ABSTRACT

KIL, CHONGKYUNG. Mechanisms for Protecting Software Integrity
in Networked Systems. (Under the direction of Associate Professor Peng Ning).

Protecting software integrity is a key to successfully maintain its own credibility and reduce the financial and technical risks caused from a lack of integrity. Although researchers have been putting effort on improving software development techniques and preventing human errors during the software development process, it is still a daunting task to make non-vulnerable software in practice. For example, the national vulnerability database shows that a set of new software vulnerabilities are discovered every day.

Since developing non-vulnerable software is hardly achievable, in this research, we look for a way to *achieve software integrity while they are used*. In particular, this dissertation investigates three mechanisms to protect software integrity at runtime. Firstly, this dissertation presents a protection mechanism that can thwart attacks that try to exploit memory corruption vulnerabilities of software. The protection mechanism is provided by randomizing the program's runtime memory address layout and the memory objects. As a result, it hinders memory corruption attacks by preventing an attacker being able to easily predict their target addresses. The protection mechanism is implemented by a novel binary rewriting tool that can randomly place the code and data segments of programs and perform fine-grained permutation of function bodies in the code segment as well as global variables in the data segment. Our evaluation results show minimal performance overhead with orders of magnitude improvement in randomness.

Secondly, this dissertation investigates a vulnerability identification mechanism named as CBones that can discover how unknown vulnerabilities in C programs are exploited by verifying program structural constraints. CBones automatically extracts a set of program structural constraints via binary analysis of the compiled program executable. CBone then verifies these constraints while it monitors the program execution to detect and isolate the security bugs. Our evaluation with real-world applications that known to have vulnerabilities shows that CBones can discover all integrity vulnerabilities with no false alarms, pinpoint the corrupting instructions, and provide information to facilitate the understanding of how an attack exploits a security bug.

Lastly, this dissertation identifies the need of dynamic attestation to overcome the limitations of existing remote attestation approaches. To the best of our knowledge, we are the first to introduce the notion of dynamic attestation and propose use of dynamic system properties to provide the integrity proof of a running system. To validate our idea, we develop an application-level dynamic attestation system named as ReDAS (Remote Dynamic Attestation System) that can verify runtime integrity of software. ReDAS provides the integrity evidence of runtime applications by checking their dynamic properties: structural integrity and global data integrity. These properties are collected from each application, representing the application's unique runtime behavior that must be satisfied at runtime. ReDAS also uses hardware support provided by TPM to protect the integrity evidence from potential attacks. Our evaluation with real-world applications shows that ReDAS is effective in capturing runtime integrity violations with zero false alarms, and demonstrates that ReDAS incurs 8% overhead on average while performing integrity measurements.

Mechanisms for Protecting Software Integrity in Networked Systems

by
Chongkyung Kil

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2008

APPROVED BY:

Dr. Douglas S. Reeves	Dr. Tao Xie
Dr. Peng Ning Chair of Advisory Committee	Dr. S. Purushothaman Iyer

DEDICATION

To my family and friends who made all of this possible, for their endless encouragement and patience.

BIOGRAPHY

Chongkyung Kil was born on August 17, 1971, in Seoul, Korea. He married Soojin Park in 1996 and they have two children, Sojung and Sangjun.

Chongkyung attended the Korea Military Academy from 1990 to 1994, earning bachelor of science degree in computer science. He started his graduate study at the Department of Electrical and Computer Engineering, Air Force Institute of Technology (Dayton, OH) in 2000 and received his Master's degree in computer science. After serving various assignments in Korean army, he continued the graduate study with the Ph.D. program at the Department of Compute Science, North Carolina State University in 2004.

During his doctoral study at North Carolina State University, he served as teaching and research assistants, and has numerous publications in journals and conference proceedings. His research interests include software and distributed system security.

He lives in Seoul, Korea, where he is working on various security projects for Korean army and still spends weekends as an ardent rooter for Wolfpack.

ACKNOWLEDGMENTS

I would like to thank all people who have helped and inspired me during my doctoral study. First of all, I would especially like to thank my advisor, Dr. Peng Ning, for his generous support and commitment. Throughout my doctoral work he continually encouraged me to develop analytical thinking and research skills. In addition, he was always accessible and willing to help his students with their research. As a result, research life became smooth and rewarding for me.

I am pleased thank Dr. Jun Xu for his insightful and constructive guide at early stage of my doctoral study. I am also very grateful for having an exceptional doctoral committee and wish to thank Dr. S. Purushothaman Iyer, Dr. Douglas S. Reeves and Dr. Tao Xie for their continual support, valuable comments, and encouragement.

I would like to thank Director of Graduate Program (DGP) Dr. David J. Thuente for his help during my Ph.D. study. I am also grateful to Ms. Margery Page for her help.

This research is supported by various funds including U.S. Army Research Office and I would like to thank their support. I also would like thank Korean army for giving the chance of my Ph.D. study and the financial support.

All my lab colleagues at the Cyber Defense Laboratory (CDL) made it a convivial place to work. In particular, I would like to thank Emre (John) Sezer for his friendship and help in the past four years. All other hardworking folks in CDL had inspired me in research and life through our interactions during the long hours in the lab.

I also would like to thank Korean army officers, Kyuyong Shin, Jungki So, Woopan Lee, Yongcheol Kim, Kiseok Jang, Changho Son, and Hesun Choi for all their help, support, and valuable time we had together at NC State.

My deepest gratitude goes to my family, Soojin, Sojung, and Sangjun. Their endless love, patience, and encouragement always help me keep moving forward in my life. I also would like to thank my extended families in Korea for their love and support.

I would regret if I did not join the Agape church. Dr. Hyungjun Kim and his family gave us tremendous help to face many challenges living in US. My grateful thanks also go to Rev. Kwanseok Kim for his spiritual guide to the Lord.

Finally, but most importantly, thanks be to God for my life and always being with me.

TABLE OF CONTENTS

LIST OF TABLES . . . . **vii**

LIST OF FIGURES . . . . **viii**

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Summary of Research Results 2
- 1.3 Organization of the Dissertation 4

2 Integrity Protection Mechanism **5**
- 2.1 Existing Approaches 8
- 2.2 Address Space Layout Permutation 9
  - 2.2.1 User Level Address Permutation 9
  - 2.2.2 Kernel Level Address Permutation 18
  - 2.2.3 Demonstration of Permutation 22
  - 2.2.4 Example of Integrity Protection with ASLP 22
- 2.3 Evaluation 24
  - 2.3.1 Security Evaluation 24
  - 2.3.2 Performance Evaluation 27
- 2.4 Limitations 29
- 2.5 Summary 30

3 Vulnerability Identification Mechanism **31**
- 3.1 Security Debugging Using Structural Constraints 34
  - 3.1.1 Program Structural Constraints 35
  - 3.1.2 Security Debugging through Constraints Verification 41
- 3.2 Implementation 43
  - 3.2.1 Constraint Extractor 43
  - 3.2.2 Monitoring Agent 44
- 3.3 Experimental Evaluation 47
  - 3.3.1 Security Debugging Using CBones: A Running Example 47
  - 3.3.2 Security Evaluation 48
  - 3.3.3 Performance Overhead 50
- 3.4 Related Work 51
- 3.5 Summary 53

4 Remote Dynamic Attestation Mechanism . . . 54
4.1 ReDAS: A Remote Dynamic Attestation System . . . 57
4.1.1 Assumptions and Threat Model . . . 57
4.1.2 ReDAS Overview . . . 58
4.1.3 Dynamic System Properties . . . 59
4.1.4 Measuring Dynamic System Properties . . . 65
4.1.5 Protection of Integrity Evidence . . . 67
4.1.6 ReDAS Dynamic Attestation Protocol . . . 68
4.1.7 Security Analysis . . . 69
4.2 Implementation . . . 72
4.3 Experimental Evaluation . . . 75
4.3.1 Experiments Setup . . . 75
4.3.2 Effectiveness . . . 77
4.3.3 Efficiency . . . 79
4.4 Related Work . . . 80
4.5 Summary . . . 81

5 Conclusion and Future Work . . . 83
5.1 Conclusion . . . 83
5.2 Future Work . . . 84

Bibliography . . . 86

LIST OF TABLES

Table 2.1 ELF Sections to Modify for ASLP . . . 13

Table 2.2 PaXtest Results . . . 25

Table 2.3 SPEC CPU2000 Benchmark Run Times (seconds) . . . 28

Table 3.1 Security Evaluation Result . . . 49

Table 4.1 Structural Constraints used in ReDAS . . . 60

Table 4.2 Data Invariants . . . 62

Table 4.3 Measurement Points of ReDAS . . . 66

Table 4.4 Test applications . . . 76

Table 4.5 Dynamic Properties of Test Applications . . . 76

Table 4.6 Test Scenarios for `ghttpd` . . . 77

Table 4.7 Effectiveness Evaluation Result . . . 78

LIST OF FIGURES

Figure 2.1 Example of Object Reference 11

Figure 2.2 Example of Program Header Rewriting 15

Figure 2.3 Example of PLT and GOT Sections Rewriting 16

Figure 2.4 Example of Relocation Section Rewriting 17

Figure 2.5 Example of Fine-Grained Permutation 19

Figure 2.6 Example of Normal Linux Process Memory Layout 20

Figure 2.7 Example of Coarse-grained Permutation 22

Figure 2.8 Example of Fine-grained Permutation 23

Figure 2.9 Example of Integrity Protection with ASLP 24

Figure 3.1 Security Debugging Process with CBones 35

Figure 3.2 Example of an ELF Program Runtime Process Memory Layout 36

Figure 3.3 Example of Stack Structure and Caller-callee Relationship 37

Figure 3.4 Example of Heap Structure 39

Figure 3.5 Example of Automated Security Bug Diagnosis Using CBones 48

Figure 3.6 Slowdown Factors of Nullgrind and CBones 52

Figure 4.1 ReDAS architecture 58

Figure 4.2 Snippet of Training Input for Test Applications 63

Figure 4.3 ReDAS Dynamic Attestation Protocol 69

Figure 4.4 Performance Overhead Evaluation Result 79

Chapter 1

Introduction

1.1 Motivation

In software engineering standards [1], software integrity is defined as ensuring the containment of risk or confining the risk exposure in software. In a security perspective, the meaning of software integrity can be translated as protecting software against various threats upon its components to ensure that the software has not been altered or compromised.

As software are taking over diverse tasks in networked computing environments (e.g., web-based applications, online banking), the importance of software integrity can not be overemphasized. When there is a lack of software integrity, it often causes both technical and financial losses. For example, US CERT [2] advises that 90% of security incidents are caused from the lack of software integrity, and a FBI report [3] shows that 90% of US organizations experienced various types of computer attacks in 2006 and 64% of them incurred the financial impact up to $32 million in total loss.

A good way of achieving software integrity is having good mind set and techniques during the software development process. Thus, software products will not have vulnerabilities that may harm to their integrity while they are used. Although researchers have been putting effort on improving software development techniques and preventing human errors during the software development process, it is still a daunting task to make non-vulnerable software in practice. For example, the national vulnerability database [4] shows that software vulnerabilities are increasing exponentially and a set of new vulnerabilities are discovered every day.

Since developing non-vulnerable software is hardly achievable, in this research, we look for a way to *achieve software integrity while they are used.* In particular, this dissertation attempts to make contributions by investigating three mechanisms that can protect software against integrity vulnerabilities (*integrity protection mechanism*), identify potential vulnerabilities of software (*vulnerability identification mechanism*), and verify software integrity at runtime (*remote dynamic attestation mechanism*).

1.2 Summary of Research Results

This dissertation includes three mechanisms to protect software integrity at runtime. The summary of these mechanisms are as follows:

Integrity Protection Mechanism: Protection of software against attacks that try to exploit their vulnerabilities is essential to achieve software integrity at runtime. In this thesis, we develop a protection mechanism named as address space layout permutation (ASLP). The core idea of ASLP is to provide a protection against memory corruption attacks (e.g., stack buffer overflow) by randomly assigning a program's critical memory regions (e.g., stack) and the memory objects (e.g., global variables) at process initialization time. Thus, it breaks the attacker's hard-coded memory address assumption that used in their attacks that try to exploit the vulnerabilities, and converts such attacks into a benign process crash, rather than allowing the attacker to take control of the program. Moreover, such crash-related information(e.g., coredump) can provide the valuable information about a yet-to- be-found integrity vulnerability in the program.

Compared to existing address space randomization approaches, ASLP introduces higher degree of randomness with minimal performance overhead. Essential to ASLP is a novel binary rewriting tool that can place the static code and data segments of a compiled executable to a randomly specified location and performs fine-grained permutation of procedure bodies in the code segment as well as static data objects in the data segment. We have also modified the Linux operating system (OS) kernel to permute stack, heap, and memory mapped regions. Together, ASLP completely permutes memory regions in an application. Our security and performance evaluation shows minimal performance overhead with orders of magnitude improvement in randomness (e.g., up to 29 bits of randomness on a 32-bit

architecture).

Vulnerability Identification Mechanism: A program may be identified as vulnerable to integrity breaches in various ways (e.g., address space randomization techniques [5, 6], Common Vulnerabilities and Exposures (CVE) [7]). However, understanding security bugs in a vulnerable program is a non-trivial task, even if the target program is known to be vulnerable and such crash information is given to users. To address this problem, we develop a novel security debugging tool called *CBones* (*SeeBones*, where *bones* is an analogy of program structures). CBones is intended to *fully automate* the analysis of a class of security vulnerabilities in C programs, the exploits of which would compromise the integrity of program structures satisfied by all legitimate binaries compiled from C source code. In other words, CBones automatically discovers how unknown vulnerabilities in C programs are exploited based on new type of program invariants called *program structural constraints*. Unlike the previous approaches, CBones can automatically identify exploit points of unknown security bugs without requiring a training phase, source code access (analysis or instrumentation), or additional hardware supports. CBones automatically extracts a set of program structural constraints via binary analysis of the compiled program executable. CBone then verifies these constraints while it monitors the program execution to detect and isolate the security bugs. To validate the effectiveness of this approach, we evaluate CBones with 12 real-world applications that contain a wide range of vulnerabilities. Our results show that CBones can discover all security bugs with no false alarms, pinpoint the corrupting instructions, and provide information to facilitate the understanding of how an attack exploits a security bug in the program.

Remote Dynamic Attestation Mechanism: Attestation of system integrity is an essential part of building trust and improve the integrity in distributed systems. A computer system usually behaves in the expected manner if the integrity of its components (e.g., applications) is intact. Several remote attestation techniques have been developed recently to provide integrity evidence of an *attester* to a remote *challenger* by hashing static memory regions, sometimes combined with language specific properties or program input/output. Unfortunately, none of these techniques include the dynamically changing components in the system, and thus do not provide complete integrity evidence. As a result, a remote party

still cannot gain high confidence in the attested system due to various runtime threats (e.g., remote buffer overflow attacks).

To address the above problem, we present a novel remote dynamic attestation system named ReDAS (Remote Dynamic Attestation System) that can provide integrity evidence for dynamic system properties. Such dynamic system properties represent the runtime behavior of the attested system, and enable an attester to provide its runtime integrity to a remote party. As an initial attempt, ReDAS currently provides two types of dynamic system properties: *structural integrity* and *global data integrity* of running applications. Such properties are collected from each application, representing the application's unique runtime behavior. In order to deal with potential attacks against ReDAS, we use the hardware support provided by Trusted Platform Module (TPM) to protect the integrity evidence. As a result, even if an attacker is able to compromise the host, he/she cannot modify the integrity evidence without being detected by the remote party. ReDAS is currently implemented as a prototype system on Linux. We have performed experimental evaluation of ReDAS using real-world applications. The result shows that ReDAS is effective in capturing runtime integrity violations with zero false alarms, and demonstrates that ReDAS incurs 8% overhead on average while performing integrity measurements.

1.3 Organization of the Dissertation

The rest of this dissertation is organized as follows. Chapter 2 presents the integrity protection mechanism. Chapter 3 discusses the vulnerability identification mechanism. Chapter 4 gives a detail of the remote dynamic attestation mechanism. Chapter 5 concludes this dissertation and points out some future research directions.

Chapter 2

Integrity Protection Mechanism

Develop a mechanism for protecting software against potential vulnerabilities is essential to achieve software integrity at runtime. Although there are different types of vulnerabilities that can lead to integrity breaches, this chapter focuses on the vulnerabilities that can allow attackers illegally access or modify memory objects in the program at runtime. This type of vulnerability is called *memory corruption vulnerability*. Memory corruption vulnerability has been one of most common methods for attackers to break software integrity. A quick survey of US-CERT cyber security alerts on both most recently updated vulnerabilities and the twenty vulnerabilities with the highest severity metric (highest impact) shows that more than half of them are memory corruption vulnerabilities [2].

Memory corruption attack is a mechanism that exploits memory corruption vulnerabilities to illegally modify memory objects in a vulnerable program's runtime memory space, so that an attacker can alter the program's behavior in a certain way. When the attacker is able to modify the contents of target program's memory at will, the results can range from system instability to execution of arbitrary code. Examples of common memory corruption attacks are stack buffer overflows [8], format string attacks [9], and double-free attacks [10]. These attacks try to modify memory objects in the stack (e.g., return address) or the heap (e.g., heap management data) in a vulnerable program.

Vulnerability to this type of attack is typically caused by a lack of input validation in the C programming language, with which the programmers are offered the freedom to decide when and how to handle inputs. This flexibility often results in improved application performance. However, the number of vulnerabilities caused by failure of input validation

indicates that programming errors of this type are easy to make and difficult to fix.

While bug prevention techniques (e.g, CCured [11]) in the development stage are preferred, continued discoveries of memory corruption vulnerabilities indicate alternatives must be sought. We believe that mitigating this kind of attacks would give attackers significantly fewer ways to exploit their targets, thereby reducing the threat they pose against software integrity. One promising method is *address space randomization* [12]. It has been observed that most memory corruption attacks hard-code the memory addresses of target memory objects in their malicious input. This input is then used to exploit the memory corruption vulnerability in the target program. Address space randomization randomizes the layout of process memory, thereby making the critical memory addresses unpredictable and breaking the attacker's hard-coded address assumption. As a result, a memory corruption attack will most likely cause a vulnerable program to crash, rather than allow the attacker to take control of the program. Moreover, such crash-related information (e.g. coredump) helps to indicate the program has integrity vulnerabilities that can be exploited by an adversary.

Several address space randomization techniques have been proposed [12, 13, 14, 15, 6]. Among the existing approaches, PaX Address Space Layout Randomization (ASLR) [6] and address obfuscation [14] are most visible. PaX ASLR randomly relocates the stack, heap, and shared library regions by modified Operating System (OS) kernel, but does not efficiently randomize locations of code and data segments. Address obfuscation expands the randomization to code and data segments by modifying a compiler, but requires source code modifications and incurs 11% performance overhead on average. Position Independent Executables (PIE) [16] allows a program to run as a shared object so the base addresses of the code and data segments can be relocatable, but it also incurs 14% performance degradation on average. While insufficient randomness allows successful brute-force attacks, as shown in recent studies, the required source code modification and performance degradation prevent these effective methods from being used for commodity software, which is the major source of exploited vulnerabilities on the Internet.

In this chapter, we present Address Space Layout Permutation (ASLP) to increase the programs' randomness with minimal performance overhead. ASLP permutes all sections (including code and data segments) in the program address space. This is done in two ways. First, we create a novel binary rewriting tool that randomly relocates the code and data

segments in a program. It also randomly re-orders functions within the code segment, and data variables within the data segment. Our rewriting tool operates directly on compiled program executables, and does not require source code modification. We only need the relocation information from the compile-time linker to perform the randomization rewriting. Such information is produced by all existing C compilers. Second, to randomly permute stack, heap, and memory mapped regions (e.g., shared libraries), we modify the Linux kernel. Our kernel changes conserve as much virtual address space as possible to increase randomness. Our binary rewriting tool can be automatically and transparently invoked before a program is launched, while our kernel level support runs without any additional change to the runtime system. To validate the practicality and effectiveness of ASLP, we have evaluated ASLP using security and performance benchmarks. Our security benchmark result shows that ASLP can provide up to 29 bits of randomness on a 32-bit architecture, while the performance benchmark result indicates that ASLP incurs less than 1% overhead. In summary, the major contributions of our work are as follows:

- ASLP provides probabilistic protection an order of magnitude stronger than previous techniques. It can protect software from wide range of memory corruption attacks such as stack/heap buffer overrun, format string, return-to-libc attacks. Especially the fine-grained randomization provides further protection against attacks such as partial overwrite attacks [13], dtors attacks [17], and data forgery attacks. These types are attacks are not addressed in existing approaches.

- ASLP randomizes regions throughout the entire user memory space, including code and data segments while maintaining correct program behavior at runtime. Program transformation is automatically done by our binary rewriting tool without the requirement of source code modification.

- The performance overhead is generally very low (less than 1%). In comparison, existing techniques that are capable of randomly relocating code and data segments, in particular PIE and Address obfuscation, incur more than 10% overhead on average.

The rest of the chapter is organized as follows. Section 2.1 discusses related work. Section 2.2 describes the design and implementation of ASLP. Section 2.3 presents the evaluation of ASLP. Section 2.4 describes the limitations of current implementation, and Section 2.5 concludes.

2.1 Existing Approaches

The seminal work on program randomization by Forrest et al. illustrated the value of diversity in ecosystems and similar benefits for diverse computing environments [18]. In short, their case for diversity is that if a vulnerability is found in one computer system, the same vulnerability is likely to be effective in all identical systems. By introducing diversity into a population, resistance to vulnerabilities is increased. Address randomization achieves diversity by making the virtual memory layout of every process unique and unpredictable. Existing address space randomization approaches can be divided into two categories: user level and kernel level. User level and kernel level approaches achieve randomization with modification to the user space applications and the OS kernel, respectively. Both approaches have their advantages and disadvantages, which should be carefully reviewed by the user before the application of the techniques.

User Level Randomization Address obfuscation [13, 14] introduced a mechanism to not only randomly shift the placement of the three critical memory regions (stack, heap, and shared library), but also randomly re-order objects in code and data segments. This approach relies on a source code transformation tool [19] to perform the randomization. It introduced special pointer variables to store actual locations of objects for the code and data segments randomization. In addition, they add randomly sized pads to stack, heap, and shared library by using a special initialization code, the wrapper function, and the junk code respectively.

Kernel Level Randomization Kernel level randomization has become a more attractive option since modification of one component provides system wide protection. Recently, kernel level address randomization techniques are being actively used in major Linux distributions: Red Hat Exec-Shield [20] can be found in Fedora Core [21]; PaX Address Space Layout Randomization(ASLR) [6] can be found in Hardened Gentoo [22]. Both PaX ASLR and Exec-Shield use the same approach by padding random size to the critical memory regions. These techniques, unfortunately, have a number of limitations. First, the pads unnecessarily waste memory space. Note that the only way to increase the program randomness is to increase the size of the pads, thereby wasting more space. Second, they keep the relative order of segments. For instance, code segment always comes first,

and data segment follows the code segment. Therefore, once an attacker detects the size of the pad, he/she can easily craft attacks to compromise the system. A recent work [23] shows the de-randomization attack that can defeat PaX ASLR in an average of 216 seconds. This is a clear indication that we need to improve the randomness in the critical memory regions.

2.2 Address Space Layout Permutation

ASLP provides both user and kernel level randomizations. For user level randomization, we create a binary rewriting tool that randomly relocates the code and data segments of an executable before it is loaded into memory for execution. Our tool not only alters the locations of the code and data segments but also changes the orders of functions and data objects within the code and data segments. For kernel level randomization, we modify the Linux kernel to permute three critical memory regions including stack, heap, and shared library. This section explains the design and implementation of ASLP in detail. This section also shows how ASLP can protect software integrity against the class of memory corruption attacks.

2.2.1 User Level Address Permutation

In a typical program development scenario, a program's source code is written in a high-level language (e.g., `helloworld.c`) and compiled into object files (e.g., `helloworld.o`) that can be linked with other objects to produce the final executable (e.g., `helloworld`). An object file contains not only code (functions) and data (variables), but also additional bookkeeping information that can be used by the compile-time linker to assemble multiple objects into an executable file. Such information includes relocation records, the type of the object file, and debugging related information. In an object file, an element (function, variable, or bookkeeping information) has its own symbolic name. Symbolic names for static code/data elements in object files are resolved to virtual addresses when the executable is assembled by the compile-time linker. Symbolic names for dynamic elements such as C library functions are usually resolved by the runtime system loader.

The goal of user level address permutation is to randomly relocate the code and data segments and their elements so that the program will have a different memory layout

each time it is loaded for execution. This permutation makes it difficult for various types of memory corruption attacks including partial overwrite attacks [13], dtors attacks [17], and data forgery attacks. These attacks are based on the assumption that the code and data segments of the target application reside at the same locations on different machines. Partial overwrite attacks change only the least significant part of the return address in the stack, so the attacker can transfer the program control to the existing vulnerable function with malicious arguments. Dtors attacks overflow a buffer (global variable) in the `.data` section to overwrite function pointers in the `.dtors` section. The `.dtors` section includes function pointers that are used after `main` function exits. When a corrupted function pointer is used, the program control is transferred to the attacker's code. Dtors attacks are possible since `.dtors` section is adjacent to the `.data` section, and the attacker knows the relative distance between the buffer in the `.data` section and the target function pointer in the `.dtors` section. Recent security report [24] shows that data forgery attacks can overwrite global variables in the data segment so the attacker can alter the values of security critical data variables. Note that even if kernel level randomization changes the base addresses of stack, heap, and shared library regions, these kinds of attacks can still be successful, since the locations of the code and data segments are fixed.

Our user level permutation makes these types of attacks significantly difficult. In the user level address permutation, we change the base addresses of code and data segments. We also randomly reorder procedure bodies within the code segment and data variables within the data segment. As a result, the virtual addresses of code elements (e.g., functions) or data variables (e.g., global variables) are not fixed anymore. To make these changes, we have to modify all cross references between the objects in the program. Otherwise, the randomized program will have many dangling references that will certainly lead to a program crash.

In order to achieve the user level permutation, we develop a binary rewriting tool that transforms an Executable and Linking Format (ELF) [25] executable file into a new one that has a different layout. Our tool allows users to choose any values between 1 and 700K for different offsets for the code and data randomization. The given values are then multiplied by virtual memory page size (usually 4096). Consequently, users can freely decide the starting addresses of the code and data segments in the entire user memory space. (Currently on a default Linux system, user space has 3GB of virtual address space.)

Our tool also allows users to change the order of the code and data segments. The default linker always places the data segment after the code segment. Therefore, an attacker can guess the target program layout once he has found certain offset values and the starting address of the code segment. By changing the order of code and data segments, it is more difficult to guess correct program layout.

Rewriting an ELF executable file for randomizing memory layout while maintaining the program's original behavior is non-trivial task. An error during the rewriting process can cause abnormal program status at runtime. Figure 2.1 shows an example. In this example program's source code (left side of the figure), `printNumber` function is called at line 11 in the `main` function. Since the virtual address of `printNumber` function is $0x804835c$, the `call` instruction at $0x80483c7$ refers the address, $0x804835c$ to execute `printNumber` function. This means that if we modify the virtual address of `printNumber` function, we also need to change the reference at $0x80483c7$. Therefore, we need to find and change all such references to `printNumber` function in the program. Otherwise, the randomized program will cause side effects such as program crash at runtime.

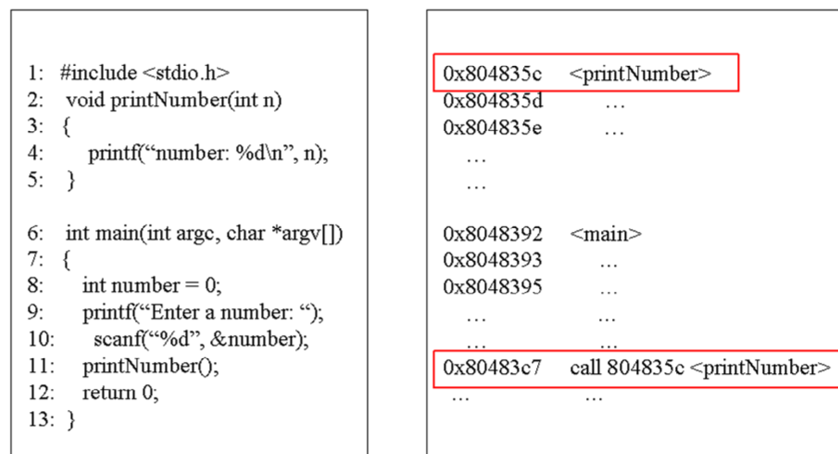


Figure 2.1: Example of Object Reference

Requirements and challenges during the binary rewriting an ELF file can be summarized in the following questions:

- What parts of an ELF executable file need rewriting?

- How to find the correct locations of those parts and rewrite them?
- How those parts are connected or affect each other at run time? (How functions and variables are referred at run time?)

The first challenge requires us to understand the ELF executable file format and how the linker and the loader create the program memory layout. An ELF executable comprises different segments including code, data, and dynamic segments. The code segment includes executable code, the data segment includes the program data, and the dynamic segment includes the information relevant to dynamic loading. Each segment further comprises different sections. For example, the data segment includes `.data` section to maintain initialized global variables and `.bss` section for uninitialized global variables. Currently, an ELF executable file can have totally 45 sections according to the specification. Each section has its own identifier called *section header* which specifies the content of the section and how to access the elements in the section. We identify that a total of 16 sections are related to program memory layout. These sections include information about the symbols, dynamic linking, relocation of code and data objects, and other data that are critical to program execution. Table 2.1 presents the detailed information about the 16 sections[1]. We also identify that both the ELF header and the program header need to be modified since they contain an entry point to start the program, the name of the dynamic loader, instructions on what portions of the file are to be loaded, and the permissions of the sections of memory (for example, `.text` section is read-only). Such information has to be changed once our tool permutes the code and data segments. (A detailed explanation of rewriting the headers will be discussed later in this section.)

The next challenge lies in the method to find out correct locations of the elements in an ELF executable file. We acquire this information by looking up the symbol table section. The symbol table holds the information that the linker needs to bind multiple object files into a final executable file. An entry of the symbol table holds the following information: symbol name, binding (linkage scope: local or global), visibility (scope to other files: hidden, protected, or internal), and the virtual address of the symbol. Since every element has its own symbol name, we can get the virtual address of the symbol by looking up its name in the symbol table.

[1]Further information about these sections and the ELF specifications can be obtained from a number of sources including [25, 26]

Table 2.1: ELF Sections to Modify for ASLP

Section Name	Semantics	Section Type
`.text`	Executable instructions	SHT_PROGBITS
`.data`	Initialized global variables	SHT_PROGBITS
`.bss`	Uninitialized global variables	SHT_PROGBITS
`.got`	Global offset table	SHT_PROGBITS
`.plt`	Procedure linkage table	SHT_PROGBITS
`.got.plt`	Read-only portion of the global offset table	SHT_PROGBITS
`.rodata`	Read-only data	SHT_PROGBITS
`.symtab`	Symbol table	SHT_SYMTAB
`.dynsym`	Dynamic linking symbol table	SHT_DYNSYM
`.dynamic`	Dynamic linking information	SHT_DYNAMIC
`.rel.dyn`	Relocation information for dynamic linking	SHT_REL
`.rel.plt`	Relocation information for `.plt` section	SHT_REL
`.rel.init`	Relocation information for `.init` section	SHT_REL
`.rel.text`	Relocation information for `.text` section	SHT_REL
`.rel.data`	Relocation information for `.data` section	SHT_REL
`.rel.fini`	Relocation information for `.fini` section	SHT_REL

The last challenge is to find out how elements in an ELF file refer to each other at run time and how to find out such references. Searching all such references (where it is defined and where it is used) in a binary file is a daunting task without additional information. We obtain such references by using a linker option (`-q, or -emit-relocs`). This option produces relocation sections that include information about where the functions and variables are used in the program. Now we can gather all cross-reference information from the following sections: global offset table (`.got`), procedure linkage table (`.plt`), relocation data (`.rel.data`), and relocation text (`.rel.text`). The global offset table includes pointers to all of the static data in the program, and the procedure linkage table stores pointers to all of the static code objects (functions) in the program. Therefore, these two sections provide the information about where the functions and variables are located in the program. Relocation sections such as `.rel.text` and `.rel.data` provide information about where the functions and variables are used in the program.

The rewriting process (user level permutation) comprises two major phases: 1) Coarse-grained permutation, and 2) Fine-grained permutation.

Coarse-grained Permutation The goal of the coarse-grained permutation is to shift code and data segments according to the given offset values from the user. Changing the order of code and data segments is also executed in this phase. To achieve the goal, the rewriting process goes through three stages: 1) ELF header rewriting, 2) Program header rewriting, and 3) Section rewriting.

ELF Header Rewriting Our tool first reads the ELF header to check if the target file is an ELF file. This can be done by reading the first four bytes of the file. If the file is an ELF object file, it should include the magic number in the *e_ident* member identifying itself as an ELF object format file. The tool then checks the sizes of the code and data segments to validate the given offset sizes from the user can fit into the user memory address space that are allowed in the Linux memory management scheme. Retrieving the location of the string table is then done. The string table holds information that represents all symbols, including section names referred in the program. Since each section header's *sh_name* member only holds an index to the string table, we need the string table during the entire rewriting process to look up the symbol name according to its index. The tool then rewrites the program entry point (*e_entry*), which is the virtual address where the system first transfers control to start the program, according to the offset value of the code segment.

Program Header Rewriting Once we modified the ELF header, we need to change two entries in the program header: *p_vaddr* and *p_paddr*. They hold the virtual/physical addresses of the code and data segments that the loader needs to know for creating the program memory layout. Our tool modifies the *p_vaddr* and *p_paddr* values of the code and data segments according to the given offset values for the code and data segments. Figure 2.2 illustrates the program header rewriting. The program entry point is randomized from $0x80482ac$ to $0x804c2ac$ after rewriting. The data segment's starting address is also changed from $0x0805d4bc$ to $0x0805d4bc$.

Section Rewriting Section rewriting is the most important stage in the coarse-grained permutation process to ensure a newly generated ELF file runs without any side effects (i.e., broken references). Since each section has different semantics and holds specific

```
Elf file type is EXEC (Executable file)
Entry point 0x80482ac
There are 6 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4
  INTERP         0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004bc 0x004bc R E 0x1000
  LOAD           0x0004bc 0x080494bc 0x080494bc 0x0010c 0x00110 RW  0x1000
  DYNAMIC        0x0004d0 0x080494d0 0x080494d0 0x000c8 0x000c8 RW  0x4
  NOTE           0x000108 0x08048108 0x08048108 0x00020 0x00020 R   0x4
```

(a) Program Header Before Rewriting

```
Elf file type is EXEC (Executable file)
Entry point 0x804c2ac
There are 6 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x0804c034 0x0804c034 0x000c0 0x000c0 R E 0x4
  INTERP         0x0000f4 0x0804c0f4 0x0804c0f4 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x0804c000 0x0804c000 0x004bc 0x004bc R E 0x1000
  LOAD           0x0004bc 0x0805d4bc 0x0805d4bc 0x0010c 0x00110 RW  0x1000
  DYNAMIC        0x0004d0 0x0805d4d0 0x0805d4d0 0x000c8 0x000c8 RW  0x4
  NOTE           0x000108 0x0804c108 0x0804c108 0x00020 0x00020 R   0x4
```

(b) Program Header After Rewriting

Figure 2.2: Example of Program Header Rewriting

information, we need to know how to handle different sections and their entries. To take the case of symbol table section (`.symtab`), it holds all symbols used in both code and data segments. Therefore, if a symbol table entry refers a function in the code segment, we need to adjust the symbol's address value according to the code segment offset. Similarly, if an entry refers to a global variable in the data segment, we need to adjust it according to the data segment's offset value.

Some sections require further understanding of how an ELF executable works during the run time to rewrite the program correctly. According to the ELF specification, some sections implicitly refer to other sections during the run time to resolve the symbol's actual address. For example, Procedure Linkage Table (PLT) section contains a jump table used when the program calls functions during the run time. Since procedure linkage table entries store addresses pointing to the entries in the Global Offset Table (GOT) to resolve actual addresses of the functions, we need to modify both PLT and GOT section entries together. Figure 2.3 shows the randomization example of the PLT and the GOT sections. A

PLT entry at $0x804829c$ in Figure 2.3(a) points the GOT entry that holds the jump address ($0x80482a2$). In Figure 2.3(b), the procedure linkage table entry, related global offset table entry, and actual content of the GOT entry are modified after the randomization.

```
0804826c <.plt>:
 804826c:      ff 35 b0 95 04 08        pushl  0x80495b0
  ...                ...                   ...
 8048297:      e9 d0 ff ff ff           jmp    804826c <.plt>
 804829c:      ff 25 c0 95 04 08        jmp    *0x80495c0
 80482a2:      68 10 00 00 00           push   $0x10

080495ac <_GLOBAL_OFFSET_TABLE_>:
 80495ac:      d0 94 04 08 00 00 00     rclb   0x8(%esp,%eax,1)
  ...                  ...                 ...
 80495be:      04 08                    add    $0x8,%al
 80495c0:      a2 82 04 08 00           mov    %al,0x80482
 80495c5:      00 00                    add    %al,(%eax)
  ...
```

(a) Before Rewriting

```
0804c26c <.plt>:
 804c26c:      ff 35 b0 d5 05 08        pushl  0x805d5b0
  ...                ...                   ...
 804c297:      e9 d0 ff ff ff           jmp    804c26c <.plt>
 804c29c:      ff 25 c0 d5 05 08        jmp    *0x805d5c0
 804c2a2:      68 10 00 00 00           push   $0x10
 804c2a7:      e9 c0 ff ff ff           jmp    804c26c <.plt>

0805d5ac <_GLOBAL_OFFSET_TABLE_>:
 805d5ac:      d0 d4                    rcl    %ah
  ...                ...                   ...
 805d5bd:      c2 04 08                 ret    $0x804
 805d5c0:      a2 c2 04 08 00           mov    %al,0x804c2
 805d5c5:      00 00                    add    %al,(%eax)
```

(b) After Rewriting

Figure 2.3: Example of PLT and GOT Sections Rewriting

We also need to modify relocation sections since they hold information about where the functions and variables are referred to in the program. In Figure 2.4(a), the relocation text section (`.rel.text`) entry tells that the `printNumber` function is referred at address $0x80483c8$ (where the function call is made). It also indicates that the $0x804835c$ is the `printNumber` function's address (where the function resides). After randomization, those values are changed according to the code section offset size (4 KB) as in Figure 2.4(b).

Fine-grained Permutation The goal of fine-grained permutation is to randomly change the order of functions and data variables within the code and data segments. This provides additional protection against memory corruption attacks (e.g., dtors attack) that previous

```
Relocation section '.rel.text' at offset 0x249c contains 37 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
080482b8  00006101 R_386_32          08048408   __libc_csu_fini
080482bd  00006801 R_386_32          080483d8   __libc_csu_init
080482c4  00006a01 R_386_32          08048392   main
080482c9  00006b02 R_386_PC32        0804828c   __libc_start_main@@GLI
080482dd  0000720a R_386_GOTPC       080495ac   _GLOBAL_OFFSET_TABLE_
080482e3  00007803 R_386_GOT32       00000000   __gmon_start__
  ...       ...      ...               ....       ....
  ...       ...      ...               ....       ....

080483bb  00001101 R_386_32          0804847c   .rodata
080483c0  00006402 R_386_PC32        0804827c   scanf@@GLIBC_2.0
080483c8  00006602 R_386_PC32        0804835c   printNumber
080483cd  00007002 R_386_PC32        0804837a   foo
080483de  00006202 R_386_PC32        08048254   _init
```

(a) `.rel.text` Section Before Rewriting

```
Relocation section '.rel.text' at offset 0x249c contains 37 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804c2b8  00006101 R_386_32          0804c408   __libc_csu_fini
0804c2bd  00006801 R_386_32          0804c3d8   __libc_csu_init
0804c2c4  00006a01 R_386_32          0804c392   main
0804c2c9  00006b02 R_386_PC32        0804c28c   __libc_start_main@@GLI
0804c2dd  0000720a R_386_GOTPC       0805d5ac   _GLOBAL_OFFSET_TABLE_
0804c2e3  00007803 R_386_GOT32       00000000   __gmon_start__
  ...       ...      ...               ...        ...
  ...       ...      ...               ...        ...

0804c3bb  00001101 R_386_32          0804c47c   .rodata
0804c3c0  00006402 R_386_PC32        0804c27c   scanf@@GLIBC_2.0
0804c3c8  00006602 R_386_PC32        0804c35c   printNumber
0804c3cd  00007002 R_386_PC32        0804c37a   foo
0804c3de  00006202 R_386_PC32        0804c254   _init
```

(b) `.rel.text` Section After Rewriting

Figure 2.4: Example of Relocation Section Rewriting

approaches can not address. Fine-grained permutation comprises three stages: 1) Information Gathering, 2) Random Sequence Generation, and 3) Entry Rewriting.

Information Gathering The following information is gathered for the fine-grained permutation: section size, section's starting address, section's offset, total number of entries, the original order of entries, each entry's size, and each entry's starting address. The program header provides most of the information except for each entry's size and the entry's starting address. We can get each entry's starting address from the symbol table and calculate the entry's size by subtracting the address value of the current entry from the address of the next entry according to the original order of the entries. We store the gathered information in the data structure for later use.

Random Sequence Generation We need to generate a randomized integer sequence to shuffle the functions and variables. To increase the randomness, we generate two separate randomized integer sequences for each code and data segment. We exploit the `random` function and related functions provided by Linux operating system to generate the random sequences. The maximum number in the randomized sequence for code(data) segment is the same as the total number of entries of code (data) segment.

Entry Rewriting Entry rewriting is the last stage of fine-grained permutation. First, we rewrite the functions and variables according to the randomized sequences in a separate memory region. We then take out the original portions of the code and data segments from the ELF file and replace them with rearranged ones. Finally, we modify all the cross-references among functions and data objects. We change the relocation sections as shown in the coarse-grained permutation. We also modify offset values of all local function calls in the code segment. Local functions can only be called within the local scope (within the same file) and they are called by relative-offset from the current program counter (PC). For example, if the current PC is $0x8048000$ and the local function is located at $0x8049000$, then the offset used in the calling instruction is $0x1000$. We also change `.rodata` section that stores read-only (static) data objects, since control flow instructions (e.g., `jmp` or `call`) may refer to the values of the objects in the `.rodata` section to transfer the current control of the program. Figure 2.5 present an example of fine-grained permutation with an example program. The order of variables and their addresses are randomly rewritten by our tool.

It is worth noting that the protection scheme of fine-grained permutation is mainly dependent on the number of variables or functions in the program. If a program has few functions or global variables, the fine-grained permutation does not add strong protection on the code and data segments. However, if a program has a large number of functions and/or variables (e.g., Apache has over 900 variables), fine-grained permutation makes it difficult to guess correct locations of functions and variables.

2.2.2 Kernel Level Address Permutation

We build the ASLP kernel [27] for the 32-bit x86 CPU with Linux 2.4.31 kernel. Each process has its own virtual 32-bit address space ranging sequentially from 0 byte to 4 GB as shown in Figure 2.6. A program code segment starts from 0x8048000, which is

```
08049664 <number1>:
 8049664:       01 00                     add    %eax,(%eax)
        ...

08049668 <number2>:
 8049668:       16                        push   %ss
 8049669:       00 00                     add    %al,(%eax)
        ...

0804966c <number3>:
 804966c:       4d                        dec    %ebp
 804966d:       01 00                     add    %eax,(%eax)
        ...

08049670 <numbers>:
 8049670:       16                        push   %ss
 8049671:       00 00                     add    %al,(%eax)
```

(a) `.data` Section Before Rewriting

```
08049664 <number2>:
 8049664:       16                        push   %ss
 8049665:       00 00                     add    %al,(%eax)
        ...

08049668 <number3>:
 8049668:       4d                        dec    %ebp
 8049669:       01 00                     add    %eax,(%eax)
        ...

0804966c <number1>:
 804966c:       01 00                     add    %eax,(%eax)
        ...

08049670 <numbers>:
 8049670:       16                        push   %ss
 8049671:       00 00                     add    %al,(%eax)
```

(b) `.data` Section After Rewriting

Figure 2.5: Example of Fine-Grained Permutation

approximately 128MB from the beginning of the address space. All data variables initialized by the user are placed in the `.data` section, and uninitialized data variables are stored in the `.bss` section. Shared libraries are placed in the dynamically shared objects (DSO) segments. The heap and the stack segments grow according to the user's request.

ASLP does not permute the top 1 GB of virtual address space (kernel space) since moving this region would require complex access control check on each memory access which introduces additional performance overhead. As a result, there are three regions for permutation: the user-mode stack, `brk()`-managed heap, and `mmap()` allocations.

The User Stack The location of the user stack is determined and randomized during process creation. In the early stage of process creation, the kernel builds a data

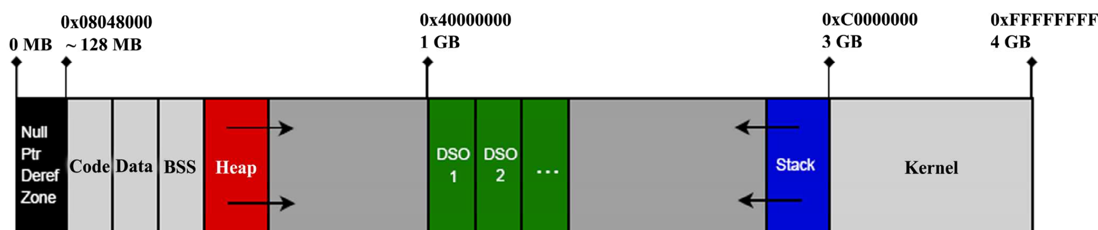


Figure 2.6: Example of Normal Linux Process Memory Layout

structure to hold process arguments and environment variables. This data structure is not yet allocated in the process memory, but rather it is prepared for the process in the kernel space memory. In this structure, the stack pointer is defined. This pointer is merely an offset into the first page of the soon-to-be stack region. We subtract a random amount between 0 and 4 KB from the stack pointer, thereby introducing randomization in low-order bits.

In the later stages of process creation, the same data structure is copied into the process address space. In this phase we introduce the large scale randomization. A random amount is subtracted from the standard 3 GB stack base location so that the region starts anywhere between approximately 128 MB and 3 GB.

To ensure the stack has room to grow, ASLP prevents subsequent allocations immediately below the stack. This feature prevents the stack from being allocated so close to another region that it cannot expand. The exact amount of reserved area is configurable, but our experiments show that 8 MB is sufficient for most applications.

The `brk()`-managed Heap Similar to the stack, the heap location is set during process creation. In an unmodified Linux kernel, the heap is allocated along with the BSS region, which is conceptually a part of the data segment. We modify the allocation code for the BSS and heap so they occur in two independent steps. Separation allows the heap location to be defined independently of the data segment. The amount of space to be allocated for the heap is then augmented by 4 KB (1 page). Then a random, page-aligned virtual address between 0 and 3 GB is generated for the start of the heap. Finally, a random value between 0 and 4 KB is added to this address to achieve sub-page randomization. Since the initial heap allocation was given an extra page, the sub-page shift will not push it beyond

the original allocation. The heap also can grow to fulfill dynamic memory requirements as the corresponding process runs. As with the stack, a comparable solution is used for the heap in which an unused region of configurable size is maintained following the heap. This prevents the heap from being placed too close to other regions so that it has enough room to grow.

`mmap()` **Allocations** The `mmap` system call is used to map objects into memory. Such objects include shared libraries as well as any other files the application may wish to bring into memory. Allocations made by `mmap` are randomized using a one-phase, major randomization that is nearly identical to the primary phase used for the stack and heap. A secondary, sub-page shift is not used for `mmap` allocations, because doing so would violate the POSIX `mmap` specification [28]. From the specification:

> The *off* argument is constrained to be aligned and sized according to the value returned by `sysconf()` when passed `SC PAGESIZE` or `SC PAGE SIZE`. When `MAP FIXED` is specified, the application shall ensure that the argument `addr` also meets these constraints. The implementation performs mapping operations over whole pages [28].

From this excerpt, the straightforward approach of applying a similar 0 to 4 KB random shift for `mmap` allocations is forbidden.

Since there can be multiple independent `mmap` allocations per process (such as for two different shared libraries), each allocation is made randomly throughout the entire available user level address space. This means that allocations for multiple shared libraries do not necessarily occupy a contiguous, sequential set of virtual memory addresses as they do in all related techniques and unrandomized kernels. This is beneficial because the location of one library will be of no use to determine the location of another library.

Although the `mmap` system call allows a user process to request a specific virtual address to store the mapping, there is no guarantee the request will be honored even in the vanilla kernel. In the ASLP kernel, if a specific address is requested it is simply disregarded and replaced by a random address. The random, page-aligned addresses are issued between 0 and 3GB. Therefore, `mmap` allocations use a one-phase major randomization rather than the two-phase approach used for the stack and heap. An exception to overriding supplied addresses exists for fixed regions, such as the code and data segments. These regions are

also brought into memory via `mmap`, but because they are flagged as fixed the supplied address is honored without modification.

2.2.3 Demonstration of Permutation

After both user level and kernel level permutations, all critical memory regions including code and data segments can be placed in different locations throughout the user memory space. Figure 2.7 shows a possible permutation of the normal process memory layout as shown in Figure 2.6. The heap is allocated independently of the data segment, and the stack is not the highest allocation in the user space. The data segment comes first than the code segment. In short, the static code, data, stack, heap, and `mmap` allocations occur randomly throughout the 3 GB user address space. ASLP further randomizes the orders of functions in the code segment and data variables in the data segment. Figure 2.8 shows an example of fine-grained permutation on the data segment. It shows that data variables (from `num1` to `num6`) are randomly re-ordered after the rewriting the data segment.

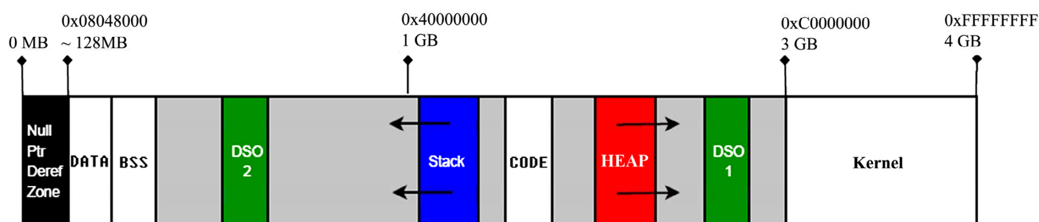


Figure 2.7: Example of Coarse-grained Permutation

2.2.4 Example of Integrity Protection with ASLP

As mentioned earlier, ASLP provides a protection mechanism that can thwart the class of memory corruption attacks. One of key characteristics of memory corruption attacks is that attackers need to locate the memory objects that they want to modify in a vulnerable program's memory space. This can be done easily when a program is not randomized since a program's runtime memory layout is generally same on different machines. However, if the program memory layout is randomized on different machines or even different processes, attackers are forced to guess the correct locations of the target

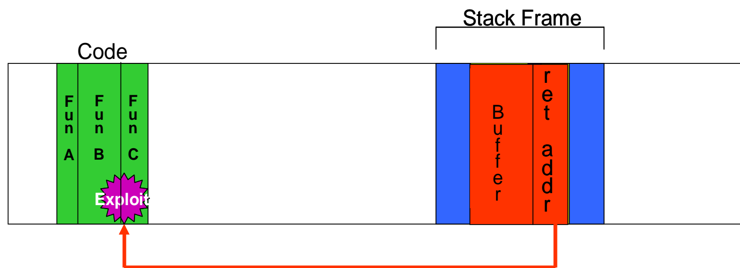
```
Disassembly of section .data:
8049684 <__data_start>:
8049684:    00 00             add   %al,(%eax)
8049688 <__dso_handle>:
8049688:    00 00             add   %al,(%eax)
804968c <p.0>:
804968c:    9c                pushf
804968d:    95                xchg  %eax,%ebp
804968e:    04 08             add   $0x8,%al
8049690 <num1>:
8049690:    01 00             add   %eax,(%eax)
8049694 <num2>:
8049694:    02 00             add   (%eax),%al
8049698 <num3>:
8049698:    03 00             add   (%eax),%eax
804969c <num4>:
804969c:    04 00             add   $0x0,%al
80496a0 <num5>:
80496a0:    05 00 00 00 06    add   $0x6000000,%eax
80496a4 <num6>:
80496a4:    06                push  %es
80496a5:    00 00             add   %al,(%eax)
```

```
Disassembly of section .data:
8049684 <__data_start>:
8049684:    00 00             add   %al,(%eax)
8049688 <__dso_handle>:
8049688:    00 00             add   %al,(%eax)
804968c <p.0>:
804968c:    9c                pushf
804968d:    95                xchg  %eax,%ebp
804968e:    04 08             add   $0x8,%al
8049690 <num4>:
8049690:    04 00             add   $0x0,%al
8049694 <num5>:
8049694:    05 00 00 00 06    add   $0x6000000,%eax
8049698 <num6>:
8049698:    06                push  %es
8049699:    00 00             add   %al,(%eax)
804969c <num2>:
804969c:    02 00             add   (%eax),%al
80496a0 <num1>:
80496a0:    01 00             add   %eax,(%eax)
80496a4 <num3>:
80496a4:    03 00             add   (%eax),%eax
```

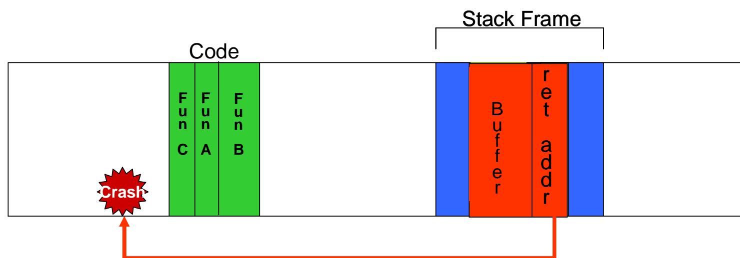
Figure 2.8: Example of Fine-grained Permutation

memory objects. Especially, ASLP makes it more challenging to locate the memory objects with fine-grained randomization capability. Figure 2.9 illustrates an example.

Figure 2.9(a) shows an attack scenario named as existing code attack. In this attack, an adversary tries to return to existing code which is known vulnerable function (e.g., Function C in Figure 2.9(a)). Previous address space randomization approaches can not address this type of memory corruption attack since they do not randomize the code segment. However, ASLP can thwart this attack by randomly locating the code segment as shown in Figure 2.9(b). ASLP also provides further protection by permuting the orders of the functions in the code segment. In other words, ASLP forces an adversary to guess not only the beginning address of the code segment but also the location of the vulnerable function in the code segment. As a result, it makes an adversary more difficult to find his/her target memory objects in the program's memory space.



(a) Example of an Existing Code Attack



(b) Example of Existing Code Attack with ASLP

Figure 2.9: Example of Integrity Protection with ASLP

2.3 Evaluation

2.3.1 Security Evaluation

As discussed previously, although every address in the x86 memory architecture is represented by 32-bits, not all of those bits can be randomized. To assess the bits of randomness in each region and for each technique, we use a third-party application called PaXtest [29]. PaXtest includes two types of tests. First, there are tests that try overwriting tiny executable code (e.g., just `ret`) in different memory regions (stack, heap, bss, data, anonymous) and check if the executable code can be run on the region. If the code works, it reports that the system is vulnerable on such region. The second type of tests measures the randomness of each region. By running locally, PaXtest is able to spawn simple helper programs that merely output the virtual addresses of their variables in a randomized region. In doing so, it can determine the number of randomized bits for such variables based on where they would normally reside in an unrandomized system. Table 2.2

Table 2.2: PaXtest Results

Segment	Vanilla	Exec-Shield	PaX ASLR	ASLP
User Stack	0 bits	17	24	28
Heap	0 bits	13	13	29
Mmap	0 bits	12	16	20
Code	0 bits	0	0	20
Data	0 bits	0	0	20

provides a summary of the PaXtest results.

The vanilla kernel has no randomization. So each region has zero bit of randomness. In other words, all segments always reside at the same locations. Exec-Shield provides the lowest amount of randomization in all five segments. PaX ASLR comes in the middle, with significantly better stack and `mmap` randomization. ASLP comes out ahead by at least 4 bits in all segments and provides randomization on the code and data segments with 20 bit randomness. This means that there are 2^{20} possible locations for placing code and data segments in 3GB user memory space.

We use the PaXtest results to estimate the probabilistic protection provided by ASLP against the de-randomization attack. The goal of de-randomization attack is to find out the randomized program memory layout. A recent de-randomization attack shown in [23] shows that it can guess the correct location of the shared library by brute-froce guessing in 216 seconds. Assuming a random distribution, the address can be guessed in a number of guesses equal to half of the possible locations. Knowing both the reported attack duration of 216 seconds to brute-force guess the address of the shared library region and the number of possible locations in this region from Table 2.2, we estimate the average guess rate. Equation 2.1 shows the calculation of guess rate.

$$\frac{2^{20}\,_{\text{possible locations}}}{2} \rightarrow \frac{524288\,_{\text{average guesses}}}{216\,_{\text{seconds}}} = 2427.2\,_{\text{seconds to guess on average}} \tag{2.1}$$

Using the rate of 2427.2 guesses per second derived from equation 2.1 and the bits of randomness returned from PaXtest in Table 2.2, we can calculate the amount of time required to brute force the randomization in each memory region for ASLP. It takes about 3 weeks to guess correct location of heap by brute force searching. The stack takes 10 days

to de-randomize. Mmap, code, and data segments cause lowest amount of time–about an hour. However, code and data segments would take additional time to brute force due to the fine-grained randomization of functions and data variables.

Against ASLP, a de-randomizing attack would take a considerable amount of time for constant guessing. However, we do not claim that ASLP is an effective deterrent to prevent a determined attacker from penetrating a single randomized target. In such micro-security level perspective, the de-randomizing attack by Shacham et al [23] can still succeed in about an hour to guess the mmap region. Instead, we argue from a macro-security perspective that ASLP provides a mechanism by which the memory corruption attack on large machines can be slowed to a rate that allows intrusion detection systems and system administrators to respond.

Consider the effect of address randomization at the macro-security level: a large scale Internet worm propagation. With ASLP, the speed at which worms can spread using memory corruption attacks is bounded not by how fast they can reach vulnerable hosts, but by how fast they can de-randomize the randomized address space of each target.

Ideally, the current fastest known spreading worm, The Sapphire/Slammer [30], is able to infect 100% of vulnerable hosts in less than a minute by doubling in size every one second with no randomization address space. We calculate the worm propagation rate of each randomization technique based on the infection rate of Sapphire/Slammer worm and the probabilistic protection of each technique discussed above. For Exec-Shield, 100% infection occurs in just over four minutes (4.275 minutes); for PaX ASLR the time is just over one hour (68.4 minutes). Our ASLP approach is able to delay 100% infection for over eighteen hours (1,083 minutes). This extension of infection time illustrates the benefit of having effective address space layout protection because fast worms that exploit memory corruption vulnerabilities must first get through address randomization before they can compromise their targets. Further, the barrage of attempts to guess the correct address should be visible by intrusion detection and prevention systems. Increasing probabilistic protection means forcing attackers to make more guesses, effectively giving intrusion detection systems a bigger target.

2.3.2 Performance Evaluation

Our goal is to incur comparable or less performance overhead to related techniques. We compare our approach with two other popular ones: PaX ASLR and Exec-Shield. It should be noted that both PaX ASLR and Exec-Shield can be configured to do more than address randomization. Where possible, we disable their extra features to make them as similar to ASLP as possible.

The various kernels are tested by installing each of them on a single test computer and selecting the desired kernel at boot time. The test computer runs Red Hat Linux 9.0 with a 2.66 GHz Intel Pentium 4 CPU, 512 MB of RAM, and a 7200 RPM ATA IDE hard disk drive. All benchmark suites were compiled using GCC version 3.4.4.

The vanilla kernel establishes the baseline by which the absolute performance impact of other configurations can be measured. Since PaX ASLR and Exec-Shield are closely related works that provide kernel-based address layout randomization, their performance provides a metric to determine if ASLP does or does not have comparable performance with kernel level randomization. For user level randomization, we compare ASLP with Position Independent Executable(PIE). PIE is used in PaX ASLR to move the static code and data regions from their traditionally fixed positions. Since PIE is a compilation option and not a kernel modification, we test it using a vanilla kernel and compile the benchmark suite with the PIE option.

We employ three popular benchmarks to measure the performance of each configuration: SPEC CPU2000 [31], LMBench micro-benchmark [32], and Apache Benchmark [33].

SPEC CPU2000 Benchmark The SPEC CPU2000 Integer benchmark suite is a set of computationally intensive integer applications that simulate the workload of scientific computation. Each benchmark in the CPU2000 suite measures the time required to complete a different integer computation. For this reason, we use the CPU2000 benchmark to measure the impact of address permutation on computational efficiency. Table 2.3 gives the result of CPU2000 Integer benchmark[2].

For the kernel level randomization comparison, Exec-Shield has an average per-

[2] The full CPU2000 Integer benchmark suite contains 12 benchmarks. One of the 12, the "eon" benchmark, did not compile on GCC version 3.4.4. Although this GCC version was newer than that recommended by the benchmark documentation, a GCC compiler of at least version 3.4.4 was needed to support PIE linking. We therefore elected to exclude the eon benchmark from the final results.

Table 2.3: SPEC CPU2000 Benchmark Run Times (seconds)

Benchmark	Vanilla	Exec-Shield	PaX ASLR	PIE	ASLP
gzip	177	178	176	207	177
vpr	284	284	284	298	284
gcc	132	134	131	148	133
mcf	408	413	409	427	410
crafty	116	116	117	142	116
parser	266	268	266	281	267
perlbmk	168	166	166	248	166
gap	126	128	125	144	128
vortex	185	185	185	218	187
bzip2	260	257	257	285	256
twolf	514	505	506	525	504
Total	2636	2634	2619	2923	2628
Avg. Overhead(%)	0	0.14	0	14.38	0

formance overhead of 0.14%, but PaX ASLR does not incur additional overhead. For the user level randomization comparison, PIE shows 14.38% average performance overhead. The overhead of PIE mainly comes from additional instructions that need to resolve actual memory addresses of program objects during the run time. Recent work by Bhatkar et al. in [14] also took a similar indirection approach to randomize static code and data segments. Such indirection causes an average runtime overhead of about 11% in their experiments. ASLP shows no performance overhead which supports our claim that ASLP has computational performance overhead comparable to closely related works.

LMBench benchmark The LMBench benchmark suite differs from CPU2000 because it strives to benchmark general operating system performance rather than the computational performance of a set of applications. The LMBench operating system benchmark suite consists of five sets of micro-benchmarks, each of which is designed to focus on a specific aspect of operating system performance. We only consider kernel level permutation in this evaluation since LMBench benchmark targets at an operating system's performance overhead rather than an application's performance overhead. The result shows that the process creation overhead is the primary source of expected overhead from address space randomization techniques like PaX ASLR, Exec-Shield, and ASLP, because additional op-

erations are inserted into the process creation functions. ASLP slows down `fork()` and `exec()` operations by 6.86% and 12.53% respectively. Both PaX ASLR and Exec-Shield have consistently higher overheads for the same tasks: 13.83-21.96% and 12.63-32.18% respectively. The context switching overhead results give a similar story to process operation results. ASLP caused 3.57% which is less than Exec-Shield and PaX ASLR. Their results are 8.15% and 13.8% respectively. In file and virtual memory (VM) system latency results, ASLP and PaX ASLR incur 12% overhead for mmap latency due to the increasing number of instructions to complete mmap allocations. However, average overheads of file and VM operations are very low in all three techniques. We do not consider local communication overhead from LMBench benchmark result, since we made no changes to networking code. Instead, we have the Apache benchmark to evaluate remote communication performance overhead.

Apache Benchmark The Apache Benchmark [33] measures the performance of an Apache HTTP server [34] in terms of how quickly it can serve a given number of HTML pages via TCP/IP. Our Apache Benchmark configuration makes 1 million requests, in simultaneous batches of 100, for a static HTML page of 1881 bytes, which includes 425 bytes of images. The result shows that only PIE incurs major overhead, about 14%. The other techniques, including ASLP, shows less than 1% overhead.

2.4 Limitations

Although ASLP can mitigate various types of memory corruption attacks, it is not a complete solution yet. Current implementation of ASLP does not support stack frame randomization. This may allow attackers to exploit a return-to-libc attack as described in [23]. We can mitigate such attack by adding pads among elements in the stack [13], but the randomization is limited and it wastes memory spaces. Further investigation is required to find a better solution. Another limitation is that ASLP might require re-linking or recompilation of source codes if a program (executable binary) does not have relocation information. Although our goal is to perform randomization without source code access, current implementation requires the relocation data from the compiler. However, we believe that this is a one time effort for life time benefit, since once the relocation data is included,

we can randomize the program repeatedly to thwart real-life attacks. Finally, the bug information that ASLP provides can be limited when the program crashes upon attacks and the entire stack is already corrupted. In this case, only the last executed program point where causes the crash will be provided.

2.5 Summary

This chapter presents Address Space Layout Permutation (ASLP) that provides a protection mechanism for achieving software integrity while they are used. ASLP randomizes all sections in the program memory space at process initialization times. This makes attackers difficult to find the correct locations of memory objects that they want to access or modify. As a result, ASLP can thwart the class of memory corruption attacks which is one of most common type of attacks in current network computing environment. In order to randomize the program memory space, we develop a novel binary rewriting tool that allows users to permute static code and data segments with fine-grained level. We also modified the Linux operating system kernel to randomize the stack, heap, and shared libraries. Combined together, ASLP permutes the program's memory layout completely within 3 GB user space memory. We use various types of benchmarks to evaluate ASLP and the result shows that it is possible to achieve better randomization for process virtual memory layout without incurring obtrusive performance overhead. The performance overhead of ASLP is low as compared with other address space randomization techniques that provide less randomization. With ASLP, runtime overhead is less than 1%, as indicated by both the SPEC CPU2000 Integer Benchmark as well as the Apache Benchmark.

Chapter 3

Vulnerability Identification Mechanism

In this chapter, we investigate a mechanism that can automatically identify integrity vulnerabilities in a program. Having such mechanism is helpful in discovering potential vulnerabilities in programs since understanding security bugs in a vulnerable program is a non-trivial task, even if the target program is known to be vulnerable. This mechanism is also useful in developing integrity patches for vulnerable programs where applying security patch is increasingly common in these days.

A program can be identified as vulnerable to integrity breaches in various ways (e.g., address space randomization techniques [5, 6], Common Vulnerabilities and Exposures [7]). However, understanding security bugs in a vulnerable program is a non-trivial task, even if the target program is known to be vulnerable. Though there are existing offline debugging tools (e.g., gdb, Purify [35]) to facilitate the vulnerability analysis and debugging process, human developers still need to manually trace the program execution most of times. This makes security debugging a difficult and tiresome task even for experienced programmers.

A critical step in the security debugging process is the analysis that recognizes the vulnerable point in a program and identifies the cause of the bug. A major challenge in this analysis process is to identify how unknown vulnerabilities can be exploited, since these vulnerabilities are often short-lived during the exploit process and thus are difficult to trace. Static analysis techniques (e.g., [36, 37, 38]) have been proposed to discover vulnerabilities

in a program via source code analysis (e.g., model checking). However, such methods are mainly focused on known vulnerabilities and often require human interventions to write assertions or specifications. In addition, some require access to source code, which is not always available (e.g., commodity software). Runtime checking methods (e.g., [35, 39, 40, 41]) have also been studied to detect security bugs dynamically by checking legitimate memory accesses, inserting canary values, or using additional hardware support. These approaches, however, still mostly focus on existing vulnerabilities and suffer from non-negligible false alarms and performance overhead.

A few approaches have been investigated recently to discover security bugs using statistical program invariants [42, 43, 44]. These approaches detect bugs by identifying deviations from program invariants statistically derived during normal program executions. DAIKON [44] and DIDUCE [43] automatically extract likely program invariants among variables through multiple normal program executions in the training phase, and use the violations of such invariants to detect security bugs at analysis time. AccMon [42] introduced Program Counter (PC)-based invariant detection (i.e., a memory object is typically accessed only by a small set of instructions), and detects security bugs when memory objects are accessed by instructions not observed during normal program executions in the training phase.

These approaches extended the capability of analyzing unknown security bugs. However, they still suffer from several limitations. In particular, all these approaches require a training phase during normal program executions to derive the program invariants, which offers no guarantee in generating all program invariants. Moreover, each of these approaches have additional limitations. For example, DAIKON [44] requires access to the program source code, while AccMon [42] requires hardware supports not available on modern computer systems.

In this chapter, we identify another type of program invariants called *program structural constraints* (or simply *structural constraints*), which are complementary to the above program invariants. Unlike the program invariants used by the previous approaches (e.g., data variable invariants used by DAIKON [44] and DIDUCE [43]), such program structural constraints are satisfied by all binaries produced by certain software development tools. Moreover, these program structural constraints can be *statically* and *entirely* extracted from program binaries. Thus, no training phase is required, and it is guaranteed

that we can extract all instances of program structural constraints.

Based on program structural constraints, we develop an automated offline security debugging tool named *CBones* (*SeeBones*, where *bones* is an analogy of program structures). CBones automatically extracts program structural constraints from a program binary, and verifies these constraints during program execution to detect and isolate security bugs. Compared with the previous approaches [44, 43, 42], CBones provides several unique benefits:

- Complete automation. CBones extracts program structural constraints (invariants) via static analysis of the compiled program executable. This has two implications. First, CBones does not require any training phase, which differentiates CBones from most runtime monitoring tools (e.g., AccMon [42], DIDUCE [43], DAIKON [44]). Second, CBones does not require manual specification as an input. This differentiates CBones from most static analysis and model checking tools (e.g., [36, 37, 38]).

- No access to source code or additional hardware is required. CBones dynamically instruments the program binary using Valgrind [45] during the program executions. Thus, it does not need to access the source code (in comparison with DAIKON [44]), nor does it need additional hardware support (in comparison with AccMon [42]).

- No false alarms. Since the program structural constraints should be satisfied by all the binaries produced by the compiler of concern, violation of any of them during runtime indicates a bug. Thus, the proposed approach produces no false alarms. Moreover, as indicated in our experiments, CBones can detect some data attacks recently discovered in [24], which many other approaches fail to recognize.

- CBones provides both coarse-grained and fine-grained debugging modes, which can be combined to isolate security bugs efficiently. The coarse-grained mode allows a user to quickly zoom into a region that has a security bug, and the fine-grained mode enables the user to pinpoint the bug.

To validate the practicality and effectiveness of the proposed approach, we evaluate CBones with 12 real-world applications that have different types of vulnerabilities. The result shows that CBones can discover all 12 security bugs with no false alarms. More-

over, CBones can pinpoint the corrupting instruction points precisely, which is critical in understanding how an attack exploits a security bug.

The rest of chapter is organized as follows. Section 3.1 describes the program structural constraints and the debugging techniques based on these constraints. Section 3.2 discusses the implementation of CBones. Section 3.3 presents the experimental evaluation of CBones. Section 3.4 discusses related work, and Section 3.5 concludes the chapter.

3.1 Security Debugging Using Structural Constraints

A process's virtual address space is divided and used according to the operating system and the compiler with which the program was compiled. Furthermore, each segment of memory is usually logically divided into smaller chunks that represent memory objects in the program or metadata used by the compiler to determine the program state. In this study, we use the structural properties that these memory objects or metadata always satisfy to derive program structural constraints for each program. These constraints can be verified at runtime to ensure that the program complies with the assumptions made by the operating system and/or the compiler. Violations of these constraints thus indicate the existence of security vulnerability.

CBones uses a combination of static analysis and dynamic monitoring to accomplish its goal. Static analysis is used to automatically determine structural constraints for a given program. These constraints are then passed to the dynamic monitoring component which verifies whether the constraints are satisfied during execution. Any violation of these constraints signals a misbehavior, which indicates an attack or a bug. For convenience, we refer to the static analysis portion as *Constraint Extractor*, and the dynamic monitoring part as the *Monitoring Agent*.

In this study, we focus our attention on Linux operating systems running applications written in C and compiled with the GCC compiler [46]. Our set of constraints include those generated for the operating system and compiler, and some others generated for the standard C library. It is worth mentioning that similar structural constraints can be generated for different operating system platforms, compilers, etc. One could derive a set of structural constraints for other platforms by examining how the program is structured at different stages (e.g, compile time and and load time) and finding the structural invariants

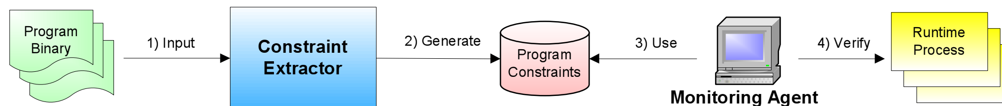


Figure 3.1: Security Debugging Process with CBones

that the program must satisfy. Indeed, the proposed method is applicable to a broad range of operating systems and compilers despite our choice in this study.

Figure 3.1 shows the security debugging process used by CBones. The binary executable is first analyzed by the Constraint Extractor to determine the structural constraints. The set of constraints are then passed to the Monitoring Agent along with the program executable. The Monitoring Agent executes the program and checks for any constraint violations, possibly with previously captured malicious inputs exploiting one or more security vulnerabilities of the target program. If a structural constraint is violated, the execution is halted and an error message is generated. The error message states the violated constraint, outputs the program state, and indicates the instruction responsible for the violation.

In the following subsections, we first present our structural constraints, and then show how these structural constraints can be used in security debugging.

3.1.1 Program Structural Constraints

The Linux executable file format ELF (Executable and Linkable Format) [26] has a typical virtual memory layout as shown in Figure 3.2. Although some of the addresses can be user-specified, by default, the program code, data, stack and other memory segments are located as depicted in the figure and the ordering of these segments are fixed. For example, the stack segment is always at a higher address than the heap and the code segments, and the heap is always higher than the code and the data segments. We present our program structural constraints with respect to these segments, namely the stack, the heap and the data segments.

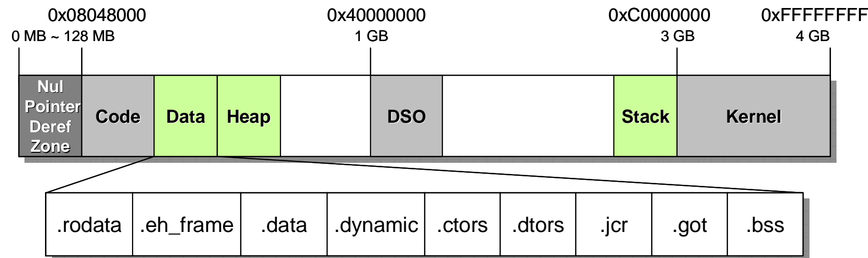


Figure 3.2: Example of an ELF Program Runtime Process Memory Layout

Stack Constraints

The stack is perhaps the most commonly attacked segment of the virtual memory. For each procedure called, an activation record of an appropriate size is pushed onto the stack. Figure 3.3 illustrates a simple program and a simplified stack layout after function `main` calls `cal_Sum`, which in turn calls `get_Num`. The figure also shows the various logical subdivisions in the activation record. For x86 architectures, the special register `$ebp` points to the return address in the activation record of the last function called. *Arguments* hold the values of the arguments that the caller passes to the callee when making the function call. *Return Address* is where the execution will return once the function completes execution. *Previous Frame Pointer* is the saved `$ebp` register value. When the function returns, the `$ebp` value is restored to point back to the previous activation record of the caller function. Sometimes additional registers can be pushed onto the stack to free up more registers for the current function. These registers are saved in the *Saved Registers* area and are restored upon function return. The rest of the activation record holds the local variables of the function.

The return address has been the most frequent target for attackers; however, a recent attack trend study [4] shows that other elements in the stack (e.g., frame pointer, saved registers) have also been exploited [47, 48, 49]. Such stack-based exploits require illegal modification of the stack structure. Therefore, security bugs in a program can be detected by monitoring the structural changes of the stack during program execution. Next, we present the program structural constraints for the stack region.

Caller-callee relationship constraint: When a function A (caller) calls another function B (callee), we say that A and B have a caller-callee relationship. A given function

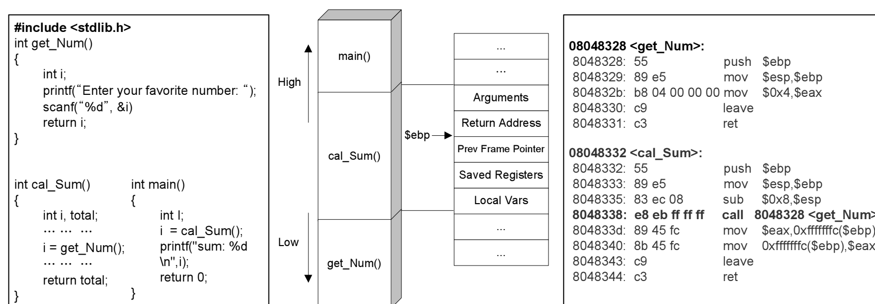


Figure 3.3: Example of Stack Structure and Caller-callee Relationship

generally calls a small number of other functions. We can find out all possible caller-callee relationship between functions, and represent such relationships in a *call-graph*, in which all functions are represented as nodes, and a possible caller-callee relationship is represented as a directed edge from the caller to the callee. The constraint here is that, at runtime, every function call should represent an edge in this call-graph.

Return address constraint: Return address is a critical element in an activation record, since it changes a program's control flow. A valid return address should always point to an executable region such as the code section or the executable section of the dynamically loaded libraries.

Frame pointer constraint: The frame pointer (`$ebp` register) is a reference point for the stack frame and is used as a base address to access memory objects in the stack that are defined as offsets from this pointer. The invariant is that the value of the frame pointer should not be changed during a function's execution once it has been set in the function's prologue. The frame pointer is changed at a function's epilogue (`leave` instruction) according to the IA-32 instruction set manual [50].

Saved registers constraint: Saved registers generally hold function pointers or values that refer to memory objects in a program's runtime process. Although saved registers are not critical to the program's control flow, they can be used as a bridge to a successful exploit. For example, one of the data attacks in [47] uses an address register(`$esi`) to execute an attacker-provided command. The saved registers constraint is that their values should not be changed during a function execution once they are set at the function's prologue.

Saved frame pointer constraint: Upon a function call, the caller's frame pointer is saved in the callee's activation record. Since the frame pointer is held in the `$ebp` register, the constraint for saved registers applies directly. However, due to its special use, we found more constraints for the saved frame pointer. First, the saved frame pointer should point to a higher memory address than where it is stored. Second, since a frame pointer actually points to the saved frame pointer in the activation record, one should be able to walk the stack (following the saved registers as a linked list), and traverse every activation record. Finally, at the top of the stack, the saved frame pointer for function `main` (or the alternate entry function) should be NULL.

Frame size constraint: The content of the activation record is determined during compile time. Therefore, at runtime, the activation record size for a function should remain constant until the function returns.

Stack limit constraint: The maximum size of a stack can be increased or decreased depending on a program's needs. The two register values, the stack pointer (`$esp`) and the frame pointer (`$ebp`) should be in-bounds during execution (i.e. point to the valid stack region). The default stack size limit in GCC is 8 MB, but it is a soft limit that can be changed during link time with the `-stack_size` linker flag or during runtime via the `setrlimit` system call. Therefore, the size of the stack and any changes made to it during runtime need to be determined.

Alignment constraint: The GCC aligns the stack as much as possible to improve a program's performance. The default alignment is word aligned, but a user can choose other alignment options to adapt different computing environments such as 64 bit or 128 bit. Therefore, each stack frame should be aligned according to the alignment option.

Heap Constraints

A powerful feature of the C language is its dynamic memory management. Generally, dynamic memory is manipulated via the `malloc` family of library functions. There are various dynamic memory management schemes, Lea [51], Hoard [52], and OpenBsd [53] to name a few. In this study, we follow Lea's scheme, which is used in Linux systems, to derive our heap based program structural constraints. Lea's scheme uses boundary tags to manage allocated and freed memory chunks. Each memory block is preceded by a small bookkeeping region that stores the information about the block in use, or in the case of

available blocks, the next available block. Figure 3.4 shows an example heap structure. A boundary tag of allocated chunk (e.g., `DATA(A)`) includes information about its size and indicates whether the chunk is in-use with the last bit of the size field (`PINUSE_BIT`). If a chunk is free, the tag includes pointers to the previous and the next free chunks. We are able to identify six structural constraints for the heap segment.

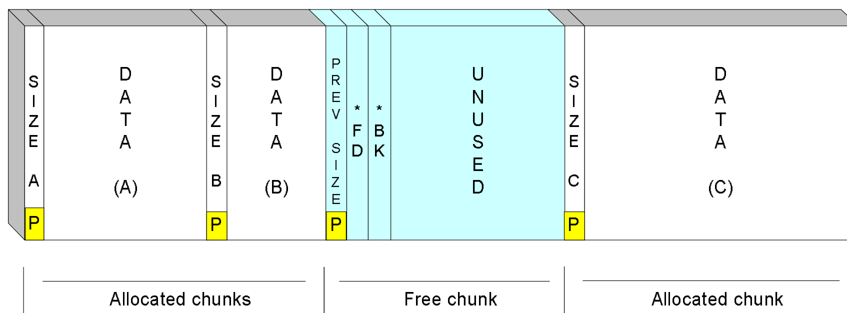


Figure 3.4: Example of Heap Structure

Memory allocation/deallocation requests constraint: Since the structure of heap is changed dynamically by `malloc` related function calls, the first thing we need to check is whether such memory allocation/deallocation requests are made from valid locations. The idea is similar to the caller-callee relationship constraint in the stack, since we verify valid structural changes in a heap using caller-callee relationships for `malloc` family of function calls.

The heap constraints are checked for programs that use the C library to manage their heap. Some applications use their own memory management utilities, either by allocating a large chunk through `malloc` and managing the block themselves or using the `brk` system call to eliminate the library entirely. In this case, CBones does not check heap constraints. It is worth noting that such systems are not vulnerable to metadata manipulation like the C library is.

Boundary tag (metadata) constraint: Boundary tags, or hereafter referred to as metadata, are used and manipulated by the library that implements the memory management scheme. In our case, the metadata should only be modified through legitimate calls to `malloc` family of functions.

Allocated memory chunks link constraint: The metadata allows the mem-

ory manager to traverse the list of allocated or available memory blocks as a linked list. Therefore, a verification program should be able to traverse the same list using the metadata. A broken link between allocated memory chunks indicates the corruption of the heap structure.

Heap boundary constraint: Dynamic memory allocation must be performed within the given boundary of the heap memory range in a program's runtime process. It is possible for an attacker to expand the heap size up to the kernel's memory space to shutdown the target system or to escalate the privilege by making invalid heap allocation requests. For example, a real threat has been found in the previous Linux kernels [54] that exploits no boundary checking in the kernel's `do_brk()` function so that an adversary can expand the heap area to the kernel space to take control of the affected system.

Chunk boundary constraint: Heap memory chunks always begin on even word boundaries. When the program requests n bytes, Lea's memory allocator provides more than $8+n$ bytes to keep the heap memory area aligned. Consequently, the returned pointer (beginning of the memory chunk) of `malloc` function call should be aligned with even word boundary.

Consecutive free chunks constraint: Upon deallocation of a memory block, before the memory block is added to the linked list of available memory, the memory manager looks for adjacent available memory blocks. If found, the two are merged together to form a larger available memory block. Therefore, after the memory manager returns from a `free` function call, the adjacent memory blocks should be in use. Having adjacent free chunks indicates a corruption of the heap structure and a heap corrupting attack. For example, double free attack puts fake free chunks with malicious forward (`*FD`) and backward (`*BK`) pointers to overwrite function pointers in the global offset table when the malicious chunk is freed.

Data Constraints

Although there are not many structural changes in the data section, we have found that some of the structural characteristics can help detect security bugs.

Shared library function pointers constraint: The global offset table (GOT) is a table of library function pointers. These function pointers initially point to the dynamic library loader and are updated via the loader when a library is loaded at runtime. Various

attacks have tried to exploit these function pointers (e.g., [55]). The structural constraint is very simple: the library function pointers should point to library functions.

Constructor function pointers constraint: Constructor section (`.ctors`) consists of pointers to functions that are called before the program begins execution. These constructor functions mainly check or initialize the runtime environment. It has been shown that a format string attack can change these pointers to hijack the control flow [56]. Being function pointers, these entries should point to the program's code section, not to stack or heap regions. Another observation is that these pointers are determined during compile time. Thus, once the program is loaded, they should remain constant.

Destructor function pointers constraint: Just like the constructors, the destructor function pointers point to the functions that are called once the program terminates. The structural constraints are the same as those of the constructor function pointers.

3.1.2 Security Debugging through Constraints Verification

As discussed earlier, CBones performs security debugging by verifying whether program structural constraints are violated at run time. To do so successfully, we have to answer two important questions related to the verification of these constraints: "what to verify", and "when to verify". We have described the program structural constraints in the previous subsection, which answered the first question. In this subsection, we clarify the second question, i.e., when to verify.

We first introduce some terms to facilitate our explanation. Most structural constraints state that a memory region should be constant. Obviously, we need to clarify the time frame during which such a constraint is valid. We define the *lifetime* of a memory region to be the duration from the time when the memory region is set for the current context to the time when the memory region is discarded or released. Take for example an activation record in the stack. We described that the return address and the saved registers are structural invariants and should be constant throughout the execution of the function. The lifetime of the memory region holding a saved register does not start at function call time, but rather after the function prologue has pushed the register value onto the stack and ends when the function call returns. A metadata's lifetime starts with the dynamic memory allocation and ends with the deallocation of the same memory region. For a data constraint, the lifetime of the `.ctors` and `.dtors` segments start with program execution

and end at program termination.

Our second term describes the program state. A program is said to be in a *transient state w.r.t. a structural constraint* if the memory region related to the structural constraint is being modified. For example, consider a heap memory region allocated in a program. The program is in a transient state w.r.t. the memory region from the time when a `malloc` family of function is called to its return. In general, for a heap related structural constraint, this transient time frame is the period from the call to a `malloc` family of function to the function return. For a stack related structural constraint, this time frame includes the period from the time a function call occurs to the time the function prologue sets up the activation record.

Most of the constraints are based on memory segments that are dynamic. The stack changes with every function call/return, and the heap is modified with every memory allocation/deallocation. In theory, we can verify all the structural constraints continuously at every instruction. Indeed, any structural constraint that relies on a specific memory region can be checked at any given time, provided that the program is not in a transient state w.r.t. that constraint. However, such an approach will introduce substantial overhead, which is prohibitive in practice. On the other hand, the structural constraints related to a memory region must be checked at least before the memory region becomes inaccessible, so that potential violation of structural constraints will not be missed.

A simple solution is to perform *coarse-grained* constraint verification. That is, to verify the structural constraints before function returns and memory deallocations, since both the activation record for the returning function and the metadata of the dynamic memory region will become inaccessible after these actions. This allows us to capture violations of program structural constraints (as long as the exploit does not "fix" such violations). However, we will not be able to pinpoint the instruction that causes the constraint violation. This is certainly highly undesirable in security debugging.

We propose to use a two-stage approach in CBones to address the dilemma between unacceptable performance overhead and inaccuracy in identifying the constraint violations. In the first stage, CBones narrows the location of a constraint violation point down to a single function call, and then in the second stage, it determines the constraint violation point precisely.

Specifically, in the first stage, CBones is executed in the coarse-grained debugging

mode, where the CBones monitoring agent verifies the structural constraints before function returns and memory deallocations. CBones then identifies the function call after which a constraint violation is detected. CBones is then re-executed with the same target program and input for the second time to start the second stage. In order to obtain more information and provide the instruction responsible for the corruption, in the second stage, CBones switches to a fine-grained debugging mode when it reaches the function call identified in the first stage. CBones then monitors all the memory writes during the function call. If a memory write instruction causes the constraints to be violated, CBones raises a flag and outputs the instruction attempting to corrupt the memory. As discussed earlier, the fine-grained debugging mode incurs high performance overheads; CBones works around this problem by only performing fine-grained monitoring during the function call identified in the first stage.

3.2 Implementation

As mentioned previously, utilizing a combination of static analysis and dynamic monitoring techniques, CBones consists of two components, the static analysis component called *Constraint Extractor*, and the dynamic monitoring component called *Monitoring Agent*. In the following, we describe the two components in detail.

3.2.1 Constraint Extractor

Written as a Ruby script [57], *Constraint Extractor* utilizes a number of programs and scripts to extract a program's structural information and constraints from the target program binary. Since a C program is generally structured by multiple user-defined functions including `main`, *Constraint Extractor* first extracts each function's information such as its name, address, activation record size and number of saved registers from the debugging information included in the program binary. In order to obtain the information from the binary code itself, *Constraint Extractor* takes as input the program executable compiled with the debugging flag (`-g`) and without any optimization. It uses `dwarfdump`, a publicly available C program that outputs the debugging information in DWARF format and we wrote a parser called `dwarf_parser` in Ruby to parse `dwarfdump` output.

To derive the caller-callee constraints from the target program, *Constraint Ex-*

tractor uses `objdump` to disassemble the program executable and extract all the `call` instructions, then parses the instruction number and the procedure name, and adds the next instruction number to the valid return address list of the procedure. Finally, the Constraint Extractor outputs the procedure information in a text file.

Note that *Constraint Extractor* takes as input an ELF binary that is compiled without any optimizations. The reason for a non-optimized binary requirement is that *Monitoring Agent* works based on a number of assumptions such as the `$ebp` being used as a frame pointer and the specific protocols during function calls and returns. When optimized, binaries may invalidate these assumptions, causing false positives and/or negatives to occur. We leave the optimized binaries and seamless integration with commodity software as future work.

3.2.2 Monitoring Agent

The CBones Monitoring Agent is responsible for verifying the program structural constraints and reporting any violations. We implement the CBones Monitoring Agent as a Valgrind skin. Valgrind is an open-source CPU emulator which has been used for debugging and profiling Linux programs [45]. When a program is run under Valgrind, Valgrind executes the program one basic block at a time. It initially translates the instructions in a basic block to an intermediate assembly-like language called UCode. The UCode basic block is then passed on to the `SK_(instrument)` function defined in the skin, which allows programmers to instrument the binary code by calling their own functions or altering the basic block. The instrumented UCode block is translated back into machine language for execution.

The Monitoring Agent uses some internal data structures to store the procedure information provided by the Constraint Extractor and to keep program state during debugging. A `Procedure` data structure is created for every entry in the input file and a `CallStack` stores activation records during the runtime. The Monitoring Agent also keeps another data structure `ChunkList` to keep track of the dynamically allocated memory regions.

The Monitoring Agent uses a procedure called `cb_check_all_constraints` to verify all program structural constraints that are available. As discussed earlier, we cannot verify whether a structural constraint is violated when the program is in transient state w.r.t. this constraint. Since the only times when such cases may happen is when the

target program makes a function call or a call to a `malloc` family of functions, in coarse-grained debugging mode, the Monitoring Agent captures function calls to validate the structural constraints. Once the program leaves its *transient* state, the Monitoring Agent calls `cb_check_all_constraints` to verify all the remaining structural constraints. Moreover, the Monitoring Agent also marks a "safe point" if no constraint violation is detected. The last "safe point" before the constraint violation will be used in stage 2 as the indication to switch to the fine-grained debugging mode.

In the following we list the events of interest and the actions taken by the Monitoring Agent.

Function calls: The Monitoring Agent handles function calls in several stages. Initially, when a `jump` due to a `call` instruction is captured, the Monitoring Agent determines the caller and the callee and verifies the caller-callee relationship constraint. This constraint is only checked for client program functions and not the library functions. The second stage occurs when the callee sets its frame pointer. The Monitoring Agent creates a new activation record for the callee and adds it to the current thread's `CallStack`. The alignment of the frame pointer is also checked. The third stage only applies to procedures that save registers in their activation record. Once all the registers are pushed onto the stack, a snapshot of the invariant region in the activation record is taken and stored in the activation record. Since no further changes to the invariant region is expected, the program is no longer in its transient state. The Monitoring Agent calls `cb_check_all_constraints` to verify the other structural constraints, and marks a "safe point" if there is no constraint violation.

Function returns: When a procedure is returning, the Monitoring Agent captures the `jump` due to a `ret` instruction and verifies that the return address is a valid return address for the returning procedure. A function epilogue contains at the very least a `leave` and a `ret` instruction. The `leave` instruction, which precedes the `ret`, restores all the saved registers. Therefore, when the Monitoring Agent captures a function return, the registers, including the frame pointer, are already restored to the caller's values. Nevertheless, the activation record of the callee is still intact and can be examined. The Monitoring Agent verifies that the invariant region of the activation record is intact and removes the activation record of the returning function from the current thread's `CallStack`. It then calls `cb_check_all_constraints`, and marks the return address as a "safe point" if there is no

constraint violation.

`malloc/free` **calls:** The Monitoring Agent intercepts `malloc` family of function calls via wrapper functions. These function allow the Monitoring Agent to perform book-keeping on dynamically allocated memory regions. For each newly allocated memory, the Monitoring Agent first calls `malloc` to allocate the memory, and then creates a new `chunk` and add it to `ChunkList`. Two additional checks verify that the heap boundary constraint and the alignment constraint are satisfied. When a `free` call is intercepted, the Monitoring Agent first verifies that the metadata is intact. It then calls `free` and finally removes the `chunk` corresponding to the memory from `ChunkList`. During deallocation, the Monitoring Agent simply calls `cb_check_all_constraints` to verify that the metadata is intact. This is possible since the Monitoring Agent determines when to actually deallocate the memory, and hence the program is not in a transient state until it does.

Memory writes: When running in the fine-grained debugging mode in the second stage, the Monitoring Agent captures all memory writes by instrumenting the binary code. If the destination address belongs to any of the invariant regions in stack or heap, a flag is raised to mark the instruction attempting to violate the corresponding structural constraint. Capturing memory writes is not always trivial, since memory can be updated through system calls and kernel functions. The Monitoring Agent's current implementation captures system calls and performs the necessary checks before the memory is updated. In one of the test cases, a large memory copy operation is performed by manipulating the page table through the use of kernel functions. Since Valgrind cannot trace into kernel space, such a memory modification would go unnoticed. This means that the current implementation of CBones would not be able to determine the instruction responsible for the corruption. It should be noted that this does not mean that the attack is unnoticed.

We briefly mentioned in section 3.1.2 that not all program structural constraints need be checked to verify our list of constraints. We deferred this discussion previously to present the events and actions related to the Monitoring Agent first. For each activation record in `CallStack`, `cb_check_all_constraints` verifies that the invariant region of the activation record is intact. This ensures that the return address and saved registers, including the saved frame pointer, have remained constant, satisfying a number of structural constraints. We assume that if every activation record is intact, and each activation record is created by a legitimate instruction conforming to the call-graph, then other structural

constraints such as the linked list of saved frame pointers and frame sizes are also satisfied. A similar approach is taken with heap related structural constraints. Our assumption is that, as long as the metadata have not been modified by any means other than the library function calls that are designated for the task, the constraints are satisfied.

3.3 Experimental Evaluation

We performed a series of experiments to evaluate CBones, using a group of 12 real-world applications with known vulnerabilities and exploits. The objective of the experiments is to understand both the security debugging capability and the performance overhead in CBones.

In the following, we first illustrate how CBones is used for security debugging through a running example, then present our security evaluation aimed at understanding the security debugging capabilities provided by CBones, and finally describe the performance evaluation.

3.3.1 Security Debugging Using CBones: A Running Example

We use one of our test cases, Sumus, to demonstrate how to use CBones for security debugging. Sumus [58] is a game server with a stack overflow vulnerability. Figure 3.5(a) shows the snippet of the source code responsible for this vulnerability. The boldface line copies the contents of `p1` into the local array `tmdCad`. `p1` points to the string after `GET` in the HTTP request. The programmer assume that this input string have a small size. An attacker may exploit this assumption by sending a message longer than the size of `tmpCad`. At a first glance, this looks like a trivial stack overflow; the overflow should first corrupt the local variables and then the return address. However, as the buffer is overflowed, the instruction first overwrites `faltan` and then `kk`, which is actually used to index `tmpCad`. With a carefully designed input, the overwrite skips to the formal arguments' memory region, not overwriting the return address. This behavior of the attack makes it much more difficult and time-consuming to debug manually. Another important note is that this attack cannot be captured by systems looking for control-hijacking attacks alone, since the return address remains intact.

For comparison purposes, we first ran Sumus under gdb. However, *gdb* was only

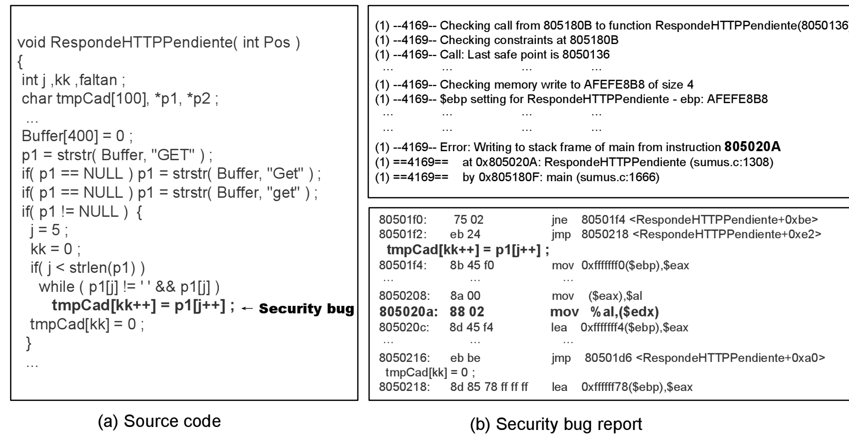


Figure 3.5: Example of Automated Security Bug Diagnosis Using CBones

able to capture the program's crash after `RespondeHTTPPendiente` calls the `send` function (not shown in figure). Therefore, it is non-trivial to locate the corrupting instruction using gdb, if not entirely impossible.

We then ran Sumus under CBones. CBones, however, was able to detect the exploit and determine the corrupting instruction fully automatically. Although this attack does not corrupt the invariant region of `RespondeHTTPPendiente`, it does corrupt the invariant region of the caller function `main`. Therefore, CBones was able to detect a violation of the caller-callee relationship constraint in the first stage. In the second stage, CBones started the fine-grained debugging mode from the last "safe point" (0x8050136), detected the (illegal) memory write into the return address in the stack frame of the `main` function, and raised a flag accordingly. Figure 3.5(b) shows the output of CBones. The error message clearly states that a memory write to an invariant region has occurred and displays the instruction number responsible for the memory write.

3.3.2 Security Evaluation

Our debugging example with CBones clearly reflects the effectiveness of using program structural constraints to discover security bugs in a vulnerable program. To further evaluate the effectiveness of our method, we used 12 real-world applications with various vulnerabilities. Seven applications have stack overflow vulnerabilities, three have heap overflow vulnerabilities, and two have format string vulnerabilities. The effectiveness is measured by

how precisely CBones can locate a security bug in a program.

Table 3.1: Security Evaluation Result

App Name	Vulnerability Type	Bug Detected?	Constraint Violation
streamripper	Stack overflow	Yes	Return address constraint
ghttpd	Stack overflow	Yes	Return address constraint
sumus	Stack overflow	Yes	Return address constraint
prozilla	Stack overflow	Yes	Return address constraint
monit	Stack overflow	Yes	Saved registers constraint
newspost	Stack overflow	Yes	Saved registers constraint
icecast	Stack overflow	Yes	Saved registers constraint
WsMp3d	Heap overflow	Yes	Boundary tag constraint
Xtelnet	Heap overflow	Yes	Boundary tag constraint
NULLhttp	Heap overflow	Yes	Boundary tag constraint
Power	Format string	Yes	Return address constraint
OpenVMPS	Format string	Yes	Return address constraint

Table 3.1 summarizes the evaluation result. In all tested applications, CBones captured constraint violations at the time of structural corruption and pinpointed the corrupting point (instruction) in the program, and raised no false alarms. The last column of Table 3.1 shows the constraint that was violated in each test case. Most stack overflow exploits violated either the return address constraint or the saved registers constraint. All heap overflow cases violated boundary tag constraint, in which the exploits overwrote metadata next to an allocated chunk in the heap. Two format string cases violated the return address constraint. Although we do not have malicious attacks to evaluate CBones with other types of attacks, it is worth noting that such attacks depend on the three types of memory corruption techniques mentioned here. For example, return-to-libc attack is a variation of stack overflow and malloc-free attack relies on a heap buffer overflow.

Discussion: False Positives and False Negatives

Our proposed approach guarantees no false positives, since all program structural constraints should be satisfied by all program binaries. The current version of CBones, however, does not reach the full potential of the proposal approach yet, due to a limitation in the implementation of Constraint Extractor. The current CBones Constraint Extractor cannot handle function pointers properly yet. A jump due to a function pointer usually

dereferences a register or an address in assembly code. Even though the Constraint Extractor detects these instructions, it does not include analysis procedures to derive all possible caller-callee relationships. As a result, the Monitoring Agent may raise a false alarm due to some missing caller-callee relationship. These false positives can be suppressed by marking `call` instructions that dereference values instead of using function addresses and suppressing the errors when they are generated at these instructions. We will extend the implementation of Constraint Extractor to handle such cases in our future work.

CBones is intended to automatically identify exploits of vulnerabilities that violate *program structural constraints*. Thus, it cannot detect exploits that are outside of this scope, for example, illegal modifications of the memory regions belonging to program variables. Such cases can be resolved using techniques and tools complementary to CBones, such as DAIKON [44] and DIDUCE [43].

3.3.3 Performance Overhead

We also performed experiments to understand how much CBones slows down the test applications, though such performance penalty is non-critical for debugging. For simplicity, we chose to use throughput as a performance metric. This performance metric certainly reflects the extra overhead due to CBones. However, it is not applicable to all the test applications. For example, the Power daemon checks the status of the UPS periodically and idles/sleeps for the rest of the time. As another example, Xtelnet is a telnet daemon, and its runtime or throughput depends on the client activity. Thus, in our experiments for performance evaluation, we focus on the subset of server/client applications in our test suite. All programs were compiled using GCC version 3.2.2 with debugging option (-g). Our testbed ran Red Hat Linux 9.0 with a 2.66 GHz Intel Pentium 4 CPU, 512 MB of RAM, and a 7200 RPM ATA IDE hard disk drive.

For comparison, we first ran the test programs without Valgrind, under Valgrind's Nullgrind skin, and finally under CBones. Nullgrind is Valgrind skin without instrumentation. Thus, this reflects the performance slowdown introduced by Valgrind. To better understand the performance impact of CBones, we run CBones in two modes. The first mode is the default, coarse-grained mode, in which the relevant program structural constraints are checked after function calls/returns and after `malloc` family function calls. The second mode is the fine-grained mode, in which CBones starts the memory monitoring from

the last “safe point”. Such a “safe point” is usually very close to the corrupting instruction. For each test program except for sumus, we measured the time it took to download two files with sizes 700KB and 12MB, respectively, and take the average of the slowdown factor over 15 iterations. As an exception, sumus uses multi-threads and only allows a small size file (up to 200KB) for data transmission. In this case, we used 200KB files.

Figure 3.6 shows the evaluation result. The Y-axis in the figure shows the slowdown factor of using Nullgrind and CBones compared to normal execution of each test program. For example, in ghttpd, Nullgrind incurs 1.88 times slowdown and CBones in the coarse-grained mode incurs 2.46 times slowdown compared to the normal execution. The overall average slowdown factors for CBones in the coarse-grained and the fine-grained modes are about 5.23 and 12.57, respectively, compared to normal program execution. It is worth noting that a good portion of the overhead came from our implementation choice (Valgrind), which incurs 2.44 times slowdown on average by itself (Nullgrind). CBones only incurs 68% additional overhead on average compared to Nullgrind. NULLhttp shows a exceptional slowdown compared to other programs. It shows about 5 times slowdown under Nullgrind, 15.21 times slowdown with CBones in the coarse-grained mode, and 53.97 times slowdown with CBones in the fine-grained mode. This significantly increases the performance overhead. Our further investigation indicates that this significant performance overhead is due to the large number of function calls and returns during the test process (e.g., 708,355 function calls in NULLhttp v.s. 1,224 function calls in ghttp when downloading a 700KB file). In contrast, the sumus test case shows very little overhead (1%), since sumus uses multi-threads and only allows a small size file (200K) for data transmission.

When the monitoring agent observed a constraint violation, we changed the constraint checking granularity to fine-mode using the last safe point reported from the monitoring agent. In such case, the average performance overhead can be increased up to 14.76 times compared to normal execution and 2.99 times compared to Nullgrind.

3.4 Related Work

A number of approaches have been proposed to provide automated debugging capability. iWatcher [41] uses expected access (e.g., read-only, read/write) to user-specified memory regions to check whether there is any access violation and triggers further investiga-

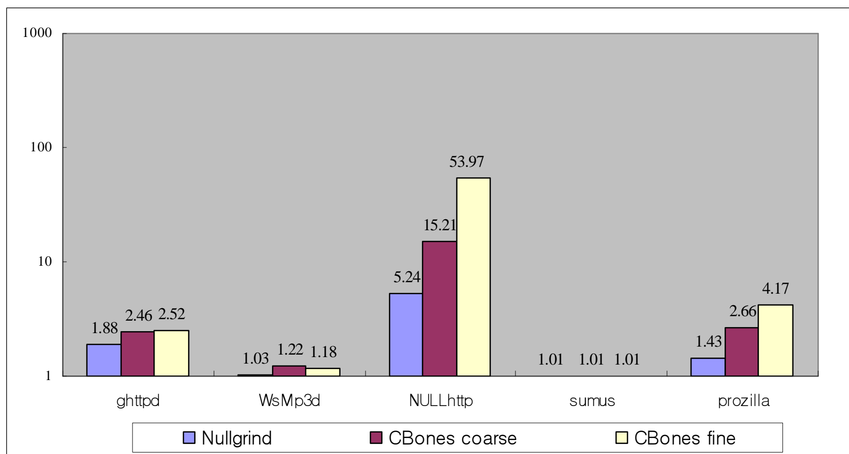


Figure 3.6: Slowdown Factors of Nullgrind and CBones

tion if there is any. Users can turn monitoring on or off at run time to reduce the overhead. However, iWatcher requires additional hardware support, such as modified L1/L2 cache, victim watchflag table, and range watch table, and thus cannot be used directly on common computer systems. AccMon [42] uses the program counter based invariants (i.e., each memory object is generally referenced only by a few instructions) to monitor program execution. AccMon incurs low overhead (0.24-2.88 times) using additional hardware support, including iWatcher [41] and Check Look-aside Buffer. AccMon monitors global objects, heap objects, and return addresses in the stack. To reduce false alarms, AccMon provides a confidence level to help users decide whether an alarm is likely to be a bug. Similar to iWatcher, AccMon cannot be used on modern computer systems due to the additional hardware requirement. DIDUCE [43] instruments Java byte code and monitors variables in a program to generate dynamic invariants. DIDUCE reports detected anomalies by checking the history of variable usage (comparing a variable's previous and current values). To reduce false alarms, it provides options for users to tighten or relax the properties. DAIKON [44] introduced variable-based invariants debugging. It instruments a program's source code and traces the program executions to find the likely invariants. It then generates pre/post conditions at certain program points such as functions or variables to check violations.

3.5 Summary

In this chapter, we present a vulnerability identification mechanism named as CBones that can automatically locate the vulnerable points in a program and provide the useful information to understand the security bugs. CBones automatically extracts program structural constraints from program binaries and analyze security bugs in vulnerable programs. Compared with the previous approaches, CBones provides several benefits: 1) full automation (neither training nor manual specification is required, 2) no need for source code analysis or instrumentation, 3) no requirement of additional hardware support, and 4) no false alarms. Our experimental evaluation with 12 real-world applications demonstrates that CBones can identify all security bugs automatically, without generating any false alarm or false negatives.

Chapter 4

Remote Dynamic Attestation Mechanism

The goal of remote attestation is to enable a computer to prove its integrity to a remote party. A computer will behave in the expected manner if the integrity of its components (e.g., applications) is intact. Remote attestation is an essential part of building trust and improving trustworthiness in networked computing environments. For example, if an integrity violation is detected, the violating component can be distrusted by other components in the system. Furthermore, the computer running the component can be excluded from the system to prevent possible error propagation, or to facilitate the forensic investigation if it is believed compromised.

Several remote attestation techniques have been developed recently to provide integrity evidence of an *attester* to a remote *challenger* by hashing static memory regions, sometimes combined with language specific properties or program input/output [59, 60, 61, 62, 63, 64, 65, 66, 67, 68]. However, all existing attestation approaches focus on the static properties of the computer system by providing hash values of static objects as the integrity evidence to the challenger. Although this *static attestation* provides a straightforward way to attest static objects in the system, it does not apply to dynamic objects in the system. (Hashing dynamic objects is not effective because their contents change dynamically.) Since a system can be compromised due to various runtime threats (e.g., remote buffer overflow attacks), a remote challenger still cannot gain high confidence in a system even if it is statically attested.

In order to address this problem, we introduce the notion of *dynamic attestation* in this research. A dynamic attestation is an attestation that provides the integrity evidence for the dynamic properties of a running system. The dynamic system properties are the properties that dynamic objects must satisfy during their lifetime. For example, even though stack is dynamically changing, a saved frame pointer in the stack must point to the caller's stack frame, and thus saved frame pointers must be linked together. In other words, dynamic system properties represent the valid runtime behavior of the attested system. Dynamic attestation can use such dynamic properties to verify the runtime integrity of the system, and provide integrity evidence to improve the confidence in system integrity.

There are some unique challenges in developing a dynamic attestation system. Firstly, due to the diversity in dynamic objects and their properties, it is non-trivial to identify and derive the "known" good states of the dynamic objects. In contrast, in static attestation, the known good state can be measured by the cryptographic checksum of static objects. Secondly, dynamic attestation needs to access a potentially large number of dynamic objects and measure their integrity. Such objects are often transient in nature, and thus require a versatile and efficient measurement mechanism. Thirdly, dynamic objects need to be monitored continuously as their values change. This differs considerably from the one-time check of static attestation techniques. Finally, dynamic attestation measures the objects that the adversary may have already modified, and the measurement process involves verification of dynamic system properties, which is more complicated than simply hashing the measurement targets as in static attestation. Thus, the attester must protect itself against potential indirect attacks from the adversary as well.

In this chapter, we present a novel dynamic attestation system named ReDAS (Remote Dynamic Attestation System) to address the challenges described above. We propose to perform an application-level dynamic attestation by attesting to the dynamic properties of running applications. As an initial attempt in our research, ReDAS provides the integrity evidence for two dynamic properties: *structural integrity* and *global data integrity* of running applications. These properties are automatically extracted from each application. They represent a program's unique runtime characteristics that must be satisfied during its execution. Thus, attacks modifying dynamic objects related to such properties (e.g., corrupting saved frame pointers) will lead to integrity (violation) evidence, which ReDAS will capture and report during attestation.

In order to deal with potential attacks against ReDAS, we use the hardware support provided by Trusted Platform Module (TPM) [69] to protect the integrity evidence. ReDAS runs in the OS kernel. As long as ReDAS sees an attack interfacing with the OS kernel, it seals the integrity evidence into the TPM right away, before the attack has any chance to compromise the kernel. Thus, even if an attacker can compromise the system, he/she cannot modify the integrity evidence without being detected by the remote challenger.

We have implemented ReDAS as a prototype system on Linux, and performed experimental evaluation of ReDAS using real-world applications. The experimental result shows that ReDAS is effective in capturing runtime integrity violations. We observe zero false alarms in our test cases and an average of 8% performance overhead.

Note that this work serves as an initial attempt for providing complete system integrity evidence through dynamic attestation, but it does not provide a complete solution for dynamic attestation yet. Several open questions remain to be addressed, such as the identification of all dynamic system properties for remote attestation and the determination of appropriate measurement points. Nevertheless, the results in this work have already advanced the state of the art of attestation (beyond static attestation), and can be used as a good starting point for additional research.

The contributions of this work are as follows. First, we raise the need of dynamic attestation and propose a framework for application-level dynamic attestation. Second, we implement a dynamic attestation system (ReDAS) on Linux to prove that runtime dynamic attestation is feasible by verifying dynamic properties. The dynamic properties being attested by the challenger are originally identified from previous works [44, 73]. We modify them to meet the requirements of dynamic attestation. Third, we use the hardware support provided by TPM to protect the integrity evidence, so that even if the attacker can compromise the OS kernel, he/she cannot tamper with the integrity evidence without being detected by the remote challenger. Finally, we perform a substantial set of experiments to evaluate the proposed techniques.

The rest of chapter is organized as follows. Section 4.1 presents the proposed dynamic attestation techniques. Section 4.2 describes the prototype implementation of the proposed dynamic attestation system. Section 4.3 presents the experimental evaluation of our approach using the prototype system. Section 4.4 discusses related work. Section 4.5 concludes the chapter and points out some future research directions.

4.1 ReDAS: A Remote Dynamic Attestation System

The ultimate goal of dynamic attestation is to provide complete system integrity evidence by measuring dynamic system properties. In this chapter, we focus on application-level dynamic attestation as a part of this effort. Specifically, we develop an application-level dynamic attestation system named ReDAS (Remote Dynamic Attestation System). As an initial attempt in our research, ReDAS currently provides two dynamic properties: structural integrity and global data integrity of running applications. In this section, we begin with assumptions and threat model of ReDAS. We then propose the attestation architecture for measuring the dynamic properties and discuss important issues regarding to dynamic attestation provided by ReDAS.

4.1.1 Assumptions and Threat Model

We assume that the attester's computer is equipped with a TPM and the BIOS supports the core root of trusted measurement (CRTM). The CRTM measures the computer's hardware configurations (e.g., ROMs, peripherals) and provides the root of trust for ReDAS. We assume that the attester's Operating System (OS) kernel is trusted at boot time. This can be ensured by using existing trusted boot techniques (e.g.,[70]). However, kernel may still be compromised at run time due to potential vulnerabilities that can be exploited remotely. (We plan to investigate how to attest to the runtime kernel's integrity in future work; it is out of scope of this chapter.) We assume that each application binary is measured before it is loaded into memory for execution. This can be done by using static attestation techniques (e.g., IMA[65]). As a result, any malicious changes in the application's binary before loading time will be visible to the remote challenger during attestation. We assume that the application's source code is available. Source code access is required to collect dynamic properties of applications. Finally, we assume that the attester and the challenger can establish an authenticated communication channel using the cryptographic support provided by TPM [71].

We assume that the adversary is capable of launching any types of remote attacks against the attester. Thus, a successful attacker can get full control (e.g., root) over the attester's system even though the kernel is trusted at boot time. However, we assume that the adversary does not have physical access to the attester's computer. In other words, the

attacker cannot launch hardware attacks (e.g., resetting the platform configuration register (PCR) values in the TPM without rebooting the computer [72]).

4.1.2 ReDAS Overview

As discussed earlier, we propose to use dynamic system properties for dynamic attestation. In the attestation process, the remote challenger is expected to send an attestation request to the dynamic attestation service running on the attester's system. The dynamic attestation service then provides the integrity evidence of the attester's system to respond to the challenger.

In order to provide the dynamic attestation and the required integrity measurement, we propose to have three components in ReDAS: 1) *dynamic property collector*, 2) *integrity measurement component*, and 3) *attestation service*. Figure 4.1 illustrates the ReDAS architecture.

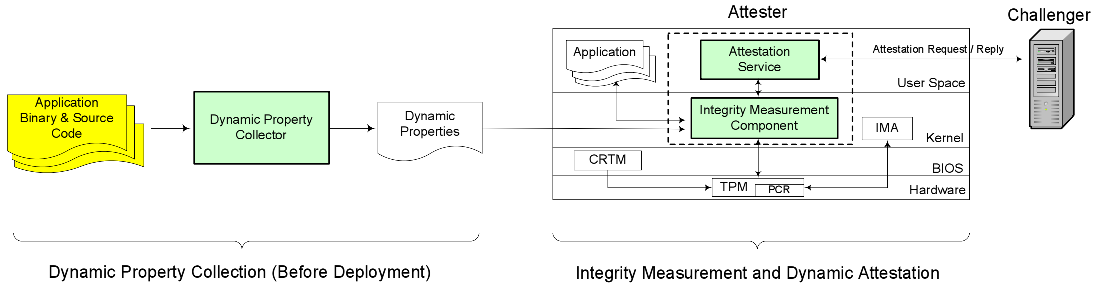


Figure 4.1: ReDAS architecture

The dynamic property collector is an off-line component that collects dynamic properties from the target applications' source code and binary. The integrity measurement component and the attestation service form the *ReDAS runtime system* (shown in the dashed box in Figure 4.1). They take the dynamic properties extracted by the dynamic property collector, and runs on the attester to perform integrity measurement and provide dynamic attestation service. To ensure only trusted software is loaded into the attester, we assume the CRTM, trusted boot technique [70], and IMA [65] along with ReDAS to provide the attester's load-time integrity measurement.

In order to securely handle the attestation request and reply, ReDAS uses a stan-

dard attestation protocol commonly used in static attestation. The only difference is that in ReDAS the reply message includes violations of dynamic properties rather than static hash values. The details of the attestation protocol can be found in Section 4.1.6.

In this chapter, we focus on two types of dynamic properties as an initial attempt. In the following subsections, we explain these dynamic properties, the runtime integrity measurement, the protection of integrity evidence against potential attacks targeting ReDAS, and security analysis.

4.1.3 Dynamic System Properties

As mentioned earlier, dynamic system properties are the properties that dynamic objects in the system must satisfy during their lifetime. Finding a complete set of dynamic system properties is a non-trivial task due to the diversity in dynamic objects and their properties. As an initial effort for dynamic attestation, we identify two types of dynamic properties that can be practically used in dynamic attestation: *structural integrity* and *global data integrity*. There are other simple dynamic system properties that can be used in ReDAS easily (e.g, the canary word in stack [40]). However, the challenging problem is how to find the set of all dynamic system properties that can be verified efficiently at runtime. We will investigate this problem in our future work.

Structural Integrity

It is observed in [73] that even though memory objects in the writable sections of a running application are dynamically modified, they still follow certain patterns, particularly when the application binary is generated by a modern compiler such as GCC. For example, the return address of a function in the (dynamic) stack must point to the instruction next to the `call` instruction in the calling function. As another example, memory chunks allocated in the heap must be linked together if they are allocated via `malloc` family functions. Such patterns are referred to as *structural constraints* [73]. We say an application (execution) satisfies the *structural integrity* if all its structural constraints are satisfied at runtime. The full list of structural constraints used in ReDAS can be found in Table 4.1.

Structural constraints are obtained by static analysis of the program binary based on the knowledge of how the program binary is generated. They do not produce false

Table 4.1: Structural Constraints used in ReDAS

Constraint Name	Section	Description
Caller-Callee	Stack	Stack frames must be constructed via valid caller-callee relationships
Return address	Stack	A return address must point its caller (call site)'s next instruction
Saved frame pointer	Stack	Saved frame pointers must be linked together
Frame size	Stack	A stack frame's size must match the size that predetermined by compiler
Stack limit	Stack	Stack pointer and frame pointer must be within the stack address range
Alignment	Stack	Stack pointer should be aligned according to the stack alignment option
Memory alloc/dealloc	Heap	Heap structure should be changed via valid requests
Boundary tag (meta data)	Heap	A memory chunk's meta data must not be modified until it is freed or reallocated
Allocated memory chunks	Heap	Allocated memory chunks should be linked together
Heap boundary	Heap	Memory allocations should not exceed the heap memory region
Chunk boundary	Heap	Allocated memory chunks should be aligned by word or pre-defined option
Consecutive free chunks	Heap	No two consecutive freed memory chunks exist in the chunk list
Constructor func. pointers	`.ctors`	Function pointers in the `.ctors` section must not modified during execution
Destructor func. pointers	`.dtors`	Function pointers in the `.dtors` section must not modified during execution

alarms, and thus can be safely used in integrity measurement.

It is desirable to reduce the performance overhead introduced by verifying structural constraints. Fortunately, some constraints can be verified together in integrity measurement. For example, a return address in the stack must satisfy both the caller-callee constraint[1] and the return address constraint[2]. Thus, these two constraints can be verified

[1]The caller-callee constraint refers to the condition that any function call represented in an application's stack (through the stack frames of the caller and the callee) must be a possible one as encoded in the application's binary (e.g., through a direct function call or a function pointer).

[2]The return address constraint is the condition that any return address of a function in an application's stack must point to the instruction next to `call` instruction in its calling function.

together using the return addresses in an application's stack.

In order to collect structural constraints from each application, we use the same techniques that provided from previous work [73]. The dynamic property collector analyzes the application's source and binary code, and extracts the set of structural constraints of the application.

Global Data Integrity

Global variables are another type of dynamic objects in a running application. Although their values may change dynamically, there are properties that they must satisfy during program execution [44]. Such properties are called *data invariants* [44] . Data invariants are either values of data variables or relations among them that must be satisfied at certain points in program execution. For example, the constant invariant is an actual numeric value that the corresponding variable should hold during program execution. As another example, the equality invariant means that two data variables must have the same value at certain points of execution. The application satisfies the global data integrity if all its data invariants involving global variables are satisfied. The full list of data invariants can be found in Table 4.2.

Daikon [44] was developed to collect data invariants through a training phase, where Daikon runs the program multiple times to generate the data traces and extracts the data invariants. However, unlike structural constraints, directly using the Daikon's data invariants output to verify global data integrity leads to two problems. Firstly, Daikon may produce false alarms due to inadequate training [74]. Secondly, using Daikon's output may lead to redundant verifications, which introduce unnecessary performance overhead. We address these two problems in the following.

Ensuring the Quality of Daikon Training Phase. In general, a good quality training is the result of using a good set of training data. In order to develop a good set of training data for each program, we first identify various usages of the program along with different options and configurations that the program provides. We then generate a set of training scenarios that triggers various program usages according to the type of an application. For server type applications (e.g., `ghttpd`), training scenarios describe various types of server/client interactions. For example, Figure 4.2 (a) shows the snippet of training input for `ghttpd`. We use the Apache benchmark (`ab`) to reflect multiple clients sending

Table 4.2: Data Invariants

Invariant Type	Example	Description
Constant	$x = 3$	A variable x is a constant
Uninitialized	$x = uninit$	A variable x is never set
Small value set	$x = 1, 3$	a variable x takes only a small number of different values
Range limits	x in [1..3]	x takes the value between minimum and/or maximum
NonZero	x is not null	A variable x is never set to zero
Modulus	$x = a \pmod{b}$	A variable x mod $b = a$ always
Non-modulus	x != $a \pmod{b}$	reported only if x mod b takes on every values besides a
Equality	$x = y$	x and y have same value
Comparison	$x > y$ or $x < y$	Comparison holds over variable x and y
Linear relationship	$y = ax + b$	A linear relationship exists in the variables
Functions	$y = fn(x)$	A function exists between two variables (e.g., $abs()$)
Invariants over $x + y$	$x + y = a$	Any invariant from the list of invariants over a single numeric variable
Invariants over $x - y$	$x - y = a$	Any invariant from the list of invariants over a single numeric variable
Element ordering	$x[]$ elements are equal	Elements of each sequence are non-decreasing, non-increasing, or equal
Sub-sequence	$x[]$ is subset of $y[]$	An array x is a sub-sequence of array y or vice versa
No-duplicate	$x[]$ contains no duplicates	An array x contains no duplicate elements
Reversal	$x[]$ is the reverse of $y[]$	An array x is the reverse of array y
Membership	x in $y[]$	x is a member of an array y

various types of requests to `ghttpd`. For client type applications (e.g., `prozilla`), training scenarios mainly describe various usages of the application. For example, Figure 4.2 (b) shows the snippet of training input for `prozilla`, which executes `prozilla` with various options to reflect its usages (e.g., download different types of files from multiple web sites).

```
# serve normal requests
ab -n 10  http://localhost:80/home/papers/
ab -n 12  http://localhost:80/home/data/example.rar
ab -n 7   http://localhost:80/home/audio/presentation.wmv
ab -n 9   http://localhost:80/home/basic/sample.pdf
ab -n 14  http://localhost:80/home/cyber.doc
 ...  ... ...
 ...  ... ...
# dynamic cgi handling
ab -n 3   http://localhost:80/cgi-bin/cgi_application/query.cgi
ab -n 3   http://localhost:80/cgi-bin/cgi_application/input.cgi
ab -n 10  http://localhost:80/cgi-bin/perl_application/list.pl
ab -n 5   http://localhost:80/cgi-bin/perl_application/input.pl
 ...  ... ...
 ...  ... ...
# error handling
ab -n 2   http://localhost:80/nowhere/nonexisting.html
ab -n 10  http://localhost:80/home/index.nonexistingmimetype
ab -n 12  http://localhost:80/cgi-bin/nonexistin.cgi
ab -n 12  http://localhost:80/cgi-bin/per_application/nonexistin.pl
 ...  ... ...
 ...  ... ...
```

a. Training Input for ghttpd

```
# basic usages
proz -v   # printing version information
proz -L   # printing license information
proz -h   # printing help information
 ...  ... ...
 ...  ... ...
# access valid web server and download directory contents with different options
proz -r www.csc.ncsu.edu
proz -r discovery.csc.ncsu.edu
proz -no-getch -f  - timeout=3, -f -r -t 4   http://www.csc.ncsu.edu/academics/graduate/news/
proz -r -retry-delay=4 k=4 --debug http://www.isoc.org/isoc/conferences/ndss/09/
 ...  ... ...
 ...  ... ...
# error handling
proz –no-getch -f -debug  http://non-existing-web-site/downloadme/
proz –no-getch -f -debug  http://discovery.csc.ncsu.edu/~ckil/nonexistingfile.htm
proz –no-getch -f -debug  http://discovery.csc.ncsu.edu/~ckil/nonexistingmime.mim
 ...  ... ...
 ...  ... ...
# access different type of server
proz -no-getch -f -r  -max-bps=1200 ftp://ftp.freebsd.org/pub/FreeBSD/doc/
proz -no-getch -f -r  http://discovery.csc.ncsu.edu/pubs.html
 ...  ... ...
 ...  ... ...
```

b. Training Input for prozilla

Figure 4.2: Snippet of Training Input for Test Applications

Generating training scenarios is a manual process and it stops when its training data fully covers target application's source code. To check whether the set of training scenarios covers the complete source code of the program, we use a test coverage tool named `Gcov` [75]. `Gcov` allows us to inspect how much of the program is executed by given input. We revise the training scenarios based on the `Gcov`'s result until it completely covers

the program's source code. Each training scenario can be further configured with different parameters. After we obtain such set of training scenarios, we change these parameters to generate various training input. For example, we create 70 different training scenarios for `ghttpd` web server. After configuring the training parameters, we obtain 13,000 training sessions.

It is worth noting that complete source code coverage does not necessarily mean 100% in `Gcov`'s output. Some parts of the program may never be executed due to the program's runtime environment or other reasons. For example, at the end of the training phase for `ghttpd`, `Gcov` reports that 87% of the program are covered by the training data. Our investigation of `Gcov`'s output shows that the rest 13% of `ghttpd`'s source code are for running `ghttpd` on different operating system environments (e.g., `#ifdef WIN`) that is never executed in our Linux training environment.

Once we acquire the final training data set for each application, we start training the application by using Daikon. We observe that the number of data invariants in the Daikon's output is increasing at the beginning of the training process. This is because Daikon has little data trace to analyze, and thus it does not see much difference in global variables. As the training continues, however, the number of data invariants shrinks gradually and stops decreasing at certain point, and remains the same until the end of the training. Note that it is non-trivial to prove the completeness of the training since the training data may not trigger all possible control and data flow paths in the target program. (It is known difficult to find a set of input that can trigger complete control and data paths in the program.) This indicates that it is possible to have false invariants in the Daikon's output even though we have a good quality of training data. In order to ensure no false invariants in the Daikon's output, we enlist the help of a Daikon's utility, `InvariantChecker`. It verifies the invariants output with the set of data traces, and reports false invariants if any. Once `InvariantChecker` reports that there is no false invariant in the invariants output, the data invariants can be used for integrity measurement.

Removing Redundant Data Invariant Verifications. It is desirable to have as few data invariants as possible during integrity measurement for performance efficiency. We observe that Daikon produces redundant invariants. There are two cases. In the first case, Daikon generates the data invariants that have logical redundancy between them (e.g., $x > y$, $y > z$, and $x > z$). In this example, if $x > y$ and $y > z$ invariants are satisfied,

it must satisfy $x > z$ invariant. Thus, we only need to verify $x > y$ and $y > z$. In the second case, the data invariants generated by Daikon are not exactly redundant, but not all of them need to be evaluated in order to verify them all. For example, Daikon may generate a constant invariant that requires a variable to hold certain value during program execution (e.g., $x = 7$), and also a nonzero invariant with the same variable (e.g., x is not null). These two data invariants are not exactly redundant. However, if the constant is not zero in the constant invariant, during integrity measurement, we can simply verify the constant invariant (e.g., $x = 7$) to cover both invariants.

In order to remove redundant data invariant verifications described above, we develop a utility program to remove logically redundant invariants. The utility program reads the Daikon's ouput and looks for any logically redundant data invariants. These redundant data invariants are then removed.

4.1.4 Measuring Dynamic System Properties

After the dynamic system properties are collected by the dynamic property collector, they are used as input to the integrity measurement component, which monitors applications at runtime and measures their integrity.

We propose to include the integrity measurement component in the OS kernel as a loadable kernel module. This allows the integrity measurement component to examine the dynamic objects in various applications. More importantly, this provides isolation between the integrity measurement component and the target applications.

A critical decision is when to measure the dynamic system properties. In general, fine-grained verification may capture more integrity violations, but it also introduces more performance overhead. On the other hand, coarse-grained verification may miss some of integrity violations, though the performance overhead is light. There is clearly trade-off between the quality of integrity measurement and the performance overhead.

As an initial attempt, we propose to use the system call times as the measurement points. The system calls are the means by which an application interacts with the OS kernel, and are also the entry points for many attacks to compromise the OS kernel. Therefore, the system call times are good candidate for verifying dynamic system properties.

Integrity measurement at system call times allows us to verify data invariants and the stack structural constraints. However, they are not good places to verify the heap

structural constraints, since the structure of the heap is only modified when a memory block is added or deleted in the heap. We propose to use `malloc` family function call times to check the heap structural constraints. In addition, the dynamic objects in the `.ctors` and `.dtors` sections are only used when the `main` function starts and after it finishes. Thus, we only need to verify them before the `main` function enters/exits. Table 4.3 summarizes the measurement points and the corresponding dynamic properties used in ReDAS.

Table 4.3: Measurement Points of ReDAS

Measurement Point	Dynamic Properties	Example(s) of Dynamic Objects to Verify
System call	Data invariants Structural constraints (stack section)	Global variables Return addresses, saved frame pointers
Dynamic memory allocation/deallocation	Structural constraints (heap section)	Boundary tag (heap management data)
`Main` function	Structural constraint (`.ctors` and `.dtors` sections)	Constructor (destructor) function pointers

There are certainly other options for measurement points. For example, we may verify dynamic system properties at every function call. This solution is potential to capture more transient integrity violations, but also introduces higher performance overhead. We will investigate alternative options in our future work.

Measuring dynamic properties of an application starts when the integrity measurement component intercepts an *execve* system call from the OS kernel. The integrity measurement component first identifies what application is being loaded into memory by checking the *path* argument of the *execve* system call. Once the application is identified as a target application, the integrity measurement component starts verifying the application's dynamic properties and continues until the application is unloaded from memory. When the application invokes a system call or a function call that is a measurement point, the integrity measurement component first blocks the system call or the function call so that the application will wait until the measurement is done. It then locates the required dynamic objects (e.g., return addresses in the stack) within the application's memory space and examines their integrity. If the application passes all integrity checks, the integrity measurement component releases the system call or the function call, and allows the appli-

cation to continue the execution. If any check fails, the integrity measurement component records the related information, including the application's name, the violated constraint (or data invariant), and current instruction point. This list of data items is called the *integrity violation proof* of this integrity violation. The collection of all the integrity violation proofs is called the *integrity evidence* of the attester. In the special case where the integrity evidence is empty, all dynamic system properties are satisfied.

When verifying heap related dynamic properties, we take advantage of the existing effort in the GNU C library. The library provides sanity check against the heap management data, which is the same dynamic object that ReDAS needs to evaluate. If any sanity check fails, the library invokes the `SIGABRT` signal to the OS kernel. Our integrity measurement component is implemented to capture such signals and identifies which application is involved. It then records the integrity violation proof in the integrity evidence. By harnessing the library support, we can reduce the burden of keeping track of dynamic memory allocations (deallocations) along with their integrity check. The details can be found in Section 4.2.

4.1.5 Protection of Integrity Evidence

As long as an attester is online to provide network-based services to others, it is subject to remote attacks, which may exploit (both known and unknown) system vulnerabilities to compromise the system. A determined attacker may be able to compromise the runtime kernel. Even if ReDAS can detect the integrity violation caused by the attacker, after compromising the kernel, the attacker will be able to modify the integrity evidence maintained by ReDAS, which is also stored in the kernel. In the following, we investigate how to protect the integrity evidence so that even if an attack is successful, the attacker still cannot erase its mark from the integrity evidence without being detected.

To achieve this goal, we propose to take advantage of the hardware support provided by TPM. The TPM contains Platform Configuration Registers (PCRs), which provide sealed storage for platform configuration measurements. These measurements are normally hash values of entities (e.g., programs, configuration data) on the platform. A subset of PCRs, which are called *static* PCRs (no 0-15), cannot be reset at runtime. Instead, they can only be extended with new measurement values. This means a new measurement value is hashed together with the previous one in the PCR, and the result is saved in the same PCR.

This property in particular aims to prevent a compromised application from tampering with a previously saved measurement.

In ReDAS, the integrity evidence is sealed into TPM PCR 8, which is one of the static PCRs. Other static PCRs are used for the CRTM (PCR 0-7) and IMA uses PCR 10. During the system operations, if none of the dynamic system properties are violated, both the integrity evidence and the PCR 8 have their default values. On the other hand, if an integrity violation occurs, the integrity evidence will include the proof of the integrity violation. We then compute the measurement (i.e., hash value) of the integrity evidence, and extend the measurement to PCR 8. During the attestation, the attester sends both the integrity evidence and the value of PCR 8 to the challenger protected by the TPM signature. Thus, any malicious modification on the integrity evidence will be visible to the challenger during the attestation.

Another challenge here is the timing of sealing (the cryptographic checksum of) integrity evidence. We have to guarantee that even if there is an on-going attack, it cannot interfere with the sealing of such evidence. When the integrity measurement happens at system call times, we do have certain guarantee. Since an attack has to interface with the OS kernel through system calls, it cannot corrupt the integrity evidence, which is stored in the kernel, before the attack vector is verified at system calls. The details of TPM related implementation can be found in Section 4.2.

4.1.6 ReDAS Dynamic Attestation Protocol

Once ReDAS runs on the attester, a remote challenger can request a dynamic attestation to the attester. The dynamic attestation process is a request/reply protocol as shown in Figure 4.3.

The ReDAS dynamic attestation protocol is straightforward, and similar to other static attestation protocols. The only difference is that the attestation response includes the integrity evidence, which is the history of all integrity violation proofs stored in the attester. The challenger (A) starts the protocol by generating a random nonce and including it with the dynamic attestation request message. If only authorized challengers are allowed to send attestation requests, the request messages can further be authenticated with a digital signature or a MAC (Message Authentication Code) using a symmetric key. Upon receiving the dynamic attestation request, the attestation service (B) generates a dynamic attestation

1. A : generate random nonce
   A → B : dynamic attestation request, nonce
2. B → A : Integrity evidence, $Sign_{AIK_B}$ {nonce, PCR 8}
3. A : verify the TPM signature and the nonce
   if the signature or nonce is invalid then exit
   else validate the correctness of the integrity evidence using PCR 8

A: Challenger **B**: Attester (runs ReDAS)

Figure 4.3: ReDAS Dynamic Attestation Protocol

response message which includes two parts. The first part is the current integrity evidence from the integrity measurement component, including the list of integrity violation proofs in the order of occurrences. (Note that this list is empty when there is no integrity violation on the attester.) The second part is the TPM signature by using its Attestation Identification Key (AIK) on the current PCR 8 value along with the nonce received from the challenger. Once the challenger received the dynamic attestation response message, it verifies the TPM's signature and the integrity evidence.

The attestation response is protected against replay attacks due to the presence of the nonce in the TPM signature. The integrity evidence cannot be forged due to the presence of the TPM signature on its hash value. If there is no integrity violation history present, the TPM signature should sign an default value in the PCR 8 and the challenger can trust that all applications passed their integrity verifications successfully. We assume that static attestation of the applications is already performed via IMA before dynamic attestation. In other words, integrity proof of static system properties (e.g., application binary) of the attester is already given to the remote challenger.

4.1.7 Security Analysis

In the following, we explain how we build the chain of trust from hardware up to the integrity evidence. We then discuss possible adversary actions against ReDAS and how we address such attacks.

Establishing Chain of Trust

Hardware and Kernel Integrity Measurement. Our trust building process starts from measuring the hardware of the attesting host. We use the CRTM to measure the initial state of the hardware and the kernel of the attester. This enables a remote challenger to verify if the hardware of the attester is genuine and whether the kernel is configured as expected and booted securely. The integrity measurement component is a part of the kernel, and is loaded before any user-level applications. Therefore, the trusted boot will also include measurement of the integrity measurement component and attest to its load time integrity.

Application Load-Time Integrity Measurement. Protection against illegal modification of application binaries is done by using IMA [65]. IMA measures a program's binary before it is loaded into memory for execution. IMA protects the integrity of the measurement result by extending its hash value into a PCR. As discussed earlier, we assume a static attestation using IMA has happened before dynamic attestation. As a result, any illegal modification of a program's binary will be visible because of IMA.

Run-Time Protection of Integrity Measurement Component. The integrity measurement component can be trusted as long as the kernel is not compromised. Since the integrity measurement component intercepts system calls and the kernel's signals, it can capture malicious attacks that try to interfere with the integrity measurement component (e.g., sends STOP or KILL signal to stop the integrity measurement component). When the integrity measurement component captures such attacks, it records the attack information in the integrity evidence and extends its measurement to a PCR. Thus, such attacks will be visible during dynamic attestation.

Integrity Evidence. Due to the protection of integrity evidence using TPM, the adversary cannot forge the PCR value without being detected. During dynamic attestation, the integrity evidence is protected by the TPM signature and the nonce. This prevents the adversary from altering the integrity evidence during transmission or reusing previous integrity evidence.

Possible Attacks and Defense

Given our threat model and a careful study of previous security issues in remote attestation, we categorize the possible threats and show how ReDAS is resilient to these attacks below.

Modification. In a modification attack, the adversary tries to disrupt the attestation process by modifying the objects that the attestation is dependent on. In ReDAS, these include the application binaries, dynamic system properties, and integrity evidence. ReDAS uses IMA to provide load-time measurement of application binaries. The dynamic system properties are statically measured and verified before it is loaded into the integrity measurement component. The integrity evidence is protected by extending its hash value into a PCR. As a result, any illegal modification to these objects will be visible to the remote challenger during dynamic attestation.

Emulation. In this attack, the adversary tries to emulate the attestation process or even emulate the entire host using a Virtual Machine Monitor (VMM). However, these techniques require changing both the hardware and software configurations. By using trusted boot and IMA, the difference in configurations is visible to the challenger, and as a result, the attestation will fail.

Misdirection. During dynamic attestation, the remote challenger establishes a secure connection with the attester. In a misdirection attack, the adversary tries to redirect the attestation request to another platform running ReDAS to pass the attestation, and then provides the actual services from the other machine without ReDAS. Fortunately, countermeasures for this attack have already been developed. Goldman et al. [71] proposed to extend the PCR with a measurement that represents the static property of the secure connection endpoint (e.g., SSL public key). Using this new technique, the remote challenger can guarantee that the TPM signature was generated using the same machine on the endpoint of the tunnel.

Runtime Memory Corruption. The adversary may try to compromise an application by exploiting a vulnerability. Even if successful, the adversary is limited in its ability to affect the system. In order to avoid detection by ReDAS, the adversary either has to refrain from making system calls, or ensure that he/she does not modify dynamic objects that ReDAS examines. This limits the adversary's ability to make the exploit useful. The

current implementation of ReDAS cannot capture all memory corruption attacks due to the limited dynamic system properties and measurement points. However, this is due to the limited scope of the current effort, rather than the dynamic attestation methodology.

Denial of Service (DoS) Attacks. The adversary may try to launch DoS attacks by repeatedly sending attestation requests. Such attacks are certainly not unique to ReDAS, and there have been existing solutions such as rate limiting (e.g., [76]) and client puzzles (e.g., [77]) to prevent or mitigate such attacks.

Limitation

As discussed earlier, the current implementation of ReDAS is not guaranteed to see integrity violations of all attacks. There are two reasons. First, some attacks may modify the dynamic objects that ReDAS does not examine. Second, some attacks may have transient effects that are not observable at current measurement points. These threats can be addressed by having fine-grained measurement points or having more dynamic properties to check. For example, a recent return-to-libc attack [78] manipulates the stack to inject attacks. If the attack does not invoke any system call, the current implementation of ReDAS cannot find the integrity violation. However, if we perform integrity measurement at function call times, we will be able to identify such violation.

Despite the limitation, ReDAS has advanced the state of the art of remote attestation. Static attestation cannot handle runtime attacks at all. As the first step of dynamic attestation, ReDAS can now partially address integrity violation caused by runtime attacks. It is certainly necessary to understand the coverage of various dynamic system properties and explore different measurement points in order to have “perfect” dynamic attestation. We will investigate these issues in our future work.

4.2 Implementation

To assess the feasibility and effectiveness of our approach, we have implemented ReDAS on Linux kernel version 2.6.22. This section describes implementation details of the main components in ReDAS.

Dynamic Property Collector: Each application monitored and attested for its integrity first runs through the dynamic property collection phase, which is done offline.

The dynamic property collector gathers the application's dynamic properties (currently structural constraints and data invariants) by using various tools including a few Ruby [57] scripts, modified Daikon, and `Gcov` [75].

The dynamic property collector uses a few Ruby scripts to obtain structural constraints. The Ruby scripts are implemented to gather the required structural information from the target applications (e.g., code segment range and stack frame sizes of user-defined functions) and output the structural constraints. It is worth noting that the structural constraint generation is fully automated and requires no human interaction. However, the requirement for debugging information in some cases requires a recompilation of the program.

As discussed earlier, data invariants are obtained via training by using `Gcov` and modified Daikon. We first generate the training data by analyzing various usages of the target application and use `Gcov` to ensure if the training data fully covers the application's source code. Once all training data for the target application are prepared, the dynamic property collector runs the application with modified Daikon. We have to perform a number of modifications to adapt the functionality offered by Daikon. Originally, Daikon produces the data trace at function call times to collect data invariants. Since the integrity measurement component performs integrity verifications of data invariants at system calls, we modify Daikon to produce the data trace at system call times. Once the data traces of the target application are obtained, the dynamic property collector runs Daikon's invariant detector to collect the data invariants of the application.

Application binaries usually do not include the symbol table for performance reasons. In order to address this problem, the dynamic property collector converts dynamic properties data into a binary format by using the Ruby scripts. For example, a data invariant, (`foo_var` == 50) is converted to (0x8049000 == 50), where 0x8049000 is the memory address of `foo_var`. After both structural constraints and data invariants are collected and converted, the dynamic property collector uses the Ruby scripts to put them together and create the input file for the integrity measurement component.

Integrity Measurement Component: The integrity measurement component is implemented as a loadable kernel module using SystemTap [79]. We write a Systemtap script that defines the functions for verifying structural constraints and data invariants at measurement points (e.g., system calls), and Systemtap translates the script to create a

loadable kernel module. Moreover, as discussed earlier, we take advantage of the GNU C library's `MALLOC_CHECK` to verify the heap structural constraints. We set `MALLOC_CHECK` to 2 (`setenv MALLOC_CHECK_ 2`) so that the library invokes the `SIGABRT` signal when it detects any heap corruption. When the integrity measurement component sees this signal, it identifies the application that causes the problem. Such signal is regarded as an integrity violation proof of heap structural constraints and recorded in the integrity evidence.

The integrity measurement component requires the input (dynamic system properties) generated by the dynamic property collector. Since a direct file handling is not desirable within the kernel due to security reasons, the integrity measurement component enlists the help of a user level program named as *verification initializer*. Communication between the integrity measurement component and the verification initializer is carried over a `proc` file generated by the integrity measurement component. The verification initializer is a small piece of program that reads the dynamic system properties file and sends it to the integrity measurement component through the `proc` file. To protect the dynamic system properties file, the permission of the file is handled by the integrity measurement component itself, and only the verification initializer is given the write permission.

The integrity measurement component uses the TPM's support to protect the integrity evidence. If any measurement fails, it logs the integrity violation proof in the integrity evidence, and seals the evidence into the TPM. Sealing is done by calculating the hash of the integrity evidence and then extending the TPM's PCR 8 with this new hash value. The initial hash of the integrity evidence is computed within the integrity measurement component. Afterwards, it extends the PCR by directly accessing the TPM device driver, which is another loadable kernel module. The integrity measurement component avoids using the TPM user level library provided by the TCG to enhance the performance and to guarantee that the whole evidence sealing operation is done within the same context. The TPM's hardware is responsible for calculating the new PCR value by hashing the new integrity evidence hash value with the current PCR value.

Attestation Service: The attestation service is implemented as a user level network service daemon that receives dynamic attestation requests from remote challengers and sends the dynamic attestation responses back to the challenger. As described in Section 4.1.6, the response from the attestation service includes the TPM's signature on the value of PCR 8 with nonce received from the challenger and the current integrity evidence.

In order to communicate with the TPM, the attestation service uses TrouSerS [80], which is an open source TPM library. TrouSers allows the attestation service to access the current value of PCR 8 by using `Tspi_TPM_PcrRead` function. The attestation service also uses `Tspi_TPM_Quote` function to get the TPM signature of the PCR 8 and the nonce value. The attestation service gets the current integrity evidence from the integrity measurement component through the `proc` file that created by the integrity measurement component.

4.3 Experimental Evaluation

We performed a series of experiments to evaluate both effectiveness and performance of ReDAS. In the effectiveness experiments, we focus on understanding how well ReDAS verify the integrity of running applications without any false alarms. In performance evaluation, we test ReDAS to estimate how efficiently it performs integrity measurement for dynamic attestation.

4.3.1 Experiments Setup

All experiments were performed on a host running Ubuntu 7.1 with dual 1.86 GHz Intel Core 2 CPU, 3 GB of RAM, and a 10,000 RPM SATA hard drive. The host is equipped with a TPM that complies with TCG TPM specification v1.2 [69]. The kernel is configured to work with IMA [65]. All test applications were compiled using GCC version 4.1.3 with a debugging flag (-g).

We chose nine real-world applications for our evaluation experiments. They are publicly available on the Internet. These applications are various types of programs (e.g., server, client) and known to have different types of integrity vulnerabilities which serves our evaluation purpose. Table 4.4 lists the test applications.

The set of dynamic properties collected from the test applications is listed in Table 4.5. The data invariants column shows the total number of data invariants collected from each application. It also shows the minimum (maximum and average) number of data invariants to examine at measurement points. The structural constraints column shows the total number of structural constraints. It includes the legitimate return addresses of function calls (the addresses of instructions next to `call` instructions), memory allocation/deallocation points (e.g., `malloc` calls), and other required structural information of

Table 4.4: Test applications

Name	Description	Known Vulnerability
ghttpd 1.4	a web server	S (CVE-2002-1904)
prozilla 1.3.6	a web download accelerator	S (CVE-2004-1120)
sumus 0.2.2	a 'mus' game server	S (CVE-2005-1110)
newspost 2.1.1	a Usenet news poster	S (CVE-2005-0101)
wsmp3d 0.0.8	a web server with audio broadcasting	H (CVE-2003-0339)
xtelnetd 0.17	a telnet daemon	H (CA-2001-21)
nullhttpd 0.5.0	a web server	H (CVE-2002-1496)
powerd 2.0.2	a UPS monitoring daemon	F (CVE-2006-0681)
openvmps 1.3	a VLAN management server	F (CVE-2005-4714)

S: Stack Overflow, H: Heap Overflow, F: Format String

the application such as segments (e.g., beginning/ending address of the code segment) and user-defined functions (e.g., a function's address range, the size of its arguments). These information is used to verify the application's structural constraints at runtime. Note that some of structural constraints do not require input from target applications. For example, saved frame pointers in an application's stack must be linked together to satisfy the saved frame pointer constraint. However, the values of saved frame pointers can not be pre-determined by the compiler rather determined at runtime.

Table 4.5: Dynamic Properties of Test Applications

Applicaiton Name	Data Invariants				Structural Constraints
	Total	Min	Max	Average	
ghttpd 1.4	34	2	3	2	37
prozilla 1.3.6	19	1	3	2	826
sumus 0.2.2	249	24	40	31	141
newspost 2.1.1	294	22	23	22	505
wsmp3d 0.0.8	331	10	45	30	143
xtelnetd 0.17	237	50	135	79	240
nullhttpd 0.5.0	481	23	33	26	146
powerd 2.0.2	110	4	8	5	50
openvmps 1.3	34	2	10	2	273
Average	199	15	33	22	262

Min(Max/Average): minimum (maximum/average) # of data invariants at the measurement points.

4.3.2 Effectiveness

We evaluate the effectiveness of ReDAS in two ways. First, we test ReDAS to see if it triggers any false alarm when the attester is normal state. Second, we evaluate ReDAS to understand how well it captures integrity violations when the attester is under attack.

To test if ReDAS triggers any false alarm, we first generate test cases for each application to emulate a normal circumstance of running the application. In order to develop the test cases, we take the similar approach that we used in the Daikon's training phase. However, we focus on generating the test input to emulate actual usages of an application in a real environment. For example, we identify various usages of running `ghttpd` and the corresponding test scenarios as shown in Table 4.6. We then configure the test parameters to emulate 20 clients sending 35 different types of requests to `ghttpd`.

Table 4.6: Test Scenarios for `ghttpd`

Usage of Application	Test Scenario
Start/shutdown	Start and shutdown the server with no client connections
Serve normal requests	Clients send various HTTP requests
Handle invalid requests	Clients send various invalid requests
Execute applications	Clients request to execute server-side applications (e.g., cgi)
Manage multiple sites	Server runs multiple web sites
Error Handling	Server handles various types of errors

We also change the test parameters to make the test data different from the training input. However, some applications (`powerd` and `openvmps`)) are limited to have different test parameters due to its limited usages. For example, the `powerd` only accepts 4 different simple type messages (`OK, CANCEL, FAIL, SHUTDOWN`). Thus, the test cases mostly overlap with the ones in the training input.

After we obtain the test cases of each application, we run the tests and check if ReDAS generates any false integrity violation proof in the integrity evidence. Our results show that no false alarms are generated in all nine applications.

To evaluate how well ReDAS captures integrity violations caused by runtime attacks, we use the attack programs that exploit the vulnerabilities of the test applications. We obtain most of the attack programs from publicly known web sites (e.g., `www.milw0rm.com`). None of the publicly available attacks we have access to involve violation of global data in-

variants. Indeed, there are applications (e.g., [81, 82]) with known attacks that violate global data integrity. However, all the ones that we know are commercial software, and we do not have access to their source code or the malicious attack program. To compensate this, we developed a few exploits targeting some known exploits. For example, we modify an attack program that exploits a format string vulnerability in `powerd`. The modified attack program is implemented to overwrite a global variable, `outtime` in `powerd`. We use powerd 2.0.2* to indicate the attack we have develop. In each experiment, we first launch the application, then use the attack program to generate the malicious input, and send it to the application. We then check the integrity evidence to see if ReDAS captures the integrity violations caused by the attacks.

Table 4.7: Effectiveness Evaluation Result

Name	Dynamic Property Violation	Measurement Point
ghttpd 1.4	Return address constraint	3 (`sys_read`)
prozilla 1.3.6	Return address constraint	3 (`sys_read`)
sumus 0.2.2	Return address constraint	4 (`sys_write`)
newspost 2.1.1	Caller-callee constraint	3 (`sys_read`)
wsmp3d 0.0.8	Boundary tag constraint	`malloc`
xtelnetd 0.17	Boundary tag constraint	`malloc`
nullhttpd 0.5.0	Boundary tag constraint	`free`
powerd 2.0.2	Return address constraint	4 (`sys_write`)
powerd 2.0.2*	Equality invariant	4 (`sys_write`)
openvmps 1.3	Return address constraint	102 (`sys_socketcall`)

Table 4.7 summarizes our evaluation result. It shows the application name, violated dynamic property, and the measurement point where ReDAS captures the integrity violation. In all cases, ReDAS captures integrity violations caused by various runtime attacks. For example, ReDAS reports that `ghttpd` violated the return address constraint at `sys_read` system call. ReDAS also provides the current program instruction pointer (value of `EIP` register) when it captures an integrity violation. Our investigation shows that the instruction pointer in the `ghttd` case points to the `Log` function, which invokes the `sys_read` system call when it processes the malicious (oversized) input. Such input corrupts the dynamic objects including the return address in the stack, which violates the return address constraint.

Overall, our effectiveness evaluation shows a promising result that ReDAS does

not generate any false alarms while capturing the integrity violations caused by all attacks.

4.3.3 Efficiency

We also perform experiments to evaluate the performance impact of ReDAS. To measure the performance overhead, we run the test applications with and without ReDAS, and compare the applications' throughputs. The throughput is measured according to the type of the application. For server type application (e.g., `ghttpd`), we use the Apache benchmark [33] to emulate the multiple clients (1-20) sending different types of requests to the application. The apache benchmark then outputs the average throughput (# of requests/sec) of the application. For client type application (e.g., `prozilla`), we compute the average speed of executing the application. For example, in `prozilla`, we check the average speed of downloading various types of files from multiple web sites. In each experiment, we iterate the test 20 times and compute the average throughput of the application. Note that some test applications are not suitable for measuring their performance by using the throughput. For `xtelnetd`, `powerd`, and `openvmps`, we could not obtain any performance benchmark program to measure their performance. It is also not appropriate to measure their throughput by using the average speed of execution since they are daemon programs that run continuously. For `newspost`, the throughput is depend on the responding party (e.g., Usenet server) rather than its own execution. Thus, we exclude these applications in the experiments.

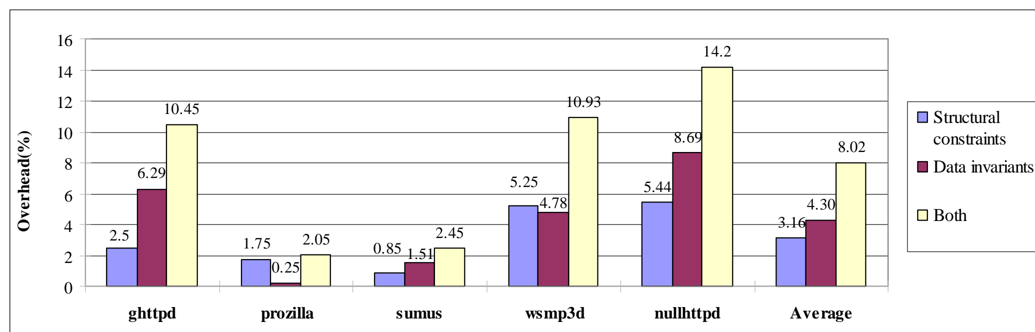


Figure 4.4: Performance Overhead Evaluation Result

Figure 4.4 illustrates the performance result. The X-axis lists the names of ap-

plications and the Y-axis shows the additional overhead (%) of running the application with ReDAS compared to running it without ReDAS. In each application, three bars show the overhead of verifying structural constraints only, data invariants only, and both. For example, `prozilla` slows down 2.05% with verifying both its structural constraints and the data invariants.

There are some factors affecting the performance overhead of ReDAS such as the total number of dynamic objects, the sizes of dynamic objects, and the number of system calls that the application makes during execution. For example, `nullhttpd` takes more time to verify the data invariants than `ghttpd`. This is because `nullhttpd` has 26 data invariants to verify on average at every measurement point, while `ghttpd` has two data invariants per measurement point to examine, as shown in Table 4.5. As another example, `sumus` shows better performance than `ghttpd` in verifying the data invariants, although it has more data invariants to verify. Our investigation shows that the sizes of data invariants (global variables) in `sumus` are much smaller (mostly 4 Bytes) than those in `ghttpd` (mostly 4KBytes). Moreover, `sumus` invokes less system calls (116 system calls) than `ghttpd` (364 system calls) in normal executions. Overall, our result shows that ReDAS incurred 8% additional overhead on average for verifying dynamic properties of test applications.

4.4 Related Work

To the best of our knowledge, there is no comprehensive study on attesting to dynamic system properties. Property based attestation [83] is closely related to our work. They proposed a delegation based attestation that requires a trusted third party certifying a mapping between the properties that the attester claims and the actual configurations of the system. Our work differs from theirs in the definition of properties. They define the properties as the attester's platform configurations, while we define the properties as the properties of dynamic objects in the system that must be satisfied during execution. In addition, their work is limited to verifying static properties of the system, while our approach is aimed at dynamic attestation.

For remote system attestation, SWATT [60] proposed to use random memory traversal on embedded systems to attest their static contents and the system's configurations. However, this work is not suitable for generic computing environment where the

memory size is much bigger than embedded systems and the content is dynamically changing. Shaneck et al. [64] developed a checksum code to measure the integrity of memory contents in the wireless sensors but their work is also limited to verify the static memory contents. Sailer et al. [65] developed an integrity measurement system named as IMA. IMA provides an integrity measurement which is the list of applications run on the attester and their corresponding hash values of binaries. This work is complimentary to our work to provide integrity measurement of static parts of applications.

For remote software attestation, Pioneer [61] proposed verifiable code execution by generating checksum code that can attest the integrity of the executing code on the untrusted host. However, the checksum code only verifies the hash of the executable where dynamic properties are excluded in generating proof. BIND [84] developed a fine-grained runtime code attestation. It provides the proof that binds the integrity of the executed code and its input/output. Flicker [62, 63] provides a hardware-supported isolation to protect the execution of security critical parts (e.g., checking an user's password) of the program. It also provides the measurement of executed code and its input/output parameters by extending their hash values into a dynamically resettable PCR (PCR 17). These works are complementary to our work since the program's input/output can be one of dynamic system properties to attest to. Our work is more general in addressing the need of dynamic attestation. Milenkovi et al. [85] use special tamper-resistant hardware to encrypt/decrypt a program with a platform unique key to verify the program's integrity. This approach shares the same limitation that only attests static parts of the programs.

There were also efforts [59, 86, 87] to measure the Linux kernel runtime integrity. Such efforts are complimentary to our work. Copilot [59] used a PCI card to directly attest the runtime kernel integrity without the operating system's support. Loscocco et al. [86] developed a kernel integrity monitoring program to verify kernel data structures. Petroni and Hicks [87] proposed a technique to monitor the runtime kernel's control-flow integrity.

4.5 Summary

In this chapter, we present a remote dynamic attestation mechanism and introduce the notion of dynamic attestation to overcome limitations of existing remote attestation techniques. Dynamic attestation uses dynamic properties of the system to verify the runtime

integrity of the system. Thus, it enables an attester to claim its runtime integrity to a remote challenger. As an initial attempt in our research, this chapter presents a novel remote dynamic attestation system called ReDAS. ReDAS provides integrity evidence of running applications in the system by measuring two dynamic properties: structural integrity and global data integrity. ReDAS uses the hardware support provided by TPM to protect the integrity evidence. Thus, even if an attacker is able to compromise the attesting host, he/she cannot modify the integrity evidence without being detected by the challenger. ReDAS is currently implemented as a prototype system on Linux. The experimental evaluation results obtained with real-world applications indicate that ReDAS is effective and practical in performing application-level dynamic attestation.

Although ReDAS extends current remote attestation capability by enabling attestation to dynamic system properties, it is not a complete solution yet. In our future work, we plan to investigate other types of dynamic system properties and different measurement points to improve the integrity measurement capability with low overhead. We will also investigate how we can perform dynamic attestation of OS kernels.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This dissertation discusses problems related to protecting software integrity at runtime and investigates three mechanisms to address such problems.

Firstly, this dissertation presents an integrity protection mechanism that can thwart attacks that try to exploit memory corruption vulnerabilities in programs. The protection mechanism is provided by randomizing the program's runtime memory address layout and the memory objects. As a result, it hinders memory corruption attacks (e.g., stack buffer overflow attack) by preventing an attacker being able to easily predict their target addresses. The protection mechanism is implemented by a novel binary rewriting tool that can randomly place the code and data segments of programs and perform fine-grained permutation of function bodies in the code segment as well as global variables in the data segment. Our evaluation results show minimal performance overhead with orders of magnitude improvement in randomness.

Secondly, this dissertation investigates a vulnerability identification mechanism named as CBones which automatically discovers how unknown vulnerabilities in C programs are exploited by verifying program structural constraints. Having such mechanism is helpful in discovering potential vulnerabilities in software since understanding security bugs in a vulnerable program is a non-trivial task, even if the target program is known to be vulnerable. This mechanism is also useful in developing integrity patches for vulnerable programs where applying security patch is increasingly common in these days. CBones

automatically extracts a set of program structural constraints via binary analysis of the compiled program executable. CBone then verifies these constraints while it monitors the program execution to detect and isolate the security bugs. Our evaluation with real-world applications that known to have vulnerabilities shows that CBones can discover all integrity vulnerabilities with no false alarms, pinpoint the corrupting instructions, and provide information to facilitate the understanding of how an attack exploits a security bug.

Lastly, this dissertation identifies the need of dynamic attestation to overcome the limitations of current state-of-the-art attestation approaches. To the best of our knowledge, we are the first to introduce the notion of dynamic attestation and propose use of dynamic system properties to provide the integrity proof of a running system. To validate our idea, we develop an application-level dynamic attestation system named as ReDAS (Remote Dynamic Attestation System) that can verify runtime integrity of software. ReDAS provides the integrity evidence of runtime applications by checking their dynamic properties: structural integrity and global data integrity. These properties are collected from each application, representing the application's unique runtime behavior that must be satisfied at runtime. ReDAS also uses hardware support provided by TPM to protect the integrity evidence from potential attacks. Our evaluation with real-world applications shows that ReDAS is effective in capturing runtime integrity violations with zero false alarms, and demonstrates that ReDAS incurs 8% overhead on average while performing integrity measurements.

5.2 Future Work

Although the substantial advances in techniques for protecting software integrity at runtime, many problems still have not been fully addressed. These problems still need further investigations.

Integrity Protection Mechanism. Further investigation is required to find a solution for fine-grained randomization on the stack region. The fine-grained stack frame randomization that can permute the stack elements (e.g., return address) will be beneficial to allow low chance of an attacker guessing where randomly placed areas are located.

Vulnerability Identification Mechanism. CBones currently cannot handle function pointers properly yet. A jump due to a function pointer usually dereferences

a register or an address in assembly code. As a result, CBones may raise a false alarm. These false positives can be suppressed by marking `call` instructions that dereference values instead of using function addresses and suppressing the errors when they are generated at these instructions. We will extend the implementation of CBones to handle such cases in our future work.

Remote Dynamic Attestation Mechanism. Although ReDAS extends current remote attestation capability by enabling attestation to dynamic system properties, it is not a complete solution yet. In our future work, we plan to investigate other types of dynamic system properties and different measurement points to improve the integrity measurement capability with low overhead. We will also investigate how we can perform dynamic attestation of OS kernels.

Bibliography

[1] ISO/IEC JTC1. JTCl/SC7/WG9 Working Draft WD 1.2, System and Software Integrity Levels, 1995.

[2] United States Computer Emergency Readiness Team (US-CERT). Software assurance. http://www.us-cert.gov/reading_room/infosheet_SoftwareAssurance.pdf.

[3] Insurance Journal. FBI: 90% of organizations face computer attack; 64% incur financial loss, January 2006.

[4] NIST. National vulerability database. http://nvd.nist.gov/.

[5] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 339–348, Washington, DC, USA, 2006. IEEE Computer Society.

[6] The PaX Team. Address space layout randomization, March 2003.

[7] The MITRE Corporation. Common vulnerabilities and exposures. http://cve.mitre.org/.

[8] Aleph One. Smashing the stack for fun and profit. Phrack Magazine, 49(14), November 1996.

[9] Scut. Exploiting format string vulnerabilities. http://julianor.tripod.com/teso-fs1-1.pdf, March 2001.

[10] Anonymous. Once upon a free(). Phrack Magazine, 11(57), August 2001.

[11] G. Necula, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, Portland, Oregon, January 2002.

[12] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd Sympomsium on Reliable Distributed Systems*, pages 260–269. IEEE Computer Society, October 2003.

[13] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *12th USENIX Security Symposium*, Washington, DC, August 2003.

[14] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *14th USENIX Security Symposium*, Baltimore, MD, August 2005.

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

[16] U. Drepper. Security enhancements in Red Hat enterprise Linux (besides SELinux). `http://people.redhat.com/drepper/nonselsec.pdf`, June 2004.

[17] SecuriTeam. Overwriting ELF .dtors section to modify program execution. `http://www.securiteam.com/unixfocus/6H00I150LE.html`, December 2000.

[18] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *6th Workshop on Hot Topics in Operating Systems*, pages 62–72, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[19] 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.

[20] I. Molnar. Exec-shield. `http://people.redhat.com/mingo/exec-shield/`, 2005.

[21] The Fedora Community. The Fedora project. `http://fedora.redhat.com/`, 2006.

[22] J. Brindle. Hardened Gentoo. `http://www.gentoo.org/proj/en/hardened/`, 2006.

[23] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, Washingtion, D.C, October 2004.

[24] SecurityTracker. Advisory 16/2005: phpMyAdmin local file inclusion, 2005.

[25] M. L. Nohr. *Understanding ELF Object Files and Debugging Tools.* Prentice Hall Computer Books, 1993.

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

[27] C. G. Bookholt. Address space layout permutation: Increasing resistance to memory corruption attacks. Master's thesis, North Carolina State University, 2005.

[28] The IEEE and The Open Group. The Open Group base specifications: mmap, 6 edition, 2004.

[29] P. Busser. Paxtest. `http://www.adamantix.org/paxtest/`, 2006.

[30] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the Sapphire/Slammer worm, 2003.

[31] Standard Performance Evaluation Corporation. Spec cpu2000 v1.2., 2006.

[32] L. McVoy and C. Staelin. Lmbench: Tools for performance analysis, 1998.

[33] Apache Software Foundation. Apache benchmark, 2006.

[34] Apache Software Foundation. Apache http server project, 2006.

[35] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter Technical Conference*, 1992.

[36] J. Choi, K. Lee, A. Loginov, R. OCallahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[37] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[38] M. Musuvathi, D.Y.W. Park, A. Chou, D.R. Engler, and D.L. Dill. CMC: A pragmatic approach to model checking real code. In *ACM Symposium on Operating Systems Design and Implementation*, December 2002.

[39] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time detection of heap-based overflows. In *Proceedings of the 17th USENIX Large Installation Systems Administration Conference*, San Diego, CA, October 2003.

[40] 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 *Proceedings of the 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.

[41] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient architecture support for software debugging. In *Proceedings of the 31st annual International Symposium on Computer Architecture*, June 2004.

[42] P. Zhou, W. Liu, L. Fei, F. Qin S. Lu, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *37th International Symposium on Microarchitecture*, November 2004.

[43] S. Hangal and M. S. Lam. Diduce: Tracking down software errors using dynamic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, May 2002.

[44] M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

[45] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, San Diego, California, June 2007.

[46] GNU Project. Programming languages supported by GCC. `http://gcc.gnu.org`, 2007.

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

[48] klog. The frame pointer overwrite. `http://doc.bughunter.net/buffer-overflow/frame-pointer.html`.

[49] E. Chien and P. Szor. Blended attacks exploits, vulnerabilities and buffer-overflow techniques in computer viruses. In *Virus Bulletin Conference*, 2002.

[50] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, September 2006.

[51] D. Lea. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`, 2000.

[52] E.D. Berger, K.S. McKinley, R.D. Blumofe, and P.R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000.

[53] The IEEE and The Open Group. The Open Group base specifications issue 6, IEEE Std 1003.1, 2004.

[54] iSec Security Research. Linux kernel do_brk() vulnerability, 2003.

[55] S. Cesare. Shared library call redirection using ELF PLT infection, April 2007.

[56] SIGMil. Format string exploit. `http://www.acm.uiuc.edu/sigmil/talks/general_exploitation/format_strings/index.html`, September 2005.

[57] Ruby Visual Identity Team. Ruby programming language. `http://www.ruby-lang.org/`.

[58] The MITRE Corporation. Common vulnerabilities and exposures (CVE) 2005-1110, April 2005.

[59] N. L. Petroni Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot- a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, August 2004.

[60] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: SoftWare-based ATTestation for embedded devices. In *IEEE Symposium on Security and Privacy*, May 2004.

[61] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *20th ACM Symposium on Operating Systems Principles*, October 2005.

[62] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. Minimal TCB code execution (extended abstract). In *IEEE Symposium on Security and Privacy*, May 2007.

[63] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys'08)*, March 2008.

[64] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim. Remote software-based attestation for wireless sensors. In *Proceedings of the 2nd European Workshop on Security and Privacy in Ad Hoc and Sensor Networks*, July 2005.

[65] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *13th Usenix Security Symposium*, San Diego, California, August 2004.

[66] T. Park and K. G. Shin. Soft tamper-proofing via program integrity verification in wireless sensor networks. *IEEE Transactions on Mobile Computing*, 4:297–309, 2005.

[67] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 193–206, October 2003.

[68] J. Garay and L. Huelsbergen. Software integrity protection using timed executable

agents. In *Proceedings of ACM Symposium on Information, Computer and Communications Security*, pages 189–200, March 2006.

[69] Trusted Computing Group. TPM specifications version 1.2, July 2005.

[70] H. Maruyama, F. Seliger, N. Nagaratnam, T. Ebringer, S. Munetho, and S. Yoshihama. Trusted platform on demand (TPod). Technical report, IBM Corporation, February 2004.

[71] K. Goldman, R. Perez, and R. Sailer. Linking remote attestation to secure tunnel endpoints. In *Proceedings of the first ACM workshop on Scalable trusted computing*, pages 21–24, Alexandria, Virginia, 2006.

[72] E. R. Sparks. A security assessment of trusted platform modules. Technical Report TR2007-597, Dartmouth College, June 2007.

[73] C. Kil, E. C. Sezer, P. Ning, and X. Zhang. Automated security debugging using program structural constraints. In *Proceedings of the Annual Computer Security Applications Conference*, pages 453–462, December 2007.

[74] C. Xiao. Performance enhancements for a dynamic invariant detector. Master's thesis, MIT Department of Electrical Engineering and Computer Science, February 2007.

[75] Free Software Foundation Inc. Gcov, a test coverage program. `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`, 2008.

[76] D. G. Andersen. Mayday: Distributed filtering for internet services. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, March 2003.

[77] A. Juels and J. Brainard. Client puzzles: A cryptographic defense against connection depletion. In *Proceedings of Networks and Distributed Systems Security Symposium (NDSS '99)*, 1999.

[78] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)*, October 2007.

[79] Systemtap. `http://sourceware.org/systemtap/`, 2008.

[80] IBM. TrouSerS: The open-source TCG software stack, September 2007.

[81] SecuriTeam. Symantec VERITAS multiple buffer overflows. `http://www.securiteam.com/securitynews/5JP0L2KI0A.html`, March 2006.

[82] SecuriTeam. Windows 2000 and NT4 IIS .HTR remote buffer overflow. `http://www.securiteam.com/windowsntfocus/5MP0C0U7FU.html`, June 2002.

[83] L. Chen, R. Landfermann, H. Loehr, M. Rohe, A. Sadeghi, and C. Stuble. A protocol for property-based attestation. In *First ACM Workshop on Scalable Trusted Computing*, 2006.

[84] E. Shi, A. Perrig, and L. van Doorn. Bind: A time-of-use attestation service for secure distributed systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[85] M. Milenkovi, A. Milenkovi, and E. Jovanov. Hardware support for code integrity in embedded processors. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, September 2005.

[86] P. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell. Linux kernel integrity measurement using contextual inspection. In *Proceedings of the 2007 ACM workshop on Scalable trusted computing*, Alexandria, Virginia, 2007.

[87] Jr. N. L. Petroni and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 103–115, October 2007.