacks that "Hijack" Access Privileges

In this section, we consider attacks that attempt to evade detection by staying within the bounds of normal accesses made by an application. These attacks are also referred to as the confused deputy attacks [43].

Consider a web browser vulnerability that allows an attack (embedded within a web page) to upload an arbitrary file f owned by the browser user without the user's consent. Since the browser itself needs to access many of the user's files (e.g., cookies), a policy that prohibits access to f may prevent normal browser operations. Instead, we need a policy that can infer whether the access is being made during the normal course of an operation of the browser, or due to an attack. One way to do this is to take the taint information associated with the file name. If this file is accessed during normal browser operation, the file name would have originated within its program text or from the user. However, if the file name originated from a remote web site (i.e., an untrusted source), then it is likely to be an attack. Similar examples include attacks on (a) P2P applications to upload (i.e., steal) user files, and (b) FTP servers to download sensitive files such as the password file that are normally accessed by the server.

A variant of the above scenario occurs in the context of *directory traversal* attacks, where an attacker attempts to access files outside of an authorized directory, e.g., the document root in the case of a web server. Typically, this is done by including "`..`" characters in file names to ascend above the document root. In case the victim application already incorporates checks for "`..`" characters, attacker may attempt to evade this check by replacing "`.`" with its hexadecimal or Unicode representation, or by using various escape sequences. A taint-enhanced policy can be used to selectively enforce a more restrictive policy on file access when the file name is tainted, e.g., accesses outside of the document root directory may be disallowed. Such a policy would not interfere with the web server's ability to access other files, e.g., its access log or error log.

The key point about all attacks discussed in this section is that conventional access control policies cannot detect them. This is because the attacks do not stray beyond the set of resources that are normally accessed by a victim program. However, taint analysis provides a clue to *infer the intended use of an access.* By incorporating this inferred intent in granting access requests, taint-enhanced policies can provide better discrimination between attacks and legitimate uses of the privileges granted to a victim application.

7.6 Discussion

The examples discussed above bring out the following important points:

- *Importance of fine-grained taint information.* If we used coarser granularity for taint-tracking, e.g., by marking a program variable as tainted or untainted, we would not be able to detect most of the attacks described above. For instance, in the case of

SQL injection example, the variable `cmd` containing the SQL query will always be marked as tainted, as it derives part of its value from an untrusted variable `name`. As a result, we cannot distinguish between legitimate uses of the web application, when `name` contains an alphanumeric string, from an attack, when `name` contains characters such as the semicolon and SQL commands. A similar analysis can be made in the case of stack-smashing and format-string attacks, cross-site scripting, directory traversal, and so on.

- *Need for taint-enhanced policies.* It is not possible to prevent these attacks by enforcing conventional access control policies. For instance, in the SQL injection example, one cannot use a policy that uniformly prevents the use of semicolons and SQL commands in `cmd`: such a policy would preclude any use of the database, and cause the web application to fail. Similarly, in the memory error example, one cannot have a working program if all control transfers through pointers are prevented. Finally, the examples in Section 7.5 were specifically chosen to illustrate the need for combining taint information into policies.

  Another point to be made in this regard is that attacks are not characterized simply by the presence or absence of tainted information in arguments to security-critical operations. Instead, it is necessary to develop policies that govern the manner in which tainted data is used in these arguments.

Chapter 8

Transformation for Taint Tracking

8.1 Runtime Representation of Taint

Our technique tracks taint information at the level of bytes in memory. This is necessary to ensure accurate taint-tracking for type-unsafe languages such as C, since the approach can correctly deal with situations such as out-of-bounds array writes that overwrite adjacent data. A one-bit taint-tag is used for each byte of memory, with a '0' representing the absence of taint, and a '1' representing the presence of taint. A bit-array `tagmap` stores taint information. The taint bit associated with a byte at address a is given by `tagmap[a]`.

8.2 Transformation for Fine-Grained Taint Tracking

The source-code transformation described in this section is designed to track *explicit information flows* that take place through assignments and arithmetic and bit-operations. Flows that take place through conditionals are addressed in Section 8.3. It is unusual in C programs to have boolean-valued variables that are assigned the results of relational or logical operations. Hence we have not considered taint propagation through such operators in this dissertation. At a high-level, explicit flows are simple to understand:

E	$T(E)$	Comment
c	0	Constants are untainted
v	$tag(\&v,\ \texttt{sizeof}(v))$	$tag(a, n)$ refers to n bits starting at $tagmap[a]$
$\&E$	0	An address is always untainted
$*E$	$tag(E,\ \texttt{sizeof}(*E))$	
$(cast)E$	$T(E)$	Type casts don't change taint.
$op(E)$	$T(E)$ 0	for arithmetic/bit op otherwise
$E_1\ op\ E_2$	$T(E_1) \mid\mid T(E_2)$ 0	for arithmetic/bit op otherwise

Figure 8.1: Definition of taint for expressions

S	$Trans(S)$
$v = E$	$v = E;$ $tag(\&v, \texttt{sizeof}(v)) = T(E);$
$S_1; S_2$	$Trans(S_1); Trans(S_2)$
`if` $(E)\ S_1$ `else` S_2	`if` $(E)\ Trans(S_1)$ `else` $Trans(S_2)$
`while` $(E)\ \ S$	`while` $(E)\ \ Trans(S)$
`return` E	`return` $(E, T(E))$
$f(a)\ \{\ S\ \}$	$f(a, ta)\ \{\ tag(\&a, \texttt{sizeof}(a)) = ta; Trans(S)\}$
$v = f(E)$ $v = (*f)(E)$	$(v, tag(\&v, \texttt{sizeof}(v))) = f(E, T(E))$ $(v, tag(\&v, \texttt{sizeof}(v))) = (*f)(E, T(E))$

Figure 8.2: Transformation of statements for taint tracking

- The result of an arithmetic/bit expression is tainted if any of the variables in the expression is tainted;
- A variable x is tainted by an assignment $x = e$ whenever e is tainted.

Specifically, Table 8.1 shows how to compute the taint value $T(E)$ for an expression E. Table 8.2 defines how a statement S is transformed, and uses the definition of $T(E)$. When describing the transformation rules, we use a simpler form of C (e.g. expressions have no side effects). In our implementation, we use the CIL [57] toolkit as the C front end to provide the simpler C form that we need.

The transformation rules are self-explanatory for most part, so we explain only the function-call related transformations. Consider a statement $v = f(E)$, where f takes a single argument. We introduce an additional argument ta in the definition of f so that the taint tag associated with its (single) parameter could be passed in. ta is explicitly assigned as the taint value of a at the beginning of f's body. (These two steps are necessary since the C language uses call-by-value semantics. If call-by-reference were to be used, then neither step would be needed.) In a similar way, the taint associated with the return value has to be explicitly passed back to the caller. We represent this in the transformation by returning a pair of values as the return value. (In our implementation, we do not actually introduce additional parameters or return values; instead, we use a second stack to communicate the taint values between the caller and the callee.) It is straight-forward to extend the transformation rules to handle multi-argument functions.

We conclude this section with a clarification on our notion of soundness of taint information. Consider any variable x at any point during any execution of a transformed program, and let a denote the location of this variable. If the value stored at a is obtained from any tainted input through assignments and arithmetic/bit operations, then `tagmap[a]` should be set. Note that by referring to the location of x rather than its name, we require that taint information be accurately tracked in the presence of memory errors. To support this notion of soundness, we needed to protect the `tagmap` from corruption, as described in Section 8.5.

8.3 Support for Implicit Information Flow

Implicit information flow occurs when the values of certain variables are related by virtue of program logic, even though there are no assignments between them. A classic example is given by the code snippet [74]:

```
x=x%2; y=0; if (x==1) y=1;
```

Even though there is no assignments involving `x` and `y`, their values are always the same. The need for tracking such implicit flows has long been recognized. [40] formalized implicit flows using a notion of *noninterference*. Several recent research efforts [56, 85, 59] have developed techniques based on this concept.

Noninterference is a very powerful property, and can capture even the least bit of correlation between sensitive data and other data. For instance, in the code:

```
if (x > 10000) error = true;
if (!error) { y = "/bin/ls"; execve(y); }
```

there is an implicit flow from `x` to `error`, and then to `y`. Hence, a policy that forbids tainted data to be used as an `execve` argument would be violated by this code. This example illustrates why non-interference may be too conservative (and hence lead to false positives) in our application. In the context of the kinds of attacks we are addressing, attackers usually need more control over the value of `y` than the minimal relationship that exists in the code above. Thus, it is more appropriate to track explicit flows. Nevertheless, there can be cases where substantial information flow takes place without assignments, e.g., in the following if-then-else, there is a direct flow of information from x to y on both branches, but our formulation of explicit information flow would only detect the flow in the else statement.

```
if (x == 0) y = 0; else y = x;
```

The goal of our approach is to support those implicit flows where the value of one variable *determines* the value of another variable. By using this criteria, we seek a balance between tracking necessary data value propagation and minimizing false positives. Currently, our implementation supports two forms of implicit flows that appear to be common in C programs.

- *Translation tables.* Decoding is sometimes implemented using a table look up, e.g.,

```
y = translation_tab[x];
```

where `translation_tab` is an array and `x` is a byte of input. In this case, the value of `x` determines the value of `y` although there is no direct assignment from `x` to `y`. To handle this case, we modify the basic transformation so that the result of an array access is marked as tainted whenever the subscript is tainted. This successfully handles the use of translation tables in the PHP interpreter.

- *Decoding using if-then-else/switch.* Sometimes, decoding is implemented using a statement of the form:

  ```
  if (x == '+') y = ' ';
  ```

  (Such code is often used for URL-decoding.) Clearly, the value of `y` can be determined by the value of `x`. More generally, `switch` statements could be used to translate between multiple characters. Our transformation handles them in the same way as a series of if-then-else statements. Specifically, consider an if-then-else statement of the form:

  `if (x ==` E`) { ... y =` E'`; ... }`

  If E and E' are constant-valued, then we add a tag update $tag(y) = tag(x)$ immediately before the assignment to y.

While our current technique seems to identify some of the common cases where implicit flows are significant, it is by no means comprehensive. Development of a more systematic approach that can provide some assurances about the kinds of implicit flows captured, while ensuring a low false positive rate, is a topic of future research.

8.4 Optimizations

The basic transformation described above is effective, but introduces high overheads, sometimes causing a slowdown by a factor of 5 or more. To improve performance, we have developed several interesting runtime and compile-time optimizations that have reduced overheads significantly. More details about the performance can be found in Section 11.2.4.

8.4.1 Runtime Optimizations

In this section, we describe optimizations to the runtime data structures.

Use of 2-bit taint values. In the implementation, accessing of taint-bits requires several bit-masking, bit-shifting and unmasking operations, which degrade performance significantly. We observed that if 2-bit taint tags are used, the taint value for an integer will be contained within a single byte (assuming 32-bit architecture), thereby eliminating these bit-level operations. Since integer assignments occur very frequently, this optimization is quite effective.

This approach does increase the memory requirement for `tagmap` by a factor of two, but on the other hand, it opens up the possibility of tracking richer taint information. For instance, it becomes possible to associate different taint tags with different input sources and track them independently. Alternatively, it may be possible to use the two bits to capture "degree of taintedness."

Allocation of `tagmap`. Initially, we used a global variable to implement `tagmap`. But the initialization of this huge array (1GB) that took place at the program start incurred significant overheads. Note that tag initialization is warranted only for static data that is initialized at program start. Other data (e.g., stack and heap data) should be initialized (using assignments) before use in a correctly implemented program. When these assignments are transformed, the associated taint data will also be initialized, and hence there is no need to initialize such taint data in the first place. So, we allocated `tagmap` dynamically, and initialized only the locations corresponding to static data. By using `mmap` for this allocation, and by performing the allocation at a fixed address that is unused in Linux (our implementation platform), we ensured that runtime accesses to `tagmap` elements will be no more expensive than that of a statically allocated array (whose base address is also determined at compile-time).

The above approach reduced the startup overheads, but the mere use of address space seemed to tie up OS resources such as page table entries, and significantly increased time for `fork` operations. For programs such as shells that fork frequently, this overhead becomes unacceptable. So we devised an *incremental allocation* technique that can be likened to user-level page-fault handling. Initially, `tagmap` points to 1GB of address space that is unmapped. When any access to `tagmap[i]` is made, it results in a UNIX signal due to a memory fault. In the transformed program, we introduce code that intercepts this signal. This code queries the operating system to determine the faulting address. If it falls within the range of `tagmap`, a chunk of memory (say, 16KB) that spans the faulting address is allocated using `mmap`. If the faulting address is outside the range of `tagmap`, the signal is forwarded to the default signal handler.

8.4.2 Compile-time Optimizations

Use of local taint tag variables. In most C programs, operations on local variables occur much more frequently than global variables. Modern compilers are good at optimizing local variable operations, but due to possible aliasing, most such optimizations cannot be safely applied to global arrays. Unfortunately, the basic transformation introduces one operation on a global array for each operation on a local variable, and this has the effect of more than doubling the runtime of transformed programs. To address this problem we modified our transformation so that it uses local variables to hold taint information for local variables, so that the code added by the transformer can be optimized as easily as the original code.

Note, however, that the use of local tag variables would be unsound if aliasing of a local variable is possible. For example, consider the following code snippet:

```
int x; int *y = &x;
x = u; *y = v;
```

If `u` is untainted and `v` is tainted, then the value stored in `x` should be tainted at the end of the above code snippet. However, if we introduced a local variable, say, `tag_x`, to store the taint value of `x`, then we cannot make sure that it will get updated by the assignment to `*y`.

To ensure that taint information is tracked accurately, our transformation uses local taint tag variables only in those cases where no aliasing is possible, i.e., the optimization is limited to simple variables (not arrays) whose address is never taken. However, this alone is not enough, as aliasing may still be possible due to memory errors. For instance, a simple variable `x` may get updated due to an out-of-bounds access on an adjacent array, say, `z`. To eliminate this possibility, we split the runtime stack into two stacks. The main stack stores only simple variables whose addresses are never taken. This stack is also used for call-return. All other local variables are stored in the second stack, also called *shadow stack*.

The last possibility for aliasing arises due to pointer-forging. In programs with possible memory errors, a pointer to a local variable may be created. However, with the above transformation, any access to the main stack using a pointer indicates a memory error. We show how to implement an efficient mechanism to prevent access to some sections of memory in the transformed program. Using this technique, we prevent all accesses to the main stack except using local variable names, thus ensuring that taint information can be accurately tracked for the variables on the main stack using local taint tag variables.

Intra-procedural dependency analysis is performed to determine whether a local variable can ever become tainted, and to remove taint updates if it cannot. Note that a local variable can become tainted *only if* it is involved in an assignment with a global variable, a procedure parameter, or another local variable that can become tainted. Due to aliasing issues, this optimization is applied only to variables on the main stack.

8.5 Protecting Memory Regions

To ensure accurate taint-tracking, it is necessary to preclude access to certain regions of memory. Specifically, we need to ensure that the `tagmap` array itself cannot be written by the program. Otherwise, `tagmap` may be corrupted due to programming errors, or even worse, a carefully crafted attack may be able to evade detection by modifying the `tagmap` to hide the propagation of tainted data. A second region that needs to be protected is the main stack. Third, it would be desirable to protect memory that should not directly be accessed by a program, e.g., the GOT. (Global Offset Table is used for dynamic linking, but there should not be any reference to the GOT in the C code. If the GOT is protected in this manner, that would rule out attacks based on corrupting a function pointer in the GOT.)

The basic idea is as follows. Consider an assignment to a memory location a. Our transformation ensures that an access to `tagmap[a]` will be made before a is accessed. Thus, in order to protect a range of memory locations l—h, it is enough if we ensure that `tagmap[l]` through `tagmap[h]` will be unmapped. This is easy to do, given our incremental approach to allocation of `tagmap`. Now, any access to addresses l through h will result in a memory fault when the corresponding `tagmap` location is accessed.

Note that l and h cannot be arbitrary: they should fall on a 16K boundary, if the page size is 4KB and if 2 bit tainting is used. This is because `mmap` allocates memory blocks whose sizes are a multiple of a page size. This alignment requirement is not a problem for `tagmap`, since we can align it on a 16K boundary. For the main stack, a potential issue arises because the bottom of the stack holds environment variables and command-line arguments that are arrays. To deal with this problem, we first introduce a gap in the stack in `main` so that its top is aligned on a 16K boundary. The region of main stack above this point is protected using the above mechanism. This means that it is safe to use local tag variables in any function except `main`.

Chapter 9

Input Marking and Policy Specification

9.1 Marking Trusted and Untrusted Inputs

Marking involve associating taint information with all the data coming from external sources. If all code, including libraries, is transformed, then marking needs to be specified for system calls that return inputs, for environment variables and command-line arguments. (If some libraries are not transformed, then marking specifications may be needed for untransformed library functions that perform inputs.) Note that we can treat command-line arguments and environment variables as arguments to `main`. Thus, marking specifications can, in every case, be associated with a function call.

Input marking are specified using a simple policy language. Each policy rule is of the form $pattern \longrightarrow action$, where $pattern$ is of the form $function \mid condition$. When the specified function returns, and (the optional) $condition$ holds, $action$ will be executed. The event corresponding to a function will take an additional argument that captures the return value from the function. Both the *condition* and the *action* can use external functions (written in C or C++). Moreover, the *action* can include arithmetic and logical operations, as well as if-then-else. Consider the following example:

```
read(fd, buf, size, rv)|(rv > 0) →
    if (isNetworkEndpoint(fd))  taint_buffer(buf, rv);
    else untaint_buffer(buf, rv);
```

This rule states that when the `read` function returns, the `buf` argument will be tainted, based on whether the read was from a network or not, as determined by the external function `isNetworkEndpoint`. The actual tainting is done using two support functions `taint_buffer` and `untaint_buffer`.

Note that every input action needs to have an associated marking rule. To reduce the burden of writing many rules, we provide default rules for all system calls that untaint the data returned by each system call. Specific rules that override these default rules, such as the rule given above, can then be supplied by a user.

9.2 Specifying Taint-Enhanced Policies

Security policies are also written in the afore-mentioned policy language, but these rules are somewhat different from the marking rules. For a policy rule involving a function f, its *condition* component is examined immediately *before* any invocation of f. To simplify the policy specification, *abstract events* can be defined to represent a set of functions that share the same security policy. (Abstract events can be thought of as macros.)

The definition of *condition* is also extended to support regular-expression based pattern matching, using the keyword `matches`. We use *taint-annotated* regular expressions defined as follows. A tainted regular expression is obtained for a normal regular expression by attaching a superscript t, T or u. A string s will match a taint-annotated regular expression r^t provided that s matches r, and at least one of the characters in s is tainted. Similarly, s will match r^T provided *all* characters in s are tainted. Finally, s will match r^u provided none of the characters in s are tainted.

Attack Type	**Policy**	**Comment**
Control-flow hijack	$\mathbf{jmp}(addr)$ | $addr$ **matches** $(any+)^t \rightarrow term()$	Tainted values cannot be used as a target of control transfer
Format string	Format = `"%[^%]"` vfprintf(fmt) | fmt **matches** $any*$ (Format)$^T any* \rightarrow reject()$	Format directives (e.g.`%n`) should not be tainted
Directory traversal	DirTraversalModifier = `".."` file_function($path$) = $open(path,)$ || $unlink(path)$ || ... file_function($path$) | $path$ **matches** $any*$ (DirTraversalModifier)$^T any*$ && $escapeRootDir(path) \rightarrow reject()$	If $path$ contains tainted directory traversal strings (e.g. ".."), then the real path of $path$ should not go outside the top level directories that are allowed to be accessed by the program, e.g. DocumentRoot and cgi-bin for httpd
Cross-site scripting	ScriptTag = `"<script"` | ... html_print_function(str) | str **matches** (StrIdNum|Delim) $*$ (ScriptTag)$^T any*$ $\rightarrow reject()$	No tainted script tags (e.g. $script$) should be output to HTML.
SQL injection	SqlMetachar = `"'"` | `";"` | `"/*"` | ... sql_query_function($query$) | $query$ **matches** (StrIdNum|Delim) $*$ SqlMetachar)$^T any*$ $\rightarrow reject()$	SQL query string should not contain tainted SQL meta-chars
Shell command injection	ShellMetachar = `";"` | `"&&"` | ... shell_command_function(cmd) | cmd **matches** (StrIdNum|Delim) $*$ (ShellMetachar)$^T any*$ $\rightarrow reject()$	cmd argument of $system$ or $popen$ should not contain tainted shell meta-chars

Figure 9.1: Illustrative security policies

The predefined pattern any matches any single character. Parentheses and other standard regular expression operators are used in the usual way. Moreover, taint-annotated regular expressions can be named, and the name can be reused subsequently, e.g., `StrIdNum` used in many sample policy rules is defined as:

```
StrIdNum = String | Id | Num
```

where `String`, `Id` and `Num` denote named regular expressions that correspond respectively to strings, identifiers and numbers. Also, `Delim` denotes delimiters.

Table 9.1 shows the examples of a few simple policies to detect various attacks. The *action* component of these policies make use of two support functions: $term()$ terminates the program execution, while $reject()$ denies the request and returns with an error.

For the control-flow hijack policy, we use a special keyword `jmp` as a function name, as we need some special way to capture low-level control-flow transfers that are not exposed as a function call in the C language. The policy states that if any of the bytes in the target address are tainted, then the program should be terminated.

For format string attacks, we only define a policy for `vfprintf`, because `vfprintf` is the common function used internally to implement all other `printf` family of functions. All format directives in a format string begin with a "`%`", and are followed by a character other than "`%`". (The sequence "`%%`" will simply print a "`%`", and hence can be permitted in the format string.)

Example policies to detect four other attacks, namely, directory traversal, cross-site scripting, SQL injection and shell command injection are also shown in Table 9.1. The comments associated with the policies provide an intuitive description of the policy. These policies were able to detect all of the attacks considered in our evaluation.

9.3 Refining Taint-Enhanced Policies

The main contribution of our approach is to show the feasibility and practicality of using fine-grained taint information in developing policy-based attack detection. The availability of fine-grained taint information makes our policies significantly more precise than traditional access-control policies. Moreover, our approach empowers system administrators and security professionals to update and refine these policies to improve protection, *without* having to wait for the patches of a newly discovered attack avenue.

Having said the above, we note that policy development effort is an important concern with any policy enforcement technique. In particular, there is a trade-off between policy precision and the level of effort required. If one is willing to tolerate false positives, policies that produce very few false negatives can be developed with modest effort. Alternatively, if false negatives can be tolerated, then false positives can be kept to a minimum with little effort. To contain both false positives and false negatives, more effort needs to be spent on policy development, taking application-specific or installation-specific characteristics.

These remarks about policy-based techniques are generally applicable to our approach as well. Although the example policies shown in Table 9.1 were able to stop the attacks in our experiments, many of them need further improvement before they can stand up to skilled attackers that are knowledgeable about the policies being enforced. We outline the ways to improve these policies. We do not discuss the refinement of the control-flow hijack policy because it is already accurate enough to capture all attacks that use corruption of code pointers as the basis to alter the control-flow of programs.

9.3.1 Lexical Confinement Policies

The policy shown in Table 9.1 does not address attacks that inject only SQL keywords (e.g., the UNION operation) to alter the meaning of a query. This can be addressed by lexical confinement taint-enhanced policies. The idea is to perform a lexical analysis on the SQL query to break it up into tokens. SQL injection attacks are characterized by the fact that multiple tokens appear in the place of one, e.g., multiple keywords and meta-characters were provided by the attacker in the place of a simple string value in the attack examples discussed earlier. Thus, systematic protection from SQL injections can be obtained using a lexical confinement policy that prevents tainted strings from spanning multiple tokens. A similar approach is suggested in [70], although the conditions are not defined as precisely.

Command injection attacks are similar to SQL injection attacks in many ways, and hence a lexical confinement policy is a good choice for them as well. Though we should note that there are some differences between SQL and command injection, e.g., shell syntax is much more complex than SQL syntax. Moreover, we may want to restrict the command names so that they are not tainted.

Lexical confinement policies requires a lexical tokenization task that is (almost invariably) implemented using regular expression based specifications. Thus, these policies are amenable to expression using our policy language. One could argue that a regular expression to recognize tokens would be complex, and hence a policy may end up using a simpler approximation to tokenization. This discussion shows that the usual trade-off in policy based attack detection between accuracy and policy complexity continues in the case of taint-enhanced policies as well. Nevertheless, it should be noted that for a given policy development effort, taint-enhanced policies will be significantly more accurate than policies that do not incorporate any knowledge about taint.

9.3.2 Syntactic Confinement Policies

Su and Wassermann proposed *syntactic confinement* to characterize and detect SQL injections based on a syntactic analysis of SQL queries [81, 82]. The syntactic confinement approach also uses taint information to identify untrusted inputs for its policy enforcement. It constructs and examines the parse tree of each generated SQL query. The syntactic confinement policies require that the tainted input be syntactically isolated within the generated SQL query. In other words, a tainted input cannot straddle different syntactic subtrees.

Both syntactic confinement and lexical confinement policies can detect injection attacks that alter the syntactical structure of a command using the tainted data. However, syntactic confinement is more general than lexical confinement. It can allow benign queries or commands in which multiple consecutive arguments of a command come from user-provided inputs, while still prevent command names and command separators from being tainted. In most applications, such policies can stop SQL and command injection attacks while permitting more flexible uses of user input data in constructing the parameters of back-end operations (e.g. SQL queries and shell commands) [76].

The work of Su et al in [81] focused on providing a precise characterization of SQL injection attacks. Their methods require a full parser of each supported language (e.g. SQL, shell commands, HTML). Recently, Sekar [76] proposed a rough-parser based syntax-aware taint-enhanced policy framework. It is accomplished by defining a simple and generic AST structure as the intermediate representation. A rough-parser of each supported language can then be used to identify only the structure of syntactic elements most relevant to attack detection, such as command names, command parameters and separators. The rough-parser does not need to understand every detail of the language grammar and differences between language variations across different platforms and products. For example, an HTML parser does not need to distinguish between many HTML tags. This approach still retains most benefits of syntactic confinement policies, while significantly reducing the efforts required to develop such policies and their enforcement techniques.

The cross-site scripting policy in Table 9.1 does not address variations of the basic attack, e.g., attackers can evade this policy by injecting the malicious script code in "`onmouseover=malicious()`" or "`<img src="javascript:malicious()">`", which is not a block enclosed by the `script` tag. To detect these XSS variations, one can develop a syntactic confinement policy that understands the different HTML tag patterns in which a malicious script can be injected into dynamic HTML pages and prevent the use of such tainted patterns in HTML outputs.

9.3.3 Other Policy Refinements

The format string attack policy shown in Table 9.1 tends to err on the side of producing false positives, by disallowing all use of tainted format directives. However, it is conceivable that some applications may be prepared to receive a subset of format directives in untrusted inputs, and handle them correctly. In such cases, this application knowledge can be used by a system administrator to use a less restrictive policy, e.g., allowing the use of format directives other than `%n`. This should be done with care, or else it is possible to write policies that prevent the use of `%n`, but allow the use of variants such as `%5n` that have essentially the same effect. Alternatively, the policy may be relaxed to permit specific exceptions to the general rule that there be no format directives, e.g., the rule:

```
vfprintf($fmt$) |
    $fmt$ **matches** $any*$ (Format)$^T any*$   &&
      (!(fmt **matches** "[^%]*%s[^%]*")) $\rightarrow reject()$
```

allows the use of a single `%s` format directive from untrusted sources, in addition to permitting format strings that contain untainted format directives.

The directory traversal policy also tends to err on the side of false positives, since it precludes all accesses outside the authorized top level directories (e.g. DocumentRoot and cgi-bin) of a web server if components of the file name being accessed are coming from untrusted sources. In devising this policy, we relied on application-specific knowledge, namely, the fact that web servers do not allow clients to access files outside the top level directories specified in the server configuration file. Another point to be noted about this policy is that variants of directory traversal attack that do not escape these top level directories, but simply attempt to fool per-directory access controls, are not addressed by our policy.

Part III

Experimental Evaluation

Chapter 10

Experiments: Memory Safety Enforcement

10.1 Implementation

We have implemented the shadow metadata structure based transformation as described in Chapter 4. In this section, we present its experimental results in Section 10.2. We leave the implementation and evaluation of the shadow metadata memory based transformation for our future work, though we expect it would have similar effectiveness and performance results.

Our approach is implemented as a syntax-directed source-source transformation tool to instrument C programs. It uses CIL [57] as the front end and Objective Caml as the implementation language. C is a very complex language to deal with for program analysis and transformation. Our transformation work was considerably simplified by the intermediate language provided by CIL which consists of a clean subset of C constructs that can be easily manipulated and then emitted as C source code.

To evaluate the performance of our implementation, we applied our transformation on a number of test programs from `Olden` [22], `SPECINT` [4, 3] benchmarks and UNIX utility programs. We used the `Olden` benchmarks included in the CCured package because the original benchmarks do not run on Linux. Our implementation is able to process many moderate-sized programs — five of our test programs are over 10K lines of code.

Some of our benchmarks made use of user-defined memory allocation functions that are functionally different from `malloc`. We have modified these programs to replace the calls to these functions with standard memory allocation functions such as `malloc` and `calloc`. No other source code modifications were required by our source code transformer.

10.2 Evaluation

10.2.1 Effectiveness

In our experiments, our implementation successfully detected several memory error bugs which have been reported in previous research [62] in the SPECINT benchmarks: `compress` contains an array out-of-boundary error, and `go` contains several array out-of-boundary errors. In addition, our implementation detected an unreported array out-of-boundary bug in the UNIX utility program `patch`. We have also written a number of test programs that contains both temporal and spatial memory errors. Our implementation was able to detect all the memory errors in these test programs at runtime. All of these provide evidences to show that our implementation indeed works to identify memory access errors.

10.2.2 Performance

We compared the performance of original programs and transformed programs. Both of them were compiled using `gcc` version 3.2.2 with `-O2` optimization, and executed on an Intel Xeon 3.06GHz workstation with 3GB RAM running Red Hat Linux 9. All the execution times were measured using the total of system time and user time reported by

Program	Lines of code	Execution time Orig. (sec.)	Execution time Trans. Ratio	Peak heap usage Orig. (MB)	Peak heap usage Trans. Ratio	Executable size Orig. (KB)	Executable size Trans. Ratio
Olden							
bh	2080	1.37	2.82	0.17	4.76	81.7	1.97
bisort	684	0.66	1.76	1.57	5.51	30.6	1.43
em3d	561	2.77	1.79	6.53	3.13	44.7	1.14
health	709	1.08	2.72	2.33	5.45	50.9	1.21
mst	592	2.43	1.76	71.34	3.36	47.8	1.10
perimeter	395	1.49	3.37	2.74	4.72	29.6	1.04
power	763	1.81	1.22	0.42	3.12	43.8	1.54
treeadd	370	0.26	3.23	25.17	5.34	34.2	0.94
tsp	565	1.23	2.28	9.44	3.36	33.8	1.41
AVG			**2.33**		**4.31**		**1.31**
SPECINT							
099.go	29262	19.29	2.60	0.00	1.00	437.8	2.51
129.compress	1939	2.69	1.85	0.00	1.00	103.6	1.17
164.gzip	8620	1.21	1.46	6.61	1.08	166.5	1.34
175.vpr	17897	0.93	3.53	0.91	1.86	408.2	3.23
181.mcf	2413	0.20	2.85	96.59	3.01	129.5	1.10
AVG			**2.46**		**1.59**		**1.87**
Utilities							
bc-1.06	14264	2.59	2.69	0.09	2.67	191.1	2.82
gzip-1.2.4	8163	1.59	1.54	0.00	1.00	168.4	1.78
patch-2.5.4	11533	0.52	1.12	10.20	1.40	234.3	2.91
tar-1.12	24339	1.01	1.14	0.04	1.25	467.1	1.98
AVG			**1.62**		**1.58**		**2.37**

Figure 10.1: Transformation performance for Olden/SPECINT benchmarks and UNIX utilities. A ratio 1.22 means that the transformed program was 22% slower/larger than the original.

`time`. Figure 10.1 shows the overall performance of our transformation.

Execution overhead The third and fourth columns of Figure 10.1 are the comparison of execution time of original and transformed programs. For the `Olden` benchmarks, the transformed programs have a slowdown of 1.22 to 3.37, with an average of 2.33; for the `SPECINT` benchmarks, the slowdown ranges from 1.46 to 3.53, with an average of 2.46; for the UNIX utility programs, the range of slowdown is between 1.12 and 2.69, and the

Program	Transformed (ratio in execution time)		
	Spatial	Spatial & Temporal	Spatial & Temporal & RTTI
Olden			
bh	1.88	2.60	2.82
bisort	1.45	1.61	1.76
em3d	1.36	1.83	1.79
health	1.97	2.47	2.72
mst	1.32	1.59	1.76
perimeter	1.82	2.11	3.37
power	1.17	1.20	1.22
treeadd	1.81	2.73	3.23
tsp	1.81	1.88	2.28
AVG	*1.62*	*2.00*	*2.33*
SPECINT			
099.go	2.44	2.56	2.60
129.compress	1.37	1.65	1.85
164.gzip	1.15	1.33	1.46
175.vpr	2.05	2.88	3.53
181.mcf	2.50	2.85	2.85
AVG	*1.90*	*2.25*	*2.46*
Utilities			
bc-1.06	1.96	2.49	2.69
gzip-1.2.4	1.46	1.47	1.54
patch-2.5.4	1.06	1.06	1.12
tar-1.12	1.05	1.19	1.14
AVG	*1.38*	*1.55*	*1.62*
AVG	**1.65**	**1.97**	**2.21**

Figure 10.2: Performance overheads of checking different memory errors

average is 1.62.

Memory overhead The fifth and sixth columns of Figure 10.1 present the peak heap memory usage of both original and transformed programs. The main source of overhead in transformed programs is the space for storing `info` structures for the pointers that would be stored in each `malloc`'d block. Since there are four fields in the `info` structure associated with a pointer, the increase in memory usage can be 5 times in the worst case. Many `Olden` benchmarks use `malloc` primarily to allocate structures that contain mostly pointers. Thus, the memory overhead in these programs is close to this worst case possibility. For some programs, the increase exceeds a factor of 5 since there are other sources for memory overhead that also need to be considered. These include a fixed-length `blkheader` for each heap memory allocation, and the memory used for HCS and SCS.

Most of the `SPECINT` benchmarks and the UNIX utilities we have tested do not store many pointers in heap, hence they do not increase heap memory usage significantly.

Analysis of performance We first studied the performance overheads due to different kinds of checks inserted by our transformation: (a) checks for detecting spatial memory errors, (b) checks for detecting temporal memory errors, and (c) checks for verifying upcasts and downcasts using run-time type information. Columns 2-4 of Figure 10.2 show the normalized execution time for (a), (a)+(b), and (a)+(b)+(c) as compared to the original execution time of each program. The average execution time overheads for the three combinations are 65%, 96% and 121% for all the test programs.

We then studied the effectiveness of the optimizations. Previous research efforts on detecting both spatial and temporal errors have reported much higher runtime overheads than what is shown in Figure 10.1. Typical slowdowns are by about a factor of 4. In order to understand the source of performance improvement in our approach, we measured the execution time of our approach with each of the optimizations discussed in Section 4.5.

The second column in Figure 10.3 shows the slowdown when we combined the `header` and `info` metadata together. Note that the slowdown has now increased to about a factor of 4, which is consistent with the results reported by previous researchers. Column 3 of this table demonstrates that significant improvement in performance is obtained as a result of splitting the metadata into `header` and `info` variables. By eliminating unnecessary SCS operations, further significant decrease in overheads is obtained, as shown in the fourth column of the table. Finally, by converting the `info` structures into individual variables, we enable `gcc` to aggressively perform other optimizations. This factor leads to a further modest decrease in overhead. Together, these optimizations reduce the runtime overhead by about a factor of two.

The effectiveness of each optimization varies depending on the characteristics of the test programs. Programs that make a large number of function calls (e.g. perimeter) can benefit more from unnecessary SCS operation eliminations than programs that make fewer function calls (e.g. `health`). Programs that have a higher ratio of the number of pointer assignments over the number of pointer dereferences (e.g. `compress`) can have more performance improvement due to the separation of `header` and `info`.

Performance comparison with related approaches Since the primary goal of our work is to detect both spatial and temporal errors, we limit our detailed comparison in this section to previous techniques that are capable of detecting both types of errors. It should be noted, however, that the comparison results should be interpreted carefully, since the set of benchmarks used in the related works are different from those used by us.

Patil and Fischer's shadow processing [68] is similar to our basic transformation, but has no support for downcasts. The approach checks all the temporal and spatial memory errors. Its overheads for a subset of the `SPECINT` benchmark programs is on an average of 538%, which is much higher than our result of 121%. The closely related approach Safe-C [13] also reports overheads over 300%.

The bounds-checking work of Jones and Kelly [49] provides better compatibility than our approach with untransformed external libraries. In particular, they do not require wrapper functions to be developed for such libraries, even if they return dynamically allocated data structures. On the downside, their approach incurs higher overheads, and moreover, does not detect dangling pointers to reallocated memory. They report a slowdown by a factor of 5 to 6 for most programs. Their work was initially distributed as an extension to `gcc 2.7`, and further work has continued on this approach, yielding `gcc` patches for

Program	Transformed (ratio in execution time)			
	H/I merged	H/I sep.	H/I sep. SCS elim.	H/I sep. SCS elim. Info split
Olden				
bh	5.24	4.18	3.92	2.82
bisort	2.61	2.17	1.76	1.76
em3d	1.92	1.83	1.81	1.79
health	3.07	2.79	2.72	2.72
mst	2.28	2.06	1.77	1.76
perimeter	7.89	6.15	3.51	3.37
power	1.71	1.66	1.22	1.22
treeadd	4.92	4.12	3.96	3.23
tsp	4.44	3.06	2.28	2.28
AVG	*3.79*	*3.11*	*2.55*	*2.33*
SPECINT				
099.go	3.46	3.06	2.65	2.60
129.compress	5.32	3.82	2.11	1.85
164.gzip	3.03	2.38	1.54	1.46
175.vpr	4.49	3.99	3.67	3.53
181.mcf	5.60	3.90	2.90	2.85
AVG	*4.38*	*3.43*	*2.57*	*2.46*
Utilities				
bc-1.06	9.60	3.74	3.51	2.69
gzip-1.2.4	2.44	2.06	1.92	1.54
patch-2.5.4	1.27	1.13	1.12	1.12
tar-1.12	1.28	1.23	1.12	1.14
AVG	*3.65*	*2.04*	*1.92*	*1.62*
AVG	**3.92**	**2.96**	**2.42**	**2.21**

Figure 10.3: Effectiveness of optimizations

subsequent versions of `gcc`. Most recent performance data for this method can be found in Ruwase and Lam [73], where they present an improvement to the original technique which has better compatibility with existing C programs. They report performance overheads that are virtually identical to that of the bounds-checking extension to `gcc` version 3.3.1. Their overheads for the common test programs such as `gzip` are about twice as much compared to our approach. In terms of worst case performance, they report slowdowns by a factor of more than 10, whereas the worst case slowdown reported for our work in Figure 10.1 is by a factor less than 4.

Yong and Horwitz [93] describe an approach which is similar to Purify [44] but is enhanced using global static analysis. They report an average overhead of 37% on the Olden benchmark. This is better than our result of 133%, but this improvement is obtained in their approach by checking pointer errors to write operations alone. Read operations, which far outnumber writes in typical programs, are not checked.

Chapter 11

Experiments: Taint-Enhanced Policy Enforcement

11.1 Implementation

We have implemented the program transformation technique described in Chapter 8. The transformer consists of about 3,600 lines of Objective Caml code and uses the CIL [57] toolkit as the front end to manipulate C constructs. Our implementation currently handles `glibc` (containing around 1 million LOC) and several other medium to large applications. The complexity and size of `glibc` demonstrated that our implementation can handle "real-world" code. We summarize some of the key issues involved in our implementation.

11.1.1 Coping with Untransformed Libraries

Ideally, all the libraries used by an application will be transformed using our technique so as to enable accurate taint tracking. In practice, however, source code may not be available for some libraries, or in rare cases, some functions in a library may be implemented in an assembly language. One option with such libraries is to do nothing at all. Our implementation is designed to work in these cases, but clearly, the ability to track information flow via untransformed functions is lost. To overcome this problem, our implementation offers two features. First, it produces warnings when a certain function could not be transformed. This ensures that inaccuracies will not be introduced into taint tracking without explicit knowledge of the user. When the user sees this warning, she may decide that the function in question performs largely "read" operations, or will never handle tainted data, and hence the warning can safely be ignored. If not, then our implementation supports *summarization functions* that specify how taint information is propagated by a function. For instance, we use the following summarization function for the `memcpy`. Summarization functions can use support functions to copy taint information. A summarization function for f would be invoked in the transformed code when f returns.

```
memcpy(dest, src, n) →
    copy_buffer_tagmap(dest, src, n);
```

So far, we had to write summarization functions for two `glibc` functions that are written in assembly and copy data, namely, `memcpy` and `memset`. In addition, `gcc` replaces calls to some functions such as `strcpy` and `strdup` with its own code, necessitating an additional 13 summarization functions.

11.1.2 Injecting Marking and Checking Code

In our current implementation, the marking specifications, security policies, and summarization code associated with a function f are all injected into the transformed program by simply inlining (or explicitly calling) the relevant code before or after the call to f. In the future, we anticipate these code to be decoupled from the transformation, and be able to operate on binaries using techniques such as library interposition. This would enable

CVE#	**Program**	**Lang.**	**Attack type**	**Attack description**
CAN-2003-0201	samba	C	Stack smashing	Buffer overflow in call_trans2open function
CVE-2000-0573	wu-ftpd	C	Format string	via SITE EXEC command
CAN-2005-1365	pico server	C	Directory traversal	Command execution via URL with multiple leading "/" characters and ".."
CAN-2003-0486	phpBB 2.0.5	PHP	SQL injection	via topic_id parameter
CAN-2005-0258	phpBB 2.0.5	PHP	Directory traversal	Delete arbitrary file via ".." sequences in avatarselect parameter
CAN-2002-1341	SquirrelMail 1.2.10	PHP	Cross site scripting	Insert script via the mailbox parameter in read_body.php
CAN-2003-0990	SquirrelMail 1.4.0	PHP	Command injection	via meta-char in"To:" field
CAN-2005-1921	PHP XML-RPC	PHP	Command injection	Eval injection
CVE-1999-0045	nph-test-cgi	BASH	Shell meta-char expansion	using '*' in `$QUERY_STRING`

Figure 11.1: Attacks used in effectiveness evaluation

a site administrator to alter, refine or customize her notions of "trustworthy input" and "dangerous arguments" without having access to the source code.

11.2 Evaluation

The main goal of our experiments was to evaluate attack detection (Section 11.2.1), and runtime performance (Section 11.2.4). False positives and false negatives are discussed in Sections 11.2.2 and 11.2.3.

11.2.1 Attack Detection

Table 11.1 shows the attacks used in our experiments. These attacks were chosen to cover the range of attack categories we have discussed, and to span multiple programming languages. Wherever possible, we selected attacks on widely-used applications, since it is likely that obvious security vulnerabilities in such applications would have been fixed, and hence we are more likely to detect more complex attacks.

In terms of marking, all inputs read from network (using `read`, `recv` and `recvfrom`) were marked as tainted. Since the PHP interpreter is configured as a module for Apache, the same technique works for PHP applications as well. Network data is tainted when it is read by Apache, and this information propagates through the PHP interpreter, and in effect, through the PHP application as well. The policies used in our attack examples were already discussed in Section 9.2.

To test our technique, we first downloaded the software packages shown in Table 11.1. We downloaded the exploit code for the attacks, and verified that they worked as expected. Then we used transformed C programs and interpreters with policy checking enabled, and verified that each one of the attacks were prevented by these policies without raising false alarms.

Network Servers in C.

- `wu-ftpd` versions 2.6.0 and lower have a format string vulnerability in `SITE EXEC` command that allows arbitrary code execution. The attack is stopped by the policy that the format directive `%n` in a format string should not be tainted.

- `samba` versions 2.2.8 and lower have a stack-smashing vulnerability in processing a type of request called “transaction 2 open.” No policy is required to stop this attack — the stack-smashing step ends up corrupting some data on the shadow stack rather than the main stack, so the attack fails.

  If we had used an attack that uses a heap overflow to overwrite a GOT entry (which is common with heap overflows), this too would be detected without the need for any policies due to the technique described in Section 8.5 for preventing the GOT from being directly accessed by the C code. The reasoning is that before the injected code gets control, the GOT entry has to be clobbered by the existing code in the program. The instrumentation in the clobbering code will cause a segmentation fault because of the protection of the GOT, and hence the attack will be prevented. Note that the GOT is normally used by the PLT (Procedure Linkage Table) code that is in the assembly code automatically added by the compiler, and is not in the C source code, so a normal GOT access will not be instrumented with checks on taint tags, and hence will not lead to a memory fault.

If the attack corrupted some other function pointer, then the "jmp" policy would detect the use of tainted data in jump target and stop the attack.

- `Pico HTTP Server (pServ)` versions 3.2 and lower have a directory traversal vulnerability. The web server does include checks for the presence of "`..`" in the file name, but allows them as long as their use does not go outside the `cgi-bin` directory. To determine this, `pServ` scans the file name left-to-right, decrementing the count for each occurrence of "`..`", and incrementing it for each occurrence of "`/`" character. If the counter goes to zero, then access is disallowed. Unfortunately, a file name such as `/cgi-bin////../../bin/sh` satisfies this check, but has the effect of going outside the `/cgi-bin` directory. This attack is stopped by the directory traversal policy shown in Section 9.2.

Web Applications in PHP.

- `phpBB2` *SQL injection* vulnerability in (version 2.0.5 of) `phpBB`, a popular electronic bulletin board application, allows an attacker to steal the MD5 password hash of another user. The vulnerable code is:

```
$sql="SELECT p.post_id FROM ... WHERE ... AND p.topic_id =
            $topic_id AND ..."
```

Normally, the user-supplied value for the variable `topic_id` should be a number, and in that case, the above query works as expected. Suppose that the attacker provides the following value:

```
-1 UNION SELECT ord(substring(user_password,5,1)) FROM phpbb_users
                         WHERE userid=3/*
```

This converts the SQL query into a union of two `SELECT` statements, and comments out (using "`/*`") the remaining part of the original query. The first `SELECT` returns an empty set since `topic_id` is set to `-1`. As a result, the query result equals the value of the `SELECT` statement injected by the attacker, which returns the 5th byte in the MD5 hash of the bulletin board user with the userid of 3. By repeating this attack with different values for the second parameter of `substring`, the attacker can obtain the entire MD5 password hash of another user. The SQL injection policy described in Section 9.2 stops this attack.

- `SquirrelMail` *cross-site scripting* is present in version 1.2.10 of `SquirrelMail`, a popular web-based email client, e.g., `read_body.php` directly outputs values of user-controlled variables such as `mailbox` while generating HTML pages. The attack is stopped by the cross-site scripting policy in Section 9.2.

- `SquirrelMail` *command injection:* `SquirrelMail` (Version 1.4.0) constructs a command for encrypting email using the following statement in the function `gpg_encrypt` in the GPG plugin 1.1.

```
$command .= " -r $send_to_list 2>&1";
```

The variable `send_to_list` should contain the recipient name in the "`To`" field, which is extracted using the `parseAddress` function of `Rfc822Header` object in

`SquirrelMail`. However, due to a bug in this function, some malformed entries in the "`To`" field are returned without checking for proper email format. In particular, by entering "⟨recipient⟩; ⟨cmd⟩;" into this field, the attacker can execute any arbitrary command ⟨cmd⟩ with the privilege of the web server. By applying a policy that prohibits tainted shell meta-characters in the first argument to the `popen` function, this attack is stopped by our technique.

- `phpBB` *directory traversal:* A vulnerability exists in `phpBB`, which, when the gallery avatar feature is enabled, allows remote attackers to delete arbitrary files using directory traversal. This vulnerability can be exploited by a two-step attack. In the first step, the attacker saves the file name, which contains "`..`" characters, into the SQL database. In the second step, the file name is retrieved from the database and used in a command. To detect this attack, it is necessary to record taint information for data stored in the database, which is quite involved. We took a shortcut, and marked all data retrieved from the database as tainted. (Alternatively, we could have marked only those fields updated by the user as tainted.) This enabled the attack to be detected using the directory traversal policy.

- `phpxmlrpc/expat` *command injection:* `phpxmlrpc` is a package written in PHP to support the implementation of PHP clients and servers that communicate using the XML-RPC protocol. It uses the `expat` XML parser for processing XML. `phpxmlrpc` versions 1.0 and earlier have a remote command injection vulnerability. Our command injection policy stops exploitations of this vulnerability.

Bash CGI Application. `nph-test-cgi` is a CGI script that was included by default with Apache web server versions 1.0.5 and earlier. It prints out the values of the environment variables available to a CGI script. It uses the code `echo QUERY_STRING=$QUERY_STRING` to print the value of the query string sent to it. If

Server Programs	**Workload**	**Orig. Response Time**	**Overhead**
Apache-2.0.40	Webstone 30 clients downloading 5KB pages over 100Mbps network	0.036 sec/page	6%
wu-ftpd-2.6.0	Download a 12MB file 10 times.	11.5 sec	3%
postfix-1.1.12	Send one thousand 3KB emails	0.03 sec/mail	7%

Figure 11.2: Performance overheads of servers. For Apache server, performance is measured in terms of latency and throughput degradation. For other programs, it is measured in terms of overhead in client response time.

Program	**Workload**	**Over-head(A)**	**Over-head(B)**	**Over-head(C)**	**Over-head(D)**
bc-1.06	Find factorial of 600.	212%	68%	61%	**61%**
enscript-1.6.4	Convert a 5.5MB text file into PS.	660%	529%	63%	**58%**
bison-1.35	Parse a Bison file for C++ syntax.	134%	92%	79%	**78%**
gzip-1.3.3	Compress a 12 MB file.	228%	161%	110%	**106%**

Figure 11.3: Performance overheads of CPU-intensive programs. Performance is measured in terms of CPU time. Overheads in different columns correspond to: (A) No optimizations, (B) Use of local tag variable, (C) B + Use of 2-bit taint value, (D) C + Use of dependency analysis.

the query string contains a "*" then `bash` will apply file name expansion to it, thus enabling an attacker to list any directory on the web server. This attack was stopped by a policy that restricted the use of tainted meta-characters in the argument to `shell_glob_filename`, which is the function used by `bash` for file name expansion. In terms of marking, the CGI interface defines the exact set of environment variables through which inputs are provided to a CGI application, and all these are marked as tainted.

11.2.2 False Positives

The policies described so far have been designed with the goal of avoiding false positives. We experimentally verified that false positives did not occur in our experiments involving the `wu-ftpd` server, the Apache web server, and the two PHP applications, `phpBB` and `SquirrelMail`. For `wu-ftpd` and Apache, we enabled the control flow hijack policy, format string policy, directory traversal policy, and shell command injection policy. For the PHP applications, we additionally enabled the SQL injection policy and cross-site scripting policy for the PHP interpreter.

To evaluate the false positives for Apache, we used the transformed server as our lab's regular web server that accepted real-world HTTP requests from Internet for several hours. For the `wu-ftpd` server, we ran all the supported commands from a ftp client. To test `phpBB` and `SquirrelMail`, we went through all the menu items of these two Web applications, performed normal operations that a regular user might do, such as registering a user, posting a message, searching a message, managing address book, moving messages between different mail folders, and so on. No false positives were observed in these experiments.

11.2.3 False Negatives

False negatives can arise due to (a) overly permissive policies, (b) implicit information flows, and (c) use of untransformed libraries without adequate summarization functions.

We discuss the implicit flows in Section 8.3 and policy refinement in in Section 9.3. As for external libraries, the best approach is to transform them, so that the need for summarization can be eliminated. If this cannot be done, then our transformation will identify all the external functions that are used by an application, so that errors of omission can be avoided. However, if a summarization function is incorrect, then it can lead to false negatives, false positives, or both.

11.2.4 Performance

Table 11.2 and 11.3 show the performance overheads, when the original and transformed programs were compiled using `gcc 3.2.2` with `-O2`, and ran on a 1.7GHz/512MB/Red Hat Linux 9.0 PC.

For server programs, the overhead of our approach is low. This is because they are I/O intensive, whereas our transformation adds overheads only to code that performs significant amount of data copying within the program, and/or other CPU-intensive operations. For CPU-intensive C programs, the overhead is between 61% to 106%, with an average of 76%.

Effect of Optimizations. The optimizations discussed in Section 8.4 have been very effective. We comment further in the context of CPU-intensive benchmarks.

- *Use of local taint variables* reduced the overheads by 42% to 144%. This is due to the reasons mentioned earlier: compilers such as `gcc` are very good in optimizing operations on local variables, but do a poor job on global arrays. Thus, by replacing global `tagmap` accesses with local tag variable accesses, significant performance improvement can be obtained.

  Most programs access local variables much more frequently than global variables. For instance, we found out (by instrumenting the code) that 99% of accesses made by `bc` are to local variables. A figure of 90% is not at all uncommon. As a result, the introduction of local tag variables leads to dramatic performance improvement for such programs. For programs that access global variables frequently, such as `gzip` that has 41% of its accesses going to global variables, the performance improvements are less striking.

- *tagmap optimizations* are particularly effective for programs that operate mainly on integer data. This is because of the use of 2-bit taint tags, which avoids the need for bit-masking and shifts to access taint information. As a result we see significant overhead reduction in the range of 7% to 466%.

- *Intra-procedural analysis* and optimization further reduces the overhead by up to 5%. The gains are modest because `gcc` optimizations have already eliminated most local tag variables after the previous step.

When combined, these optimizations reduce the overhead by a factor of 2 to 5.

Explanation of the final performance numbers. Given that our transformation introduces a taint-tag assignment for each original assignment in the program, and that `tagmap` accesses are generally more complex (involving multiple bit-masking operations) and are less likely to be optimized, one might expect that overheads should be 100% or higher. There are two reasons why the overheads are low. For server applications, they are mainly I/O-bound. Moreover, they spend a significant fraction of time within the OS, and our transformation does not add overheads to the system time. Finally, since our optimizations are particularly effective in eliminating taint-tracking for local variables, the performance overheads are likely to be low for programs that mainly access local variables. We instrumented the applications shown in the tables above, and found that often, over 90% of accesses are to local variables. For instance, `bc` has 99% of its accesses to local variables. `gzip` is an exception, with 41%of its accesses going to global variables, and hence it has a higher performance overhead.

Chapter 12

Conclusions

12.1 Dissertation Summary

Software vulnerabilities are a wide spread problem and have been the biggest culprit behind today's cyber attacks. It has been shown from the history that security vulnerabilities are hard to eliminate from software systems. In this dissertation, we present source-code transformation based techniques to automatically detect at runtime the exploits of the two most important classes of software vulnerabilities: memory error vulnerabilities and injection vulnerabilities due to input validation errors.

The presented source-code transformation for addressing attacks that exploit memory error vulnerabilities features a complete coverage of spatial and temporal memory errors at the memory block level. It works as a comprehensive approach that can stop all the attacks that exploit these memory errors. Our technique offers backwards compatibility with existing C programs. It changes neither the layout of data structures in C programs, nor the explicit memory management model in C. In addition, it supports type casts and pointer arithmetic that are pervasive in C programs.

The presented taint-enhanced policy enforcement framework is based on an efficient fine-grained dynamic taint tracking, which is enabled by instrumenting C programs with an automated source-to-source transformation. This framework offers the automated detection of injection attacks that exploit input validation errors, such as format string attacks, SQL injection, command injection, cross-site scripting, and path traversal.

We believe that the injection vulnerabilities arise due to the fact that security checks are interspersed throughout the program, and it is often difficult to check if the correct set of checks are being performed on every program path, especially in complex programs where the control flows through many, many functions. By decoupling policies from application logic, our approach can provide a higher degree of assurance on the correctness of policies. Moreover, the flexibility of our approach allows site administrators and third parties to quickly develop policies to prevent new classes of attacks, without having to wait for patches.

We have established the effectiveness and performance of these approaches through experimental evaluations on real-world applications.

12.2 Future Research Directions

12.2.1 Extending Memory Safety Enforcement

Performance optimization is one of the most important future research areas for extending our existing memory error detection approaches. There are two possible directions for performance optimization. First, we can explore techniques to dramatically reduce the metadata size and thus speed up the metadata lookup and update operations. To achieve this, we may need to constrain the memory allocation sizes, the locations and properties of spatial and temporal metadata, while still retaining the memory error coverage and backwards-compatibility. Secondly, we can explore static analysis techniques, such as those developed in CCured, to eliminate or minimize unnecessary spatial and temporal metadata maintenance for statically known safe pointers.

We would also like to investigate techniques that can improve the memory error detection coverage when interfacing with uninstrumented libraries. These techniques need to recover as much as possible the metadata information of pointers passed between instrumented functions and unmodified library functions.

12.2.2 Extending Taint-Enhanced Policy Enforcement

Policy development is a key research area for taint-enhanced policy enforcement. To be able to more effectively detect injection attacks, we need taint-enhanced policies that can more accurately characterize the intended usage of the untrusted input data. We can enrich the policy specification language and develop more application-tailored policies.

It is also important to improve the fine-grained dynamic taint tracking techniques. We currently only support taint tracking within a single program. However, the untrusted inputs might be propagated across multiple components from different applications before the use in a security-sensitive operation, where the taint-enhanced policy enforcement takes place. In such cases, we need to have the ability to propagate and track the taint information across different components.

Implicit flows and quantitative taint tracking is another interesting problem, in which the target is not fully tainted by the source. Instead, we should define the taint quantity of the taint propagation (i.e. to what extent the target is tainted by the source). The taint-enhanced policies should also take into account the amount of tainting.

12.2.3 Supporting Multi-Threaded Programs

To extend the presented transformations to support multi-threaded programs, we need to solve two essential problems: synchronizing access to the metadata of global program variables and synchronizing access to the shared global metadata.

To synchronize access to the metadata of a global program variable, we may be able to leverage the existing program locking that protects the access to the global program variable to guard the access to its metadata as well.

To synchronize access to the shared global metadata (e.g. `tagmap`, temporal capability store), we need to introduce our own locks. Care needs to be taken to avoid common concurrency issues such as deadlocks and race conditions.

We can use thread-local storage to avoid synchronizing the access to the global buffers for passing the metadata of function arguments.

Bibliography

[1] CVE - Common Vulnerabilities and Exposures. http://cve.mitre.org.

[2] Microsoft security bulletin. http://www.microsoft.com/technet/security/bulletin.

[3] SPEC CINT2000 benchmarks. Standard Performance Evaluation Corporation. http://www.spec.org/cpu2000/CINT2000/.

[4] SPEC CINT95 benchmarks. Standard Performance Evaluation Corporation. http://www.spec.org/cpu95/CINT95/.

[5] US-CERT vulnerability notes database. http://www.kb.cert.org/vuls/.

[6] Code Red worm. http://www.cert.org/advisories/CA-2001-19.html, 2001.

[7] Gates: Security is top priority. http://news.cnet.com/2100-1001-816880.html, January 2002.

[8] SQL Slammer worm. http://www.cert.org/advisories/CA-2003-04.html, 2003.

[9] W32/Blaster worm. http://www.cert.org/advisories/CA-2003-20.html, 2003.

[10] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *IEEE Symposium on Security and Privacy*, 2008.

[11] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, 2009.

[12] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. In *IEEE Symposium on Security and Privacy*, 2005.

[13] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 290–301, June 1994.

[14] K. Avijit, P. Gupta, and D. Gupta. TIED, LibsafePlus: Tools for runtime buffer overflow protection. In *USENIX Security Symposium*, pages 45–55, 2004.

[15] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX Annual Technical Conference*, pages 251–262, Berkeley, CA, June 2000.

[16] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[17] E. D. Berger and B. G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[18] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, Washington, DC, August 2003.

[19] S. Bhatkar, D. C. DuVarney, and R. Sekar. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, August 2005.

[20] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. In *Software - Practice and Experience*, pages 807–820, 1988.

[21] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *International Conference on Applied Cryptography and Network Security (ACNS)*, pages 292–302, 2004.

[22] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 29–38, Santa Barbara, CA, USA, 1995. ACM Press.

[23] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[24] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2005.

[25] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, 2005.

[26] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *International Conference on Distributed Computing Systems (ICDCS)*, April 2001.

[27] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 232–244, June 2003.

[28] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium*, 2001.

[29] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium*, Washington, D.C., August 2003.

[30] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, pages 63–78, San Antonio, Texas, January 1998.

[31] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[32] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for c with very low overhead. In *ACM International Conference on Software Engineering (ICSE)*, 2006.

[33] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2006.

[34] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in c. In *Proceedings of the 2003 ACM SIGPLAN Conference on Prgramming Language Design and Implementation (PLDI)*, San Diego, CA, USA, June 2003. ACM Press.

[35] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. Published on World-Wide Web, June 2000.

[36] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/main.html, June 2000.

[37] J. S. Fenton. Memoryless subsystems. *Computing Journal*, 17(2):143–147, May 1974.

[38] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, GA, May 1999.

[39] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *ACM Conference on Computer and Communication Security (CCS)*, pages 345–354, Washington D.C., USA, 2003.

[40] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, 1982.

[41] M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. 2009.

[42] W. Halfond and A. Orso. AMNESIA: Analysis and monitoring for neutralizing SQL-injection. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2005.

[43] N. Hardy. The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, October 1988.

[44] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, 1992.

[45] E. Haugh and M. Bishop. Testing C programs for buffer overflow vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, February 2003.

[46] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *International World Wide Web Conference*, 2004.

[47] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, June 2002.

[48] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, 2004.

[49] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *International Workshop on Automated and Algorithmic Debugging*, pages 13–26, 1997.

[50] S. Kaufer, R. Lopez, and S. Pratap. Saber-C: an interpreter-based programming environment for the C language. In *Proceedings of the Summer USENIX Conference*, pages 161–171, 1988.

[51] S. C. Kendall. Bcc: run–time checking for c programs. In *Proceedings of the USENIX Summer Conference*, El. Cerrito, California, USA, 1983. USENIX Association.

[52] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, pages 177–190, 2001.

[53] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, 2005.

[54] A. Loginov, S. H. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. In *Fundamental Approaches to Software Engineering*, 2001.

[55] M. T. Louw and V. Venkatakrishnan. Blueprint: Precise browser-neutral prevention of cross-site scripting attacks. In *IEEE Symposium on Security and Privacy*, 2009.

[56] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *IEEE Symposium on Security and Privacy*, pages 79–93, May 1994.

[57] S. McPeak, G. C. Necula, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for C program analysis and transformation. In *Conference on Compiler Construction*, 2002.

[58] B. P. Miller, L. Fredriksen, and B. So. An empirical study of reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[59] A. C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, Jan. 1999.

[60] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Network and Distributed System Security Symposium (NDSS)*, 2009.

[61] S. Nagarakatte, J. Zhao, M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[62] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 128–139, January 2002.

[63] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Workshop on Runtime Verification (RV)*, Boulder, Colorado, USA, July 2003.

[64] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[65] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, 2005.

[66] N. I. of Standards and Technology. Software errors cost u.s. economy $59.5 billion annually. http://www.nist.gov/public_affairs/releases/n02-10.htm, 2002.

[67] OWASP. Top 10 web application vulnerabilities. http://www.owasp.org/index.php/Top_10_2007, 2007.

[68] H. Patil and C. N. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Software - Practice and Experience*, 27(1):87–110, 1997.

[69] H. G. Patil and C. N. Fischer. Efficient run-time monitoring using shadow processing. In *International Workshop on Automated and Algorithmic Debugging*, 1995.

[70] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection (RAID)*, 2005.

[71] M. Rinard, C. Cadar, D. Dumitran, D. Roy, and T. Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors. In *Annual Computer Security Applications Conference (ACSAC)*, December 2004.

[72] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 182–195. ACM Press, 2000.

[73] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Network and Distributed System Security Symposium (NDSS)*, pages 159–169, February 2004.

[74] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1), Jan. 2003.

[75] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *International Symposium on Code Generation and Optimization (CGO)*.

[76] R. Sekar. An efficient black-box technique for defeating web application attacks. In *Network and Distributed System Security Symposium (NDSS)*, 2009.

[77] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communication Security (CCS)*, pages 298–307, 2004.

[78] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, August 2001.

[79] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. W. Reps. Coping with type casts in C. In *European Software Engineering Conference / ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 180–198, 1999.

[80] J. L. Steffen. Adding run-time checking to the portable c compiler. *Software - Practice and Experience*, 22(4):305–316, April 1992.

[81] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2006.

[82] Z. Su and G. Wassermann. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[83] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, Boston, MA, USA, 2004.

[84] K. suk Lhee and S. J. Chapin. Type-assisted dynamic buffer overflow detection. In *USENIX Security Symposium*, pages 81–88, 2002.

[85] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[86] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, 2000.

[87] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl.* O'Reilly, 1996.

[88] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, 2006.

[89] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *European Software Engineering Conference / ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 327–336. ACM Press, 2003.

[90] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Symposium on Reliable and Distributed Systems (SRDS)*, Florence, Italy, October 2003.

[91] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, 2006.

[92] W. Xu, D. C. Duvarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, California, November 2004.

[93] S. H. Yong and S. Horwitz. Protecting c programs from attacks via invalid pointer dereferences. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 307–316, 2003.

[94] X. Zhang, A. Edwards, and T. Jaeger. Using CQual for static analysis of authorization hook placement. In *USENIX Security Symposium*, 2002.