Stony Brook University

OFFICIAL COPY

The official electronic file of this thesis or dissertation is maintained by the University Libraries on behalf of The Graduate School at Stony Brook University.

© All Rights Reserved by Author.

Defeating Memory Error Exploits Using Automated Software Diversity

A Dissertation Presented

by

Sandeep Bhatkar

to

The Graduate School

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2007

Stony Brook University

The Graduate School

Sandeep Bhatkar

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

Professor R. C. Sekar, (Advisor)
Computer Science Department, Stony Brook University

Professor Scott Stoller, (Chairman)
Computer Science Department, Stony Brook University

Professor Rob Johnson, (Committee Member)
Computer Science Department, Stony Brook University

Professor Somesh Jha, (External Committee Member)
Computer Science Department, University of Wisconsin

This dissertation is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Defeating Memory Error Exploits Using Automated Software Diversity

by

Sandeep Bhatkar

Doctor of Philosophy

in

Computer Science

Stony Brook University

2007

The vast majority of today's security vulnerabilities, accounting for as much as 88% of US-CERT advisories in the past few years, are caused by memory errors such as buffer overflows, heap overflows, integer overflows, and format-string bugs. Previous research has developed techniques for preventing known attack classes exploiting these vulnerabilities. However, attackers quickly developed alternative exploits to circumvent these protections. In contrast to these approaches, comprehensive memory error detection techniques can help track down memory-related bugs, as well as provide full runtime protection from known and future exploits of buffer overflows. However, they typically introduce very high overheads, slowing down programs by a factor of 2 or

more; or require significant modifications to existing code that is too expensive in practice. In contrast, we develop low-overhead techniques that can provide probabilistic protection against all memory error exploits, while preserving full code compatibility. Our techniques are based on automated software diversity. In this dissertation, we undertake a systematic study of possible automated transformations that can defeat memory error exploits, and develop (a) *address space randomization*, which disrupts exploits by making it difficult to predict the object that would be overwritten as a result of a memory corruption attack, and (b) *data space randomization*, which randomizes the interpretation of overwritten data. These randomization techniques make the effect of memory-error exploits non-deterministic, with only a very small chance of success. Thus, an attacker is forced to make several attack attempts, and each failed attempt typically results in crashing the victim program thereby making it easy to detect the attack. Our implementation approaches are based on automatic source-level or (where feasible) binary-level transformations. We present experimental results on several large pieces of software.

To my loving wife Bertille,

my wonderful daughter Lisa,

and my parents.

Contents

List of Tables **ix**

List of Figures **x**

Acknowledgments **xi**

1 Introduction **1**

1.1 Dissertation Organization 6

2 Memory Error Vulnerabilities **7**

2.1 Buffer Overflows 8

2.1.1 Stack Buffer Overflows 8

2.1.2 Static Buffer Overflows 9

2.1.3 Heap Overflow and Double Free Attacks 10

2.1.4 Integer Vulnerabilities 11

2.2 Format String Vulnerabilities 13

3 Motivation for Diversity-Based Defense **14**

4 Address Space Randomization **19**

4.1 Overview of the Technique 19

4.2 Implementation Approaches 23

4.2.1 Operating System-Based Transformation 23

4.2.2 System Tools-Based Transformation 25

4.2.3 Binary Transformation 25

4.2.4 Source Code Transformation 26
4.3 Binary-Only Transformation Approach 27
4.3.1 Code/Data Transformations 28
4.3.2 Stack Transformations 28
4.3.3 Heap Transformations 31
4.3.4 Shared Library Transformations 32
4.3.5 SELF: a Transparent Security Extension for ELF Binaries 34
4.4 Source Code Transformation Approach 44
4.4.1 Static Data Transformations 44
4.4.2 Code Transformations 48
4.4.3 Stack Transformations 51
4.4.4 Heap Transformations 53
4.4.5 Shared Library Transformations 54
4.4.6 Other Randomizations 55
4.4.7 Other Implementation Issues 57
4.5 Performance Results 58
4.5.1 Binary-Only Transformations 58
4.5.2 Source Code Transformations 59
4.6 Effectiveness 63
4.6.1 Memory Error Exploits 64
4.6.2 Attacks Targeting ASR 70

5 Data Space Randomization 73
5.1 Transformation Approach 75
5.1.1 Pointer Analysis 77
5.1.2 Determination of Masks Using Points-to Graph 81
5.2 Implementation 83
5.2.1 Implementation Issues 86
5.3 Performance Results 91
5.4 Effectiveness 92
5.4.1 Memory Error Exploits 92
5.4.2 Attacks Targeting DSR 95

6 Related Work **97**
6.1 Runtime Guarding 97
6.2 Runtime Bounds and Pointer Checking 97
6.3 Compile-Time Analysis Techniques 98
6.4 Randomizing Code Transformations 98
6.4.1 System Call Randomization 99
6.4.2 Instruction Set Randomization 100
6.4.3 Pointer Randomization 100
6.4.4 Address Space Randomization 101

7 Conclusions **102**

List of Tables

1 Runtime performance overhead introduced by binary-only ASR transformations. . . . 58

2 Test programs and workloads for performance evaluation of source code-based ASR transformations. . . . 60

3 Performance overhead introduced by the source code-based ASR transformations on Apache. . . . 61

4 Runtime performance overheads introduced by the source code-based ASR transformations on benchmark programs. . . . 62

5 Distribution of variable accesses in the source code-based ASR transformations. . . . 63

6 Calls and buffer stack allocations in the source code-based ASR transformations. . . . 64

7 Runtime performance overhead introduced by transformations for DSR. . . . 91

List of Figures

1 Stack smashing attack: a buffer overflow in which the current function's return address is replaced with a pointer to injected code. 8

2 Potential locations of padding inserted between stack frames. . 29

3 Format of a typical SELF object file. 37

4 Layout and interpretation of a SELF memory block descriptor. 39

5 (a) Example program, (b) Points-to graph computed by a flow-insensitive analysis, (c) Points-to graphs computed at different program points using flow-sensitive analysis. 79

6 Figure (a) above shows a sample C program for which points-to graph is computed. Figures (b) and (c) show the points-to graphs computed by Andersen's algorithm and Steensgaard's algorithm respectively. 80

7 Properties of Steensgaard's points-to graph: Each node has at most one outdegree and zero or more indegree. Typically a connected component in Steensgaard's graph has a tree like structure as shown in the Figure. However, there is a possibility of only one cycle, which is formed by an edge from the root node to one of the nodes in the component. In the above graph component, such an edge is represented by a dashed line. . . 82

8 A sample example illustrating basic DSR transformations. . . 84

Acknowledgments

I would like to express my deepest gratitude to my advisor, Prof. R. Sekar, for providing me motivation, encouragement and support during the course of the Ph.D. program. I have been greatly benefited from his professional and personal advice. I am truly inspired by his passion, enthusiasm and boundless energy for undertaking practical research that addresses real world problems. I wish to follow his footsteps in my career as a researcher.

I want to thank my dissertation committee members Prof. Scott Stoller, Prof. Rob Johnson, and Prof. Somesh Jha for their insightful comments and suggestions.

I feel privileged to have been a member of Secure Systems Lab. The lab provided me access to excellent computing facilities needed for my research. Most of all, the folks in the lab made it a wonderful workplace. I am especially grateful to Daniel C. DuVarney for his immense help and guidance during the initial stage of this dissertation work. I also want to acknowledge a number of other lab mates who, either directly or indirectly, have provided assistance in my research: V.N. Venkatakrishnan, Wei Xu, Zhenkai Liang, Abhishek Chaturvedi, Weiqing Sun, Vishwas Nagaraja, Shruthi Murthy, and Yogesh Chauhan. I very much appreciated Ajay Gupta's table tennis lessons that brought some welcome relief from stressful times at work. I would like to mention other lab members with whom I have had enjoyable experience: Prem Uppuluri, Alok Tongaonkar, Krishna Kumar, Divya Padbhanabhan, Mohan Channa, Tapan Kamat, Varun Katta, Tianning Li, and several others. It was a pleasure meeting these people, and I thank them all.

In addition, I would like to thank my other friends in Stony Brook, in

particular, Rahul Agarwal, Pavan Vaddadi, Dipti Saha, and Abhiram Govindaraju, for their friendship and support.

Finally, my special thanks are to my family: my parents for their continued support; my brother Sachin for always showing interest in my progress; for my wonderful infant daughter Lisa for bringing immense joy and happiness in my life; and most notably my wife Bertille for her patience, understanding, and care.

CHAPTER 1

Introduction

In recent years, a vast majority of the security attacks have targeted vulnerabilities due to memory errors in C and C++ programs. Since 2003, CERT/CC (now US-CERT) [10] has issued 136 distinct advisories involving COTS software, of which 119 (around 88%) are due to memory errors. We can totally avoid these vulnerabilities by developing more reliable and secure software using type-safe and memory-safe languages such as Java and ML. However, these languages provide little control over memory management and data representation. This control is the main reason that C and C++ are continued to be preferred languages for writing systems software. Moreover, software producers are obsessed with faster runtime performance of C and C++ programs. These facts indicate that C and C++ languages will continue to be used, and hence memory errors will be a dominant source of vulnerabilities in the foreseeable future.

To date, a number of attacks which exploit memory errors have been developed. The earliest of these to achieve widespread popularity was the *stack smashing* attack [17, 33], in which a stack-allocated buffer is intentionally overflowed so that a return address stored on the stack is overwritten with the address of injected malicious code (see Figure 1). To thwart such attacks, several attack-specific approaches were developed, which, in one way or another, prevent undetected modifications to a function's return address. They include the StackGuard [17] approach of putting *canary values* around

the return address, so that stack smashing can be detected when the canary value is clobbered; saving a second copy of return address elsewhere [13, 5]; and others [20].

The difficulty with the above approaches is that while they are effective against stack smashing attacks, they can be defeated by attacks that modify code pointers in the static or heap area. In addition, attacks where control-flow is not changed, but security-critical data such as an argument to `chmod` or `execve` system call are changed, are not addressed. Moreover, recently several newer forms of attacks such as integer overflows, heap overflows, and format-string attacks have emerged: 54 of the 119 memory error related CERT/CC advisories since 2003 are attributed to these newer forms of attacks. (Note that some advisories report multiple vulnerabilities together.) This indicates that new types of memory error related vulnerabilities will continue to be discovered in the future. Therefore, we need to address all the existing exploits as well as exploits which may be discovered in the future.

While we can totally eliminate memory error exploits by using complete memory error detection techniques, unfortunately existing techniques face a few problems that prevent their widespread acceptance. Backwards-compatible bounds checking [25] and its descendant CRED [42] are associated with high overheads, sometimes exceeding 1000%. Lower overheads are reported in [53], but the overheads can still be over 100% for some programs. Overheads are low in approaches such as CCured [34] and Cyclone [24], but they have compatibility issues with legacy C code. Precompiled libraries pose additional compatibility problems with these techniques. Finally, CCured and Cyclone both rely on garbage collection instead of explicit memory management model used in C programs, which can pose another obstacle to their widespread acceptance.

Having considered the progress of research towards defeating memory error exploits, we feel that there is a need for a middle ground between attack-specific and complete memory detection techniques: an approach that offers broad protection, but does not compromise performance or code compatibility. In this dissertation, we explore use of software diversity as a means to achieve

this objective. As a first step towards this direction, we observe that memory error vulnerabilities are a part of a broader issue: vulnerabilities of software monocultures.

Prevalence of software monocultures in today's networked computing environments is considered to be a serious threat to computer security. Because of software monocultures, an attacker capable of exploiting even a single vulnerability in an application can compromise multiple instances of the application across the Internet. For instance, over 90% of the world's computers run Microsoft's operating systems. Thus, a virus or a worm capable of exploiting a vulnerability in Microsoft's software could potentially harm most of the computers in the world. Because of the Internet, worms and viruses can spread very fast over the network leaving insufficient time for users and administrators to patch their systems. Time and again, we have seen worms such as Slammer, Blaster and Code Red that have exploited vulnerabilities of software monocultures to launch such large-scale attacks.

Diversity-based defenses are motivated by the vulnerabilities of software monocultures. These defenses introduce diversity in software systems in such a way that the attackers are forced to customize their techniques. The intention is to cause significant increase in the attack workload, thereby reducing its propagation rate. One of the first proposed approaches for software diversity was N-version programming [4], wherein N different versions of the same software component are implemented independently by N programming teams. The assumption underlying this approach is that the probability of different teams introducing the same software faults is small. This approach is not practical primarily because of the high cost involved in manual development and maintenance of different versions. Hence, approaches that introduce diversity automatically are more desirable.

Automatic introduction of diversity also presents some practical challenges. For instance, it is hard to automatically create diversity at the level of functional behavior of programs such as design and algorithms. For this reason, diversity-based defense is unsuitable for vulnerabilities that involve design or logical errors such as input validation errors and configuration errors.

Given this limitation, we need to consider automatic program transformations that preserve functional behavior. To introduce diversity, such transformations need specification of intended program behavior, which is usually not available. One of the alternatives is to make transformations that preserve programming language semantics as they are readily available. This implies that we can only change low-level program parameters that are not fixed by language semantics. Such parameters include the interface to the operating system, the program memory layout, and representation of data and code. Unfortunately on existing platforms, these parameters lack diversity, thus leading to vulnerabilities that are exploited by many attacks. It turns out that all the attacks on memory errors also exploit the lack of diversity in these low level program parameters.

Several recent defense techniques have been based on introducing diversity in the low level program parameters. Some of them require changes to the operating systems or require architecture-level support. Whereas in this dissertation, our focus is on automated program transformation techniques.

In our diversity-based approach, we do not detect memory errors or their exploits, but rather target a different goal. *Our goal is to maintain the benign program behavior, but disrupt only the attack behavior.* What this means is that attackers can still launch attacks, however, the attacks typically cause a program to behave unpredictably, but do not compromise its security. All known attacks on memory errors are based on corrupting some critical data in the writable memory of a process. For a successful attack, the attackers are required to know two things: (1) location of the critical data in the memory, and (2) how the critical data is used in the program. For instance, in a stack smashing attack, an attacker needs to know the relative location of the return address from the stack buffer, as well as the absolute location of the buffer. Also, the attacker needs to know how the return address will be used, which in this particular case is the common knowledge that the return address represents the location where control is transfered after a function's return. Based on this understanding of memory error exploits, we disrupt the attack behavior by making the effect of memory errors non-deterministic. To this end,

we have developed two randomization techniques to automatically introduce the diversity needed for disrupting attack behaviors:

- Our first technique *Address Space Randomization (ASR)* comprehensively randomizes absolute locations of all code and data objects, as well as their relative distances. This causes memory errors to corrupt random objects. So it becomes difficult for an attacker to know which objects are corrupted in a memory error, and also the attacker cannot control a memory error to corrupt a security-critical object of her choice. Additionally, in the case of attacks involving pointer data corruption, the attacker does not know the value to be used for overwriting the pointer data as all the objects are located at random addresses.

- Our second randomization technique *Data Space Randomization (DSR)* makes the use of corrupted data highly unpredictable by randomizing the representation of data objects. This causes attacker's corrupted data to be interpreted with a random value. As a result, an application may behave unpredictably, and in most cases, it may crash, but at least the security would not be compromised. DSR technique is designed to provide stronger protection than that of ASR. DSR achieves this by providing larger range of randomizations, and by overcoming a few weaknesses of ASR technique.

With these techniques, we address the principal weakness of memory errors: their predictable effect. This, as we shall show, enables our approach to provide broad protection against a wide range of memory error exploits. However, the protection provided is probabilistic, meaning that attackers still have a chance of success. Nonetheless, we shall show that for all known classes of attacks, the odds of success are very low. We present two implementation approaches for ASR: (1) a binary-level transformation that mainly performs absolute address randomizations, and (2) a source-level transformation that performs fine grained relative as well as absolute address randomizations. Our implementation approach for DSR is based on a source-level transformation.

The average runtime overheads for ASR and DSR techniques are around 10% and 15% respectively.

1.1 Dissertation Organization

The rest of the dissertation is organized as follows. In Chapter 2, we describe some of the common memory error vulnerabilities. In Chapter 3, we provide our motivation behind developing diversity-based defense techniques. In Chapter 4, we present our address space randomization technique. Chapter 5 describes our data space randomization technique. Related work is covered in Chapter 6. Finally, we conclude this dissertation in Chapter 7.

CHAPTER 2

Memory Error Vulnerabilities

This chapter gives an overview of various memory error vulnerabilities exploited by attackers. Typically, an attacker exploits a memory error vulnerability to corrupt some data in the writable memory of a process. The exploits can be divided based on the attack mechanism and the attack effect. The primary attack mechanisms known today are:

- *Buffer overflows.* These can be further subdivided, based on the memory region affected: namely, *stack, heap* or *static area overflows.* We note that *integer vulnerabilities* also fall into this category.

- *Format string vulnerabilities.*

Attack effects can be subdivided into:

- *Non-pointer corruption.* This category includes attacks that target security-critical data, e.g., a variable holding the name of a file executed by a program.

- *Pointer corruption.* Attacks in this category are based on overwriting *data* or *code pointers.* In the former case, the overwritten value may point to *injected data* that is provided by the attacker, or *existing data* within the program memory. In the latter case, the overwritten value may correspond to *injected code* that is provided by the attacker, or *existing code* within the process memory.

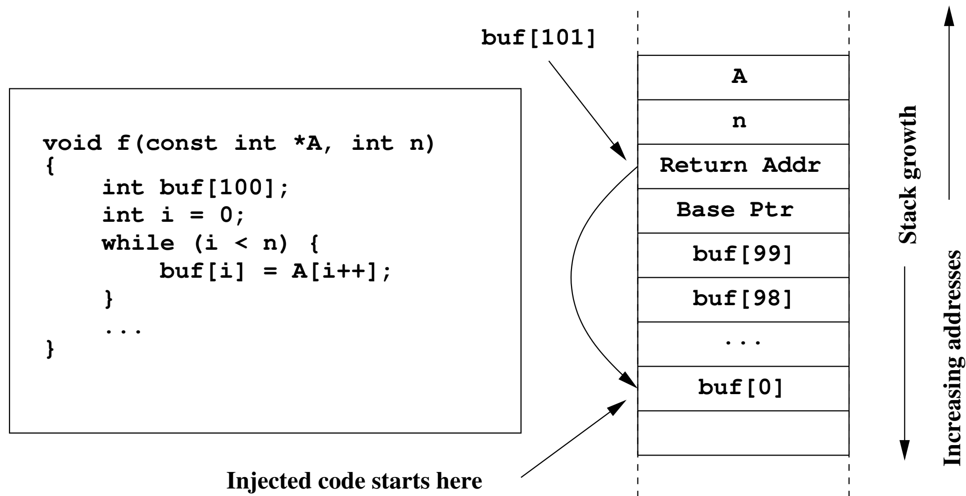


Figure 1: Stack smashing attack: a buffer overflow in which the current function's return address is replaced with a pointer to injected code.

2.1 Buffer Overflows

A buffer overflow occurs when the data written to a buffer, due to insufficient or incorrect bounds checking, corrupts the data in the memory region adjacent to the buffer. Depending upon where the buffer is located, overflows can occur in the stack, static data, or the heap area of the memory.

2.1.1 Stack Buffer Overflows

These overflows typically target the return address, saved base pointer or other pointer-type local variables.

Figure 1 shows an example of a stack smashing attack. On the left of the figure, there is a vulnerable piece of code. The function `f()` includes the definition of a local variable `buf` whose memory is allocated on the stack. In the body of the function, there is a while loop with a buffer overflow vulnerability that ends up writing past the boundary of `buf`. On the right of the figure, there is a snapshot of the stack frame corresponding to an invocation of the

function `f()`. Here the stack is growing in the downward direction, and the addresses are increasing in the upward direction.

As a result of an overflow in `buf`, the adjacent memory gets corrupted. In this case, this memory includes the base pointer, the return address, parameters to the function `f()`, and so on. In a typical stack smashing attack, an attacker exploits the overflow to overwrite the return address with a pointer to the buffer `buf`. In addition, in the same overflow, the attacker injects malicious code into the buffer. Thus when the function returns, the control is automatically transferred to the attacker's injected malicious code.

In a similar way, the attacker can also corrupt function pointers appearing on the stack. Also, the attacker can make the corrupted pointer point to an existing code to gain control. For instance, the attacker can make the control transfer to standard C library functions such as `execve` and `system` that are already mapped in the memory when the program is running. All that is required is to pass appropriate command argument to these functions from the stack. Typically the command used is `/bin/sh`, which gives a shell to the attacker. If the program is run with a root privilege as in the case of setuid to root servers, then the attacker gets a root shell.

Other variations of this attack include corrupting base pointer, or other stack allocated security-critical data such as user-ids, the file name argument of `execve` command, and so on.

2.1.2 Static Buffer Overflows

Like stack buffer overflows, buffer overflows in the static data region can be exploited to corrupt either function pointers or other security-critical data. In the static data region, certain function pointers can be always found at predictable locations. For instance, in the memory image of an ELF binary, addresses of shared library functions are stored in the section `.got` containing the *global offset table* (GOT), and addresses corresponding to constructor and destructor function calls are stored in sections `.ctors` and `.dtors` respectively. These sections are present in static data region and are popular targets of static

buffer overflows.

2.1.3 Heap Overflow and Double Free Attacks

In a typical implementation of a heap memory management, the heap's book-keeping data structure is stored around the blocks of memory allocated for `malloc` requests. So if there is an overflow in the memory allocated in the heap, the data structures get corrupted resulting in erroneous behavior of the heap. The way corrupted data structure is interpreted by the heap management code, gives attackers an ability to overwrite an arbitrary word of memory with an arbitrary value [26]. Double free attacks exploit a similar corruption which occurs in a buggy code that tries to free the same block twice [2]. In both these types of attacks, the vulnerable code pertains to operations over doubly-link lists associated with either free or allocated chunks of memory. For example, the code for deleting an element `N` from a doubly-link list is implementation as follows:

```
N->next->prev = N->prev;  // statement 1
N->prev->next = N->next;  // statement 2
```

Here, the values `N->next` and `N->prev` are a part of the heap data structure that an attacker can corrupt. The statement 1 is equivalent to `*(N->next + C) = N->prev`, where `C` is a constant associated with the offset of the field `prev` of a doubly-link list element. Thus by corrupting the heap memory corresponding to `N->next` and `N->prev` with values `(A - C)` and `V` respectively, an attacker gets an ability to corrupt a word at the memory address `A` with a value `V`.

The other buffer overflow attacks require knowledge of the relative distance of a buffer from the data to be corrupted. This distance could change every execution of the victim program, or even during runtime. If the distance is fixed, it is still difficult to identify it without having the proper knowledge of the memory layout of the program. On the other hand, in the heap overflows described above, the relative distance of the heap data structure from the heap buffer is fixed. An attacker can identify this distance just by knowing the type

of the heap buffer. As heap overflows give an ability to corrupt a data at a chosen absolute address, their exploits are absolute address-dependent.

The memory for C++ objects dynamically created using `new` operator is allocated in the heap region. A C++ object typically has a pointer to a virtual function table (**vtable**) containing addresses of virtual functions used for dynamic dispatching. Attackers also exploit heap overflows to corrupt pointers to **vtable**, and make them point to their injected table of pointers. Thus, dynamic dispatching of virtual functions results in the transfer of control to the locations chosen by the attackers.

2.1.4 Integer Vulnerabilities

An integer type in C language has a fixed width in terms of number of bytes depending upon the underlying machine representation. For instance, on 32-bit architectures, types `short` and `int` have width of typically 2 and 4 bytes respectively. Also, a machine representation has a specific format for representing signed or unsigned integers (e.g., two's complement vs. one's complement). At a high level, integer vulnerabilities arise because programmers do not take into account the machine representation of integers while writing code involving integer computations. The integer vulnerabilities could be of following types:

- **Overflows and underflows.** An arithmetic operation, such as a sum or multiplication, over two or more integers may result in a value greater than the maximum value possible for the machine representation. Such a value when assigned to the fixed width integer representation, gets changed in an undefined manner. Typically, the value "wraps around" from a large positive value to a small negative one. This effect is called integer overflow. Similarly, when the computation results in a value smaller than the minimum value possible with the machine representation, the effect is called integer underflow. In this case, the value typically "wraps around" from a small negative value to a large positive one.

  Due to the wrap-around, boolean conditions which test the values of

integers resulting from arithmetic overflow or underflow are often incorrectly evaluated. For example, if i is sufficiently large, the expression $i + 5$ can overflow, resulting in a negative value, and causing the condition $i + 5 > limit$ to evaluate to false, when it should be true. This effectively disables the bounds checking, allowing an overflow attack to be performed in spite of the bounds checking. In the most common form of this attack, the integer value is used as an index into an array, giving attackers an ability to corrupt an arbitrary word in the memory.

- **Signedness bugs.** Signedness bugs occur when an unsigned integer is interpreted as signed, or when a signed integer is interpreted as unsigned. Following example illustrates this problem.

```
1.   char gbuf[100];
2.   int copybuf(char *buf, int len) {
3.       if (len > 100) return -1;
4.       else return strncpy(gbuf, buf, len);
5.   }
```

  In the above code, a check on line 3 is performed on the length of `buf` before copying it into `gbuf`. Here the problem is that the last parameter `len` of `strncpy` is interpreted as an unsigned integer, whereas the check on line 3 is performed on a signed integer. So for example, if a negative value is passed in `len`, it is possible to escape the check on line 3, but when `strncpy` interprets the negative integer, it is interpreted with a large positive value. Thus, `strncpy` causes a buffer overflow in `gbuf` past its boundary.

  The overflows and underflows described before can also result in similar confusion in the interpretation of signed and unsigned integers.

- **Truncation errors.** When an integer with a larger width is assigned to a smaller width integer, a truncation error occurs. For instance, a type cast from an `int` to a `short` discards the leading bytes of an integer.

This causes wrong interpretation of the resulting integer value, which could be exploited in different ways as described before.

2.2 Format String Vulnerabilities

A format-string vulnerability [44] occurs whenever a program contains a call to the `printf` family of functions with a first parameter (format string) that is provided by an attacker. The common form of this attack uses the somewhat obscure `%n` format parameter, which takes a pointer to an integer as an argument, and writes the number of bytes printed so far to the location given by the argument. The number of bytes printed can be easily controlled by printing an integer with a large amount of padding, e.g., `%432d`. The `printf` function assumes that the address to write into is provided as an argument, i.e., it is to be taken from the stack. If the attacker-provided format string is stored on the stack, and if `printf` can be tricked into extracting arguments from this portion of the stack, then it is possible for an attacker to overwrite an arbitrary, attacker-specified location in memory with attacker-specified data.

Certain types of format-string vulnerabilities are exploited by information leakage attacks. As an example, consider `sprintf` function that takes an attacker-provided format-string and then prints the output into a buffer which is then sent back to the attacker. Now if the attacker sends a format string such as `"%x %x %x %x"`, 4 words from the top of the stack will be printed and disclosed to the attacker. In this way, the attacker can inspect the contents of the stack, and if the stack is known to contain pointers to objects, then the attacker can learn locations of these objects.

CHAPTER 3

Motivation for Diversity-Based Defense

This chapter provides our intuition behind using diversity-based defense against exploits of memory errors. First we understand all the attack techniques and also the role of memory errors which lead to exploitable vulnerabilities.

The goal of an attacker is to cause the target program to execute attack-effecting code. This code itself may be provided by the attacker (*injected code*), or it may already be a part of the program (*existing code*). A direct way to force execution of such code is through a change to the control-flow of the program. This requires the attacker to change a code pointer stored somewhere in memory, so that it points to the code chosen by the attacker. In such a case, when the corrupted code pointer is used as the target of a jump or call instruction, the program ends up executing the code chosen by the attacker. Some natural choices for such code pointers include the return address (stored on the stack), function pointers (stored on the stack, the static memory region, or the heap), the global offset table (GOT) used in dynamic linking, and buffers storing `longjmp` data. An indirect way to force execution of attack-effecting code is to change security-critical data that is used by the program in its normal course of execution. Examples of such data include arguments to a `chmod` or `execve` system call, variables holding security-critical data such as a flag indicating whether a user has successfully authenticated or

not, etc. In summary, the success of an exploit of a vulnerability depends on the following two factors:

1. *The vulnerability allows corruption of a critical target data.* A vulnerability involving a memory error may allow corruption of data only in specific ways. For example, a buffer overflow attack requires the knowledge of the relative distance of the buffer from the target data. On the other hand, in a heap overflow or format-string attack, the attacker needs to know the absolute location of the target data. Thus, depending on the type of the vulnerability, its exploits may either be *relative address-dependent* or *absolute address-dependent*.

2. *The target data is overwritten with a correct value.* The target data corrupted in an exploit is either a pointer value or non-pointer data. Since code sections cannot be overwritten in most modern operating systems, there are three possible types of data: code-pointer, data-pointer, and non-pointer data. An attack involving corruption of pointer values requires the attacker to know the absolute address of a data or code object for overwriting the pointer values. Therefore, this kind of attack is *absolute address-dependent*. In contrast, a non-pointer data can be corrupted with a value that is independent of memory addresses. Hence attacks on non-pointer data depend only on the first factor mentioned above.

We now try to understand how memory errors lead to security vulnerabilities. For this, it is helpful to know how memory is allocated for a program.

A program's memory consists of different regions such as code, data, the stack, and the heap. In most modern operating systems, a program's code and data regions are mapped at fixed addresses in the memory, and also the relative distances between the individual data and code objects remain fixed. Memory for objects on the stack and the heap is allocated dynamically. However, for a given program execution, the stack and the heap objects are allocated at predictable locations, and these locations can be learnt using memory monitoring tools such as debuggers.

The attack techniques mentioned above exploit a particular weakness of memory errors, which is that the effect of memory errors is predictable. To illustrate this, we will first describe different types of memory errors, and then show how attackers exploit predictable effect of each of these types. Intuitively, a memory error occurs in C programs when the object accessed via a pointer expression is different from the one intended by the programmer. The intended object is called the *referent* of the pointer. Memory errors can be classified into *spatial* and *temporal errors:*

I. *A spatial error* occurs when dereferencing a pointer that is outside the bounds of its referent. It may be caused as a result of:

  (a) *Dereferencing non-pointer data,* e.g., a pointer may be (incorrectly) assigned from an integer, and dereferenced subsequently. If an attacker has control over the non-pointer value assigned to the target pointer, she can make the pointer dereference to access any object as objects are located at predictable locations in the memory.

  (b) *Dereferencing uninitialized pointers.* This case differs from the first case only when a memory object is reallocated. The contents of uninitialized pointers may become predictable if the previous use of the same memory location can be identified. For instance, suppose that during an invocation of a function `f`, its local variable `v` holds a valid pointer value. If `f` is invoked immediately by its caller, then `v` will continue to contain the same valid pointer even before its initialization. This gives attackers an opportunity to exploit the predictable value of the pointer.

  (c) *Valid pointers used with invalid pointer arithmetic.* The most common form of memory access error, namely, out-of-bounds array access, falls in this category. Since relative distances between memory objects are predictable, attackers can determine the target object accessed as a result of invalid pointer arithmetic.

II. *A temporal error* occurs when dereferencing a pointer whose referent no longer exists, i.e., it has been freed previously. If the invalid access goes

to an object in the free memory, then it causes no errors. But if the memory has been reallocated, then temporal errors allow the contents of the reallocated object to be corrupted using the invalid pointer. The result of such errors become predictable only when the purpose of reuse of the memory location is predictable.

It may appear that temporal errors, and errors involving uninitialized pointers, are an unlikely target for attackers. In general, it may be hard to exploit such errors if they involve heap objects, as heap allocations tend to be somewhat unpredictable. However, stack allocations are highly predictable, so these errors can be exploited in attacks involving stack-allocated variables.

Based on our understanding of various attack techniques and types of memory errors, we conclude that a memory error which allows corruption of a critical data value leads to a security vulnerability. As locations of objects in the memory are predictable, attackers can determine the data corrupted in a memory error, and in most cases attackers can actually control the effect of the memory error to corrupt the data of their choice. Additionally, attackers are also able to control the use of corrupted data value. If the data corresponds to a pointer, attackers know the pointer value to be used for corruption as locations of objects are predictable; value of a non-pointer data is independent of memory addresses, so attackers do not need any information to corrupt it.

As a first step towards a comprehensive solution, our diversity-based defense approach seeks to make the effect of memory errors unpredictable. To achieve this, we need to randomize the objects that are corrupted as a result of a memory error vulnerability, and/or randomize the use of corrupted value. To this end, we have developed two randomization techniques.

Our first technique ASR randomizes absolute and relative distances of all the objects in the memory. The effect of this is that the objects that are corrupted become unpredictable, and also the use of corrupted pointer data becomes unpredictable. Note that this technique can not make the use of corrupted non-pointer data unpredictable.

Our second technique DSR randomizes the representation of data. In this technique, attacker can determine the data object that gets corrupted in a memory error. However, on corruption, the object gets assigned a wrong value because of its randomized representation. The wrong value of the object results in an unpredictable interpretation of the object.

Since our approach uses randomization, the protection offered is probabilistic. This means that attackers still have a chance of success. However, as we shall show later, our approach performs comprehensive randomizations leaving a very small chance of success for attackers. So attackers are forced to make many attempts on average before an attack succeeds, with each unsuccessful attack causing the target program to crash, increasing the likelihood that the attack will be detected. Moreover, an attack that succeeds against one victim will not succeed against another victim, or even for a second time against the same victim. This aspect makes it particularly effective against large-scale attacks such as Code Red, since each infection attempt requires significantly more resources, thereby greatly reducing the propagation rate of such attacks.

The randomization techniques will be described in more details in subsequent sections.

CHAPTER 4

Address Space Randomization

4.1 Overview of the Technique

This technique makes the memory locations of program objects (including code as well as data objects) unpredictable. This is achieved by randomizing the absolute locations of all objects, as well as the relative distance between any two objects. These objectives can be achieved using a combination of the following transformations:

I. Randomize the base addresses of memory regions. By changing the base addresses of code and data segments by a random amount, we can alter the absolute locations of data resident in each segment. If the randomization is over a large range, say, between 1 and 100 million, the virtual addresses of code and data objects become highly unpredictable. Note that this does not increase the physical memory requirements; the only cost is that some of the virtual address space becomes unusable. The details depend on the particular segment:

1. *Randomize the base address of the stack.* This transformation has the effect of randomizing all the addresses on the stack. A classical stack smashing attack requires the return address on the stack to be set to point to the beginning of a stack-resident buffer into which the attacker has injected malicious code. This becomes very difficult

when the attacker cannot predict the address of such a buffer due to randomization of stack addresses. Stack-address randomization can be implemented by subtracting a large random value from the stack pointer at the beginning of the program execution.

2. *Randomize the base address of the heap.* This transformation randomizes the absolute locations of data in the heap, and can be performed by allocating a large block of random size from the heap. It is useful against attacks where attack code is injected into the heap in the first step, and then a subsequent buffer overflow is used to modify the return address to point to this heap address. While the locations of heap-allocated data may be harder to predict in long-running programs, many server programs begin execution in response to a client connecting to them, and in this case the heap addresses can become predictable. By randomizing the base address of the heap, we can make it difficult for such attacks to succeed.

3. *Randomize the starting address of dynamically-linked (shared) libraries.* This transformation has the effect of randomizing the location of all code and static data associated with shared libraries. This will prevent *existing code* attacks (also called *return-into-libc* attacks), where the attack causes a control-flow transfer to a location within the library that is chosen by the attacker. It will also prevent attacks where static data is corrupted by first corrupting a pointer value. Since the attacker does not know the absolute location of the data to be corrupted, it becomes difficult to use this strategy.

4. *Randomize the locations of routines and static data in the executable.* This transformation has the effect of randomizing the locations of all functions in the executable, as well as the static data associated with the executable. The effect is similar to that of randomizing the starting addresses of shared libraries.

We note that all of the above four transformations are also implemented in the PaX ASLR [36] system, but their implementation relies on kernel patches rather than program transformations. The following three classes of transformations are new to our system. The first two of them have the effect of randomizing the relative distance between the locations of two routines, two variables, or between a variable and a routine. This makes it difficult to develop successful attacks that rely on adjacencies between data items or routines. In addition, it introduces additional randomization into the addresses, so that an attacker that has somehow learned the offsets of the base addresses will still have difficulty in crafting successful attacks. As an effect of the last class of transformation, certain attacks become impossible.

II. Permute the order of variables/routines. Attacks that exploit relative distances between objects, such as attacks that overflow past the end of a buffer to overwrite adjacent data that is subsequently used in a security-critical operation, can be rendered difficult by a random permutation of the order in which the variables appear. Such permutation makes it difficult to predict the distance accurately enough to selectively overwrite security-critical data without corrupting other data that may be critical for continued execution of the program. Similarly, attacks that exploit relative distances between code fragments, such as partial pointer overflow attacks (see Section 4.6.2), can be rendered difficult by permuting the order of routines. There are three possible rearrangement transformations:

1. *permute the order of local variables in a stack frame*
2. *permute the order of static variables*
3. *permute the order of routines in shared libraries or the routines in the executable*

III. Introduce random gaps between objects. For some objects, it is not possible to rearrange their relative order. For instance, local variables

of the caller routine have to appear at addresses higher than that of the callee. Similarly, it is not possible to rearrange the order of `malloc`-allocated blocks, as these requests arrive in a specific order and have to be satisfied immediately. In such cases, the locations of objects can be randomized further by introducing random gaps between objects. There are several ways to do this:

1. *Introduce random padding into stack frames.* The primary purpose of this transformation is to randomize the distances between variables stored in different stack frames, which makes it difficult to craft attacks that exploit relative distances between stack-resident data. Additionally, gaps could be introduced within the variables stored in the same stack frame. The size of the padding should be relatively small to avoid a significant increase in memory utilization.
2. *Introduce random padding between successive* `malloc` *allocation requests.*
3. *Introduce random padding between variables in the static area.*
4. *Introduce gaps within routines, and add jump instructions to skip over these gaps.*

IV. Introduce inaccessible memory regions between objects. The main purpose of this transformation is to isolate buffer variables from other variables, so that any buffer overflow cannot corrupt non-buffer variables. The transformation can be implemented by separating buffer variables from other variables, and by introducing inaccessible (neither readable nor writable) memory pages between them. For additional protection, we can introduce inaccessible pages periodically within the memory region containing the buffer variables. This is to bound the span of buffer-to-buffer overflows.

Similarly, we can introduce inaccessible pages within the code segment of the memory. Since absolute and relative addresses of routines are

randomized, attackers are forced to guess code addresses. The inaccessible pages provide additional protection because any guessed addresses within the inaccessible pages would result in immediate program crash.

4.2 Implementation Approaches

There are two basic issues concerning the implementation of the transformations mentioned in the previous section. The first implementation issue is concerned with the time when the randomization amounts are determined. Possible choices here are (a) transformation time, (b) beginning of program execution, and (c) continuously changing during execution. Clearly, choice (c) increases the difficulty of attacks, and is hence preferred from the point of security. Choices (a) or (b) may be necessitated due to performance or application binary interface compatibility considerations. For instance, it is not practical to remap code at different memory locations during program execution, so we cannot do any better than (b) for this case.

The second issue concerns the timing of the transformations: they may be performed at compile-time, link-time, installation-time, or load-time. Generally speaking, higher performance can be obtained by performing transformations closer to compilation time. On the other hand, by delaying transformations, we avoid making changes to system tools such as compilers and linkers. In contrast to these options, there exists an alternative approach that does not require transformation of applications — incorporating randomizing transformations as a part of operating system functionality. In this section, we describe some of the possible implementation approaches.

4.2.1 Operating System-Based Transformation

Virtual memory layout of a process is determined by the underlying operating system. Typically, the code generated for an executable uses absolute virtual addresses (determined at link-time) to access code and data objects. Therefore, the operating system is forced to map code and data segments of the

executable at fixed virtual memory addresses in order to ensure correct execution. On the other hand, a shared library contains *position-independent code* (PIC), which allows it to be mapped to almost any virtual address. However in most cases, the dynamic linker maps shared libraries at predictable virtual addresses.

Memory for stack and heap objects is allocated dynamically at runtime. However, the base addresses of the stack and the heap are predictable. For example, the base of the stack of all the processes in RedHat Linux systems on 32 bit x86 architectures is fixed at the virtual address 0xC0000000. On the same platform, the base of the heap is different for each program and is fixed at an address following the end of the data segment of the program. Hence the base of the heap is also predictable for each program and it remains the same for different execution runs of the program. As a result of this, for a given execution path, even the individual heap and stack objects are allocated at the same addresses for different runs of the program.

Some of the address space randomization features can be easily implemented as a part of operating system functionality. For instance, simple operating system changes can be done to achieve absolute address randomization by randomizing the base addresses of different memory regions, e.g., the operating system can choose random addresses to assign the base of the stack and shared libraries. Randomization of code and data segments of an executable is a difficult task for an operating system, and it involves significant increase in the performance overheads [36]. A practical approach is to compile an executable with PIC so that it can be mapped at random virtual addresses just like shared libraries. As the heap follows the data segment of the executable, its base address is automatically randomized.

The main benefit of this approach is that it is transparent to existing applications; source code of applications is not required, and there is no need to modify existing tools such as compilers or linkers. This approach, however, cannot be used for performing relative address randomizations. This is because relative address randomization requires information about the internal details of a program to control memory allocation for individual code or data objects,

and an operating system has no way of getting this information directly from program binaries.

4.2.2 System Tools-Based Transformation

We can use modified system tools such as compilers and linkers to transform programs for randomizations. There are some difficulties associated with this approach. As an example, consider the option of using modified compilers. The problem here is that the compilers are language-specific, and so typically there are different compilers to handle different languages. Even for the same language, there may exist multiple compilers distributed by multiple vendors. Thus implementing the required transformations using compilers is not a good option because in order to make the approach widely applicable and practical, we need to modify all the compilers. The same problem applies to the other system tools such as linkers, assemblers, and loaders.

4.2.3 Binary Transformation

A binary transformation approach is preferable because it does not require changes to the system tools. Moreover, performing transformations at binary-level means that the transformations can be applied to proprietary software that is distributed in binary-only form. This makes it easier for the approach to be widely accepted and used. Binary transformations can be performed statically or dynamically.

Static Binary Transformation. It is well known that binary analysis and transformations are complex problems. More specifically, binary files lack sufficient relocation information to distinguish code from data, which makes it difficult to disassemble instructions correctly. Other contributing factors include: variable instruction size, indirect jump or call instructions, etc. (Refer to Section 4.3 for additional information.) Due to these issues and a few others, existing binary analysis and editing tools have restricted applications.

Consequently, some of our transformations, more specifically, those required for relative distance randomizations are not possible on binaries.

Dynamic Binary Transformation. The difficulties associated with static analysis of binaries can be overcome by dynamic analysis. For example, for accurate and complete disassembly of code, it is necessary to identify the target of an indirect jump or call instruction. Static analysis may not be able to identify the target as it could be the result of a computation performed at runtime. On the other hand, the dynamic analysis technique can find out the target at runtime and continue the analysis with disassembly from that target address. In order to perform the randomizing transformations, additional effort is needed to transform the analyzed code at runtime. For fine grained randomizations, the transformations would be required at the level of individual instructions. Considering that a lot of work is required for analysis as well as transformation of binaries at runtime, the runtime performance is likely to be high. DynamoRIO [9] is a popular dynamic binary transformation tool because it incorporates various optimizations to minimize the runtime overheads for basic analysis of binaries. Even so, these overheads could be high in the range of 20% to 70%. So the actual overheads would be much higher if we also consider the overheads due to instruction-level transformations.

This approach is dependent on the ability of binary transformation tools, which in turn depends on architectures and binary file formats. However, the approach itself is independent of programming languages, binary file formats, and even the operating system.

4.2.4 Source Code Transformation

Availability of source code gives more freedom for performing the randomizing transformations. It allows all the transformations required for absolute as well as relative address randomizations. The only problem with this approach is that it is language-dependent. Nevertheless, the approach is transparent to the system tools and the operating system.

In this dissertation, we present two implementation approaches for ASR. The first approach, which is based on a binary-level transformation, is useful for software that is distributed in binary-only form. This approach mainly provides absolute address randomization, and is effective against most of the known exploits of memory errors. The second approach, which is based on a source code transformation, improves over the first approach by providing absolute as well as relative address randomizations. As a result, it offers wider protection against memory error exploits.

Our implementation targets Intel x86 architectures running ELF-format [35] executables on a Linux operating system.

4.3 Binary-Only Transformation Approach

In our implementation of transformation of programs at binary-level, we have added our transformation code in LEEL binary-editing tool [54]. One of the complications associated with this implementation approach is that on most architectures, safe rewriting of machine code is not always possible. This is due to the fact that data may be intermixed with code, and there may be indirect jumps and calls. These two factors make it difficult to extract a complete control-flow graph, which is necessary in order to make sure that all code is rewritten as needed, without accidentally modifying any data. Most of our transformations are simple and not impacted by the difficulty of extracting an accurate control-flow graph. For instance, the stack base randomization requires instrumentation of only one routine, and it doesn't need the control-flow graph of that routine. However, a few transformations (e.g., stack frame padding) are dependent on control-flow graphs of routines. This is a challenge when some routines cannot be accurately analyzed. Therefore, we take a conservative approach to overcome this problem — rewriting only those routines that can be completely analyzed. Further details can be found in the description of relevant transformations.

4.3.1 Code/Data Transformations

Relocation of a program's code (text) and data segments is desirable in order to prevent attacks which modify a static variable or jump to existing program code. The easiest way to implement this randomization is to convert the program into a shared library containing position-independent code. In the case of GCC, such code is generated using the compiler option `-fPIC`. The final executable is created by introducing a new `main` function which loads the shared library generated from the original program (using `dlopen`) and invokes the original `main`. This allows random relocation of the original program's code and data segments at a page-level granularity. (Finer granularity of randomization is achieved by using the link-time shared library transformation described in a later subsection.) Since position-independent code is less efficient than its absolute address-dependent counterpart, it introduces a modest amount of extra runtime overhead.

An alternative approach is to relocate the program's code and data at link-time. In this case, the code need not be position-independent, so no runtime performance overhead is incurred. Link-time relocation of the base address of code and data segments can be accomplished by simple modifications to the linker script used by the linker.

Our implementation supports both of these approaches. Section 4.5.1 presents the runtime overheads we have observed with both the approaches.

4.3.2 Stack Transformations

Stack base address randomization: The base address of the stack is randomized by extra code which is added to the code segment of the program. The code is spliced into the execution sequence by inserting a jump instruction at the beginning of `main` routine. The new code generates a random number between 1 and 10^8, and decrements the stack pointer by this amount. In addition, the memory region corresponding to this "gap" is write-protected using `mprotect` system call. The write-protection ensures that any buffer overflow attack that overflows beyond

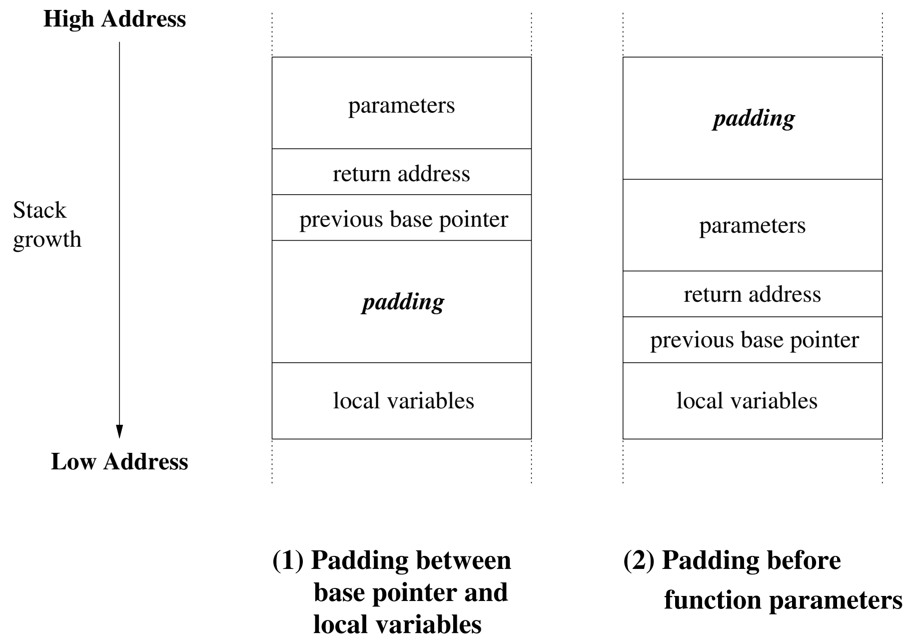


Figure 2: Potential locations of padding inserted between stack frames.

the base of the stack into the read-only region will cause the victim program to crash.

Random stack frame padding: Introducing padding within stack frames requires that extra storage be pushed onto the stack during the initialization phase of each subroutine. There are two basic implementation issues that arise.

The first issue is the randomization of the padding size, which could be static or dynamic. Static randomization introduces practically no runtime overhead. Dynamic randomization requires the generation of a random number at regular intervals. Additionally, the amount of extra code required for each function preamble is significant. Moreover, if the randomization changes the distance between the base of the stack frame and any local variable (from one invocation of a function to the next) then significant changes to the code for accessing local variables

are required, imposing even more overheads. For these reasons, we have currently chosen to statically randomize the padding, with a different random value used for each routine.

The second issue concerns the placement of the padding. As shown in Figure 2, there are two basic choices: (1) between the base (or frame) pointer and local variables, or (2) before parameters to the function:

1. *Between the base pointer and local variables.*
   This requires transformation of the callee to modify the instruction which creates the space for the local variables on the stack. Local variables are accessed using instructions containing fixed constants corresponding to their offset from the base pointer. Given that the padding is determined statically, the transformation simply needs to change the constants in these instructions. The main benefit of this approach is that it introduces a random gap between local variables of a function and other security-critical data on the stack, such as the base pointer and the return address, and hence makes typical stack smashing attacks difficult.

2. *Before parameters to the function.*
   This is done by transforming the caller. First, the set of argument-copying instructions is located (usually `PUSH` instructions). Next, padding code is inserted just before these instructions. The primary advantage of this approach is that the amount of padding can change dynamically. Disadvantages of the approach are (a) in the presence of optimizations, the argument-pushing instructions may not be contiguous, which makes it difficult to determine where the padding is to be introduced, and (b) it does not make stack smashing attacks any harder since the distance between the local variables and the return address is left unchanged.

We have implemented the first option. As mentioned earlier, extraction of accurate control-flow graphs can be challenging for some routines. To

ensure that our transformation does not lead to an erroneous program, the following precautions are taken:

- Transformation is applied to only those routines for which accurate control-flow graphs can be extracted. The amount of padding is randomly chosen, and varies from 0 to 256, depending on the amount of storage consumed by local variables, and the type of instructions used within the function to access local variables (byte- or word-offset). From our experience on instrumentation of different binaries, we have found that around $95 - 99\%$ of the routines are completely analyzable.
- Only functions which have suitable behavior are instrumented. In particular, the function must have at least one local variable and manipulate the stack in a standard fashion in order to be instrumented. Moreover, the routines should be free of non-standard operations that reference memory using relative addressing with respect to the base pointer.
- Only *in place* modification of the code is performed. By *in place*, we mean that the memory layout of the routines is not changed. This is done in order to avoid having to relocate the targets of any indirect calls or jumps.

These precautions have limited our approach to instrument only 65% to 80% of the routines. We expect that this figure can be improved to 90+% if we allow modifications that are not in-place, and by using more sophisticated analysis of the routines.

4.3.3 Heap Transformations

The base address of the heap can be randomized using a technique similar to the stack base address randomization. Instead of changing the stack pointer, code is added to allocate a randomly-sized large chunk of memory, thereby

making heap addresses unpredictable. In order to randomize the relative distances between heap data, a wrapper function is used to intercept calls to `malloc`, and randomly increase the sizes of dynamic memory allocation requests by 0 to 25%. On some operating systems, including Linux, the heap follows the data segment of the executable. In this case, randomly relocating the executable causes the heap to also be randomly relocated.

4.3.4 Shared Library Transformations

In the ELF binary file format, the program header table (PHT) of an executable or a shared library consists of a set of structures which hold information about various segments of a program. Loadable segments are mapped to virtual memory using the addresses stored in the `p_vaddr` fields of the structures (for more details, see [35]). Since executable files typically use (non-relocatable) absolute code, the loadable segments must reside at the addresses specified by `p_vaddr` in order to ensure correct execution.

On the other hand, shared object segments contain position-independent code (PIC), which allows them to be mapped to almost any virtual address. However, in our experience, the dynamic linker almost always chooses to map them starting at `p_vaddr`, e.g., this is the case with `libc.so.6` (the Standard C library) on RedHat Linux distributions. The lowest loadable segment address specified is `0x42000000`. Executables start at virtual address `0x08048000`, which leaves a large amount of space (around 927 MB) between the executable code and the space where shared libraries are mapped. Typically, every process which uses the dynamically-linked version of `libc.so.6` will have it mapped to the same base address (`0x42000000`), which makes the entry points of the `libc.so.6` library functions predictable. For example, if we want to know the virtual address where function `system()` is going to be mapped, we can run the following command:

```
$ nm /lib/i686/libc.so.6 | grep system
42049e54 T __libc_system
2105930 T svcerr_systemerr
```

```
42049e54 W system
```

The third line of the output shows the virtual address where `system` is mapped.

In order to prevent existing code attacks which jump to library code instead of injected code, the base address of the libraries should be randomized. There are two basic options for doing this, depending on when the randomization occurs. The options are to do the randomization (1) once per process invocation, or (2) statically. The trade-offs involved are as follows:

1. *Dynamically randomize library addresses using* `mmap`*.* The dynamic linker uses `mmap` system call to map shared libraries into memory. The dynamic linker can be instrumented to instead call a wrapper function to `mmap`, which first randomizes the load address and then calls the original `mmap`. The advantage of this method is that in every program execution, shared libraries will be mapped to different memory addresses.

2. *Statically randomize library addresses at link-time.* This is done by dynamically linking the executable with a "dummy" shared library. The dummy library need not be large enough to fill the virtual address space between the segments of the executable and standard libraries. It can simply introduce a very large random gap (sufficient to offset the base addresses of the standard libraries) between the load-addresses of its `text` and `data` segments. Since shared libraries use relative addressing, the segments are mapped along with the gap.

   On Linux systems, the link-time gap can be created by using the `ld` options `-Tbss`, `-Tdata` and `-Ttext`. For example, consider a dummy library which is linked by the following command:

   ```
   $ ld -o libdummy.so -shared dummy.o  -Tdata 0x20000000
   ```

   This causes the load address of the text segment of `libdummy.so` to be `0x00000000` and the load address of data segment to be `0x20000000`,

creating a gap of size `0x20000000`. Assuming the text segment is mapped at address `0x40014000` (Note: addresses from `40000000` to `40014000` are used by the dynamic linker itself: `/lib/ld-2.2.5.so`), the data segment will be mapped at address `0x60014000`, thereby offsetting the base address of `/lib/i686/libc.so.6`.

The second approach does not provide the advantage of having a freshly randomized base address for each invocation of the program, but does have the benefit that it requires no changes to the loader or rest of the system. With this approach, changing the starting address to a different (random) location requires the library to be re-obfuscated (to change its preferred starting address). Our implementation supports both the above approaches.

A potential drawback of the above approaches is that they do not provide sufficient range of randomization on 32-bit architectures. In particular, with a page size of 4096 ($= 2^{12}$) bytes on Linux, uncertainty in the base address of a library cannot be much larger than 2^{16}, which makes them susceptible to brute-force attacks [45]. We address this problem by a link-time transformation, i.e., a binary-level transformation. However, we will describe this transformation when we discuss our source code transformation approach. Using this link-time transformation, we can increase the space of randomization for shared libraries up to 2^{26} addresses.

4.3.5 SELF: a Transparent Security Extension for ELF Binaries

Ideally, we would like to choose an implementation approach purely based on binary transformations. Such an approach is desirable as it facilitates transformation of propriety software distributed only in binary form. Moreover, the approach would be independent of programming languages. Also, we do not need to alter system tools such as compilers and linkers. Unfortunately, existing binary rewriting techniques offer inadequate support to implement all of our randomizing transformations. For instance, in our current implementation, we are unable to apply transformations required for randomization of

relative distances between code and data objects.

In this section, we first describe problems associated with analysis and transformation of binaries. We then propose an extension to a binary file format ELF (Executable and Linking Format), which contains additional information that allows proper analysis and transformation of binary files.

Limitations of binary rewriting techniques stem from the difficulties associated with analysis of binaries. For any kind of analysis or transformation on binaries, it is important to be able to retrieve information from binaries and also to manipulate it. However, many practical difficulties arise in statically analyzing a binary file.

Problems with existing formats: Some of the difficulties encountered with current binary file formats are:

- *Distinguishing code from data:* The fundamental problem in decoding machine instructions is that of distinguishing code (i.e., instructions) from data within an executable. Machine code in the text segment often contains data embedded between machine instructions. For example, in C programming language, typical compilers generate code for a `case` statement as an indirect jump to an address loaded from some location in a *jump table*. This table, which contains the target addresses, is also placed along the instructions in the text segment. Another example is that of the data inserted in instructions for alignment purpose, presumably to improve instruction-fetch hit rates. Such data causes disassembly problems in architectures such as Intel x86, which has a dense instruction set. Thus, most data bytes are likely to appear as valid beginning bytes of instructions. This is a major source of problem for disassembly based on *linear sweep* algorithm [43], which in the process of decoding bytes sequentially, misinterprets the embedded data as instructions.

- *Indirect jumps/calls:* One of the ways to avoid misinterpretation of data as instructions is to use *recursive traversal* disassembly algorithm [43] in which disassembly starts from the entry point of the program, and

whenever there is a jump instruction, it continues along the control-flow successors of the instruction. However, this approach fails to obtain complete disassembly in presence of indirect jumps because of the difficulty involved in identifying the targets of the instructions. A similar difficulty to statically predicting the targets of function calls is presented by indirect call instructions.

- *Variable-length instruction sets:* Unlike RISC architectures, in which all instructions are fixed-sized, CISC architectures (such as `x86`) often have variable-length instructions, which complicates their disassembly. In presence of variable-length instructions, a single disassembly error increases the likelihood of errors in disassembly of many of the subsequent instructions. On the other hand, a disassembly error in fixed-length instructions does not propagate to subsequent instructions.

- *Distinguishing address and non-address constants:* It is difficult to distinguish between addresses and non-address constants. Making this distinction is necessary in order to perform any modification to a binary which causes code or data to be relocated. For existing binary file formats, there is no general mechanism to correctly make this distinction in every case.

- *Instructions generated through non-standard mechanisms:* Sometimes executables contain instructions generated through non-standard mechanisms (such as hand-written assembly code). Such instruction sequences may violate high-level invariants that one normally assumes hold true for compiler generated code. For example, in many mathematical libraries it not uncommon for control to branch from one function into middle of another, or fall through from one function into another, instead of using a function call. This kind of code complicates analysis of binaries considerably.

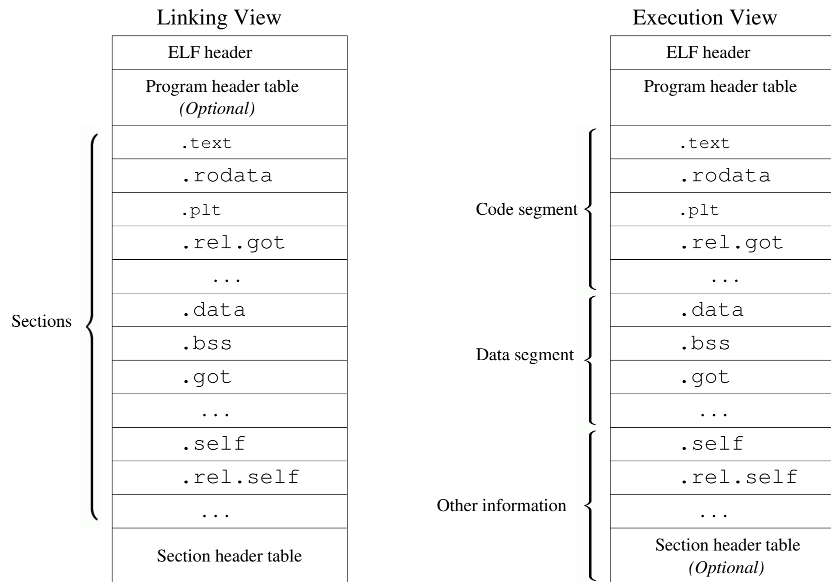


Figure 3: Format of a typical SELF object file.

4.3.5.1 ELF format

ELF files fall into the following three types:

- *Executable files* containing code and data suitable for execution. This specifies the memory layout of the process image of a program.
- *Relocatable (object) files* containing code and data suitable for linking with other object files to create an executable or a shared object file.
- *Shared object files* (shared libraries) containing code and data suitable for the link editor (`ld`) at the link-time and the *dynamic linker* (`ld.so`) at runtime.

A binary file typically contains various headers that describe the organization of the file, and a number of sections which hold various information about the program such as instructions, data, read-only-data, symbol table, relocation tables and so on.

Executable and shared object files (as shown in the execution view of Figure 3) are used to build a process image during execution. These files must

have a *program header table*, which is an array of structures, each describing a segment or other information needed to prepare the program for execution. An *object file segment* contains one or more sections. Typically, a program has two segments: (1) a code segment comprised of sections such as `.text` (instructions) and `.rodata` (read-only data) (2) a data segment holding sections such as `.data` (initialized data) and `.bss` (uninitialized data). The code segment is mapped into virtual memory as a read-only and executable segment so that multiple processes can use the code. The data segment has both read and write permissions and is mapped exclusively for each process into the address space of that process.

A relocatable file (as shown in the linking view of Figure 3) does not need a program header table as the file is not used for program execution. A relocatable file has sufficient relocation information in order to link with other similar relocatable files. Also, every relocatable file must have a *section header table* containing information about the various sections in the file.

By default, an ELF executable file or a shared object file does not contain relocation information because it is not needed by the loader to map the program into process memory. Relocation information identifies address-dependent byte-streams in the binary that need modification (relocation) when the linker re-maps the binary to different addresses. A single entry in a relocation table usually contains the following: (1) an offset corresponding to either the byte-offset from the beginning of the section in a relocatable file, or the virtual address of the storage unit in an executable file, and (2) information about the type of relocation (which is processor specific) and a symbol table index with respect to which the relocation will be applied. For example, a call instruction's relocation entry would hold the symbol table index of the function being called.

Many binary tools rely on relocation information for analysis and transformation of binaries. Transformation of a binary file often requires modifications in which the subsequences of the machine code are moved around. When this is done, the data referenced by relocation entries must be updated to reflect the new position of corresponding code in the executable. In the absence

Tag	Address	Alignment	Width

Field	**Meaning**
Tag	Summary of block contents
Address	Starting address of block
Alignment	Alignment requirements of block
Width	Block size in bytes

Figure 4: Layout and interpretation of a SELF memory block descriptor.

of relocation information, binary tools resort to nontrivial program analysis techniques [31, 14, 54, 37]. These techniques are inadequate and hence the tools adopt conservative strategies, thereby restricting the their efficacy in performing various transformations. Also, due the fact that relocation information is not required for execution, many linkers do not have option flags to retain the information in the executables. In addition, even the presence of relocation information does not help in certain kinds of binary transformations. In particular, the relocation table does not give sufficient information about the data and instructions used in the machine code. Hence, certain transformations which require complete disassembly of instructions and modification of data are not possible. The SELF extension, described in the following section, is specifically intended to deal with this problem.

4.3.5.2 SELF extension

The SELF extension will reside in a section named `.self` which will be indicated by the section header table in both execution and linking views as shown the Figure 3. The purpose of the extension is to provide additional information about instructions and data used in various sections. As discussed before, much of the information is available in relocation tables and symbol table. An object file compiled with the debug flag option contains debug information which can also provide useful information such as type and size of data, addresses of functions, etc. However, in a typical software distribution

model, binary files are compiled with optimizations which render debug information incorrect. Also, binaries are stripped, which means that they do not have a symbol table. Here, the objective is to distribute slim and efficient binaries containing no superfluous information and which are not easy to reverse engineer. The SELF extension is designed with these objectives in mind. It concisely captures only the relevant information required to perform post-link-time transformations of binary code. This information is described in the form of a table of *memory block descriptors*. A memory block is a contiguous sequence of bytes within a program's memory. Each memory block descriptor has four fields, as shown in Figure 4. The fields are interpreted as follows:

1. *Memory Tag* - type of the block of memory. This includes various kinds of data and code and their pointers. Also includes a bit which indicated whether or not it is safe to relocate the block.

2. *Address* - the virtual address of a block of memory within the data or code of the shared object/executable. This field is meaningful only for executable or shared object files where locations of code and data of the program have been finalized.

3. *Alignment* - this indicates alignment constraints for certain type of data or instructions.

4. *Width* - size of data for the entries that correspond to a certain data related memory tag.

During code generation, the compiler adds entries to the table depending upon the *memory tag* of data or instructions. The memory tags fall into the following categories:

- *Data stored between instructions.* This memory tag corresponds to data used in the code, either in the form of jump tables or padding bytes which are used to enforce alignment restrictions. This helps to identify and disassemble all of the machine instructions in the program.

- *Code address.* Code addresses appear in the program mainly in the form of operands corresponding to targets of jump or call instructions. The addresses could be used in instructions either as relative displacements or as absolute values stored in register or memory. Also, there could be other types of instructions which use code addresses. Typical examples are (1) a `PUSH` instruction used to pass a constant function-address as a parameter to a callback function and (2) code addresses contained in jump tables. During transformation of binaries, the code at these addresses might be relocated. Therefore, such operands or locations must be changed to point to the new addresses.

- *Data address constant.* Static data in the program is referenced using data address constants in the instructions. Entries of these types are required if the data segment of the binary undergoes reorganization.

- *Offset from the GOT (global offset table).* This corresponds to the constant offsets which are used to access static data in position-independent code (PIC). Such offsets will be modified if the GOT or the data is relocated.

- *Offset used to obtain base address of the GOT.* This pertains mainly to x86-specific position-independent code generated for shared objects. For this purpose, the code is generated in such a way that the `%EBX` register is assigned the value of the GOT's base address. During the generation of this code, a constant value is used which corresponds to the relative offset of the program counter from the GOT. This constant requires modification if the GOT or the code containing the program counter undergoes relocation during binary transformation.

- *PLT (procedure linkage table) entry address.* In an executable or a shared library, a position-independent function call (e.g., a shared library function) is directed to a PLT entry. The PLT entry in turn redirects it to its absolute location, which is resolved by the dynamic linker at runtime.

Code addresses associated with these function calls need different memory tags as some binary transformation may require relocation of only the PLT.

- *Offset from base pointer*. This memory tag identifies the locations of constant offsets from the base pointer (`%EBP`) that are used by instructions which access stack-allocated objects. These constants have to be changed if there is a binary transformation that relocates stack-allocated objects.

- *Routine entry point*. This memory tag identifies the entry points of all the routines in the code segment.

- *Stack data*. Stack data is mainly associated with the local variables of functions in the program. Memory for such data is allocated on the stack dynamically during function invocations. Therefore, the virtual memory addresses of stack data can not be determined statically. However, each stack data is allocated on the stack at a fixed constant negative offset from the base pointer. The *address* field in an entry of this tag contains this offset instead of the virtual memory address.

- *Static data*. Static data corresponds to different storage units allocated in the code segment for global or static variables used by the program. This memory tag is used to identify the locations of each such storage unit in the code segment.

Apart from these, there are other memory locations which contain data required for dynamic linking. The entries of these types are retained in the binary file and hence we do not require to save them separately. The above types are all that is needed to effectively disassemble executables and thereby make program analysis and transformation of executables possible.

For static or stack-allocated data, additional information is available through the fields *alignment* and *width* of the entries. A compiler generates the memory layout of the program data depending on their types. Data could

either have scalar or aggregate types. A datum of a scalar type holds a single value, such as an integer, a character, a float value, etc. An aggregate type, such as an array, structure, or union, consists of one or more scalar data type objects. The ABI (application binary interface) of a processor architecture specifies different alignment constraints for different data types in order to access the data in an optimum way. Scalar types align according to their natural architectural alignment, e.g., in the Intel IA-32 architecture, the integer type requires word (4 byte) alignment. The alignment of an aggregate type depends on how much space it occupies and how efficiently the processor can access the individual scalar members objects within it. The data entries hold alignment and width of only the scalar and the aggregate objects and not for the members inside the aggregate objects. Thus, relocation can be performed only on the scalar or the aggregate objects as a whole.

4.3.5.3 SELF generation

Generating SELF from within a compiler is a straightforward process, as most of the information required can be gleaned directly from the compiler's internal symbol tables. Also required will be a `.rel.self` section, which will contain the relocation entries used by the linker to update the `.self` section when the program layout is finalized. A good implementation strategy for adding a SELF generation option to a typical compiler is to modify the code used to generate debugging information, since there is a lot of overlap between the debugging information and SELF. The `.self` section contents can be viewed as a copy of the debugging information with unneeded information removed, such as variable names and types, and extra information added, such as information about pointers embedded within machine instructions.

4.3.5.4 Application to address space randomization

Using the SELF extension, binary files can be analyzed completely, enabling all the randomizing transformations that we proposed. In particular, we can implement relative address randomizations on binary files, which was otherwise

impossible. For instance, we can permute the order of static data objects. For this we simply need to modify the instructions in the code segment that contain static data addresses. Similarly we can randomize the base addresses of the data and code segments, permute the order of routines, introduce gaps between objects, and so on.

4.4 Source Code Transformation Approach

In our binary transformation approach, we are able to achieve all the absolute address randomizations, but not all the relative address randomizations. On the other hand, working with source code gives us more flexibility in implementing all the absolute as well as the relative address randomizations.

The main component of this implementation approach is a source code transformer which uses CIL [32] as the front-end, and Objective Caml as the implementation language. CIL translates C code into a high-level intermediate form which can be transformed and then emitted as C source code, considerably facilitating the implementation of our transformation.

Our implementation also includes a small platform-specific component that supports transformations involving executable code and shared libraries.

The implementation of these components are described in greater details below. Although the source code transformation is fairly easy to port to different operating systems, the description below refers specifically to our implementation on an x86/Linux system.

4.4.1 Static Data Transformations

One possible approach to randomize the location of static data is to recompile the data into position-independent code (PIC). This is the approach taken in PaX ASLR [36], as well as in the binary transformations described in the previous section. A drawback of this approach is that it does not protect against relative address attacks, e.g., an attack that overflows past the end of a buffer to corrupt a security-critical data that is close to the buffer. Moreover,

an approach that relies only on changes to the base address is very vulnerable to information leakage attacks, where an attacker may mount a successful attack just by knowing the address of any static variable, or the base address of the static area. Finally, on operating systems such as Linux, the base address of different memory sections for any process is visible to any user with access to that system, and hence the approach does not offer much protection from this class of attacks.

For the reasons described above, our approach is based on permuting the order of static variables at the beginning of program execution. In particular, for each static variable `v`, an associated (static) pointer variable `v_ptr` is introduced in the transformed program. All accesses to the variable `v` are changed to reference `(*v_ptr)` in the transformed program. Thus, the only static variables in the transformed program are these `v_ptr` variables, and the program no longer makes any reference to the original variable names such as `v`.

At the beginning of program execution, control is transferred to an initialization function introduced into the transformed program. This function first allocates a new region of memory to store the original static variables. This memory is allocated dynamically so that its base address can be chosen randomly. Next, each static variable `v` in the original program is allocated storage within this region, and `v_ptr` is updated to point to the base of this storage.

To permute the order of variables, we proceed as follows. If there are n static variables, a random number generator is used to generate a number i between 1 and n. Now, the ith variable is allocated first in the newly allocated region. Now, there are $n-1$ variables left, and one can repeat the process by generating a random number between 1 and $n-1$ and so on.

Note that bounds-checking errors dominate among memory errors. Such errors occur either due to the use of an array subscript that is outside its bounds, or more generally, due to incorrect pointer arithmetic. For this reason, our transformation separates buffer-type variables, which can be sources of bounds-checking errors, from other types of variables. Buffer-type variables

include all arrays and structures/unions containing arrays. In addition, they include any variable whose address is taken, since it may be used in pointer arithmetic, which can in turn lead to out-of-bounds access.

All buffer-type variables are allocated separately from other variables. Inaccessible memory pages (neither readable nor writable) are introduced before and after the memory region containing buffer variables, so that any buffer overflows from these variables cannot corrupt non-buffer variables. The order of buffer-type variables is randomized as mentioned above. In addition, inaccessible pages are also introduced periodically within this region to limit the scope of buffer-to-buffer overflows.

Finally, all of the `v_ptr` variables are write-protected. Note that the locations of these variables are predictable, but this cannot be used as a basis for attacks due to write-protection.

We illustrate the exact implementation steps through an example code:

```
int a = 1;
char b[100];
extern int c;

void f() {
    while (a < 100) b[a] = a++;
}
```

We transform the above declarations, and also add an initialization function to allocate memory for the variables defined in the source file as shown below:

```
int *a_ptr;
char (*b_ptr) [100];
extern int *c_ptr;

void __attribute__ ((constructor)) data_init() {

    struct {
      void *ptr;
      unsigned int size;
```

```
      BOOL is_buffer;
    } alloc_info[2];

    alloc_info[0].ptr = (void *) &a_ptr;
    alloc_info[0].size = sizeof(int);
    alloc_info[0].is_buffer = FALSE;
    alloc_info[1].ptr = (void *) &b_ptr;
    alloc_info[1].size = sizeof(char [100]);
    alloc_info[1].is_buffer = TRUE;

    static_alloc(alloc_info, 2);

    (*a_ptr) = 1;
}

void f() {
      while ((*a_ptr) < 100)
        (*b_ptr)[(*a_ptr)] = (*a_ptr)++;
}
```

For the initialization function `data_init()`, we use `constructor` attribute so that it is invoked automatically before execution enters `main()`. Each element in the array `alloc_info` stores information about a single static variable, including the location of its pointer variable, its size, etc. Memory allocation is done by the function `static_alloc`, which works as follows. First, it allocates the required amount of memory by using a `mmap`. (Note that `mmap` allows its caller to specify the start address and length of a segment, and this capability is used to randomize the base address of static variables.) Second, it randomly permutes the order of static variables specified in `alloc_info`, and introduces gaps and protected memory sections in-between some variables. Finally, it zeroes out the memory allocated to static variables. After the call to `static_alloc`, code is added to initialize those static variables that are explicitly initialized.

Other than the initialization step, the rest of the transformation is very

simple: replace the occurrence of each static variable to use its associated pointer variable, i.e., replace occurrence of `v` by `(*v_ptr)`.

All of the `v_ptr` variables are write-protected by initialization code that is introduced into `main`. This code first figures out the boundaries of the data segment, and then uses the `mprotect` system call to apply the write protection.

Section `.got` contains the GOT, whose randomization will be discussed in the context of randomization of PLT in Section 4.4.6.

4.4.2 Code Transformations

As with static data, one way to randomize code location is to generate PIC code, and map this at a randomly chosen location at runtime. But this approach has several drawbacks as mentioned before, so our approach involves randomizing at a much finer granularity. Specifically, our randomization technique works at the granularity of functions. To achieve this, a function pointer `f_ptr` is associated with each function `f`. It is initialized with the value of `f`. All references to `f` are replaced by `(*f_ptr)`.

The above transformation avoids calls using absolute addresses, thereby laying the foundation for relocating function bodies in the binary. But this is not enough: there may still be jumps to absolute addresses in the code. With C-compilers, such absolute jumps are introduced while translating `switch` statements. In particular, there may be a jump to location `jumpTable[i]`, where `i` is the value of the `switch` expression, and `jumpTable` is a constant table constructed by the compiler. The `i`th element of this table contains the address of the corresponding case of the `switch` statement. To avoid absolute address dependency introduced in this translation, we transform a `switch` into a combination of `if-then-else` and `goto` statements. Efficient lookup of case values can be implemented using binary search, which will have $O(log\ N)$ time complexity. However, in our current implementation we use sequential search. In theory, this transformation can lead to decreased performance, but we have not seen any significant effect due to this change in most programs.

On a binary, the following actions are performed to do the actual randomization. The entire code from the executable is read. In addition, the location of functions referenced by each f_ptr variable is read from the executable. Next, these functions are reordered in a random manner, using a procedure similar to that used for randomizing the order of static variables. Random gaps and inaccessible pages are inserted periodically during this process in order to introduce further uncertainty in code locations, and to provide additional protection. The transformation ensures that these gaps do not increase the overall space usage for the executable by more than a specified parameter (which has the value of 100% in our current implementation). This limit can be exceeded if the original code size is smaller than a threshold ($32KB$).

After relocating functions, the initializations of f_ptr variables are changed so as to reflect the new location of each function. The transformed binary can then be written back to the disk. Alternatively, the transformation could be done at load-time, but we have not implemented this option so far.

It is well known that binary analysis and transformation are very hard problems [37]. To ease this problem, our transformation embeds "marker" elements, such as an array of integers with predefined values, to surround the function pointer table. These markers allow us to quickly identify the table and perform the above transformation, without having to rely on binary disassembly.

As a final step, the function pointer table needs to be write-protected. Actual transformation of source code is shown through following example:

```
char *f();
void g(int a) { ... }
void h() {
  char *str;
  char *(*fptr)();
  ...
  fptr = &f;
  str = (*fptr)();
  g(10);
```

```
}
```

The above code will be transformed as follows:

```
void *func_ptrs[] =
  {M1, M2, M3, M4, (void *)&f, (void *)&g,
   M5, M6, M7, M8};

char *f();
void g(int a) { ... }
void h() {
  char *str;
  char *(*fptr)();
  ...
  fptr = (char *(*)())func_ptrs[4];
  str = (*fptr)();
  (*((void (*)(int)) (func_ptrs[5])))(10);
}
```

The function pointer array in each source file contains locations of functions used in that file. The `func_ptrs` array is bounded on each end with a distinctive, 128-bit pattern that is recorded in the marker variables `M1` through `M8`. This pattern is assumed to be unique in the binary, and can be easily identified when scanning the binary. The original locations of functions can be identified from the contents of this array. By sorting the array elements, we can identify the beginning as well as the end of each function. (The end of a function is assumed to just precede the beginning of the next function in the sorted array.) Now, the binary transformation simply needs to randomly reorder function bodies, and change the content of the `func_ptr` array to point to these new locations. We adapted LEEL binary-editing tool [54] for performing this code transformation.

In our current implementation, we do not reorder functions at load time. Instead, the same effect is achieved by modifying the executable periodically.

4.4.3 Stack Transformations

To change the base address of the stack, our transformation adds initialization code that subtracts a large random number (of the order of 10^8) from the stack pointer. In addition, all of the environment variables and command line arguments are copied over, and the original contents erased to avoid leaving any data that may be useful to attackers (such as file names) at predictable locations. Finally, the contents of the stack above the current stack pointer value are write-protected. (An alternative to this approach is to directly modify the base address of the stack, but this would require changes to the OS kernel, which we want to avoid. For instance, on Linux, this requires changes to `execve` implementation.). In our current implementation, this initialization is done inside the loader code which is executed before the invocation of `main`.

The above transformation changes the absolute locations of stack-resident objects, but has no effect on relative distances between objects. One possible approach to randomize relative distances is to introduce an additional level of indirection, as was done for static variables. However, this approach will introduce high overheads for each function call. Therefore we apply this approach only for buffer-type local variables. Buffer-type variables include those whose address is explicitly or implicitly used in the program. For each buffer-type variable, we introduce a pointer variable to point to it, and then allocate the buffer itself on a second stack called the *buffer stack*. Consider a local variable declaration `char buf[100]` within a function, `func`. This variable can be replaced by a pointer with the following definition:

```
char (*buf_ptr)[100]
```

On entry of `func`, memory for `buf` is allocated using:

```
buf_ptr = bufferstack_alloc(sizeof(char [100]))
```

Allocations of multiple buffers are performed in a random order similar to static variables. Also, the allocator function allocates extra memory of a random size (currently limited to a maximum of 30%) between buffers, thereby creating

random gaps between adjacent buffers. Finally, all occurrences of `buf` in the body of `func` are replaced with `(*buf_ptr)`.

Our transformation does not change the way other types of local variables are allocated, so they get allocated in the same order. However, since the addresses of these variables never get taken, they cannot be involved in attacks that exploit knowledge of relative distances between variables. In particular, stack smashing attacks become impossible, as the return address is on the regular stack, whereas the buffer overflows can only corrupt the buffer stack. In addition, attacks using absolute addresses of stack variables do not work, as the absolute addresses are randomized by the (random) change to the base address of the stack.

Note that function parameters may be buffer-type variables. To eliminate the risk of overflowing them, we copy all buffer-type parameters into local variables, and use only the local variables from there on. Buffer-type parameters are never accessed in code, so there is no possibility of memory errors involving them. (An alternative to this approach is to ensure that no buffer-type variables are passed by value. But this requires the caller and callee code to be transformed simultaneously, thereby potentially breaking the separate file compilation approach.)

As a final form of stack randomization, we introduce random gaps between stack frames. This makes it difficult to correlate the locations of local variables across function invocations, thereby randomizing the effect of uninitialized pointer access and other temporal errors. Before each function call, code is added to decrement stack pointer by a small random value. After the function call, this padding is removed. The padding size is a random number generated at runtime, so it will vary for each function invocation.

Introduction of random-sized gaps between stack frames is performed using the `alloca` function, which is converted into inline assembly code by `gcc`. There are two choices on where this function is invoked: (a) immediately before calling a function, (b) immediately after calling a function, i.e., at the beginning of the called function. Note that option (b) is weaker than option (a) in a case where a function f is called repeatedly within a loop. With (a),

the beginning of the stack frame will differ for each call of f. With (b), all calls to f made within this loop will have the same base address. Nevertheless, our implementation uses option (b), as it works better with some of the compiler optimizations.

Handling setjmp/longjmp. The implementation of buffer stack needs to consider subroutines such as `setjmp()` and `longjmp()`. A call to `setjmp()` stores the program context which mainly includes the stack pointer, the base pointer and the program counter. A subsequent call to `longjmp()` restores the program context and the control is transferred to the location of the `setjmp()` call. To reflect the change in the program context, the buffer stack needs to be modified. Specifically, the top of buffer stack needs to be adjusted to reflect the `longjmp`. This is accomplished by storing the top of the buffer stack as a local variable in the main stack and restoring it at the point of function return. As a result, the top of buffer stack will be properly positioned before the first allocation following the `longjmp`. (Note that we do not need to change the implementation of `setjmp` or `longjmp`.)

4.4.4 Heap Transformations

Heap-related transformations may have to be implemented differently, depending on how the underlying heap is implemented. For instance, suppose that a heap implementation allocates as much as twice the requested memory size. In this case, randomly increasing a request by 30% will not have much effect on many memory allocation requests. Thus, some aspects of randomization have to be matched to the underlying heap implementation.

For randomizing the base of heap, we could make a dummy `malloc()` call at the beginning of program execution, requesting a big chunk of memory. However, this would not work for `malloc()` as implemented in GNU `libc`: for any chunk larger than 4 KB, GNU `malloc` returns a separate memory region created using the `mmap` system call, and hence this request doesn't have any impact on the locations returned by subsequent `malloc`'s.

We note that `malloc` uses the `brk` system call to allocate heap memory. This call simply changes the end of the data segment. Subsequent requests to `malloc` are allocated from the newly extended region of memory. In our implementation, a call to `brk` is made before any `malloc` request is processed. As a result, locations returned by subsequent `malloc` requests will be changed by the amount of memory requested by the previous `brk`. The length of the extension is a random number between 0 and 10^8. The extended memory is write-protected using the `mprotect` system call.

To randomize the relative distance between heap objects, calls to `malloc()` are intercepted by a wrapper function, and the size of the request increased by a random amount, currently between 0% and 30%.

Additional randomizations are possible as well. For instance, we can intercept calls to `free`, so that some of the freed memory is not passed on to `malloc`, but simply result in putting the the buffer in a temporary buffer. The implementation of the `malloc` wrapper can be modified to perform allocations from this buffer, instead of passing on the request to `malloc`. Since heap objects tend to exhibit a significant degree of randomness naturally, we have not experimented with this transformation.

4.4.5 Shared Library Transformations

Ideally, shared libraries should be handled in the same way as executable code: the order of functions should be randomized, and the order of static variables within the libraries should be randomized. However, shared libraries are shared across multiple programs. Randomization at the granularity of functions, if performed at load time on shared libraries, will create *copies* of these shared libraries, and thus rule out sharing. To enable sharing, randomization can be performed on the disk image of the library rather than at load time. Such randomization has to be performed periodically, e.g., at every restart of the system.

A second potential issue with shared libraries is that their source code may not be available. In this case, the base address of the shared library

can be randomized in a manner similar to [36] or the technique used in the previous section. However, this approach does not provide sufficient range of randomization on 32-bit architectures. In particular, with a page size of 4096 ($= 2^{12}$) bytes on Linux, uncertainty in the base address of a library cannot be much larger than 2^{16}, which makes them susceptible to brute-force attacks [45]. We address this problem by a link-time transformation to prepend each shared library with junk code of random size between 0 and page size. The size of this junk code must be a multiple of 4, so this approach increases the space of randomization to $2^{16} * 2^{12}/4 = 2^{26}$.

Load-time randomization has been implemented by modifying the dynamic linker `ld.so` so that it ignores the "preferred address" specified in a shared library, and maps it at a random location. Note that there is a bootstrapping problem with randomizing `ld.so` itself. To handle this problem, our implementation modifies the preferred location of `ld.so`, which is honored by the operating system loader. This approach negatively impacts the ability to share `ld.so` among executables, but this does not seem to pose a significant performance problem due to the relatively small size and infrequent use (except during process initialization) of this library.

4.4.6 Other Randomizations

Randomization of PLT and GOT. In a dynamically linked ELF executable, calls to shared library functions are resolved at runtime by the dynamic linker. The GOT (global offset table) and the PLT (procedure linkage table) play crucial roles in resolution of library functions. The GOT stores the addresses of external functions, and is a part of the data segment. The PLT, which is a part of the code segment, contains entries that call addresses stored in the GOT.

From the point of view of an attacker looking to access system functions such as `execve`, the PLT and the GOT provide "one-stop shopping," by conveniently collecting together the memory locations of all system functions in one place. For this reason, they have become a common target for attacks.

For instance,

- if an attacker knows the absolute location of the PLT, then she can determine the location within the PLT that corresponds to the external function `execve`, and use this address to overwrite a return address in a stack smashing attack. Note that this attack works even if the locations of all functions in the executable and libraries have been randomized
- if an attacker knows the absolute location of the GOT, she can calculate the location corresponding to a commonly used function such as the `read` system call, and overwrite it with a pointer to attack code injected by her. This would result in the execution of attack code when the program performs a `read`.

It is therefore necessary to randomize the locations of the PLT and the GOT, as well as the relative order of entries in these tables. However, since the GOT and the PLT are generated at link-time, we cannot control them using a source code transformation. One approach for protecting the GOT is to use the eager linking option, and then write-protect it at the beginning of the `main` program. An alternative approach that uses lazy linking (which is the default on Linux) is presented in [52].

The main complication in relocating the PLT is to ensure that any references in the program code to PLT entries be relocated. Normally, this can be very difficult, because there is no way to determine through a static analysis of a binary whether a constant value appearing in the code refers to a function, or is simply an integer constant. However, our transformation has already addressed this problem: every call to an entry `e` in the PLT will actually be made using a function pointer `e_ptr` in the transformed code. As a result, we treat each entry in the PLT as if it is a function, and relocate it freely, as long as the pointer `e_ptr` is correctly updated.

Randomization of read-only data. The read-only data section of a program's executable consists of constant variables and arrays whose contents are guaranteed not to change when the program is being run. Attacks which

corrupt data cannot harm read-only data. However, if their location is predictable, then they may be used in some attacks that need meaningful argument values, e.g., a typical return-to-libc attack will modify a return address on the stack to point to `execve`, and put pointer arguments to `execve` on the stack. For this attack to succeed, an attacker has to know the absolute location of a string constant such as `/bin/bash` which may exist in the read-only section.

Note that our approach already makes return-to-libc attacks very difficult. Nevertheless, it is possible to make it even more difficult by randomizing the location of potential arguments in such attacks. This can be done by introducing variables in the program to hold constant values, and then using the variables as arguments instead of the constants directly. When this is done, our approach will automatically relocate these constants.

4.4.7 Other Implementation Issues

Random number generation. Across all the transformations, code for generation of random numbers is required to randomize either the base addresses or the relative distances. For efficiency, we use pseudo-random numbers rather than cryptographically generated random numbers. The pseudo-random number generator is seeded with a real random number read from `/dev/urandom`.

Debugging support. Our transformation provides support for some of the most commonly used debugging features such as printing a stack trace. Note that no transformations are made to normal (i.e., non-buffer) stack variables. Symbol table information is appropriately updated after code rewriting transformations. Moreover, conventions regarding stack contents are preserved. These factors enable off-the-shelf debuggers to produce stack traces on transformed executables.

Unfortunately, it isn't easy to smoothly handle some aspects of transformation for debugging purposes. Specifically, note that accesses to global variables (and buffer-type local variables) are made using an additional level

Program	Combination (1) % Overhead	Combination (2) % Overhead
tar	-1	0
wu-ftpd	0	2
gv	0	2
bison	1	8
groff	-1	13
gzip	-1	14
gnuplot	0	21

Table 1: Runtime performance overhead introduced by binary-only ASR transformations.

of indirection in the transformed code. A person attempting to debug a transformed program needs to be aware of this. In particular, if a line in the source code accesses a variable `v`, he should know that he needs to examine `(*v_ptr)` to get the contents of `v` in the untransformed program. Although this may seem to be a burden, we point out that our randomizing transformation is meant to be used only in the final versions of code that are shipped, and not in debugging versions.

4.5 Performance Results

4.5.1 Binary-Only Transformations

We have collected performance data on the implementation of randomization using binary-only transformations. The following randomizations were implemented:

- relocating the base of the stack, heap, and code regions
- introduction of random gaps within stack frames, and at the end of memory blocks requested by `malloc`. The stack frame gaps were determined

statically for each routine, while the `malloc` gaps can change with each `malloc` request.

We studied two different approaches for randomizing the start address of the executable:

- *Combination 1:* static relocation performed at link-time.
- *Combination 2:* dynamic relocation performed at load-time.

Both approaches incorporate all of the transformations mentioned above. Note that dynamic relocation requires the executable be compiled into position-independent code, which introduces additional runtime overheads.

Table 1 shows the runtime performance overheads due to the two combinations of transformations. All measurements were taken on an 800 MHz, Pentium III, 384 MB RAM machine with RedHat 7.3 Linux operating system. Average execution (system + user) time was computed over 10 runs. The overheads measured were rounded off to the nearest integral percentage.

From the table, we see that combination (1) incurs essentially no runtime overhead.

Combination (2) has noticeable runtime overhead. This is because it requires position-independent code, which is less efficient, since it performs extra operations before every procedure call, and every access to static data. On the other hand, when code is already being distributed in a shared library form, combination (2) provides broad protection against memory error exploits without any additional overhead.

4.5.2 Source Code Transformations

We have collected data on the performance impact of the source code-based randomizing transformations. The transformations were divided into the following categories, and their impact was studied separately.

- **Stack**: transformations which randomize the stack base, move buffer-type variables into the buffer stack, and introduce gaps between stack frames.

Program	Workload
`Apache-1.3.33`	WebStone 2.5, client connected over 100 Mbps network
`sshd-OpenSSH_3.5p1`	Run a set of commands from ssh client
`wu-ftpd-2.8.0`	Run a set of different ftp commands
`bison-1.35`	Parse C++ grammar file
`grep-2.0`	Search a pattern in files of combined size 108 MB
`bc-1.06`	Find factorial of 600
`tar-1.12`	Create a tar file of a directory of size 141 MB
`patch-2.5.4`	Apply a 2 MB patch-file on a 9 MB file
`enscript-1.6.4`	Convert a 5.5 MB text file into a postscript file
`ctags-5.4`	Generate tag file of 6280 C source code files with total 17511 lines
`gzip-1.2.4`	Compress a 12 MB file

Table 2: Test programs and workloads for performance evaluation of source code-based ASR transformations.

- **Static data**: transformations which randomize locations of static data.
- **Code**: transformations which reorder functions.
- **All**: all of the above, plus randomizing transformations on heap and shared libraries.

Table 2 shows the test programs and their workloads. Table 4 shows runtime performance overheads due to each of the above categories of transformations. The original and the transformed programs were compiled using `gcc-3.2.2` with `-O2` optimization, and executed on a desktop running RedHat Linux 9.0 with 1.7 GHz Pentium IV processor, and 512 MB RAM. Execution times were averaged over 10 runs.

For Apache server, we studied its performance separately after applying all the transformations. To measure performance of the Apache server accurately, heavy traffic from clients is required. We generated this using WebStone [50], a standard web server benchmark. We used version 2.5 of this benchmark, and ran it on a separate computer that was connected to the server through a 100 Mbps network. We ran the benchmark with two, sixteen

#clients	Degradation (%)	
	Connection Rate	Response Time
2-clients	1	0
16-clients	0	0
30-clients	0	1

Table 3: Performance overhead introduced by the source code-based ASR transformations on Apache.

and thirty clients. In the experiments, the clients were simulated to access the web server concurrently, randomly fetching html files of size varying from 500 bytes to 5 MB. The benchmark was run for a duration of 30 minutes, and the results were averaged across ten such runs. Results were finally rounded off to the nearest integral values.

We analyzed the performance impact further by studying the execution profile of the programs. For this, we instrumented programs to collect additional statistics on memory accesses made by the transformed program. Specifically, the instrumentation counts the total number of accesses made to local variables, variables on buffer stack, global variables and so on.

Table 5 shows the dynamic profile information. (We did not consider servers in this analysis due to the difficulties involved in accurately measuring their runtimes.) From this result, we see that for most programs, the vast majority of memory accesses are to local variables. Our transformation doesn't introduce any overheads for local variables, which explains the low overheads for most programs in Table 4. Higher overheads are reported for programs that perform a significant number of global variable accesses, where an additional memory access is necessitated by our transformation.

A second source of overhead is determined by the number of function calls made by a program. This includes the overhead due to the additional level of indirection for making function calls, the number of allocations made on buffer stack, and the introduction of inter-stack-frame gap. To analyze

Program	Orig. CPU time	% Overheads			
		Stack	Static	Code	All
grep	0.33	0	0	0	**2**
tar	1.06	2	2	1	**4**
patch	0.39	2	0	0	**4**
wu-ftpd	0.98	2	0	6	**9**
bc	5.33	7	1	2	**9**
enscript	1.44	8	3	0	**10**
bison	0.65	4	0	7	**12**
gzip	2.32	6	9	4	**17**
sshd	3.77	6	10	2	**19**
ctags	9.46	10	3	8	**23**
Avg. Overhead		**5**	**3**	**3**	**11**

Table 4: Runtime performance overheads introduced by the source code-based ASR transformations on benchmark programs.

this overhead, we instrumented the transformed programs to collect number of function calls and number of buffer stack allocations. The results, shown in Table 6, illustrate that programs that make a large number of function calls per second, e.g., `ctags` and `gzip` incur higher overheads. Surprisingly, `bison` also incurs high overheads despite making small number of function calls per second. So we analyzed `bison`'s code, and found that it contains several big switch statements. This could be the main reason behind the high overheads, because our current implementation performs sequential lookup for the case values. However, with binary search-based implementation, we should be able to get better performance.

We point out that the profile information cannot fully explain all of the variations in overheads, since it does not take into account some of the factors involved, such as register usage, compiler optimizations, and the effect of cache hits (and misses) on the additional pointer dereferences introduced in the transformed program. Nevertheless, the profile information provides a broad indication of the likely performance overheads due to each program.

Program	%age of variable accesses		
	Local		Global
	(non-buffer)	(buffer)	(static)
grep	99.9	0.004	0.1
bc	99.3	0.047	0.6
tar	96.5	0.247	3.2
patch	91.8	1.958	6.2
enscript	90.5	0.954	8.5
bison	88.2	0.400	10.9
ctags	72.9	0.186	26.9
gzip	59.2	0.018	40.7

Table 5: Distribution of variable accesses in the source code-based ASR transformations.

4.6 Effectiveness

Effectiveness can be evaluated experimentally or analytically. Experimental evaluation involves running a set of well-known exploits (such as those reported on `www.securityfocus.com`) against vulnerable programs, and showing that our transformation stops these exploits. We have not carried out a detailed experimental evaluation of effectiveness because today's attacks are quite limited, and do not exercise our transformation at all. In particular, they are all based on a detailed knowledge of program memory layout. We have manually verified that our transformation changes the memory locations of global variables, local variables, heap-allocated data and functions for each of the programs discussed in the previous section. It follows from this that none of the existing buffer overflow attacks will work on the transformed programs.

In contrast with the limitations of an experimental approach, an analytical approach can be based on novel attack strategies that haven't been seen before. Moreover, it can provide a measure of protection (in terms of the probability of a successful attack), rather than simply providing an "yes" or "no" answer. For this reason, we rely primarily on an analytical approach in this section. We first analyze memory error exploits in general, and then discuss

Program	**# calls**	**calls/ sec.**	**Buffer stack allocations**	
	$\times 10^6$	$\times 10^6$	per sec.	per call
`grep`	0.02	0.06	24K	0.412
`tar`	0.43	0.41	57K	0.140
`bison`	2.69	4.11	423K	0.103
`bc`	22.56	4.24	339K	0.080
`enscript`	9.62	6.68	468K	0.070
`patch`	3.79	9.75	166K	0.017
`gzip`	26.72	11.52	0K	0.000
`ctags`	251.63	26.60	160K	0.006

Table 6: Calls and buffer stack allocations in the source code-based ASR transformations.

attacks that are specifically targeted at randomization.

4.6.1 Memory Error Exploits

Recall from Chapter 2 that memory error exploits are based on corrupting some data in the writable memory of a process. For the purpose of overwriting, the exploits use different attack mechanisms such as buffer overflows and format string vulnerabilities. The targeted data for overwriting is either a pointer or non-pointer data. The randomizations make the attacker's job difficult in two ways. First, the overwrite step becomes difficult. Second, in the attack involving overwrite of a pointer data, the attacker is forced to guess the correct value of the pointer. Thus, for a specific vulnerability V, the probability of a successful attack A that exploits it is given by:

$$P(A) = P(Owr) * (P(Eff_1) * P(Eff_2) * ... * P(Eff_N)), \text{ where } N > 0, \text{ and}$$

$P(Owr)$: probability that V can be used to overwrite a specific data item of interest to the attacker,

$P(Eff_i)$: probability of correctly guessing the value to be used for overwriting the ith pointer.

Note that some attacks (e.g., heap overflows) involve corrupting multiple pointers. In absolute address randomization, once one address is guessed, other addresses can be derived as relative distances are fixed. In such a case, typically $P(Owr) = P(Eff_2) * P(Eff_3) * ... * P(Eff_N) = 1$. On the other hand, relative address randomization produces multiplicative effect, further reducing the probability of a successful attack.

In arriving at the above formula, we make either of the following assumptions:

- (a) the program is re-randomized after each failed attack. This happens if the failure of the effect causes the victim process to crash, (say, due to a memory protection fault), and it has to be explicitly restarted.

- (b) the attacker cannot distinguish between the failure of the overwrite step from the failure of the effect. This can happen if (1) the overwrite step corrupts critical data that causes an immediate crash, making it indistinguishable from a case where target data is successfully overwritten, but has an incorrect value that causes the program to crash, or (2) the program incorporates error-handling or defense mechanisms that explicitly masks the difference between the two steps.

Note that (a) does not hold for typical server programs that spawn children to handle requests, but (b) may hold. If neither of them hold, then the probability of a successful attack is given by $min(P(Owr), P(Eff_1) * P(Eff_2) * ... * P(Eff_N))$.

We estimate the probability factors for various types of attacks, separately for the two implementation approaches.

4.6.1.1 Estimating $P(Owr)$

We estimate $P(Owr)$ separately for each attack type.

Buffer overflows

- *Stack buffer overflows.* These overflows typically target the return address, saved base pointer or other pointer-type local variables.

  In the binary transformation approach, the stack padding creates an uncertainty of the order of 128 bytes in the distance between the vulnerable buffer and the return address or the base pointer. However, attacker can still successfully corrupt the return address (or the base pointer) by writing to the stack buffer a block containing copies of guessed address G (enough copies to be relatively sure that the return address is overwritten). Another disadvantage is that the other local variables in the same stack frame are not protected from corruption.

  In the source code transformation approach, the buffer stack transformation makes these attacks impossible, since all buffer-type variables are on the buffer stack, while the target data is on the main stack. Attacks that corrupt one buffer-type variable by overflowing the previous one are possible, but unlikely. As shown by our implementation results, very few buffer-type variables are allocated on the stack. Moreover, it is unusual for these buffers to contain pointers (or other security-critical data) targeted by an attacker.

- *Static buffer overflows.* The binary transformation approach provides no protection from these overflows.

  In contrast, the source code transformation approach is very effective against this type of attacks. As in the case of stack overflows, the likely targets are simple pointer-type variables. However, such variables have been separated by our transformation from buffer-type variables, and hence they cannot be attacked.

  For attacks that use an overflow from one buffer to the next, the randomization introduced by our transformation makes it difficult to predict the target that will be corrupted by the attack. Moreover, unwritable pages have been introduced periodically in-between buffer-type static variables, and these will completely rule out some overflows. To estimate the probability of successful attacks, let M denote the maximum size of a buffer

overflow, and S denote the granularity at which inaccessible pages are introduced between buffer variables. Then the maximum size of a useful attack is $min(M, S)$. Let N denote the total size of memory allocated for static variables. The probability that the attack successfully overwrites a data item intended by the attacker is given by $min(M, S)/N$. With nominal values of $4KB$ for the numerator and $1MB$ for the denominator, the likelihood of success is about 0.004.

- *Heap overflows.* Heap transformations are similar in the binary and the source code-based approaches. Therefore, the probability of the overwrite step is same for both the approaches.

  In general, heap-allocations are non-deterministic, so it is hard to predict the effect of overflows from one heap block to the next. This unpredictability is further increased by our transformation to randomly increase the size of heap-allocation requests. However, control data exist in heap blocks, and these can be more easily and reliably targeted. For instance, heap overflow attacks generally target two pointer-valued variables that are used to chain free blocks together, and appear at their beginning.

  The transformation to randomly increase `malloc` requests makes it harder to predict the start address of the next heap block, or its allocation state. However, the first difficulty can be easily overcome by writing alternating copies of the target address and value many times, which ensures that the control data will be overwritten with 50% probability. We believe that the uncertainty on allocation state doesn't significantly decrease the probability of a successful attack, and hence we conclude that our randomizations do not significantly decrease $P(Owr)$. However, as discussed below, $P(Eff)$ is very low for such attacks.

Format string attacks. These attacks exploit the (obscure) `"%n"` format specifier. The specifier needs an argument that indicates the address into which the `printf`-family of functions will store the number of characters that

have been printed. This address is specified by the attacker as a part of the attack string itself. The inter frame padding implemented in the binary transformation does not help in this case. On the other hand, in the source code-based implementation, the argument corresponding to the `"%n"` format specifier will be taken from the main stack, whereas the attack string will correspond to a buffer-type variable, and be held on the buffer stack (or the heap or in a global variable). As a result, there is no way for the attacker to directly control the address into which `printf`-family of functions will write, and hence the usual form of format-string attack will fail.

It is possible, however, that some useful data pointers may be on the stack, and they could be used as the target of writes. The likelihood of finding such data pointers on the stack is relatively low, but even when they do exist, the inter-stack frame gaps of the order of 2^8 bytes implemented in both of our approaches reduces the likelihood of successful attacks to $4/2^8 = 0.016$. This factor can be further decreased by increasing the size of inter-frame gaps in functions that call `printf`-family of functions.

In summary, the source code transformation approach significantly reduces the success probability of most likely attack mechanisms, which include (a) overflows from stack-allocated buffers to corrupt return address or other pointer-type data on the stack, (b) overflows from a static variable to another, and (c) format-string attacks. This is a far better improvement over binary transformation approach which has a very little effect on $P(Owr)$, or other address space randomization-based techniques [36, 52, 21] which have *no effect at all* on $P(Owr)$. Their preventive ability is based entirely on reducing $P(Eff)$ discussed in the next section.

4.6.1.2 Estimating $P(Eff)$

Corruption of non-pointer data. This class of attacks target security-critical data such as user-ids and file names used by an application. With our technique, as well as with the other address space randomization techniques, it can be seen that $P(Eff) = 1$, as they have no bearing on the interpretation

of non-pointer data. The most likely location of such security-critical data is the static area, where our approach provides protection in the form of a small $P(Owr)$. This contrasts with the other approaches that provide no protection from this class of attacks.

Pointer corruption attacks.

- *Corruption with pointer to existing data.* The probability of correctly guessing the absolute address of any data object is determined primarily by the amount of randomization in the base addresses of different data areas.

  In both of our implementation approaches, this quantity can be in the range of 2^{27}, but since the objects will likely be aligned on a 4-byte boundary, the probability of successfully guessing the address of a data object is of the order of 2^{-25}.

- *Corruption with pointer to injected data.* Guessing the address of some buffer that holds attacker-provided data is no easier than guessing the address of existing data objects. However, the odds of success can be improved by repeating the attack data many times over. If it is repeated k times, then the odds of success is given by $k \times 2^{-25}$. If we assume that the attack data is 16 bytes and the size of the overflow is limited to $4KB$, then k has the value of 2^{8}, and $P(Eff)$ is 2^{-17}.

- *Corruption with pointer to existing code.* The probability of correctly guessing the absolute address of any code object is determined primarily by the amount of randomization in the base addresses of different code areas.

  In the binary transformation-based approach, the uncertainty in the locations of functions within the executable is $2^{27}/4 = 2^{25}$, so $P(Eff)$ is bounded by 2^{-25}.

  In the source code-based implementation, the uncertainty in the locations of functions within the executable is $2^{16}/4 = 2^{14}$. We have already

argued that the randomization in the base address of shared libraries can be as high as 2^{-26}, so $P(\mathit{Eff})$ is bounded by 2^{-14}. This probability can be decreased by performing code randomizations at load-time. When code randomizations are performed on disk images, the amount of "gaps" introduced between functions is kept low (of the order of $64KB$ in the above calculation), so as to avoid large increase in the size of files. When the randomization is performed in main memory, the address space of randomization can be much larger, say, $128MB$, thereby reducing the probability of successful attacks to 2^{-25}.

- *Corruption with pointer to injected code.* Code can be injected only in data areas, and it does not have any alignment requirements (on x86 architectures). Therefore, the probability of guessing the address of the injected code is 2^{-27}. The attacker can increase the success probability by using a large NOP-padding before the attack code. If a padding of the order of $4KB$ is used, then $P(\mathit{Eff})$ becomes $4K \times 2^{-27} = 2^{-15}$.

4.6.2 Attacks Targeting ASR

The other address space randomization approaches and also our binary transformation-based approach are vulnerable to the classes of attacks described below. We describe how our source code transformation-based approach defends against them.

- **Information leakage attacks.**
  Programs may contain vulnerabilities that allow an attacker to "read" the memory of a victim process. For instance, the program may have a format string vulnerability such that the vulnerable code prints into a buffer that is sent back to the attacker. (Such vulnerabilities are rare, as pointed out in [45].) Armed with this vulnerability, the attacker can send a format string such as `"%x %x %x %x"`, which will print the values of 4 words near the top of the stack at the point of the vulnerability. If some of these words are known to point to specific program objects,

e.g., a function in the executable, then the attacker knows the locations of these objects.

We distinguish between two kinds of information leakage vulnerabilities: *chosen pointer leakage* and *random pointer leakage*. In the former case, the attacker is able to select the object whose address is leaked. In this case, the attacker can use this address to overwrite a vulnerable pointer, thereby increasing $P(Eff)$ to 1. With random pointer leakage, the attacker knows the location of some object in memory, but not the one of interest to him. Since relative address randomization makes it impossible in general to guess the location of one memory object from the location of another memory object, random pointer leakages don't have the effect of increasing $P(Eff)$ significantly.

For both types of leakages, note that the attacker still has to successfully exploit an overflow vulnerability. The probability of success $P(Owr)$ for this stage was previously discussed.

The specific case of format-string information leakage vulnerability lies somewhere between random pointer leakage and chosen pointer leakage. Thus, the probability of mounting a successful attack based on this vulnerability is bounded by $P(Owr)$.

- **Brute force and guessing attacks.**

  Apache and similar server programs pose a challenge for address space randomization techniques, as they present an attacker with many simultaneous child processes to attack, and rapidly re-spawn processes which crash due to bad guesses by the attacker. This renders them vulnerable to attacks in which many guesses are attempted in a short period of time. In [45], these properties were exploited to successfully attack a typical Apache configuration within a few minutes. This attack doesn't work with our source code transformation-based approach, as it relies on stack smashing. A somewhat similar attack could be mounted by exploiting some other vulnerability (e.g., heap overflow) and making repeated attempts to guess the address of some existing code. As discussed earlier,

this can be done with a probability between 2^{-14} to 2^{-26}. However, the technique used in [45] for passing arguments to this code won't work with heap overflows.

- **Partial pointer overwrites.**

  Partial pointer overwrites replace only the lower byte(s) of a pointer, effectively adding a delta to the original pointer value. These are made possible by off-by-one vulnerabilities, where the vulnerable code checks the length of the buffer, but contains an error that underestimates the size of buffer needed by 1.

  These attacks are particularly effective against randomization schemes which only randomize the base address of each program segment and preserve the memory layout. By scrambling the program layout, our approach negates any advantage of a partial overwrite over a full overwrite.

CHAPTER 5

Data Space Randomization

The basic idea of this approach is to *randomize representation of different data objects*, so that even if an attacker can predictably corrupt a data object, the effect of this corruption cannot be predicted. One way to modify data representation is to xor each data object in the memory with a different random mask (encryption), and to unmask it (decryption) before its use. Now, even if an attacker uses a buffer overflow to overwrite a variable x with a value v, it will be interpreted as $v \oplus m_x$ by the program, where m_x is the random mask (unknown to attacker) associated with x. Thus, the use of the random and wrong value $v \oplus m_x$ may cause an application to crash, but is very unlikely to compromise it.

As compared to ASR, DSR provides a much larger range of randomization. For instance, on 32-bit architectures, we can randomize integers and pointers over a range of 2^{32} values, which is much larger than the range possible with ASR. Moreover, DSR can address the weakness of ASR concerning its inability to randomize relative distance between two data objects because of certain languages semantics. For instance, ASR cannot randomize relative distances between the fields of the same structure. An interesting point to be noted here is that even the complete memory error detections techniques do not provide protection from overflows within the fields of a structure because according to C language specifications, an overflow within a structure is not considered as a memory error.

Previous approaches have used data space randomization technique in restricted forms. Instruction set randomization approaches [6, 28] use randomized representation of only the code objects, thus providing protection against only the injected code attacks. The PointGuard [16] approach randomizes only the pointer representation: a single random mask is used for xor'ing all the pointers. Their transformation approach is dependent on the accuracy of type information, which may not be always available. In particular, their transformations are unsound if a variable is an alias associated with the data of pointer as well as non-pointer types, but this fact cannot be inferred from just the type information. As a concrete example, consider the standard C library function `bzero` which takes an untyped parameter. This function may be used to zero out different structures containing combinations of pointer and non-pointer data. However, the type information of a structure that is passed as a parameter is not available in the body of `bzero`, because of which the transformation will end up assigning null value to a pointer present in the structure. Such a pointer is expected to have a value equal to the random mask, but its null value will cause the program to malfunction. Similar problem is faced with unions that store pointer and non-pointer values in the same memory location.

In contrast to the previous approaches, we undertake a more general interpretation of data space randomization, wherein all types of data can be randomized. This allows us to protect not only pointer type data, but also non-pointer security-critical data such as file names, command names, user ids, and so on. Furthermore, our approach uses different random masks for different data. This is particularly useful in defeating data corruption attacks that involve overflows into adjacent data. In addition, it also provides protection from exploits of a vulnerability that reveals the value of a masked variable, making it possible for attackers to retrieve masks of all other variables.

5.1 Transformation Approach

Our transformation approach for DSR is based on a source-to-source transformation of C programs. The basic transformations are very simple. For each data variable `v`, we introduce another variable `m_v` which stores the mask value to be used for randomizing the data stored in `v` using a xor operation. The mask is a random number that can be generated at the beginning of program execution or during runtime. The size of `m_v` depends on the size of the data stored in `v`. Ideally, we can store a fixed size (say, word length) random number in the mask variable, and depending on the size of the associated variable, we can generate bigger or smaller masks from the random number. However, for simplicity of notation, we will use mask variables having the same size as that of the variables being masked.

The variables appearing in expressions and statements are transformed as follows. Values assigned to variables are randomized. Thus, after every statement that assigns a value to a variable `v`, we add the statement `v = v ^ m_v` to randomize the value of the variable in the memory. Also, wherever a variable is used, its value is first derandomized. This is done by replacing `v` with `v ^ m_v`.

So far the transformations seem straightforward, but we have not yet considered a case in which variable data is accessed indirectly by dereferencing pointers. This concerns aliasing in C language. As an example, consider following code snippet:

```
    int x, y, z, *ptr;
    ...
    ptr = &x;
    ...
    ptr = &y;
    ...
L1: z = *ptr;
```

In the above code, the expression `*ptr` is an alias for either `x` or `y`. Since `*ptr` is used in the assignment statement at `L1`, we need to unmask it before

using its value in the assignment. Therefore, the line should be transformed as:

```
z = m_z ^ (m_starptr ^ *ptr),
```

where `m_z` and `m_starptr` are respectively masks of `z` and `*ptr`. Unfortunately, statically we cannot determine the mask `m_starptr` to be used for unmasking; it can be the mask of either variable `x` or `y`.

One way to address this problem is to dynamically track the masks to be used for *referents*[1] of all the pointers. This requires storing additional information (henceforth called metadata) about pointers. Similar information is maintained in some of the previous techniques that detect memory errors. In particular, they store metadata using different data structures such as *splay tree* [25] and *fat pointers* [3, 34]. These metadata storing techniques lead to either high performance overheads or code compatibility problems. For this reason, we chose to avoid dynamic tracking of masks.

Our solution to the above problem is based on using static analysis. More specifically, we use the same mask for variables that can be pointed by a common pointer. Thus, when the pointer is dereferenced, we know the mask to be used for its referents statically. This scheme requires "points-to" information which can be obtained by using *pointer analysis*. In the above example, from the results of any conservative pointer analysis technique, we can conclude that both variables `x` and `y` can be pointed by the pointer variable `ptr`. Hence we can use the same mask for both `x` and `y`, and this mask can be then used for unmasking `*ptr`, i.e., `m_x = m_y = m_starptr`.

Since our method statically determines the masks, it is likely to yield better performance than a dynamic tracking-based method. Moreover, our method does not create any code compatibility problems. However, there is one potential problem with this solution: if two adjacent objects are assigned the same mask, and there is a vulnerability that allows corruption of one object using an overflow from the other object, then an attacker can exploit the vulnerability to corrupt the overwritten object with a correct value. Our solution addresses this problem. The idea is something similar to what we

[1] A *referent* of a pointer is an object that the pointer points to.

did in ASR. The objects which are responsible for overflows correspond to buffer-type data objects. We access buffer-type data using an extra level of indirection (refer to Section 4.4.1) using pointers. The memory for buffer-type data is allocated either at the beginning of program execution or during runtime. If two buffers have the same mask, we ensure that the memory for these buffers is allocated in different memory regions, so that an overflow from one buffer into the other becomes impossible.

For the static data buffers, memory regions are created at the beginning of program execution. Whereas for stack data buffers with the same mask, we maintain multiple buffer stacks in different memory regions, and we allocate memory for the buffers in different stacks. Maintaining multiple buffer stacks could be an expensive operation. However, typically programs contain very few buffer-type stack data objects as compared to other data objects. Thus, in practice, we need only a small number of buffer stacks, and they can be efficiently maintained. For the heap objects, memory regions are created at runtime.

5.1.1 Pointer Analysis

Pointer analysis is a compile-time analysis that attempts to determine what storage locations a pointer can point to.

As mentioned before, we use pointer analysis to determine the masks of all the variables. In this section, we first identify the precision requirements for our pointer analysis, and then provide details on the kind of pointer analysis that we use.

Ideally, we need a distinct mask for each variable. Unfortunately, presence of pointers in C language potentially forces assignment of the same masks for different variables. As a result, variables are divided into different equivalence classes. All the variables in a class are assigned the same mask, and those belonging to different classes are assigned different masks. The number of the equivalence classes depend on the precision of pointer analysis. Intuitively, greater the precision, there will be more number of the equivalence classes. To

illustrate this point, consider the program in Figure 5(a), and points-to graph computed for this program using flow-insensitive and flow-sensitive pointer analyses in Figure 5(b) and Figure 5(c) respectively. Points-to graph captures points-to information in the form of a directed graph, wherein a node represents an equivalence class of symbols and edges represent pointer relationships. The flow-insensitive analysis computes the points-to information for the overall program. According to this information, the same mask is required for variables `x`, `y` and `z`. On the other hand, the flow-sensitive analysis computes the points-to information for each program point, and hence it is more precise. Using this analysis, variables `x` and `y` need the same mask, however, the extra precision permits a different mask for the variable `z`.

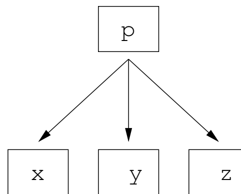
A pointer analysis is, in general, computationally undecidable [39]. As a result, existing pointer analysis algorithms use approximations that provide varying degree of precision and efficiency. The worst-case time complexities of these algorithms range from linear to exponential. We need to consider the time complexity, for the analysis to be efficient and scalable. There are several factors that affect precision and efficiency of analysis. Here are some of the important factors:

- **Flow-sensitivity:** whether the solution is computed for the whole program or for each program point
- **Context-sensitivity:** whether calling context of a function is considered while analyzing the function
- **Modeling of heap objects:** whether heap objects are named by allocation site, or more sophisticated shape analysis is used
- **Modeling of aggregate objects:** whether elements of aggregate objects such as structures, unions, and arrays are distinguished or collapsed into one object
- **Representation of alias information:** whether explicit alias information is used, or a more compact points-to graph representation is used

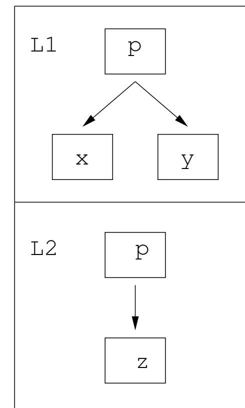
```
void main() {
     int *p, x, y, z, w;
     ...
     if (...) {
       p = &x;
     else {
       p = &y;
     }

L1:  w = *p;
     p = &z;
     ...
L2:  w = *p;
     ...
}
```

(a)



(b)



(c)

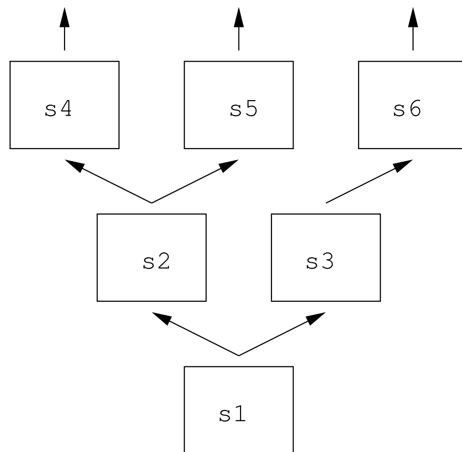
Figure 5: (a) Example program, (b) Points-to graph computed by a flow-insensitive analysis, (c) Points-to graphs computed at different program points using flow-sensitive analysis.

We need to consider these factors while choosing the analysis. Algorithms involved in existing flow-sensitive analyses [11, 19, 51, 40] are very expensive in terms of time complexity (high order polynomials). All the context-sensitive approaches have exponential time complexity. We avoid these two types of analyses as they do not scale to large programs. Among the flow-insensitive and context-insensitive algorithms, Andersen's algorithm [1] is considered to be the most precise algorithm. This algorithm has the worst case cubic time complexity, which is still high for it to be used on large programs. On the other hand, Steensgaard's algorithm [47] has linear time complexity, but it gives less precise results. Interestingly, as we shall show in the next section, it turns out that the results of Andersen's and Steensgaard's analyses give us the same equivalence classes of variable masks. Therefore, we implemented Steensgaard's algorithm for our purpose.

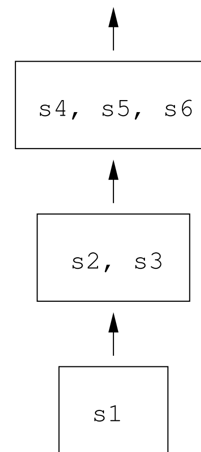
```
s2 = &s4;
s2 = &s5;
s3 = &s6;
foo(&s2);
foo(&s3);

void foo(int **s1) { ...}
```

(a)



(b)



(c)

Figure 6: Figure (a) above shows a sample C program for which points-to graph is computed. Figures (b) and (c) show the points-to graphs computed by Andersen's algorithm and Steensgaard's algorithm respectively.

5.1.2 Determination of Masks Using Points-to Graph

Consider points-to graphs computed by Steensgaard's and Andersen's algorithms as shown in Figure 6. Points-to information computed by Andersen's algorithm is more precise than that computed by Steensgaard's algorithm. For instance, according to Steensgaard's graph, `s2` may point to `s6`. However, this relationship appears unlikely if we look at the program. Andersen's graph does not capture this relationship, hence it is more precise. In Steensgaard's analysis, two objects that are pointed by the same pointer are unioned into one node. This may lead to unioning of the points-to sets of formerly distinct objects. This kind of unioning makes the algorithm faster, but results in less precise output as shown in the above example.

Now let us see how we can use the points-to information to determine the equivalence classes of masks for the above example; we do this for Andersen's graph. Objects `s2` and `s3` can be accessed using the pointer dereference `*s1`. This suggests that `s2` and `s3` should have the same mask, and therefore they belong to the same equivalence class. Similarly, pointer dereference `**s1` can be used to access any of the objects pointed by `s2` or `s3`. This implies that the objects pointed by `s2` and `s3` should have the same mask, and hence objects `s4`, `s5` and `s6` should be merged into the same equivalence class. This merging is similar to the unioning operation in Steensgaard's algorithm. Therefore, the equivalence classes of masks will be the same even in the case of Steensgaard's graph. For the above example, the complete set of equivalence classes of masks is {{`s1`}, {`*s1`, `s2`, `s3`}, {`**s1`, `*s2`, `*s3`, `s4`, `s5`, `s6`}}.

As Steensgaard's and Andersen's graphs are equivalent from the point of view of determining masks, it makes sense to use Steensgaard's algorithm for our purpose as it is more efficient than Andersen's algorithm. So we formally define the procedure for determining masks using Steensgaard's points-to graphs.

In general, a points-to graph of a program consists of disconnected components. Hence we consider the procedure only for one component which can be similarly applied to all the graph components. For this, let us first look at the properties of a Steensgaard's points-to graph. The unioning operation in

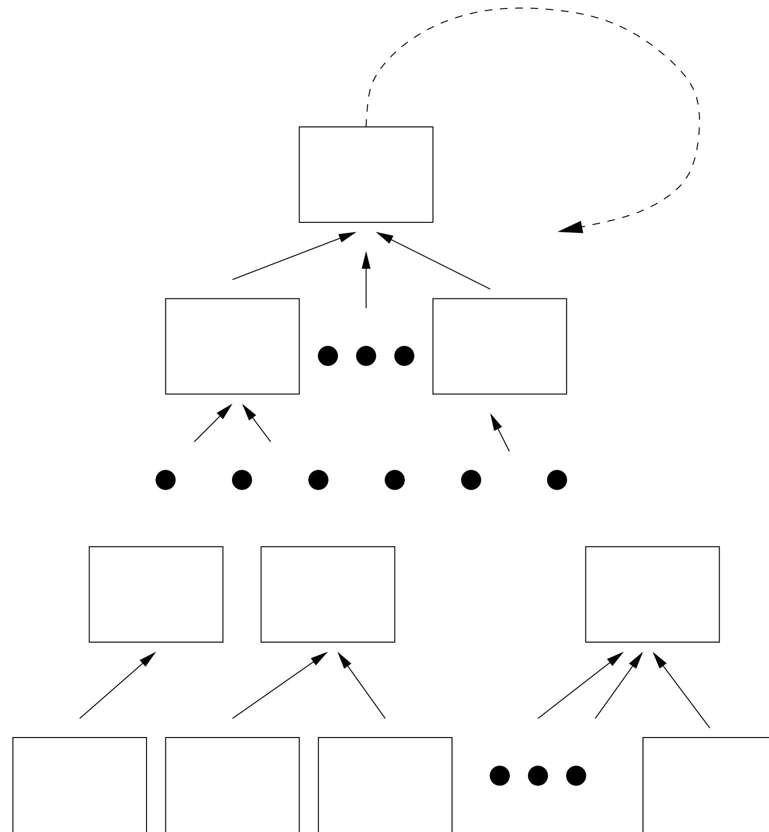


Figure 7: Properties of Steensgaard's points-to graph: Each node has at most one outdegree and zero or more indegree. Typically a connected component in Steensgaard's graph has a tree like structure as shown in the Figure. However, there is a possibility of only one cycle, which is formed by an edge from the root node to one of the nodes in the component. In the above graph component, such an edge is represented by a dashed line.

Steensgaard's algorithm enforces following properties in the points-to graph. A node in the graph has at most one outdegree and zero or more indegree. Owing to this, a connected component in the graph assumes a tree like structure, where a node can have multiple children corresponding to the indegree edges, but at most one parent depending on the presence of an outdegree edge. However, this does not imply that the component is always a tree. There is a possibility that the root node of the tree like structure may have an outward edge pointing to any of the nodes in the component, resulting in a cycle. Figure 7 shows such an edge as a dashed line in a typical connected component of a Steensgaard's points-to graph.

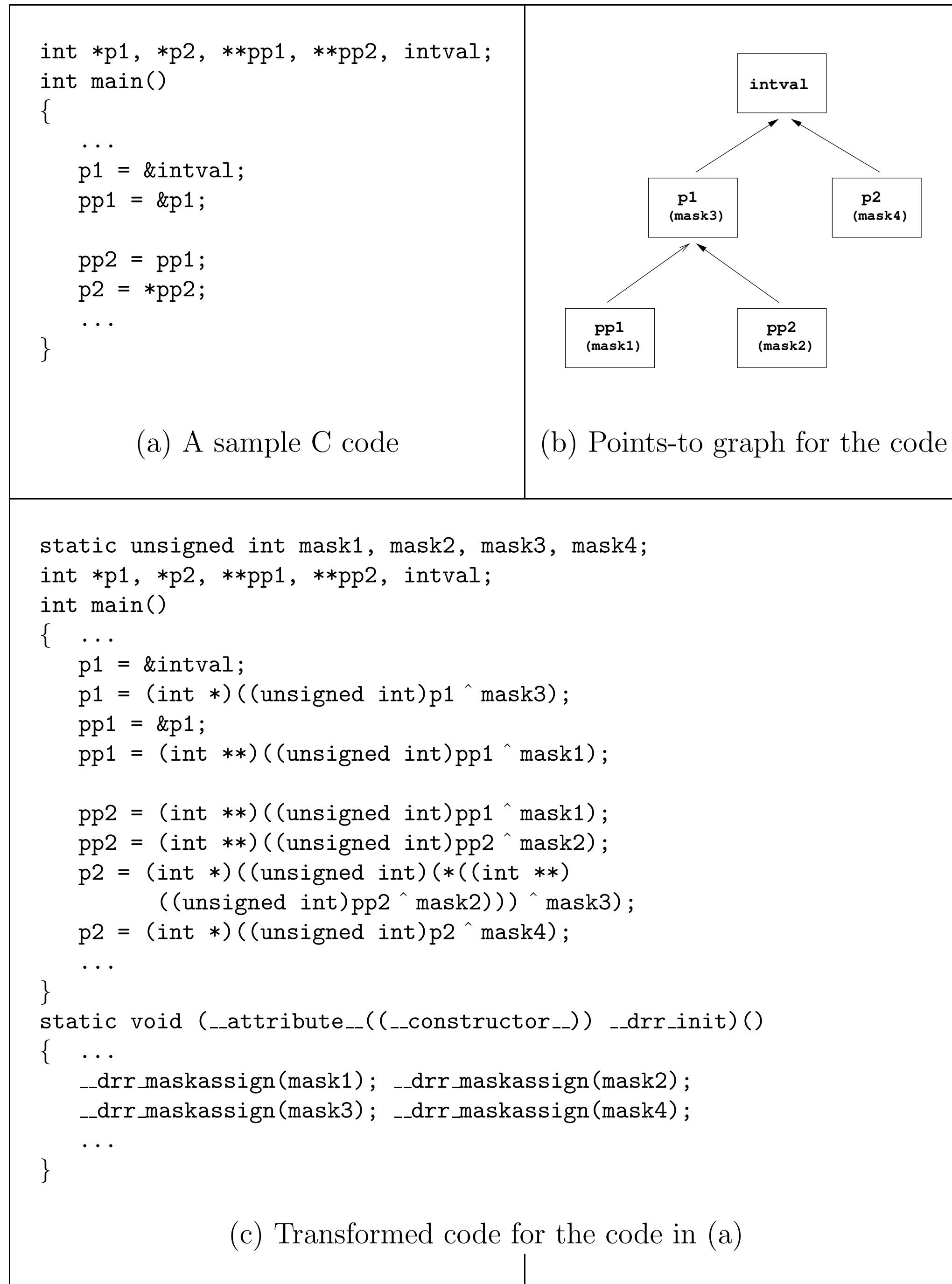
We assign a distinct mask to each node of the points-to graph. Note that a node may correspond to multiple variables. The mask of the node is thus used for masking all of its variables.

The mask of an object that is accessed using a pointer dereference is determined as follows. Let `ptr` be the pointer variable. First, the node N corresponding to the pointer variable is located in the points-to graph. For the object `*ptr`, its mask is the mask associated with the parent node $parent(N)$. Similarly, the mask of `**ptr` is the mask associated with $parent(parent(N))$, and so on. Since each node has at most one parent, we can uniquely determine the masks of objects accessed through pointer dereferences. Note that this procedure also works for dereferences of a non-pointer variable that stores an address because points-to graph captures the points-to relation involved. The procedure for deferences of pointer expressions involving pointer arithmetic is similar.

5.2 Implementation

Our transformation approach is applicable to C programs. We use CIL as the front end, and Objective Caml as the implementation language. We describe our implementation approach for a 32-bit x86 architecture and Linux operating system.

As a first step in the transformation of a program, we first perform pointer



(a) A sample C code

(b) Points-to graph for the code

(c) Transformed code for the code in (a)

Figure 8: A sample example illustrating basic DSR transformations.

analysis in order to determine masks associated with different data. Our current implementation supports Steensgaard's pointer analysis. The analysis has linear time complexity, and is thus scalable. One of the limitation of our current implementation is that it is based on whole program analysis and transformation. The whole program analysis approach requires a merged source file. The CIL toolkit provides an option to automatically generate such a merged file. Sometimes this kind of merging can fail due to type mismatch of variable declarations present in different files. Such cases can be handled by manual changes to the declarations. Notwithstanding this limitation, with some extra effort, our implementation could have been also extended to a separate file compilation-based transformation approach. Even with the current implementation approach, we have demonstrated its practicality by transforming several large "real-world" programs without any manual changes.

In the second step, we generate points-to graph, from which we then compute the equivalence classes needed for assigning random masks to data variables. In the third step, we transform the code as per the transformations described in the previous section.

The example shown in Figure 8 illustrates the above transformation steps.[2] To mask pointers `p1`, `p2`, `pp1`, `pp2`, we respectively introduce mask variables `mask3`, `mask4`, `mask1`, and `mask2`. Each of these mask variables are initialized with a different random value using the macro `__drr_maskassign` in the constructor function `__drr_init()` that is automatically invoked before the start of the execution in `main()`. The statements are transformed in such a way that if a variable is assigned a value, the value is masked and then stored in the memory; if a variable is used, its masked value is unmasked before its use.

Now we discuss a few issues concerning the basic implementation approach.

[2]This example does not show the effect of transformations for optimizations discussed later in this section.

5.2.1 Implementation Issues

5.2.1.1 Overflows within structures

According to C language specifications, overflows within structures are not considered as memory errors. However, attackers can potentially exploit such overflows also. For instance, an overflow from an array field inside a structure corrupting adjacent fields in the same structure may lead to an exploitable vulnerability. Thus, it is desirable to have some protection from these overflows. Unfortunately, complete memory error detection techniques do not provide defense against these types of overflows. ASR too fails to address this problem due to the inherent limitation of not being able to randomize relative distances between fields of a structure because of language semantics. DSR can be used to provide some level of protection in this case. The basic idea is to use field-sensitive points-to analysis so that we can assign different masks to different fields of the same structure.

Our current implementation does not support field-sensitive points-to analysis. As a part of future enhancement, we plan to implement Steensgaard's points-to analysis [46] to handle field-sensitivity. The time complexity of this analysis, as reported in [46], is likely to be close to linear in the size of the program in practice. Hence, this enhancement would not affect the scalability of our approach. Moreover, it does not increase runtime performance overhead.

5.2.1.2 Optimization

Masking of all the data may obviously lead to poor runtime performance. To improve performance, we have developed a few interesting runtime optimizations.

- *Avoid masking of all the data.* If we mask and unmask each type of data, we are expected to get high runtime performance overhead. If, somehow, we can avoid masking of all the data, we can reduce the overhead. An obvious question that arises here is whether we can avoid masking of some data without compromising the protection level offered by DSR. We show that this is possible.

To understand how this is possible, we first note that the essential goal of DSR is to prevent predictable interpretation of corrupted data. We achieve this goal as follows. We mask only pointer and buffer-type data, and make corruption of non-masked data either very difficult or impossible.

Attacks which corrupt data are either absolute address-dependent or relative address-dependent. In an absolute address-dependent attack, an attacker needs to corrupt a pointer data with a correct address. This is almost impossible as pointers are masked. An important point to be noted here is that absolute address-dependent attack may target corruption of either a pointer, or non-pointer value. In the case of non-pointer data corruption, the attack involves multiple steps, one of which is corruption of a pointer to the non-pointer data. Unsuccessful corruption of pointer objects defeats attacks of both the cases.

A relative address-dependent attack typically involves a buffer overflow corrupting some data at a predictable distance from the buffer. Since we mask all buffer-type data, the attacker cannot corrupt any buffer with a predictable value. Moreover, the attacker also cannot exploit buffer overflow to corrupt any adjacent data, because such data will be corrupted with a random mask (that of the buffer) unknown to the attacker. Furthermore, we separate memory for buffer-type data from other data with inaccessible memory pages. Owing to this, overflows from buffers into non-buffers become impossible.

As a further optimization, we avoid masking of pointers allocated on the main stack. The protection to such pointers is obtained in the following way. It is impossible to exploit buffer overflows in order to corrupt these pointers. And we make absolute address-dependent attacks difficult by randomizing the base of the main stack as we did in ASR. Because of this, the absolute addresses of the main stack data are highly randomized. So attackers cannot predict the locations of the non-masked pointers on the main stack, thereby making it difficult to corrupt them using absolute

address-based attack.

This optimization works very well in practice. Recall from Table 5 that relatively high percentage of data accesses are to the stack-allocated non-buffer-type data. As we do not add to the overhead in accessing these data objects, we gain significant performance improvement. Actual experiments to study the impact of this optimization would give us a better understanding, but we are leaving that analysis for future work. Nevertheless, the Table 5 itself serves as a good empirical evidence of huge gains obtained through this optimization.

- *Use of local mask variables wherever possible.* For static data (global) variables, we store their mask variables in the static memory region. For local data variables, which require masking, we store their masks in local data variables. These local mask variables have the same scope and lifetime as that of the variables that they mask. They are initialized with random mask values at the function entry point. Because the local mask variables are of simple data type (more precisely unsigned integers) and tend to go into registers, access to these variables are usually much faster.

  We only use local mask variables for those local variables that have no aliases, or in particular whose addresses are not taken. For a variable whose address is taken, it may be accessed in the scope of its function f, but in the body of a different function, say g, that is invoked from f. Such an access is possible using aliases, and for such accesses, we need to ensure that appropriate mask variables are used. This requires tracking all aliases to make sure that accesses to these aliases are performed with the same local mask variable. This is rather difficult, especially when the local variables are passed beyond the function scope.

  To keep the transformation conservative yet simple, we restrict the use of local mask variables only to mask the local variables whose addresses are not assigned to other variables, i.e., there are no aliases to those local variables.

5.2.1.3 Masking heap objects

Heap objects are typically allocated using standard C library functions such as `malloc` and `calloc`. If we treat heap objects as buffer-type data and mask them, we will get natural protection from their exploitable corruption. However, there is a potential issue involving buffer-to-buffer overflows within the heap. For buffer-to-buffer overflows, DSR protection is provided by ensuring that the adjacent buffers are masked with different random masks. This requires taking control of memory allocations for buffer-type data. We have shown how this is done for the static and the stack-allocated buffers. Whereas for heap objects, we do not have control over their memory allocations. As a result, there exists a possibility of adjacent heap objects having the same mask, and hence successful buffer-to-buffer overflow attacks are also possible. Thus, it is important to address this problem.

Ideally, we can get strongest protection if we are able to mask each heap data object with a different random mask. However, as we saw in the previous section, aliasing does not permit this. Aliasing may occur when a pointer may point to different heap objects. A common form of aliasing occurs in the way heap objects are created dynamically. For instance, the statement `charPtr = (char *) malloc(100)`, if executed in a loop, results in `charPtr` pointing to multiple heap objects. These heap objects, which are aliased together, require the same mask. The heap objects with the same mask should be allocated in different memory regions to prevent overflows among them. For this, we need to take control of heap-allocations. This implies that we need to either change implementation of existing heap code or wrap the allocation functions and take control of heap allocations. As heap objects tend to be large in numbers, we may need to create several memory regions each with minimum of a page size, separated by an inaccessible page. With a limited address space, especially for 32 bit architectures, it may not be always possible to create sufficient number of memory regions. In which case, we can randomly select memory regions and allocate the heap objects. In our current implementation, we take control of heap-allocations and use a simple scheme to randomly distribute heap objects over different memory regions. We also perform

relative address randomization on heap objects that are located in the same region. More complex address space randomization based-transformations, e.g., DieHard technique [7], can be also efficiently implemented.

5.2.1.4 Coping with untransformed libraries

Ideally, all the libraries used by an application need to undergo the transformations. However, in practice, source code may not be available for some libraries. Such libraries cannot be directly used with our DSR technique.

Untransformed shared libraries affect our transformations in two ways. First, certain external shared library functions may produce points-to relationships that our pointer analysis step could miss. Examples of such functions include `memcpy()` and `bcopy()` present in `glibc`. As a concrete example, consider an assignment statement `x = y` that could be emulated using function call `memcpy((void *)&x, (void *)&y, sizeof(x))`. In pointer analysis, the effect the statement `x = y` is such that any object that `y` points to may now be pointed by `x` also. However, this effect will not be captured in the case of `memcpy()` function as its code is not available for analysis. Currently, such functions are modeled manually to capture their effect in pointer analysis.

Second, if external variables or library function arguments are masked, their use in the library would result in an erroneous behavior. For instance, the arguments to library function `strcpy(char *dest, const char *src)` include pointers to two character arrays (i.e., buffer-type data), each of which could be masked with a different random mask. Invocation of this function would result in failure because the data present in the arrays is masked and would appear as a random byte-stream very different from characters. Thus failure could result from — missing terminating character `'\0'`, buffer overflow, copying with wrong mask, and so on.

One option to deal with this problem is to identify such external data and avoid masking it. However, this would result in a weaker protection level. Other option is to identify and flag warnings for such external data during transformation, and also give an interface to the users so that they can provide code to unmask data before corresponding external function call and mask it

Program	% Overhead
patch-1.06	4
tar-1.13.25	5
grep-2.5.1	7
ctags-5.6	11
gzip-1.1.3	24
bc-1.06	27
bison-1.35	28
Average	15

Table 7: Runtime performance overhead introduced by transformations for DSR.

again after the call.

In our current implementation, we do not mask external variables. Here we assume that external library data is protected by ASR. Also, for those library function call arguments which are masked, we provide an interface in the form of wrapper functions, allowing users to mask and unmask the data at entry and exit points of library calls. This interface provides access to the data as well as their masks. This way of coping with untransformed libraries could be also implemented as an automatic transformation. However, our current implementation supports only the manual transformation. For the test programs used in our experiments, we added 52 wrapper `glibc` functions.

5.3 Performance Results

Table 7 shows the runtime overheads, when the original and the transformed programs were compiled using `gcc-3.2.2` with optimization flag `-O2`, and run on a desktop running RedHat Linux 9.0 with 1.7 GHz Pentium IV processor and 512 MB RAM. We used the same workload for the test programs as in the experimentations of ASR, and execution times were averaged over 10 runs.

For DSR transformations, the runtime overhead depends mainly on memory accesses that result in masking and unmasking operations. In IO-intensive

programs, such as `tar` and `patch`, most of the execution time is spent in IO operations, and hence we see low overheads for such programs. On the other hand, CPU-intensive programs are likely to spend substantial part of the execution time in performing memory accesses. That is why we observe higher overheads for CPU-intensive programs. The average overhead is around 15%, which is slightly higher than the overheads that we see for ASR technique. Nonetheless, DSR technique is still practical and provides much stronger level of protection.

5.4 Effectiveness

For the reasons mentioned in Section 4.6, we use an analytical approach instead of an experimental approach in evaluating the effectiveness of DSR technique.

5.4.1 Memory Error Exploits

In Section 4.6, we considered the effect of transformations on different types of attacks. Here too, we will use the same method. In particular, we will compute the probability of a successful attack in terms of $P(Owr)$ and $P(Eff)$.

5.4.1.1 Estimating $P(Owr)$

We estimate $P(Owr)$ separately for each attack type.

Buffer overflows

- *Stack buffer overflows.*

  Memory for all the buffer-type local variables is allocated on a buffer stack. So for the stack buffer overflows, which typically target the main stack data, such as the return address and the saved base pointer, $P(Owr)$ is 0 because it is impossible to cause an overflow from a buffer stack to the main stack.

Attacks that corrupt a buffer-type variable by overflowing another buffer-type variable in the same buffer stack is possible. However, such attacks have very low probability of success ($P(\mathit{Eff}) = 2^{-32}$) because we ensure that the buffers on the same buffer stack have different masks.

- *Static buffer overflows.*

  Static buffers are separated from static non-buffers with inaccessible pages. So static buffer overflows cannot be used for corrupting non-buffer static data.

  Overflows from a static buffer into another buffer that is allocated in a different memory region are impossible. However, buffer-to-buffer overflows within the same static data region are possible. For such overflows, protection is provided in the form of relative address randomization. But more importantly, we ensure that all the buffers in the same region have different masks. For this reason, successful corruption using such overflows is nearly impossible ($P(\mathit{Eff}) = 2^{-32}$). Note that this protection is guaranteed only when we are able to create the required number of separate memory regions. This may not be always possible as the available virtual address space may not be sufficient and because of the facts that each region has to be at least a page size and there exists at least one inaccessible page between two adjacent regions. However, in our experience, this case never arises for static data buffers. The number of required regions depends on the size of the largest equivalence class of masks. In our experiments, we found that this number is typically small, less than 150 for all our test programs. So if consider 150 different regions, each of an average 4 page size, we require a virtual address space of less than 3 MB. However, this scenario is very different for heap objects, which we will consider in heap overflows, as discussed next.

- *Heap overflows.* One of the type of heap overflows is an attack targeting heap control data consisting of two pointer-valued variables appearing at the end of the target heap block. The overwrite step of this attack is

easily possible, however, the corruption of pointers involve guessing the mask of the target heap object, which is very difficult.

Overflows from one heap block to the next are possible. However, such overflows are useless if the two heap objects are masked with different masks. And in fact we try to ensure that the adjacent heap objects have different associated masks. However, heap objects tend be large in numbers, and moreover, aliasing may force us to assign the same mask to several heap objects. An important point to be noted in this case is that the number of different memory regions required for heap objects is a property of input to the program, rather than the program itself. Hence we use a probabilistic approach, and distribute heap objects randomly over a bounded number of different memory regions. Moreover, we also perform relative address randomizations over heap objects. We believe that these transformations are instrumental in significant decrease in the probability $P(Owr)$.

Format string attacks. $P(Owr)$ for format string attacks is estimated in the same way as in ASR.

In summary, DSR technique significantly reduces the success probability of most likely attack mechanisms by reducing $P(Owr)$. For attacks with high $P(Owr)$ values, now we demonstrate that a very low $P(Eff)$ gives us strong protection.

5.4.1.2 Estimating $P(Eff)$

Corruption of non-pointer data. Unlike ASR which has no bearing on the interpretation of non-pointer data (i.e., $P(Eff) = 1$), DSR technique makes the interpretation of corrupted non-pointer data highly non-deterministic.

In DSR transformations, non-pointer data can be either masked or unmasked. For unmasked non-pointer data, protection is provided in the form of a small $P(Owr)$. A successful attack involving corruption of a masked non-pointer data requires correctly guessing the mask value. The probability of

correctly guessing a mask value is 2^{-32}. However, in a buffer-to-buffer overflow attack, where the target and the source buffers have the same mask, an attacker can successfully corrupt a non-pointer data in the target buffer without guessing its mask value. Protection against these attacks is additionally provided in the form of a small $P(Owr)$.

Pointer corruption attacks. Pointers could be either masked or unmasked. For unmasked pointers, protection is provided in the form a small $P(Owr)$. The probability of a successful masked pointer corruption is 2^{-32}. This probability is applicable to attacks involving corruption of pointers to point to either existing code or data.

For attacks that corrupt pointers to point to injected code or data, the odds of success can be improved by repeating the attack data many times over. For an overflow with $4K$ limit, a 16 byte data or code can be repeated 2^8 number of times. Therefore, the probability of a successful pointer corruption in this case is 2^{-24}.

5.4.2 Attacks Targeting DSR

Some of the attacks that target the weaknesses in ASR, can also be used against DSR technique.

- **Information leakage attack.**
  This attack allows an attacker to read memory of a victim process. Known examples of such an attack include format string attacks that allow inspection of main stack contents. This could be exploited to leak the values of any local mask variables stored on the stack. This causes the value of $P(Eff)$ to become 1 for corrupting the corresponding buffer stack data. However, an attacker still has to overcome the low probability $P(Owr)$ of the overwrite step.

  If a masked data is leaked, it does not help in retrieving other masks because they are independently assigned.

- **Brute force and guessing attacks.**

  These attacks particularly become very difficult because of very low probability values of $P(Owr)$ and/or $P(Eff)$.

- **Partial pointer overwrites.**

  These attacks involve corruption of the lower byte(s) of a pointer. Since pointers are masked, we obtain protection against these attacks. Although $P(Eff)$ may have a high value for these attacks, the attacks could still be very difficult because of low $P(Owr)$ values.

CHAPTER 6

Related Work

6.1 Runtime Guarding

These techniques transform a program to prevent corruption of return addresses or other specific values. StackGuard [17] provides a *gcc* patch to generate code that places *canary values* around the return address at runtime, so that any overflow which overwrites the return address will also modify the canary value, enabling the overflow to be detected. StackShield [5] and RAD [13] provide similar protection, but keep a separate copy of the return address instead of using canary values. Libsafe and Libverify [5] are dynamically loaded libraries which provide protection for the return address without requiring recompilation. ProPolice [20] further improves these approaches to protect pointers among local variables. FormatGuard [15] transforms source code to provide protection from format-string attacks.

6.2 Runtime Bounds and Pointer Checking

Several techniques [29, 3, 48, 25, 23, 27, 34, 42, 53] have been developed to prevent buffer overflows and related memory errors by checking every memory access. These techniques currently suffer from one or more of the following drawbacks: runtime overheads that can often be over 100%, incomaptibility

with legacy C-code, and changes to the memory model or pointer semantics.

6.3 Compile-Time Analysis Techniques

These techniques [22, 41, 49, 18, 30] analyze a program's source code to detect potential array and pointer access errors. Although useful for debugging, they are not very practical since they suffer from high false alarm rates, and often do not scale to large programs.

6.4 Randomizing Code Transformations

Our randomization techniques are instances of the broader idea of introducing diversity in nonfunctional aspects of software, an idea first suggested by Forrest, Somayaji, and Ackley [21]. The basic idea is that the diverse software replicas maintain the same functionality, but differ only in their implementation details. This makes diverse replicas less prone to sharing common vulnerabilities. In the context of memory error vulnerabilities, several recent works have demonstrated the usefulness of introducing automated diversity in the low level program parameters, such as memory layout and instruction set, as a practical defense. Diversity in the low level parameters is effective because attacks typically depend on the precise knowledge of these parameters. Diversity gives probabilistic protection. The level of protection depends on how much diversity is introduced. Recently, Pucella and Schneider [38] have shown that diversity is a form of *probabilistic type checking.*, in which type-incorrect operations cause programs to halt with some probability. While *strong typing* (no memory errors) offers complete protection against memory errors, it is not practical because of high overheads involved in dynamic type checking. On the other hand, the effect of diversity in the form of probabilistic type checking can be efficiently obtained with varying degree of protection.

We provide an overview of several recent works on diversity-based defenses and compare our work with them.

6.4.1 System Call Randomization

Some exploits inject attack code containing instructions to make direct system calls in order to interact with the operating system. Following instructions show how system call `execve("/bin/sh")` is invoked in a Linux operating system running on an x86 architecture.

```
<setup arguments of execve>
movl  $0xb, %eax
int   $0x80
```

The constant value `0xb` corresponding to the index of the system call `execve` is first loaded in the register `eax` and then an interrupt instruction `int $0x80` is invoked, thereby transferring the control to the kernel code which handles execution of the system call.

A possible defense against the above attack is to randomize the mapping of system calls [12]. Against this defense, attackers are forced to guess the system call numbers. If a wrong number is used, the correct system call will not be made. The implementation of this technique requires recompilation of the kernel with the randomized system call mapping. Additionally, it requires rewriting of existing binaries such that the old system call numbers are replaced to reflect the new mapping.

This approach has several disadvantages. First, re-randomization of system call number mapping is not practical as it would require kernel recompilation and rewriting of all the binaries. Second, the static instrumentation of binaries is a complicated task (see Section 4.3.5), and it might not be always feasible. Dynamic instrumentation approach [9] can be used, but it causes moderate increase in the performance overheads. Third, the defense is effective against only the injected code attacks, but is vulnerable to other types of attacks such as existing code attacks and attacks on security sensitive data.

6.4.2 Instruction Set Randomization

In this approach [28, 6], process-specific instruction set is randomized with a private randomization key. Each instruction is interpreted by a virtual machine which first derandomizes the instruction and then passes it over to the processor for execution. As an alternative, the architecture of processors can be modified to directly interpret the randomized instructions with the help of the private key.

This approach also provides limited protection as it counters only the injected code attacks.

6.4.3 Pointer Randomization

PointerGuard [16] approach randomizes ("encrypts") representation of all the stored pointer values. The encryption is achieved by xor'ing pointer values with a random integer mask generated at the beginning of program execution. This provides protection against any attack that involves corruption of pointer values. For a successful attack, attackers are forced to guess the value of the random integer mask to generate a desired pointer value for corruption.

The principal disadvantage of this approach is that it does not protect against attacks that do not corrupt pointer values, e.g., a buffer overflow to corrupt adjacent security-critical data. It should also be noted that PointGuard is dependent on the availability of accurate type information. Many C-language features, such as the ability to operate on untyped buffers (e.g., `bzero` or `memcpy`), functions that take untyped parameters (e.g., `printf`), unions that store pointers and integer values in the same location, can make it difficult or impossible to get accurate type information, which means that the corresponding pointer value(s) cannot be protected.

Our DSR technique does not suffer from these issues. The technique sound as it addresses aliasing between pointers and non-pointers. Moreover, it also provides protection against non-pointer data corruption attacks.

6.4.4 Address Space Randomization

Other address space randomization techniques [36, 52, 21] were primarily focused only on randomizing the base address of different sections of memory. In contrast, our address space randomization technique implements randomizations at a much finer granularity, achieving relative as well as absolute address randomization. Moreover, it makes certain types of buffer overflows impossible. Interestingly, our ASR implementation can achieve all of this, while incurring runtime overheads comparable to that of the other techniques.

CHAPTER 7

Conclusions

In this dissertation, we presented software diversity-based techniques to defend against exploits of memory errors. Previous defense techniques either offered incomplete protection, or were impractical to use because of problems such as high runtime overheads and/or code incompatibility. On the other hand, we showed that by randomizing the effect of memory errors, we can develop practical defense mechanisms against a broad range of memory error exploits. In this regard, we presented two randomization techniques.

By re-arranging various code and data objects in the memory space of the program, our first technique, address-space randomization, addresses the basis of the attacks — predictable locations of code and data objects. This technique comprehensively randomizes absolute as well as relative addresses of all the objects in the memory. Our initial implementation was based on binary-level program transformation for a wide potential impact. With this approach, we could achieve the absolute address randomization of all the memory regions, and some level of relative address randomization in only the stack and the heap area. However, for comprehensive randomizations, we need relative address randomization in code and data regions also. Transformation for these randomizations on binaries is not always feasible due to the difficulties associated with binary analysis. To counter this problem, we proposed an enhancement to ELF binary file format to augment binaries with an extra section that contains the information required to safely perform the randomizing

transformations. We also presented an alternative implementation strategy in case source code is available. It involves a source-to-source transformation that produces self-randomizing programs, which performs all the absolute and relative address randomizations at load-time or continuously at runtime. Experimental and analytical evaluation established that our approach is practical as it incurs low overheads, completely blocks a number of known exploits, and provides probabilistic protection against other exploits with a very small chance of success. We have released the transformation tool [8] for ASR under GPL.

Our second randomization technique, data space randomization, is based on randomization of data representation. The basic idea is to randomize the representation by xor'ing each data object with a random mask. Using this technique, we showed how the use of any corrupted data becomes highly unpredictable. When an attacker corrupts a data in the process memory, the victim program will first xor the data with its random mask before using it. As a result, the data will have an interpretation different from what the attacker intended. Similar idea was suggested before in the PointGuard technique, but for randomization of only the pointer representations. Our work showed improvement over PointGuard in two significant ways. First, we showed that we can apply our technique to any type of data – pointer or non-pointer data. Moreover, we showed that different data can have different representations by xor'ing them with different random masks. Second, our technique is sound, i.e., it does not break any correct program, whereas PointGuard technique breaks programs. PointGuard technique is not sound because it does not handle aliasing between pointer and non-pointer data. On the other hand, we presented a sound way to handle such aliasing. So effectively, what we get is a sound approach that provides strong and comprehensive protection against all the exploits of memory errors. For this reason, data space randomization technique is expected to become the next-frontier in the line of defense against memory error exploits.

In software industry, often computer security is totally overlooked in favor of increased performance, productivity and budgetary concerns. Security

has mostly been regarded as an afterthought. Security features are added only after attacks are discovered. Very often the features, which are added in *ad-hoc* manners, do not provide comprehensive protection against attacks. As modern software systems are becoming progressively complex, we are seeing more and more number vulnerabilities and their attacks, many of which are capable of causing significant damages. It is high time security be considered as a core component of software development process. The research presented in this dissertation offers some new directions for incorporating practical security mechanisms into the development process. The defense techniques presented in this dissertation are fully automated, needing little or no programmer intervention, and can be seamlessly applied using the existing compiler infrastructures. Their key features such as comprehensive protection, no compatibility issues, and low overheads, are attractive in terms of finding widespread acceptance. In countering the most important class of attacks of the past decade, these techniques would form a formidable arsenal of tools for software developers. The techniques would also empower administrators and common users to defend their software without having to depend on vendors to provide patches for vulnerabilities.

Bibliography

[1] L. O. Andersen. Program analysis and specialization for the C programming language. PhD Thesis, DIKU, University of Copenhagen, May 1994. Available at ftp.diku.dk/pub/diku/semantics/papers/D-203.dvi.Z.

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

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

[4] A. Avižienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, December 1985.

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

[6] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanović, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *ACM conference on Computer and Communications Security (CCS)*, Washington, DC, October 2003.

[7] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 158–168, Ottawa, Canada, June 2006.

[8] S. Bhatkar. Transformation tool for comprehensive address space randomization. Available for download from `http://www.seclab.cs.sunysb.edu/asr`, 2005.

[9] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization*, March 2003.

[10] CERT advisories. Published on World-Wide Web at URL http://www.cert.org/advisories, September 2007.

[11] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[12] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.

[13] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *IEEE International Conference on Distributed Computing Systems*, Phoenix, Arizona, April 2001.

[14] C. Cifuentes, M. van Emmerik, N. Ramsey, and B. Lewis. The university of queensland binary translator (UQBT) framework. Technical report, The University of Queensland, Sun Microsystems, Inc, 2001.

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

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

[17] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive

detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, pages 63–78, San Antonio, Texas, January 1998.

[18] N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 194–212. Springer Verlag, June 2001.

[19] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[20] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. Published on World-Wide Web at URL http://www.trl.ibm.com/projects/security/ssp/main.html, June 2000.

[21] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Workshop on Hot Topics in Operating Systems*, pages 67–72, Los Alamitos, CA, 1997. IEEE Computer Society Press.

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

[23] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *USENIX Winter Conference*, pages 125–138, Berkeley, CA, USA, January 1992.

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

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

[26] M. Kaempf. Vudo malloc tricks. *Phrack*, 11(57), August 2001.

[27] S. Kaufer, R. Lopez, and S. Pratap. Saber-C: An interpreter-based programming environment for the C language. In *USENIX Summer Conference*, pages 161–171, San Francisco, CA, June 1988.

[28] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM conference on Computer and Communications Security (CCS)*, pages 272–280, Washington, DC, October 2003.

[29] S. C. Kendall. Bcc: run–time checking for C programs. In *USENIX Summer Conference*, El. Cerrito, CA, 1983.

[30] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, Washington, DC, August 2001.

[31] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, La Jolla, California, June 1995.

[32] 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.

[33] Mudge. How to write buffer overflows. Published on World-Wide Web at URL http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html, 1997.

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

[35] M. L. Nohr. *Understanding ELF Object Files and Debugging Tools*. Number ISBN: 0-13-091109-7. Prentice Hall Computer Books, 1993.

[36] PaX. Published on World-Wide Web at URL http://pax.grsecurity.net, 2001.

[37] M. Prasad and T. Chiueh. A binary rewriting defense against stack-based buffer overflow attacks. In *USENIX Annual Technical Conference*, San Antonio, TX, June 2003.

[38] R. Pucella and F. Schneider. Independence from obfuscation: A semantic framework for diversity. In *IEEE Computer Security Foundations Workshop*, pages 230–241, Venice, Italy, July 2006.

[39] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, September 1994.

[40] E. Ruf. Context-insensitive alias analysis reconsidered. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–22, June 1995.

[41] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, Vancouver, British Columbia, Canada, 2000.

[42] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Network and Distributed System Security Symposium*, pages 159–169, San Diego, CA, February 2004.

[43] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *IEEE Working Conference on Reverse Engineering*, October 2002.

[44] scut. Exploiting format string vulnerabilities. Published on World-Wide Web at URL http://www.team-teso.net/articles/formatstring, March 2001.

[45] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM conference on Computer and Communications Security (CCS)*, pages 298–307, Washington, DC, October 2004.

[46] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Conference on Compiler Construction*, LNCS 1060, pages 136–150, April 1996.

[47] B. Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 32–41, January 1996.

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

[49] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, San Diego, CA, 2000.

[50] WebStone, the benchmark for web servers. http://www.mindcraft.com/webstone.

[51] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

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

[53] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Newport Beach, CA, November 2004.

[54] L. Xun. A linux executable editing library. Masters Thesis, 1999. available at http://www.geocities.com/fasterlu/leel.htm.