BOUNDWARDEN: THREAD-ENFORCED SPATIAL MEMORY SAFETY THROUGH COMPILE-TIME TRANSFORMATIONS

Mr. Smith Dhumbumroong

A Dissertation Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy (Computer Engineering) in Computer Engineering

Department of Computer Engineering

Faculty of Engineering

Chulalongkorn University

Academic Year 2018

ผู้คุมขอบเขต: สายโยงใยรักษาความปลอดภัยเชิงพื้นที่สำหรับหน่วยความจำผ่านทางการแปลงชุดคำสั่งโดยตัวแปลโปรแกรม

นายสมิทธ์ ธรรมบำรุง

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรดุษฎีบัณฑิต

สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

ปีการศึกษา 2561

Dissertation Title	BOUNDWARDEN: THREAD-ENFORCED SPATIAL MEMORY SAFETY THROUGH COMPILE-TIME TRANSFORMATIONS
By	Mr. Smith Dhumbumroong
Field of Study	Computer Engineering
Dissertation Advisor	Assistant Professor Dr. Krerk Piromsopa

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Doctoral Degree

Dean of the Faculty of Engineering

. . . .

(Professor Dr. Supot Teachavorasinskun)

DISSERTATION COMMITTEE

. . . . Chairman

(Assistant Professor Dr. Natawut Nupairoj)

. . . . Dissertation Advisor

(Assistant Professor Dr. Krerk Piromsopa)

. . . . Examiner

(Assistant Professor Dr. Veera Muangsin)

. . . . Examiner

(Professor Dr. Prabhas Chongstitvatana)

. . . . External Examiner

(Assistant Professor Dr. Putchong Uthayopas)

สมิทธ์ ธรรมบำรุง: ผู้คุมขอบเขต: สายโยงใยรักษาความปลอดภัยเชิงพื้นที่สำหรับหน่วยความจำผ่านทางการแปลงชุดคำสั่งโดยตัวแปลโปรแกรม. (BOUNDWARDEN: THREAD-ENFORCED SPATIAL MEMORY SAFETY THROUGH COMPILE-TIME TRANSFORMATIONS) อ.ที่ปรึกษาวิทยานิพนธ์หลัก : ผศ. ดร. เกริก ภิรมย์โสภา, 87 หน้า.

วิทยานิพนธ์ฉบับนี้นำเสนอผู้คุมขอบเขต โดยผู้คุมขอบเขตคือระบบรักษาความปลอดภัยเชิงพื้นที่สำหรับหน่วยความจำที่สามารถทำการตรวจจับและป้องกันบัฟเฟอร์โอเวอร์โฟลว์และข้อผิดพลาดต่างๆ ที่เกิดขึ้นจากการอ่านหรือเขียนข้อมูลที่อยู่นอกขอบเขตของบัฟเฟอร์ ไม่ว่าบัฟเฟอร์นั้นจะอยู่ในหน่วยความจำแบบกองซ้อน, ฮีป, BSS, หรือ ดาต้าเซกเมนต์ก็ตาม ผู้คุมขอบเขตใช้ประโยชน์จากความแพร่หลายของตัวประมวลผลหลายแกน โดยทำการผลักภาระการตรวจสอบขอบเขตไปให้สายโยงใยตรวจสอบขอบเขตให้มากที่สุด เพื่อลดผลกระทบทางลบของการตรวจสอบขอบเขตต่อประสิทธิภาพในการทำงานของโปรแกรม นอกจากนี้ ผู้คุมขอบเขตยังทำการบันทึกขอบเขตของแต่ละบัฟเฟอร์ลงในตารางบันทึกขอบเขตแทนที่จะต้องทำการแก้ไขโครงสร้างของหน่วยความจำ ทำให้ลดปัญหาที่อาจจะเกิดขึ้นจากการทำงานร่วมกับคลังโปรแกรมและไบนารี่ต่างๆ ผลจากการทดลองแสดงให้เห็นว่า ผู้คุมขอบเขตมีประสิทธิภาพในการรักษาความปลอดภัยเชิงพื้นที่ โดยผู้คุมขอบเขตนั้นสามารถตรวจจับและป้องกันการโจมตีทั้ง 850 แบบของชุดทดสอบ RIPE และผ่านการประเมินของแบบทดสอบ 1,092 แบบ จากทั้งหมด 1,164 (คิดเป็น 94%) ของชุดทดสอบหมายเลข 89 ของ SARD ในขณะที่ผลจากการทดสอบสมรรถนะการทำงานโดยใช้เกณฑ์เปรียบเทียบสมรรถนะ Olden แสดงให้เห็นว่า โปรแกรมที่ใช้ผู้คุมขอบเขตมีสมรรถนะการทำงานลดลงโดยประมาณ 2.25 เท่าโดยเฉลี่ย เมื่อเทียบกับโปรแกรมที่ไม่ได้ใช้ผู้คุมขอบเขต

ภาควิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่อนิสิต	....
สาขาวิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่ออ.ที่ปรึกษาหลัก	....
ปีการศึกษา	2561		

5571432021: MAJOR COMPUTER ENGINEERING

KEYWORDS: SPATIAL MEMORY SAFETY, BUFFER OVERFLOW, THREAD, CONCURRENCY, LLVM, COMPILER

SMITH DHUMBUMROONG : BOUNDWARDEN: THREAD-ENFORCED SPATIAL MEMORY SAFETY THROUGH COMPILE-TIME TRANSFORMATIONS. ADVISOR : Asst. Prof. Dr. Krerk Piromsopa, 87 pp.

This dissertation presents BoundWarden, a novel runtime spatial memory safety enforcement technique that comprehensively detects and prevents buffer overflow and other out-of-bound errors in buffers on stack, heap, and BSS and data segments of memory. BoundWarden leverages the ubiquity of multi-core processors by offloading most of the works to a dedicated bound checking thread, which performs bound checking and manages metadata, thus reducing the runtime overhead. Since BoundWarden stores base and bound of buffers in a dedicated bound table, the memory layout of programs remains unchanged, thus preserving compatibility with existing libraries and binaries. Experiments showed that the prototype of BoundWarden is effective at enforcing spatial memory safety by successfully detected all 850 attacks of RIPE test suite, and 94% (1,092 out of 1,164 tests) of NIST SARD Test Suite 89, while the results from Olden benchmark showed that on average BoundWarden introduced roughly 2.25x overhead, compared to the uninstrumented code.

Department	:	Computer Engineering	Student's Signature	....
Field of Study	:	Computer Engineering	Advisor's Signature	....
Academic Year	:	2018		

ACKNOWLEDGEMENTS

I would like to express my utmost gratitude to my advisor, Dr. Krerk Piromsopa, who provided invaluable guidance and assistance throughout my time as a student at Chulalongkorn University. Dr. Krerk's intellectual heft is matched only by his boundless patience, compassion, and genuinely good nature, and I consider myself really fortunate to have had the opportunity to be his advisee.

I would also like to express my deepest gratitude to my dissertation committee: Dr. Natawut Nupairoj, Dr. Prabhas Chongstitvatana, Dr. Veera Muangsin, and last but not least Dr. Putchong Uthayopas from Kasetsart University. They provided essential and valuable guidance, especially in finishing my dissertation, and their invaluable comments, suggestions, and criticisms improved the quality of my dissertation immensely.

During my time at Chulalongkorn University, I have been very fortunate to enjoy the support, encouragement, and advise of many friends and colleagues, too numerous to list here, to whom I would like to express my sincere gratitude.

Also, I would like to thank my wife, who always supports me and has the patience of a saint.

Finally, I would like to dedicate this dissertation to my late father and mother, both of whom passed away too soon. Both of them always believed in me, even when no one else does, and encouraged me to always strive to improve myself. Thank you, Mom and Dad, for all the love, support, and dedication.

Smith Dhumbumroong

CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

For the past four decades since the idea of an attack based on buffer overflow bugs was first documented in the 1970s [3] and three decades since the infamous Morris worm [29], buffer overflow errors still remain one of the major source of security vulnerabilities. In fact, according to recent data [1] from MITRE Corporation's Common Vulnerabilities and Exposures (CVE) database [41], as shown in Figure 1.1, buffer overflow bugs have been consistently ranked as one of the top three leading cause of vulnerabilities for the past two decades. Not only that, but the latest data from 2018 showed that buffer overflow errors finally overtook denial-of-service (DoS) attacks as the second leading cause of security vulnerabilities. Clearly, these data show that the class of attacks that exploit buffer overflow errors and other memory corruption bugs is still as relevant and dangerous today as it has ever been, and likely would remain so in the future for as long as languages that lack bound checking and allow unrestricted pointer arithmetic/ manipulations, such as C or C++, are in widespread usage.

Various techniques have been proposed over the years to try to prevent or mitigate exploits based on buffer overflow. However, despite being effective at mitigating or preventing buffer overflows, very few of these techniques have actually seen widespread deployment, which is due to the following reasons [40]:

1. The majority of the techniques have high runtime overheads, especially the majority of pure software-based approaches;
2. Others cause incompatibility issues with existing libraries or binaries; and
3. The rest have significant barriers to adoption: some require users to adopt a new language or an unofficial extension to a language, while others require new hardware.

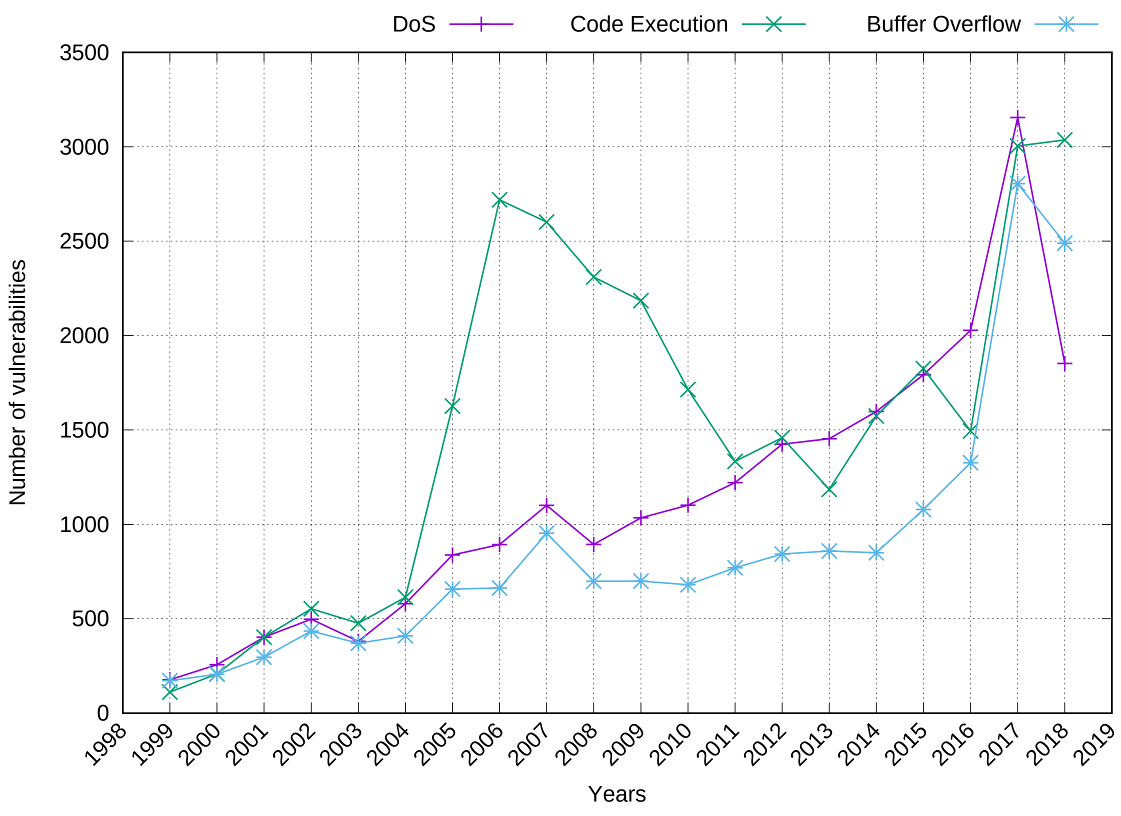


Figure 1.1: Top three types of vulnerabilities according to CVE vulnerability database over the period of 20 years — from 1999 to 2018 [1].

Clearly, the main challenge is in coming up with a solution that can satisfied the following three requirements: performance, compatibility, and ease of use. That is, an ideal solution should be compatible with existing libraries and binaries, while also has an acceptable overheads and is easy to deploy.

In this paper, we propose BoundWarden, a spatial memory safety enforcement approach that utilizes a bound checking thread to detect and prevent any buffer overflow error in a process during runtime.

One of the major design constrained of ours is that we aim to preserve the semantic of C programming language, as well as maintain compatibility with existing C libraries and binaries. We achieve this goal by storing base and bound metadata of buffers inside a dedicated bound table, thus preserving the existing pointer presentation and memory layout.

To reduce performance overhead, we exploit the fact that multi-core processors are now the norm, even among embedded devices, and perform bound checking during runtime on a dedicated bound checking thread, running on its own CPU core in parallel to the process's main thread. We also implemented various optimizations to further reduce the runtime overhead.

In order to ensure that our proposed solution has the lowest possible adoption barrier, our approach is purely software-based, requiring no new hardware. We also implement the compiler extension component of BoundWarden prototype of our system as an LLVM pass, using the standard API, and the runtime component as a static library. Thus, it is trivial for anyone with access to the Clang/LLCM compiler toolchain [19] to apply our proposed countermeasure to his or her programs.

1.1 Objectives

The followings are the objectives of our research:

1. To propose a design of BoundWarden, a new, purely software-based runtime bound checking approach that is capable of preventing buffer overflow attacks;
2. To implement the proposed design of BoundWarden and evaluate its effectiveness, specifically its performance and protection coverage, against other equivalent bound checking methods.

1.2 Threat Model and Scope

We assume that attackers are trying to exploit memory corruption bugs, such as buffer overflow or buffer over-read, to either read or write memory. We assume that the aims of the attackers are to either leak sensitive information or to modify memory contents to redirect control flow to execute malicious code. This means that we assume that the attackers will try to induce spatial memory safety violations on the stack, heap, as well as BSS and data segments.

We assume that the attackers will try to perform both local and remote attacks, and that the attackers, if their goal is to hijack control flow, will choose to either inject code into the application and redirect control flow towards the malicious code or try to perform attacks based on Return-Oriented Programming technique.

We assume that the attackers are targeting user-space applications, such as web servers or other services. In addition, we also assume that the underlying OS and the system libraries are secure, and that the attackers do not have the ability to modify source code of the applications on the system that they try to attack.

Given the above assumptions, the following are the scope of our work:

1. BoundWarden only works on C source code;
2. BoundWarden only works with Clang/LLVM compiler and toolchain;
3. BoundWarden only prevent buffer overflow attacks that rely on overflowing C's arrays, structs, and unions;
4. BoundWarden protects buffers on stack (control data), heap (non-control data), as well as BSS and data segments;
5. BoundWarden only works with programs in userspace;
6. BoundWarden only support 32-bit and 64-bit computers that are compatible with the Intel 64/AMD64 instruction set architecture;
7. BoundWarden does not protect against temporal safety violations.

1.3 Contributions

The following are the contributions of our work:

1. A software-based spatial memory safety enforcement approach that utilizes a thread running on a dedicated CPU core to enforce spatial safety by actively fol-

lowing the execution of the threads of a program and performing bound checking on buffers that are being accessed.

2. We showed that the idea of using concurrent threads to detect and prevent buffer overflows, first proposed in [46], can be extended to cover all buffer in userspace. Our approach can comprehensively detect and prevents spatial safety violations in buffers on stack, heap, and BSS and data segments, as well as in buffers that are allocated inside a structure or a union. Also, in addition to buffer overflows, our approach can also detect all kind of out-of-bound errors, such as buffer underflow, buffer over-read, and buffer under-read.

3. We give a very in-depth description of the design and implementation of our system and the optimizations that we implemented. We then evaluate the prototype of our approach using publicly available benchmark and test suites, and describe the strengths and weaknesses of our design.

CHAPTER II

BACKGROUND

In this chapter, we begin by first giving a precise definition of buffers and their bounds in Section 2.1. Next, in Section 2.2, we describe out-of-bound errors and buffer overflows in terms of the previously defined buffers and bounds, and how they can be used to craft various kind of exploits. Section 2.3 describes memory safety and how it relates to the notions of spatial and temporal memory safeties. Section 2.4 briefly touch on the problems one faced when trying to design the purely software-based spatial memory safety enforcement technique.

2.1 Buffers and Their Bounds

In this paper, we use the word buffer to refer to either an array, a structure (or struct), or a union, *i.e.*, a continuous sequence of objects, which are in turn defined as continuous sequences of one or more bytes, in memory that can be either of the same or different types [13]. As an example, given the following code snippet:

```
int foo[5] = {1, 2, 3, 4, 5};
```

Assuming that the above snippet is compiled and ran on a 64-bit machine that is compatible with the Intel 64/AMD64 instruction set architecture, and that during runtime the beginning of the array `foo` is located at memory address `0x7fff6b33cb20`, then the “bounds” of the array `foo` are located between memory addresses `0x7fff6b33cb20` and `0x7fff6b33cb34` (assuming that the size of the `int` type on the machine is 4 bytes and that the compiler didn’t add any additional padding).

2.2 Out-of-Bound Errors

Since the C programming language lacks bound checking, there are two types of out-of-bound errors that can occur: buffer overflow that occurs when writing operation goes out-of-bound, and buffer over-read which occurs when reading operation goes out-of-bound.

2.2.1 Buffer Overflow

Buffer overflow occurs when either a read or write operation occurs outside the bounds of a buffer. For example, consider the following code snippet:

```c
int bar[7];
bar[7] = 42;
```

In the above example, a buffer overflow occurs because we are trying to assign a value using an invalid index and thus ended up writing into a memory address which is outside the bound of the array `bar`, potentially overwriting other values and causing errors or undefined behaviors.

The following program shows a more common pattern of buffer overflow vulnerability:

```c
int main(void) {
  int alpha = 42;
  int beta[10];
  for (int i = 0; i < 11; i++) {
    beta[i] = i;
  }
  printf("The value of alpha is: %d\n", a);
  return 0;
}
```

Which will produces the following output when executed:

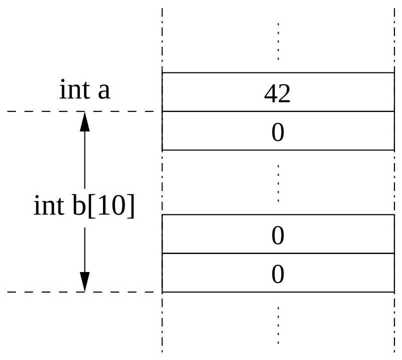
```
The value of alpha is: 10
```

Which is not the result we expected (the value of the `alpha` variable should be 42).

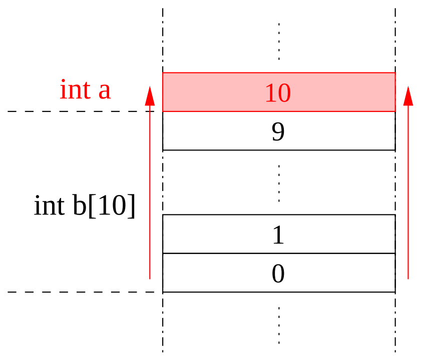
This error is caused by a bug in the for loop, which causes the loop to go out of the bound of the array `beta` during the 11th iteration (when the value of the variable `i` is equal to 10), and caused the assignment operator (=) to assign the value 10 to a memory address that is outside the bound of the array `beta` (or “overflowing” into a memory address that is adjacent to the array), thus resulting in the error/undefined behavior.

We can visualize the sequence of what happen on the stack as follows.

The following figure shows the initial state of the stack when we declare the variables `alpha` and `beta` (lines 2 and 3 from the above code snippet):



The next figure shows the state of the stack during the 11th iteration of the for loop (line 4 and 5 from the code snippet):



As can be seen from the above figures and code snippets, on the 11th iteration, the for loop goes outside the bound of the `beta` array and overwrote the value of the `alpha` variable that is located next to the `beta` array.

2.2.2 Exploiting Buffer Overflow

Traditionally, a classical approach to exploit a buffer overflow error to create an exploit generally consists of two steps: inject malicious code (*e.g.*, shell code) and utilize buffer overflow error to modify a function's return address to point to the injected code [28]. This approach relies on exploiting the standard layout of a virtual memory address space of a process on a modern computer system with an MMU, as shown in Figure 2.1.

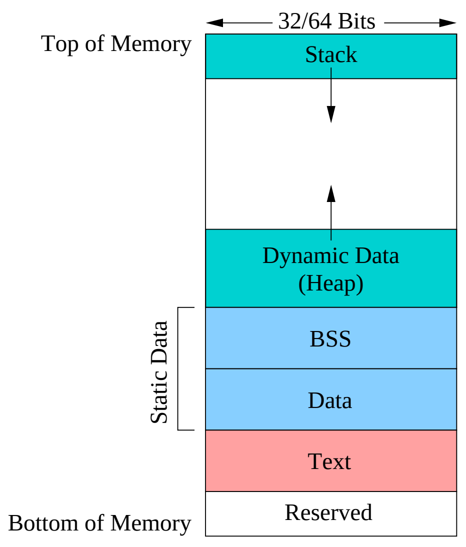


Figure 2.1: Virtual memory address space layout of a process on a modern computer system with an MMU.

As can be seen from Figure 2.1, the virtual address space layout of each process begins at a certain logical address, *e.g.*, 0, and exists in contiguous memory. At the lowest segment in the process address space is the text segment, which is a read-only map of the process's binary file into memory and contains all the process's code and other miscellaneous data such as string literals and the current value of the program counter.

Next is the data and BSS segments, both of which store contents of static and global variables. The difference is that the BSS segment stores the contents of uninitialized global and static variables, while the data segment stores the contents of global and static variables that have been initialized in source code.

The next segment is the heap, which is memory that is dynamically allocated during runtime. In C and C++, programmers are required to manually manage the memory allocated on the heap via API such as `malloc` and `free`, while in other languages, such as Python or Java, the memory on the heap is managed automatically by a garbage collector.

The topmost segment of the process address space is the stack, which contains temporary data, such as local (also known as automatic) variables and arguments that are passed to a function, as well as return addresses. Calling a function pushes a new stack frame onto the stack. When the function returns, the stack frame of that function is destroyed.

To better illustrate, consider the following code snippet from a simple program, in which a `main` function calls out to the `fib` function to calculate the Fibonacci number for $n = 40$:

```c
int fib(int input)
{
    int n = input;
    if (n < 2) {
        return n;
    } else {
        return (fib(n - 1) + fib(n - 2));
    }
}
int main(void)
{
    int fib_input = 40;

    return fib(fib_input);
}
```

Figure 2.2 shows the state of the stack after the function fib has been invoked by the function `main`.

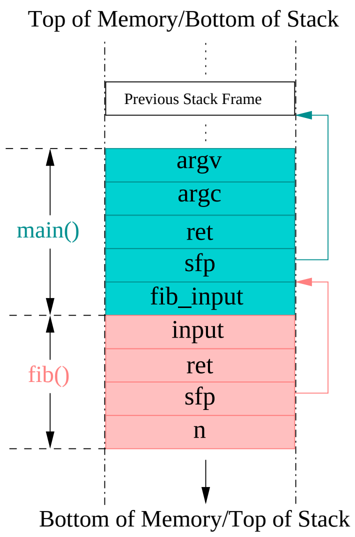


Figure 2.2: The stack frames of the `main` and fib functions.

As can be seen from the above figure, when a user run the program, the stack frame of the `main` function is pushed onto the stack. Then, when the `main` function calls out to the fib function, the stack frame of the fib function is the next one that get pushed onto the stack after the stack frame of the `main` function. Once the fib function finishes executing and returns, its stack frame is destroyed in a strict LIFO order as the control returns to the `main` function.

Armed with the knowledge of a process virtual address space and how stack frame works, we are now ready to understand the inner workings of the buffer overflow attack presented in [28]. Consider the following code, which is the `overflow1.c` example from [28] that we have modified slightly:

```
char shell code[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
        "\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
```

```
5           "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
6           "\x80\xe8\xdc\xff\xff\xff/bin/sh";
7   char large_string[128];
8   void main(void)
9   {
10    char buffer[96];
11    int i;
12    long *long_ptr = (long *) large_string;
13    for (i = 0; i < 32; i++)
14      *(long_ptr + i) = (int) buffer;
15    for (i = 0; i < strlen(shellcode); i++)
16      large_string[i] = shellcode[i];
17    /* the following is equivalent to strcpy(buffer,
       large_string) */
18    i = 0;
19    while (1) {
20      buffer[i] = large_string[i];
21      if (buffer[i] == '\0') {
22        break;
23      }
24      i++
25    }
26  }
```

Observe how the two for loops, one in the lines 12 and 13 and another in the line 14 and 15, are crafting the `large_string` array into a payload that will be used later to perform the attack. Specifically, the for loop on the lines 12 and 13 is filling the array `large_string` with the addresses of the array `buffer`, while another for loop on the lines 14 and 15 is filling the first 2/3 of the `large_string` array with the shellcode, leaving the last 1/3 with the addresses of the `buffer` array that were inserted earlier. Then, the `large_string` array is copied into the `buffer` array using code that is equivalent to the `strcpy`, which lacks bound checking. As Figures 2.3 shows, this will results in the contents inside the `large_string` array overflows onto the adjacent return address on the stack frame of the `main` function, overwriting it with the address of the `buffer` array.

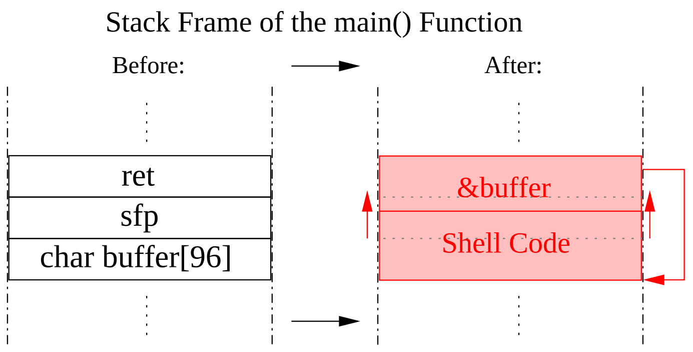


Figure 2.3: The stack frame of the `main` function: before and after the buffer overflow attack.

Then, once the `main` function returns, the control will jump to the address of the `buffer` array, and execute the shellcode that we inserted earlier to spawn a shell.

As can be seen, by using this technique, an attacker can execute shellcode, or any arbitrary code with the same privilege as those of the process that has buffer overflow errors. Also, while the examples of buffer overflow attacks described in [28] require the following two conditions to be satisfied: (1) injecting malicious code (in this case shellcode) and (2) redirecting the program control flow to execute that code via buffer overflow, it is worth pointing out that, as observed by Piromsopa and Enbody [32], the first condition, injecting code, is not necessary since the attacker can utilize existing code in the program or shared libraries to perform the attack.

This idea forms the basis of Return-Oriented Programming (ROP) [39], which is a technique that allows attacker to exploit buffer overflow and other memory errors to execute arbitrary malicious operations (since it has been shown that the ROP-based attacks are Turning-complete), without having to inject new code into the address space of the program by redirecting control flow to existing code in the program's address space.

In addition, the attacks that utilize ROP also render techniques that only protect against code injection, such as Non-Executable Data [6] ineffective. While probabilistic defense techniques such as Address Space Layout Randomization (ASLR), which randomize the locations of variables on both stack and heap, can be used to mitigate the problem, it has been shown that ASLR itself is vulnerable to information leakage attacks [11].

2.2.3 Buffer Over-Read

Another type of out-of-bound error that is closely related to buffer overflow is buffer over-read. As it name implies, buffer over-read occurs when a read operation goes out of the bound of a buffer and reads value of a memory address that is outside the bound. The following example demonstrates a simple buffer over-read error:

```
int secret = 42;
int buffer[5] = {6, 7, 8, 9, 10};
int i;
for (i = 0; i < 7; i++) {
    printf("%d\n", buffer[i]);
}
```

Here, the out-of-bound error in the `for` loop caused the index `i` to go out of the bound of the `buffer` array, causing the `print` function to erroneously print out the value of the variable `secret` that is adjacent to the `buffer` array.

2.2.4 Exploiting Buffer Over-Read

In general, buffer over-read errors are exploited to leak sensitive data, either by itself or as a prelude to other attacks. As the example above shown, it is infinitely easier to exploit buffer over-read, provided that the conditions are right. Making the matter worse is the fact that most of the modern out-of-bound prevention mechanisms in general do not protect against buffer over-read.

2.3 Memory Safety

Memory safety is a state in which all memory access is well defined, *i.e.*, there are no memory corruption errors that violate either spatial memory safety or temporal memory safety.

Spatial memory safety means that all memory accesses occur within the bounds of the object or buffer that is being access. Buffer overflow and buffer over-read errors are examples of spatial memory safety violations.

Temporal memory safety means ensuring that all pointers point to valid memory when they are dereferenced. Examples of temporal safety violations include the various use-after-free vulnerabilities.

Since temporal memory safety lies outside the scope of this dissertation, we will focus on enforcing spatial memory safety from this point forward.

2.4 Enforcing Spatial Memory Safety

We can prevent spatial memory safety violations, such as buffer overflow and buffer over-read exploits, by preventing out-of-bound write and read operations from occurring in the first place. One of the way to accomplish that is by performing bound checking before all read or write operations in a program. For example, consider the following trivial program:

```c
int main(void) {
    int i, foo[10];
    i = 10;
    foo[i] = 42; /* buffer overflow! */
    return 0;
}
```

In order to prevent buffer overflow in the above program, all we need to do is to insert bound checking code before the assignment operation (=), as the following listing illustrates:

```c
int main(void) {
    int i, foo[10];
    i = 10;
    /* perform bound checking */
    if (i < 0 || i >= 10) {
        /* buffer overflow detected, abort! */
        exit(-1);
    }
    foo[i] = 42; /* buffer overflow! */
    return 0;
}
```

This seemingly simple “solution” to the problem belies the complexity one faced when trying to design a bound checking technique that is robust enough to be practical. For example, while performing bound checking before all read and write operations is the surest way to eliminate out-of-bound errors, the performance overhead could potentially be prohibitive. The challenge, therefore, lies in how to design a bound checking technique in such a way as to maximize protection coverage while trying to minimize runtime overhead and preserve existing semantic as much as possible.

Clearly, this challenge highlights various design decisions that must be made when designing and implementing a purely software-based spatial memory safety enforcement mechanism:

- Which type of object should we track and check its bound?
- How and when do we retrieve the base and bound of an object?
- How and where to store the retrieved bases and bounds?

- How and where to instrument the bound checking code?

In the next chapter, we review related works, with spacial emphasis on the approaches that aim to enforce spatial memory safety during runtime.

CHAPTER III

RELATED WORK

In this chapter, we provide an overview of various approaches that have been proposed to enforce spatial memory safety.

Numerous countermeasure techniques against spatial memory safety violations have been developed over the years. Since it is impractical to cover all of them, we focus on the approaches that aim to enforce complete spatial memory safety, which in general means performing bound checking at every memory access. For a more comprehensive survey of various memory safety enforcement techniques, we defer the reader to the following survey papers [32, 40].

3.1 Runtime Spatial Memory Safety Enforcement Approaches

Many techniques enforce spatial memory safety by performing bound checking at runtime. In recent years, most of these runtime, dynamic approaches also employed static analysis techniques to help transform and instrument source code in order to improve detection rate and reduce runtime overhead.

3.1.1 Object-Based Approaches

Many early software-based solutions, such as [43, 18, 45], store the base and bound of each buffer inside a pointer that points to the said buffer, which almost always involve changes to how a pointer is represented (into what is called a fat pointer). While this approach has many benefits, such as low memory and performance overhead, its major downside is the incompatibility between the code that uses fat pointers that the code that doesn’t, which is a major concern because in practice it is not possible to recompile every program and libraries to use the same pointer representation as the instrumented code.

This incompatibility problem is the main motivation that gave rise to the approach that is typically called the object-based approach. In this approach base and bound of each buffer is stored in a separated data structure, sometimes called a disjointed meta-data, and the address of a buffer is used to look up its associated bounds information, which essentially means that we are tracking each buffer directly.

One of the earliest work in the object-based approach is Jones and Kelly [14], which stores base and bound of buffers in a splay tree, thus preserving memory layout of a process. Jones and Kelly also proposed a solution to the problem where, in C and C++, a pointer can go out of bounds of a buffer as long as they are not dereferenced, which can cause false positives if not handle properly. The solution that Jones and Kelly came up is to pad each buffer with an extra byte. The overhead of Jones and Kelly approach was reported to be around 11x and 12x.

CRED [35] improves upon Jones and Kelly's work [14] by reducing the overhead to around 2x, but by performing bound checking only on character arrays.

Dhurjati et al. [10] further extend the works of Jones and Kelly [14] and Ruwase and Lam [35] by proposing to partition buffers into pools during compile time using a technique called Automatic Pool Allocation [20] and use splay tree to track base and bound in each pool instead of using one large splay tree for the entire process address space like in the previous approaches. This technique help reduce overhead down to around 120%.

One of the notable work in the early object-based approach is the work by Akritidis et al. [2], which stores allocation bounds instead of precise bounds of each buffer. These allocation bounds are usually larger than the actual bounds of a buffer, while their size and alignment are constrained to facilitate efficient bounds lookups, which results in significant lower overhead during runtime of around 60%.

One recent notable work is Tag-Protector [36]. The key idea of this work is that since the majority of overhead in both the object-based and pointer-based approaches came from the looking up metadata from a table, this work proposed a technique to completely eliminate this overhead during runtime, by retrieving base and bound of buffers during compile time and them insert them as tags before and after their respective buffer. Then the bound checking code is inserted, which will lookup base and bound of each buffer from the buffer's tags. Thanks to the clever placement of the tags, this approach also preserve memory layout and compatibility with other libraries, while greatly reduced runtime overhead. One of the drawback of this approach is that, in some rare circumstances, the tags might get overwritten or modified. As can be seen, the local bound checking optimization of BoundWarden is inspired by Tag-Protector. However, instead of storing base and bound inside the tags, BoundWarden stored them in the bound table, thus side-stepping the issue of the tags being modified.

Since BoundWarden also associate base and bound to each buffer by the buffer own address, it can be considered as an object-based approach, but unlike many earlier object-based approaches, BoundWarden is capable of detecting and preventing out-of-bound errors in buffers inside structures or unions, thanks to how BoundWarden leverage how Clang translates C source code into LLVM IR.

3.1.2 Pointer-Based Approaches

However, despite solving the incompatibility issue, the early works in the object-based approach has its own set of issues, one of which being the fact that since they associate base and bound to each buffer using the address of the buffer, they cannot differentiate between a structure and its first element, which makes it impossible to detect buffer overflows in nested composite types, for example a buffer inside a struct or a union [22]. In contrast, pointer-based approaches, where base and bound of a buffer is associate with a pointer that points to the buffer, do not have this restriction.

CCured [25] uses a sophisticated whole-program static analysis technique to identify pointers that do not require bound checking, thus significantly reduce runtime overhead. However, CCured introduces incompatibility by using fat pointers to store base and bound with each pointer.

In contrast, Deputy [47] avoid having to perform type inference on the whole program by introducing dependent type that allows programmers to add annotations that specify relationship between existing data elements, which allows the compiler to perform bound checking without having to modify process memory layout.

In order to solve this problem, [23, 22] proposes a new scheme, in which the base and bounds information are again associated with a pointer to a buffer, like the previous fat pointer approaches. But unlike the fat pointer approaches, [23, 22] stores the base and bound information of a pointer inside a separated data structured and use the address of the pointer itself (and not the buffer) to lookup the bound information, essentially combined the best features of the fat pointer approaches and the object-based approaches, namely compatibility between instrumented and uninstrumented code, while also being able to detect overflow in nested composite types.

3.2 Hardware-Based Approaches

One of the major drawback of software-based solution is high performance overhead. The seemingly obvious solution to this problem is to offload some or all of the work to hardware (or to use hardware to accelerate bound checking operation). This is the general idea behind many hardware-based approaches described in this section.

In general, hardware-based approaches, such as [7, 5, 30, 17, 8, 4, 30, 31], have lower runtime and memory overhead than their software-based counterparts. However, one of the major disadvantage of the majority of hardware-based approaches is that they are difficult to deploy, usually require new hardware, new revision of existing hardware, or even custom hardware. Not to mention that some hardware approaches still suffered from high runtime and memory overhead, bugs, and design limitations that can only be fixed by replacing existing hardware with new ones [27]. In contrast, software-based approaches are more flexible and can be deployed much more easily.

Nevertheless, both approaches complement each other, as in the case of the infamous Spectre [15] and Meltdown [21] attacks, where the software-based defenses were being rolled out globally almost immediately after the attacks became public, despite the fact that these defenses have considerable runtime overhead, while the more efficient hardware solutions are being developed.

3.3 Concurrent Monitor Approaches

In recent years, with rise of multi-core processors, a new approach emerged: software cruising, a technique that uses concurrent threads to perform bound checking. The first work that utilizes this technique is Cruiser [46], which uses a monitor thread is used to monitor the integrity of buffers on heap in userspace. In addition to the idea of using thread, Cruiser also utilized a novel, custom lock-free data structure and algorithm to send metadata to the monitor list, which is used by the monitor thread to enforce the integrity of heap buffers. Another work in this approach is Kruiser [42], which extends the concept to protecting heap integrity in kernel space.

BoundWarden could also be considered as another work that utilizes software cruising technique. Indeed, at first glance, the goal and design of BoundWarden very closely resemble to those of Cruiser. However, the major difference between BoundWarden and Cruiser lies in how each approach uses the thread. The monitor thread of Cruiser works completely independent of the threads of a program, capable of checking the integrity of any buffers in the program, whether they are the ones that are being currently accessed or not. This, however, creates a latency problem, where an out-of-bound error might occurred in one buffer, while the monitor thread was working on another. In contrast, the bound checking thread of BoundWarden closely follows the execution of the threads of the program, performing bound checking on buffers that are being accessed by the running threads. This means that, while BoundWarden also suffered from the latency problem, the severity is much less than the one that Cruiser's encountered. Another different between BoundWarden and Cruiser is that BoundWarden protect the integrity of buffers in stack, heap, and BSS and data segments, while Cruiser only protect the integrity of buffers in heap.

Another technique that exploits the ubiquity of multi-core processors is Multi-

Variant Environments Execution [37, 38]. The key idea of this approach is that several slightly different variants of a single program is executed simultaneously, while the MVEE monitor feeds the variants the same inputs and monitor for any discrepancy. Once the monitor detects any divergences in behavior, it will halt the execution of the offending variant. However, this approach is not suitable for multi-threaded programs that utilizing multiple CPU cores during runtime.

CHAPTER IV

DESIGN AND IMPLEMENTATION

In this chapter, we describe the design and implementation of BoundWarden. We start with a high level overview of the design of BoundWarden in Section 4.1, then describe each component of BoundWarden in detail, with Section 4.2 describes the compiler extension component, and Section 4.3 contains the in-depth description of the runtime component.

4.1 Design Overview

BoundWarden consists of two components: a compiler extension and a runtime component, as seen in Figure 4.1. BounWarden pass, the compiler extension, is responsible for instrumenting the source code and inserting code that initialize the runtime component, which include spawning the bound checking thread, and code that send metadata needed to perform bound checking to the bound checking thread. BoundWarden runtime, the runtime component is responsible for using the information sent by the instrumented code to enforce spatial memory safety, as well as managing the runtime metadata.

4.2 Compiler Extension

The compiler extension of BoundWarden transforms source code of C programs and insert code that perform the following tasks: retrieve base and bound of each buffer and send them to the bound checking thread; invoke bound checking thread to perform bound checking by sending the base of a buffer and an offset to the bound checking thread; and initialize the runtime component.

We implement BoundWarden compiler extension to the Clang/LLVM compiler [19] as an LLVM pass, using LLVM C++ API. This means that instead of having to transform C source code via a source-to-source transformation, a daunting task even with modern

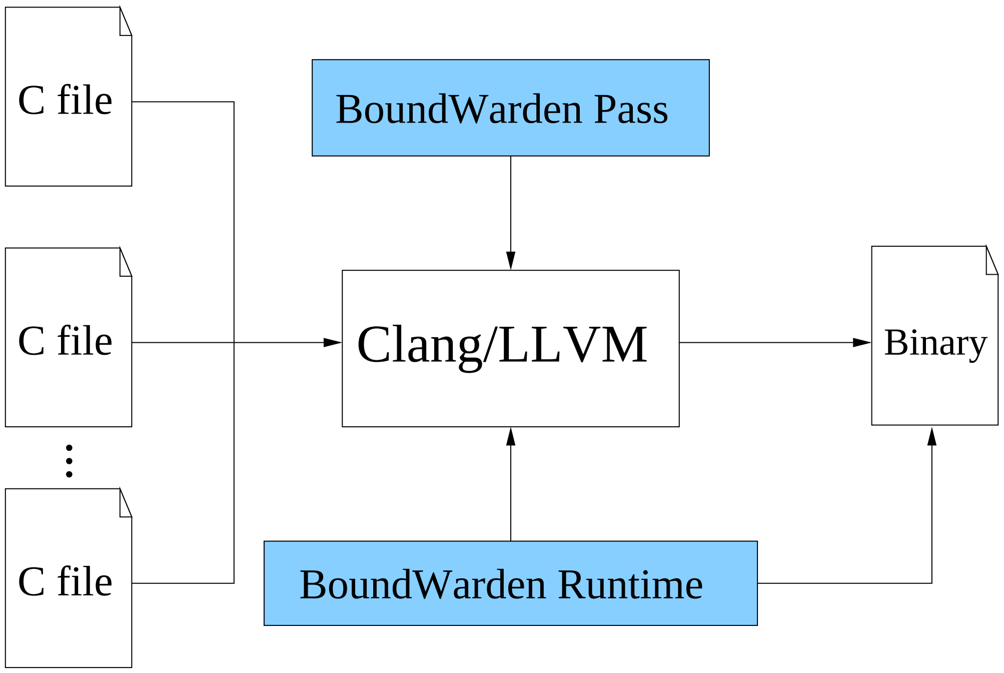


Figure 4.1: Block diagram showing a high-level design overview of BoundWarden components: BoundWarden Pass, a compiler extension implemented as an LLVM pass, and BoundWarden Runtime, which is implemented as a static library and linked to the binary.

tools due to the sheer complexity of the syntax and semantic of the C programming language, we will be performing a transformation on the LLVM Intermediate Representation (IR) that is created from the original C source code.

BoundWarden follows the objected-based approach practices of using each buffer own address in the memory, *i.e.,* the address returned by C address-of operator (`&`), to associate the buffer with its base and bound and store the metadata inside a separated data structure. The major advantage of this approach is that no changes to the memory layout are required, thus preserving compatibility with external libraries and binaries.

Next, we describe the transformations that BoundWarden pass performed to insert code to capture base and bound of buffers that are allocated on the stack, heap, and data and BSS segments in Section 4.2.1, 4.2.2, and 4.2.3, respectively. Section 4.2.4 describes how BoundWarden pass inserts code that invoke the runtime component to perform bound checking as well as manage metadata, while Section 4.2.5 describes the technique we used to detect out-of-bound errors in composite types. In Section 4.2.6, we describe how BoundWarden pass initializes the runtime component.

4.2.1 Retrieving Base and Bound of Buffers on Stack

For buffers that are allocated on the stack region of memory, both its base and bound can be obtained from the LLVM IR `alloca` instruction, which is used to allocate memory on the stack frame of the function currently executing. For example, consider the following C code snippet:

```
int foo[5];
```

Which Clang will transform into the following LLVM IR on 64-bit machines:

```
%foo = alloca [5 x i32], align 16
```

From the above code snippets, we can see that the declaration of the array `foo` is transformed into an `alloca` instruction, with the size of the array and a constant alignment as the arguments. The `alloca` instruction returns a pointer to the allocated memory, which is stored in the `%foo` local identifier.

In order to record the base address of the array `foo` at runtime, BoundWarden pass inserts a `call` instruction after the `alloca` instruction that calls out to a C function named `__bw_send` with the `%foo` identifier, which stores the address of the array `foo`, and the constant integer value of 20, which is the size in bytes of the array, as the arguments, as can be seen in the following code snippet:

```
%foo = alloca [5 x i32], align 16
call void @__bw_send(i8* %foo, i64 20)
```

The function `__bw_send`'s job is to send base and bound of buffers to the runtime component via the ring buffer. After receiving the base and bound, the runtime component will store the received metadata in a data structure for later use. The function `__bw_send` has the following definition:

```c
void __bw_send(uintptr_t base, uintptr_t offset)
{
  ring_buffer[ring_buffer_head].base = base;
  ring_buffer[ring_buffer_head].offset = offset;
  ring_buffer[ring_buffer_head].flag = 0;
  ring_buffer_head = (ring_buffer_head + 1) % QUEUE_SIZE;
}
```

As can be seen from the definition of `__bw_send`, the ring buffer is implemented using an array of structures. We describe the structure of the ring buffer that acts as the main communication channel between program threads and the runtime component in detail in Section 4.3.2.

Algorithm 1, shown below, describes the process in which BoundWarden pass instruments C source code to insert a call to the function `__bw_send` with appropriate arguments after each `alloca` instruction in a program in order to record base and bound of every buffer on the stack.

Algorithm 1 Retrieving the base and bound of buffers on stack

```
for all $Instruction$ do
    if $Instruction$ has the type AllocaInst then
        $AI \leftarrow Instruction$
        $Base \leftarrow AI$
        $Bound \leftarrow AI.getAllocSize()$
        $CallInstArgs \leftarrow [$ __bw_send, $Base$, $Bound$ $]$
        $AI.insertInstAfter($CallInst, $CallInstArgs)$
    end if
end for
```

4.2.2 Retrieving Base and Bound of Buffers on Heap

For buffers that are allocated on the heap via `malloc` or its cousins (*e.g.,* `calloc`), we use similar technique to track and record their base and bound, but instead of inserting code after each `alloca` instruction, we instead match all the `store` instruction, then check whether its first argument is an identifier that points to a `call` instruction.

If it is, then BoundWarden pass will follow the indirection and check whether the `call` instruction calls out to the `malloc` function.

If the `call` instruction calls the `malloc` function, BoundWarden pass will insert code that calls the `__bw_send` function with the identifier that stores the address returned by the `malloc` function call and the argument to the `malloc` function, which is the size of the buffer on the heap, as the arguments. This process is described in Algorithm 2.

Algorithm 2 Retrieving the base and bound of buffers on heap

```
for all $Instruction$ do
    if $Instruction$ has the type StoreInst then
        $Opd0 \leftarrow Instruction.getOperand(0)$
        if $Opd0$ has the type CallInst then
            $CI \leftarrow Opd0$
            $CalledFn \leftarrow CI.getCalledFunction()$
            if $CalledFn.getName()$ is "malloc" then
                $Base \leftarrow CI$
                $Bound \leftarrow CI.getArg(0)$
                $CallInstArgs \leftarrow [$ __bw_send, $Base, Bound ]$
                $CI.insertInstAfter($CallInst, $CallInstArgs)$
            end if
        end if
    end if
end for
```

As an example, consider the following code snippet in C that allocates an array `bar` in the heap using the `malloc` function:

```c
double *bar = malloc(sizeof(double) * 7);
```

Which will be transformed into the following equivalent IR by the compiler:

```
%bar = alloca double*, align 8
%8 = call noalias i8* @malloc(i64 56) #3
%9 = bitcast i8* %8 to double*
store double* %9, double** %bar, align 8
```

Following the process described in Algorithm 2, BoundWarden pass will insert a call to function __bw_send with appropriate arguments after the call to malloc in line 2 to send the base and bound of the bar array to the runtime:

```
1 %bar = alloca double*, align 8
2 %8 = call noalias i8* @malloc(i64 56) #3
3 call void @__bw_send(i8* %8, i64 56)
4 %9 = bitcast i8* %8 to double*
5 store double* %9, double** %bar, align 8
```

As can be seen from the above code snippet, a call to the function __bw_send is inserted after the call to function malloc with the %8 identifier—which contains the address of the buffer that is being allocated—and the argument to the function malloc—which is the size of the buffer—as the arguments.

4.2.3 Retrieving Base and Bound of Buffers on BSS and Data Segments

For buffers that are allocated on the BSS and data segments, *i.e.,* global variables and static variables, we need a different approach to retrieve their base and bound. Because unlike buffers that are allocated on stack and heap, buffers that are allocated on the BSS and data segments are stored in an LLVM IR module's list of global variables, which can be access via the getGlobalList method of the class Module.

Therefore, we iterate over each variable in the global list, and insert a call to the __bw_send function with appropriate arguments at the beginning of the program's main function to record base and bound of each variable, as described in Algorithm 3.

4.2.4 Invoking Bound Checking Thread to Perform Bound Checking

In order to insert code that invoke the bound checking thread in the runtime component to perform bound checking into the program's source code, we leverage the design of the LLVM IR. Specifically the way address computations when indexing into aggregate data structures, which in the context of LLVM means either an array, a struct,

Algorithm 3 Retrieving the base and bound of buffers on BSS and data segments

```
$MFirstBlock \leftarrow MainFunction.getFirstBlock()$
for all $GlobalVariable$ in $GlobalList$ do
    if $GlobalVariable$ is not a constant and has initializer then
        $GV \leftarrow GlobalVariable$
        $GvName \leftarrow GV.getName()$
        $Base \leftarrow Module.getNamedValue(GvName)$
        $Bound \leftarrow GV.getAllocSize()$
        $CallInstArgs \leftarrow [$ __bw_send, $Base,$ $Bound$ $]$
        $MFirstBlock.insertInstAfter($CallInst, $CallInstArgs)$
    end if
end for
```

or a union, are performed. That is, in LLVM IR, all address computation are performed via the `GetElementPtr` (`GEP`) operator. Consider the following C code:

```c
int baz[10];
baz[7] = 42;
```

Which will be transformed into the following LLVM IR:

```
%baz = alloca [10 x i32], align 16
store i32 0, i32* %1, align 4
%2 = getelementptr inbounds [10 x i32], [10 x i32]* %baz
    , i64 0, i64 7
store i32 42, i32* %2, align 4
```

Notice how the operation that assigns a number 42 to the 8th position in the `baz` array using 7 as an index is compiled into two distinct LLVM IR instructions: a `GEP` operator that, when given an identifier of the `baz` array and the index value, calculate the memory offset needed for indexing into the given buffer; and a `store` instruction that actually stores the number 42 into the memory address returned by the `GEP` operation. Also, it is worth emphasizing that no memory access occurs when the `GEP` operator is performing index calculation.

Algorithm 4 Performing bound checking before reading or writing buffers

```
for all $Instruction$ do
    if $Instruction$ has the type GEPOperator then
        $GEPOptr \leftarrow Instruction$
        $Base \leftarrow GEPOptr.getOperand(0)$
        $Bound \leftarrow GEPOptr$
        $CallInstArgs \leftarrow [$ __bw_check, $Base, \ Bound \ ]$
        $GEPOptr.insertInstAfter($CallInst, $CallInstArgs)$
    end if
end for
```

These properties of the GEP operator makes it an ideal candidate for us to insert code to invoke the runtime component to perform bound checking, which can be done using a technique similar to the ones we used to insert the code to capture base and bound of stack and heap buffers.

More specifically, as describes in Algorithm 4, for each and every GEP operator in an LLVM IR module, we extract the base address of a buffer that is being indexed into from the first argument of the GEP operator and the associated offset from the identifier that stores the result of the GEP operator, and insert a call to the function __bw_check, which invoke the runtime component to perform bound checking, with the base of the buffer and the offset as the arguments, as shown in the following example in which we instrumented the earlier C code snipper:

```
%baz = alloca [10 x i32], align 16
store i32 0, i32* %1, align 4
%2 = getelementptr inbounds [10 x i32], [10 x i32]* %baz
    , i64 0, i64 7
call void @__bw_check(i8* %baz, i8* %2)
store i32 42, i32* %2, align 4
```

Note how the call instruction that calls out to the C function __bw_check is inserted after the GEP operator, and that we use the identifier %2 that stores the memory address computed by the GEP operator as the offset value when performing bound checking. Also, since the GEP operator is used to calculate memory address before both the write operation (LLVM IR `store` instruction) and the read operation (LLVM IR `load` instruction) are performed, all we need to do to ensure that we perform bound checking before any memory access operations occur is to match and insert a call to the function `__bw_check` after all instance of the `GEP` operator in every translation unit of a program.

The function `__bw_check` has the following definition:

```
void __bw_check(uintptr_t base, uintptr_t offset)
{
  ring_buffer[ring_buffer_head].base = base;
  ring_buffer[ring_buffer_head].offset = offset;
  ring_buffer[ring_buffer_head].flag = 1;
  ring_buffer_head = (ring_buffer_head + 1) % QUEUE_SIZE;
}
```

Its job is to send a base address of a buffer and an offset to the runtime component to be used to perform bound checking. One of the major difference between the function `__bw_check` and function `__bw_send` is the value of the `flag` variable. This variable controls how the runtime component will interpret the data that it received, with a value of 0 instructs the runtime component to interpret the data inside the structure as base and bound of a buffer and to store them inside a data structure. On the other hand, a value of 1 means that the structure contains a base address of a buffer and an offset for indexing into the buffer, which tells the runtime to perform bound checking using the base and the offset it received.

4.2.5 Detecting and Preventing Spatial Memory Violations of Buffers Inside Structs and Unions

It is well-known that one of the drawback of the object-based approaches is that they cannot protect against out-of-bound errors in buffers that are nested inside either a structure or a union.

Consider the following code snippet:

```
struct foo {
  int a;
  int b;
  int c;
};
```

Suppose that the structure `foo` is located on an address `0x7ffc03715d90` on a 64-bit machine, then it is safe to assume that the variable `a`, `b`, and `c` inside the structure `foo` will have the following memory addresses: `0x7ffc03715d90`, `0x7ffc03715d94`, and `0x7ffc03715d98`, respectively. And here lies the problem: the memory address of the structure and its first element is the same (in this case `0x7ffc03715d90`). This causes a major problem for most object-based approach, because since object-based approach associated metadata to a buffer via the memory address of the buffer, this means that the addresses of the structure and its first element are indistinguishable from one another and can only be used once, either to map the structure's bound to the structure or the first element's bound to the first element inside the structure. This fact also holds true for unions as well.

However, it is possible to overcome this problem by leveraging how LLVM transforms the code that involving accessing elements inside either a structure or a union. Consider the following code snippet:

```
struct foo {
  int a;
  int b[7];
  int c;
};
struct foo s_foo;
s_foo.a = 1;
s_foo.b[3] = 42;
s_foo.c = 3;
```

Which transforms into the following IR:

```
%s_foo = alloca %struct.foo, align 4
store i32 0, i32* %1, align 4
%2 = getelementptr inbounds %struct.foo, %struct.foo* %
    s_foo, i32 0, i32 0
store i32 1, i32* %2, align 4
%3 = getelementptr inbounds %struct.foo, %struct.foo* %
    s_foo, i32 0, i32 1
%4 = getelementptr inbounds [7 x i32], [7 x i32]* %3,
    i64 0, i64 3
store i32 42, i32* %4, align 4
%5 = getelementptr inbounds %struct.foo, %struct.foo* %
    s_foo, i32 0, i32 2
store i32 3, i32* %5, align 4
```

Notice how when trying to generate the IR for computing the offset for indexing into the array `b` inside the structure `foo`, Clang/LLVM generates two `GEP` operators to perform offset calculation: the first `GEP` operator calculates the offset into the structure, while the second `GEP` operator calculates the offset into the array `b`.

By leveraging this design decision of LLVM, it is possible to add a support for sub-object bound checking to BoundWarden. Specifically, we first modify Algorithm 1, 2, and 3 to check whether the type of the buffer being allocated is either a struct or a union. If it is, then the algorithms will now iterate over each element of a struct or a union and inserting code that calls a modified version of the function `__bw_send` in that will capture and send base and bound of each member of a structure or a union to the runtime component. Next, we extend Algorithm 4 so that it can differentiate between a `GEP` operator that calculate an offset into structures or unions and a `GEP` operator that calculate an offset into buffers inside, so that we can tell BoundWarden pass to instrument only the `GEP` operator that calculate an offset into the nested buffers with a call to the function `__bw_check`.

4.2.6 Initializing the Runtime Component

BoundWarden pass is also responsible for inserting auxiliary code needed by the functions `__bw_send` and `__bw_check`, and the runtime component, as well as initializing various parts of the runtime, which consists of a bound table for storing base and bound of buffers in a program, a ring buffer that acts as a communication channel between program's threads and the runtime, and a bound checking thread, which is responsible for performing bound checking and managing metadata. Specifically, the pass insert declarations of the functions `__bw_send` and `__bw_check`, as well as declarations of variables needed by the instrumented code and the runtime system to function, such as the bound table and the ring buffer. After BoundWarden pass finished inserting the declarations into the program's modules, it will then insert a `call` instruction at the beginning of the program's `main` function that calls out to the function `__bw_init` that initializes the components of BoundWarden's runtime. The following is the definition of the function `__bw_init`:

```
void __bw_init(void)
{
  int status;
  init_bgdp(&bgdp);
  if (bgdp == NULL) {
    /* failed to allocate bound table */
    abort();
  }
  ring_buffer = malloc(sizeof(struct queue_elt) *
   QUEUE_SIZE);
  if (ring_buffer == NULL) {
    /* failed to allocate ring buffer */
```

```
        abort();
    }
    status = pthread_create(&bw_tid, NULL, bw_monitor,
     NULL);
    if (status != 0) {
        /* failed to allocate bound checking thread */
        abort();
    }
}
```

Therefore, when we execute the instrumented binary that has been transformed by BoundWarden pass, the following sequence will happen immediately after the system calls the `main` function:

1. The bound table is allocated;
2. The ring buffer is allocated;
3. The bound checking thread is spawn.

Similarly, BoundWarden will also insert a `call` instruction at the end of the `main` function that calls out to the function `__bw_cleanup` which is responsible for cleaning up and gracefully wind down the runtime component. The definition of the function `__bw_cleanup` is given below:

```
void __bw_cleanup(void)
{
    pthread_cancel(bw_tid);
    free(ring_buffer);
    free(bgdp);
}
```

Thus, when the program finishes executing, the `__bw_cleanup` will send a cancellation request to the bound checking thread, and freeing up all the memory that are used by the bound table and the ring buffer.

4.3 Runtime

The runtime component of BoundWarden is implemented as a single self-contained C static library that is linked to the instrumented source code during compilation. This makes it trivial to modify or swap one implementation of the runtime for another. The runtime component has three parts: the bound table, the ring buffer, and the bound checking thread. Figure 4.2 shows the architecture of BoundWarden's runtime component.

The bound checking thread is responsible for performing bound checking and managing the bound table. It performs actions according to the message it received from the program's threads via the ring buffer, such as inserting base and bound of a newly allocated buffer into the bound table, or using the base of a buffer and the offset to perform bound checking by comparing the offset with the bound of the buffer that is stored in the bound table. If the bound checking thread detects an out-of-bound violation, it will dump debug information and terminate the running process. We describe the bound checking thread in detail in Section 4.3.3.

The bound table is where metadata of all buffer in a program is stored. It is a simple key-value table, where the value is the bound of a buffer and the base address of the buffer itself is used as a key to index into the table. More details on the bound table can be found in Section 4.3.1.

The ring buffer, explained in-depth in Section 4.3.2, acts as the main communication channel between the bound checking thread and the program's threads.

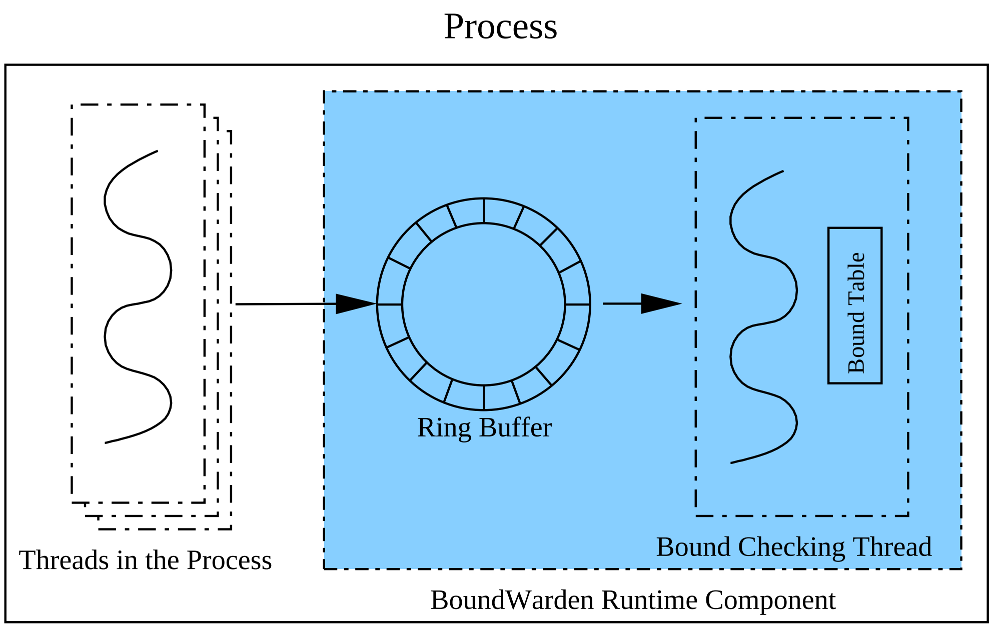


Figure 4.2: The architecture of BoundWarden runtime component, which consists of three parts: the ring buffer, the bound checking thread, and the bound table.

4.3.1 Bound Table

The bound table is a simple table that maps each buffer to its base and bound via the address of the buffer. It is used to store base and bound of all buffer in a running process. The design of our bound table is based on the design of the page table used by modern 64-bit architectures, such as AMD64 and Intel 64. That is, our bound table is also hierarchical, where each table contains a pointer that points to the next lower table in the hierarchy and the (virtual) address/base of each buffer are partitioned/translated into offsets that are used to index into the table.

We define a table at the lowest level of the hierarchy, called the bound table entries (`bte`) table, as an array of `uintptr_ts` that can be used to store the address of the bound of a buffer:

```c
typedef uintptr_t *bte_t;
```

The `bte` table is allocated via the following code snippet:

```c
bte_t = calloc(BTE_SIZE, sizeof(uintptr_t));
```

The bound table itself is pointed to by a pointer that points to the `bdg_t` type, and is allocated and initialized via the `init_bgdp` function.

Currently, we have three implementations of the bound table, each differentiated by the number of levels in the hierarchy: 5-Level, 4-Level, and 3-Level.

The 5-Level (5-L) bound table, shown in Figure 4.3a, has five levels (from highest to lowest): 256-entry Bound Global Directory, 256-entry Bound Upper Directory, 256-entry Bound Middle Directory, 256-entry Bound Lower Directory, and 65,536-entry Bound Table Entries.

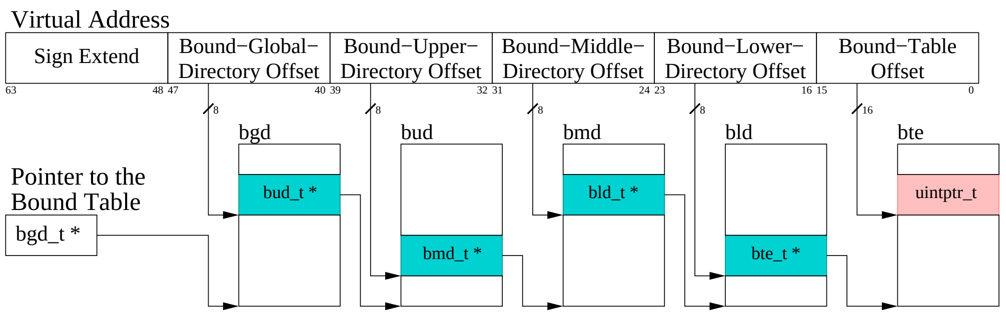
In this configuration, the base address of a buffer is divided into five fields and are used to index into the bound table according to the following scheme:

- Bits 63–48 are a signed extension of bit 47, and are discarded.
- Bits 47–40 index into the bound global directory table.
- Bits 39–32 index into the bound upper directory table.
- Bits 31–24 index into the bound middle directory table.
- Bits 23–16 index into the bound lower directory table.
- Bits 15–0 index into the bound table entries table.

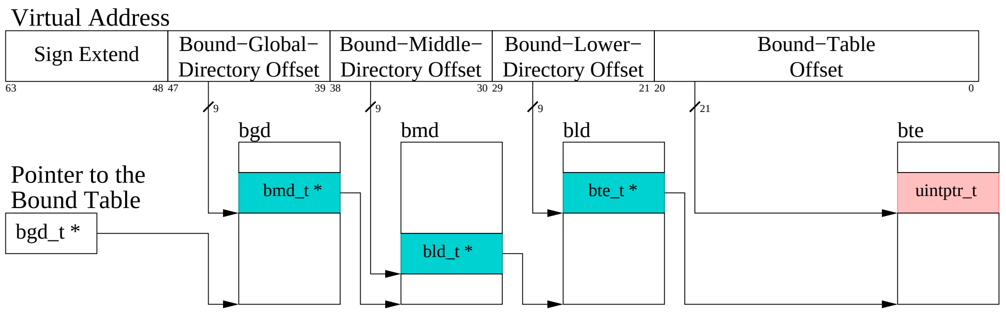
The 4-Level (4-L) bound table, shown in Figure 4.3b, has four levels. These are (from highest to lowest): 512-entry Bound Global, 512-entry Bound Middle Directory, 512-entry Bound Lower Directory, and 2,097,152-entry Bound Table Entries.

Similar to the 5-L bound table, the base address of a buffer is divided into four fields and are used to index into the bound table based on the following scheme:

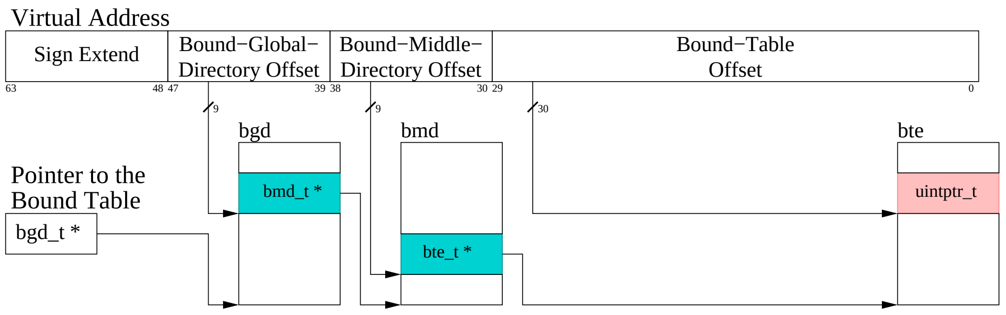
- Bits 63–48 are a signed extension of bit 47, and are discarded.


Virtual Address
Sign Extend
Bound−Global−Directory Offset
Bound−Upper−Directory Offset
Bound−Middle−Directory Offset
Bound−Lower−Directory Offset
Bound−Table Offset
63
48 47
40 39
32 31
24 23
16 15
0
8
8
8
8
16
bgd
bud
bmd
bld
bte
Pointer to the Bound Table
bgd_t *
bud_t *
bmd_t *
bld_t *
bte_t *
uintptr_t

(a) 5-L Bound Table



(b) 4-L Bound Table



(c) 3-L Bound Table

Figure 4.3: Block diagrams showing the three configurations of the bound table: 5-Level (5-L), 4-Level (4-L), and 3-Level (3-L).

- Bits 47–39 index into the bound global directory table.
- Bits 38–30 index into the bound middle directory table.
- Bits 29–21 index into the bound lower directory table.
- Bits 20–0 index into the bound table entries table.

Figure 4.3c shows the 3-Level (3-L) bound table, which has three levels (from highest to lowest): 512-entry Bound Global, 512-entry Bound Middle Directory, and 1,073,741,824-entry Bound Table Entries.

In this setup, the base address of a buffer is divided into three fields, which are then used to index into the bound table according to the scheme shown below:

- Bits 63–48 are a signed extension of bit 47, and are discarded.
- Bits 47–39 index into the bound global directory table.
- Bits 38–30 index into the bound middle directory table.
- Bits 29–0 index into the bound table entries table.

Each implementation of the bound table is compiled into a static library, which is later statically linked to other runtime components to produce the static runtime library. The interface to the bound table is implemented using void pointers to simulate generic functions in C, therefore switching between each implementation of the bound table is as simple as recompiling the runtime library with a new bound table library.

4.3.2 Ring Buffer

The ring buffer serves as the main communication channel between running process's threads and the bound checking thread. The ring buffer in our system is implemented as an array of the `queue_elt` structures, along with two variables that act as indexes into the array:

```c
struct queue_elt *ring_buffer;
unsigned long ring_buffer_head = 0;
unsigned long ring_buffer_tail = 0;
```

The `ring_buffer_head` variable is used as an index into the ring buffer by the program's threads, and is incremented only by the functions `__bw_send` and `__bw_check` in order to write and send new data to the bound checking thread. The `ring_buffer_tail` variable, on the other hand, is incremented only by the bound checking thread in order to read data that were sent by the program's threads. This access scheme ensures that the bound checking thread works in tandem with the program's running threads, performing bound checking on buffers that are being accessed, thus minimizing the "lag" time between when a pointer goes out-of-bound and when the bound checking thread detects and aborts the offending program. The trade-off of this approach is that it is possible for the ring buffer to be starved of data, causing the bound checking thread to become idle when/if the program threads slow down or stall for whatever reasons.

It is also possible to configure how closely the bound checking thread "follows" the executions of the program's threads. That is, we can configure how long the bound checking thread waits for new data in the ring buffer. This setting also affects the overall performance of BoundWarden. From preliminary testing, the longer that the bound checking thread spent waiting for new data, the better the performance, despite the fact that we pinned the bound checking thread on a separated, dedicated CPU core when performing the tests. However, the downside of setting the bound checking thread's sleeping period too high is that the lag time between the occurrence of spatial memory safety violations and the time that the bound checking thread detects and aborts them increase significantly. In the worse case from one of our tests, a buffer overflow bug caused the entire program to segmentation faulted before the bound checking thread even had the chance to perform bound checking. This essentially force us to choose between performance or security.

The default setting we ended up choosing for the rest of this work was to have the bound checking thread waits as little as possible. That is, we choose to trade performance for more security guarantee, in order to ensure that the bound checking thread is able to detect and prevent spatial safety violations as fast as possible.

The ring buffer is allocated when the `main` function of a program is called to a predetermined size, as shown in the following code snippet from the function `__bw_init`:

```
ring_buffer = malloc(sizeof(struct queue_elt) *
    QUEUE_SIZE);
```

The queue_elt structure itself has three elements and is defined as follows:

```
struct queue_elt {
  uintptr_t base;
  uintptr_t offset;
  uint_fast8_t flag;
};
```

The first element of the queue_elt structure is a base address of a buffer, while the second element can be either a bound (size) of the buffer or an offset for indexing into the said buffer. The third element of the queue_elt structure is a flag that is used by the bound checking thread to determine how to interpret the contents of the queue_elt structure. Its value can be either 0, or 1. We describe how the bound checking thread will interpret the meaning of this flag in Section 4.3.3.

4.3.3 Bound Checking Thread

The bound checking thread is responsible for managing metadata in the bound table and performing bound checking. It is implemented using the standard POSIX thread API. The threads in a running process interact with the bound checking thread through the ring buffer, either by sending base and bound of a buffer to be inserted into the bound table or base and offset of an existing buffer to be checked for out-of-bound error. In essence, the "messages", *i.e.*, the queue_elt structure, specifically the flag, in the ring buffer is what control the action and state of the bound checking thread.

The following code snippet shows the main loop of the bound checking thread:

```
struct timespec ts;
ts.tv_sec = 0;
ts.tv_nsec = 1;
while (true) {
  while (ring_buffer_head == ring_buffer_tail) {
    nanosleep(&ts, NULL);
  }
  while (ring_buffer_head > ring_buffer_tail) {
    flag = ring_buffer[ring_buffer_tail].flag;
    if (flag == 0) {
      base = ring_buffer[ring_buffer_tail].base;
      offset = ring_buffer[ring_buffer_tail].offset;
      bound = base + offset + PADDING_BYTES;
      btable_update(base, bound, bgdp);
    }
    if (flag == 1) {
      base = ring_buffer[ring_buffer_tail].base;
      bound = btable_find(base, bgdp);
      if (bound != 0) {
        offset = ring_buffer[ring_buffer_tail].offset;
        if (OOB_CHECK(base, bound, offset)) {
          abort();
        }
      }
    }
    ring_buffer_tail = (ring_buffer_tail + 1) % QUEUE_SIZE
   ;
  }
}
```

The bound checking thread retrieves a message from the ring buffer by using the variable `ring_buffer_tail` to index into the ring buffer. As previously described in Section 4.3.2, the interaction between `ring_buffer_tail` and `ring_buffer_head` also indirectly controls the bound checking thread, particularly on how often the bound checking thread is wake up to process messages in the ring buffer. That is, the longer we allow the bound checking thread to sleep to wait for new messages in the ring buffer, the faster the performance will be, and *vice versa*. On the other hand, setting the sleeping time too high could potentially introduce lag between the time a spatial memory violations occur and the time that the bound checking thread find them, in essence sacrificing security for performance. As previously mentioned, for the rest of this work, we set choose security over performance and set the sleeping time of the bound checking thread to be as little as possible.

There are three states that the bound checking thread can be in at any given moment: The first is a sleeping state, where the bound checking thread sleeps and waits for new messages in the ring buffer. How long the bound checking thread will remain in this state is determined by the amount of messages that threads of the program will send to the ring buffer, as well as how long the bound checking thread sleeps between the period when there is no new data in the ring buffer. The second is an inserting state, in which the message that the bound checking thread received via the ring buffer contains the flag value of 0, which tells the bound checking thread to calculate the bound of a buffer from the given base and size and insert the base and bound into the bound table, using the `btable_update` function. The third state is the bound checking state, where the value of the flag is 1, indicates that the bound checking thread should use the given base to lookup the corresponding bound in the bound table, and then use the base, bound, and offset to perform bound checking by invoking the `OOB_CHECK` macro, which has the following definition:

```
#define OOB_CHECK(base, bound, offset) \
  (offset < base || bound <= offset)
```

If the test fails, then the bound checking thread immediately halts the program execution and dump debug information.

After the bound checking thread finishes processing a message in the ring buffer, it will increment the variable `ring_buffer_tail`, as shown in the following code snippet:

```
ring_buffer_tail = (ring_buffer_tail + 1) % QUEUE_SIZE;
```

Again, note that the bound checking thread does not modify the value of the `ring_buffer_head` index, which is used exclusively by the threads in the program.

4.4 Summary

In this chapter, we describe the design and implementation of BoundWarden and its various components. The key idea of BoundWarden is to leverage the ubiquity of multi-core processors by offloading the runtime overhead incurred by bound checking to a dedicated thread. To accomplish this, we split BoundWarden into two components: the compiler extension and the runtime. The compiler extension, implemented as an LLVM pass, is responsible for transforming source code and inserting code needed to invoke the runtime component to perform bound checking, while the runtime component, which includes the aforementioned bound checking thread, is responsible to performing all bound checking during runtime.

To avoid having to modifying memory layout, we use the disjoint metadata technique [22] and store base and bound of buffers in a separated bound table. We associate each buffer with its metadata using the buffer's memory address. This greatly simplified the transformation needed to perform by the compiler extension. Also, by leveraging how Clang transforms C source code into LLVM IR, we showed that it is possible to detect buffer overflows in a buffers inside a structure, which used to be impossible in many previous object-based approaches.

As can be seen, the majority of the techniques used by the compiler extension component rely heavily on the semantic and structure of LLVM IR. This means that currently BoundWarden can only work with Clang/LLVM compiler toolchain. With that said, there is nothing preventing BoundWarden from working with other LLVM frontends, provided those frontends generate LLVM IR for accessing and manipulating memory in a manner that is compatible with BoundWarden pass transformation algorithms.

We try to keep the design and implementation of the runtime component as simple as possible, mainly to minimize runtime overhead and to maximize portability and ease of use.

The first prototype of BoundWarden proved to be effective at enforcing spatial memory safety. However, the runtime overhead of this first prototype was unacceptably high: averaging at around 7.5 times slower than the uninstrumented code. In the next chapter, Chapter 5, we describe the optimization techniques that we implemented to reduce runtime overhead of BoundWarden.

CHAPTER V

OPTIMIZATIONS

Our first prototype of BoundWarden was proven to be effective at detecting and preventing buffer overflow errors, but it had an unacceptable high runtime overhead: on average 7.5 times slower than the uninstrumented code. In this chapter, we describe the three optimizations that we implemented to help mitigate high runtime overhead. The first optimization that we implemented, inline functions, is explained in Section 5.1. Section 5.2 describes the local bound check optimization, and the last optimization, disable checks on local non-composite types, is explained in Section 5.3.

5.1 Inline Functions

The first optimization we implemented was to transform the `call` instructions that calls out to the function `__bw_send` and `__bw_check` into the equivalent LLVM IR that performs the same tasks. As can be seen from the function definition of both functions in Section 4.2.1 and Section 4.2.4, the only difference in the body of both functions is the value of the `flag` variable, which is 0 for `__bw_send` and 1 for `__bw_check`. This makes it trivial to create a function `insertSentBaseBoundToRingBuffer` in BoundWarden pass that, depending on the arguments passed, generates either the IR that functions like the `__bw_send` function or the IR that functions like the `__bw_check` function.

As an example, consider the following C code:

```c
long foo[5];
```

Which transforms into the following LLVM IR:

```llvm
%foo = alloca [5 x i64], align 16
```

Normally, BoundWarden pass would insert a `call` instruction that calls the function `__bw_send` after the `alloca` instruction:

```
%foo = alloca [5 x i64], align 16
%7 = bitcast [5 x i64]* %foo to i8*
call void @__bw_send(i8* %7, i64 40)
```

But with the inline function optimization, BoundWarden pass will call the function `insertSentBaseBoundToRingBuffer` with appropriate arguments to generate and insert LLVM IR that is functionally equivalent to the body of the function `__bw_send` after the `alloca` instruction, as shown in the following code snippet:

```
%foo = alloca [5 x i64], align 16
%27 = bitcast [5 x i64]* %foo to i8*
%28 = load i64, i64* @ring_buffer_head, align 8
%29 = load %struct.queue_elt*, %struct.queue_elt**
   @ring_buffer, align 8
%bw.gep.ptr15 = getelementptr inbounds %struct.queue_elt
   , %struct.queue_elt* %29, i64 %28
%bw.gep.ptr16 = getelementptr inbounds %struct.queue_elt
   , %struct.queue_elt* %bw.gep.ptr15, i32 0, i32 0
store i8* %27, i64* %bw.gep.ptr16, align 8
%bw.gep.ptr17 = getelementptr inbounds %struct.queue_elt
   , %struct.queue_elt* %bw.gep.ptr15, i32 0, i32 1
store i64 40, i64* %bw.gep.ptr17, align 8
%bw.gep.ptr18 = getelementptr inbounds %struct.queue_elt
   , %struct.queue_elt* %bw.gep.ptr15, i32 0, i32 2
store i8 0, i8* %bw.gep.ptr18, align 8
%30 = add i64 %28, 1
%31 = urem i64 %30, 1000000
store i64 %31, i64* @ring_buffer_head, align 8
```

Obviously, this greatly increase the size of the code, but as we will see later in

Section 6.2.3 in Chapter 6, in this case the performance gain is well worth the trade-off.

5.2 Local Bound Checking

Another optimization that we implemented is the local bound checking technique. That is, for some local variables that are allocated on the stack, instead of performing bound checking by sending the base and offset of these type of variables to the bound checking thread, BoundWarden pass will insert a `call` instruction that calls the function `__bw_local_check`, which performs bound checking locally in the program's thread, and pass the base, bound, and offset to directly to the function.

As an example, consider the following code snippet:

```
int foo[10];
foo[7] = 42;
```

Which will be translated into the following IR:

```
%foo = alloca [10 x i32], align 16
...
%4 = getelementptr inbounds [10 x i32], [10 x i32]* %foo
    , i64 0, i64 7
store i32 42, i32* %4, align 4
```

With local bound checking, BoundWarden will transform the above snippet into the following:

```
%foo = alloca [10 x i32], align 16
...
%4 = getelementptr inbounds [10 x i32], [10 x i32]* %foo
    , i64 0, i64 7
%10 = bitcast [10 x i32]* %foo to i8*
%11 = bitcast i32* %9 to i8*
call void @__bw_local_check(i8* %10, i64 40, i8* %11)
```

```
store i32 42, i32* %4, align 4
```

As can be seen from the above code snippet, we pass the LLVM identifier that stores the address of the `foo` array and another one that stores the result of the offset calculation from the `GEP` operator directly to the `__bw_local_check` function. As for the bound of the `foo` array, we create a new LLVM integer constant to hold the value of the bound which is later pass to the function.

It is worth emphasizing that in order to obtain base, bound, and offset of local variables without having to reply on the runtime component, we had to modify BoundWarden pass so that it process each LLVM module in two passes. The first pass involves scanning through the module and looking for local variables that have suitable characteristics. BoundWarden will extract base and bound of these variables and store them in a variable called `VariableMap`, which is a simple associative data structure that maps local variable name to its base and bound. Note that the base and bound that are stored in the `VariableMap` are different from the ones that are stored in the bound table, as the base and bound in the `VariableMap` are LLVM identifiers.

Once the first pass is done, the second pass will use the information in the `VariableMap` to insert local bound checking to the appropriate local variables. This requires a modification to Algorithm 4 in Section 4.2.4 to check each local variable to determine whether a local variable's name is in the `VariableMap`. If it is, then the modified algorithm will insert a `call` instruction to the function `__bw_local_check` with corresponding arguments from the `VariableMap`.

At first glance, this optimization should greatly help reduce the runtime overhead of BoundWarden, since this approach completely bypass the need to first send the metadata to the ring buffer and having to wait for the bound checking thread to perform bound checking. In practice, however, the benefit of this optimization depends on the structure of the program and where the bottleneck or the hotspot is. If most of the program's data structures are located on the heap, then this optimization offers little benefits. Nevertheless, we enable this optimization for the rest of this paper.

5.3 Disable Checking on Local Non-Composite Types

The last optimization we implemented is a simple check that disable bound checking on local variables that are not composite types, *i.e.,* any type that are not arrays, structures, or unions. While this optimization is simple enough to implement, care must be made to properly handle edge cases, especially the case where local variables have pointers point to them. In order to handle this case, we utilized the two pass approach (that we previously used to implement local bound checking optimization) to implement a check in the first pass that determines the type of local variables and disable bound checking on the ones that are not composite types in the second pass.

5.4 Summary

In this chapter, we describe optimizations that we implemented in the latest version of the prototype of BoundWarden to help reduce its runtime overhead. It is worth pointing out that all three optimizations that we implemented are in BoundWarden pass, the compiler extension component of BoundWarden, which, according to the results from the profiler, accounts for the majority of performance bottlenecks.

In the next chapter, Chapter 6, we evaluate effectiveness and efficiency of the latest version the prototype of BoundWarden, which has been extended with these three optimizations.

CHAPTER VI

EVALUATION

In this chapter, we describe the experiments that we performed to evaluate BoundWarden effectiveness and efficiency. Section 6.1 describes the experiments that we conducted to evaluate BoundWarden effectiveness at protecting spatial memory safety, while the experiment in Section 6.2 measured BoundWarden runtime performance.

6.1 Protection Effectiveness

We use two publicly available test suites to evaluate BoundWarden effectiveness at preventing spatial memory safety violations: RIPE test suite, and NIST SARD Test Suite 89.

6.1.1 Evaluate Protection Effectiveness using RIPE

Runtime Intrusion Prevention Evaluator (RIPE) test suite [44] is an attack synthesizer/generator that is capable of generating attacks that can be used to evaluate buffer overflow defense mechanisms. RIPE is a well-known benchmark that has been used to evaluate many works in the literature. It can generate up to 850 valid attack forms from the following five attributes:

- **Location** in memory of the buffer to be overflown, which includes stack, heap, BSS, and Data segment.
- **Target code pointer** that RIPE is going to redirect towards the attack code. Available options are return address, old base pointer, five types of function pointers, five types of longjump buffers, and four types of vulnerable structures (*i.e.*, structures that contains buffers and function pointers).
- **Overflow technique** used, of which two are supported: direct and in-direct overflowing techniques.

Runtime Environment	Successful Attacks	Partially Successful Attacks	Failed Attacks	Overall Effec-tiveness
Ubuntu 6.06				
GCC 4.0	806 (94%)	34 (4%)	10 (2%)	2%
Clang 3.8.1	442 (52%)	6 (1%)	402 (47%)	47%
BoundWarden	0 (0%)	0 (0%)	850 (100%)	100%
Debian 9.9				
BoundWarden	0 (0%)	0 (0%)	850 (100%)	100%

Table 6.1: Summary of protection effectiveness of BoundWarden using RIPE.

- **Attack code** that is the target of the redirection. The following are supported: shellcode without NOP sled, shellcode with NOP sled, shellcode with polymorphic NOP sled, return-to-libc, and Return-Oriented-Programming (ROP).
- **Function abused** that is used to perform buffer overflow. RIPE can perform buffer overflow using one of the following function: `memcpy`, `strcpy`, `strncpy`, `sprintf`, `snprintf`, `strcat`, `strncat`, `sscanf`, `fscanf`, and `homebrew`, a loop-based equivalent of `memcpy`.

Note that since RIPE was created and designed to run on 32-bit architecture, we modified the prototype of BoundWarden and add support for 32-bit architecture. Thanks to the modular design of BoundWarden, the changes needed to be made are minimum and mostly involves modifying the bound table to support 32-bit addressees.

As can be seen from the results in Table 6.1, BoundWarden successfully prevented all 850 attacks that RIPE generated on both Ubuntu 6.06 Desktop Edition, which is the environment used in original the RIPE paper [44] that has many serious vulnerabilities, and on a modern, fully updated Debian 9.9 system.

6.1.2 Evaluate Protection Effectiveness using NIST SARD Test Suite 89

Another test suite that we used to evaluate BoundWarden protection coverage is NIST Software Assurance Reference Dataset (SARD) [24] Test Suite 89 [34], which is based on a taxonomy of C buffer overflows developed by Kratkiewicz et al. [16], and contains 291 test cases, each with four variants (*i.e.*, each test case contains three "bad" tests, those that contain buffer overflow errors, and one "good" test, one which does not contain buffer overflow errors), for the total of 1,164 tests. As per the suggestions in [16], we report the results using three metrics:

- **Detection Rate**, which is defined as the number of the test cases in which BoundWarden correctly detects buffer overflows in the bad tests divided by the number of all test case, as shown in the following equation:

  $$DR = \frac{TC_{bad}}{TC_{all}},$$

  where DR is the detection rate, TC_{bad} is the number of the test cases that BoundWarden detects buffer overflows in the bad tests, and TC_{all} is the number of all test cases.

  Detection rate measures the ability of BoundWarden to detect and prevent the buffer overflows in various circumstances. The higher the detection rate, the better BoundWarden is at enforcing spatial memory safety.

- **False Alarm Rate**, which is defined as the number of the test cases that BoundWarden incorrectly identifies the good tests as having buffer overflows divided by the number of all test case:

  $$FR = \frac{TC_{good}}{TC_{all}},$$

  where FR is the false alarm rate, TC_{good} is the number of the test cases that BoundWarden detects buffer overflows in the good tests, and TC_{all} is the number of all tests that are used.

  False alarm rate tell us how likely it is that BoundWarden will (incorrectly) detect buffer overflows in programs that do not have any buffer overflow error in them. The lower the false alarm rate, the lower the chance that BoundWarden will make this mistake.

Test ID	Description
155–157, 159–161, 171–173, 175–177	Buffer overflows in global and static variables[1]
179–181, 219–221, 223–225, 227–229, 231–233, 235–237, 239–241, 243–245, 213, 217	Buffer overflows in nested composite types
291–293, 295–297, 299–301, 303–305, 307–309, 311–313, 315–317, 319–321, 839–841, 843–845, 1263–1265, 1267–1269, 1271–1273, 1275–1277	Buffer overflows when using external or library functions to copy data from one buffer to another
163–165	Buffer overflows in shared memory
277	A buffer overflow when performing pointer arithmetic

Table 6.2: Tests from SARD Test Suite 89 that BoundWarden failed to pass, grouped into 5 categories.

- **Confusion Rate**, which is defined as the number of the test cases where BoundWarden detects buffer overflows in both the bad and good tests, divided by the number of test cases that BoundWarden correctly detects buffer overflows in the bad tests, as shown below:

  $$CR = \frac{TC_{bad+good}}{TC_{bad}},$$

  where CR is the confusion rate, $TC_{bad+good}$ is the number of the test cases that BoundWarden mistakenly detects buffer overflows in both the good and bad tests, and TC_{bad} is the number of the test cases where BoundWarden detects buffer overflows in the bad tests.

  Confusion rate indicates how well BoundWarden can distinguish between the bad and good tests. The lower the confusion rate, the better BoundWarden is at differentiating between programs that have buffer overflow errors and ones that don't.

[1]Note that BoundWarden passed all 12 tests in this category once we applied the fix described in Section 6.1.2.1.

Against the SARD Test Suite 89, BoundWarden initially had the detection rate of 93%, that is, it successfully detected and prevented spatial memory violations in 1,080 tests out of the total 1,164, with 0 false alarm and confusion rates. Table 6.2 lists all the tests that BoundWarden failed to pass, which we grouped into five categorizes. Next, we will go over each category and explain the reasons why BoundWarden failed each one in detail.

6.1.2.1 Out-Of-Bound Errors on Static Arrays

The first category of test cases that BoundWarden failed to pass involve tests that perform out-of-bound array indexing on static arrays in various locations in the memory. As described in Section 4.2.3, BoundWarden already supports retrieving base and bound of global variables, as well as performing bound checking on global variables. Since static variables, both local and global, are treated by LLVM as more or less equivalent to global variables, these failures are unexpected at first. However, a closer inspection at how LLVM generate IR for static variables reveals the source of the problem. Namely, in the case of static variables, the GEP operations for computing the index into static variables are “inlined” into the store and load instructions. This means that Algorithm 4 cannot be used to match and insert code to invoke bound checking thread. To solve this problem, we implement a new algorithm that matches each `store` instruction and checks whether its second argument is an inlined `GEP` operator. If it is, then the new algorithm will extract the base, bound, and offset of the static arrays, and insert the same bound checking code used by Algorithm 4 to perform bound checking.

With this modification, BoundWarden successfully pass all 12 tests in this category, which improves the overall detection rate to 94% — now passing 1,092 tests out of the total 1,164 with 0 false alarm and confusion rates.

6.1.2.2 Out-Of-Bound Errors in Nested Composite Types

The second category of tests that BoundWarden failed to pass involves buffer overflows inside nested composite types. These include multidimensional arrays, arrays inside a structure, arrays inside a union, etc. On closer inspection, it is revealed that the majority of tests in this group involves buffer overflows between members of either a structure or a union, as the following code from test case ID 232 illustrate:

```c
typedef struct {
  char buf[10];
  int int_field;
} my_struct;
int main(int argc, char *argv[])
{
  my_struct array_buf[5];
   /*  BAD  */
  array_buf[4].buf[17] = 'A';
   return 0;
}
```

While it is unfortunate that BoundWarden currently cannot detect this category of buffer overflow, it has no issue detecting buffer overflows in the related cases where the index or pointer goes out-of-bound of the outmost composite type.

To better illustrate this, the following is the code from test case ID 209, where BoundWarden correctly detected and prevented buffer overflow:

```c
typedef struct {
  int int_field;
  char buf[10];
} my_struct;
int main(int argc, char *argv[])
{
  my_struct s;
  s.int_field = 10;
  /*  BAD  */
  s.buf[s.int_field] = 'A';
```

```
    return 0;
}
```

As can be seen in the above code snippet, the test involves an attempt to access an element inside the array `buf` inside the structure `s` with an invalid index value (10, in this case). From our experiment, BoundWarden has no problem detecting the out-of-bound error in this and other tests which have this kind of out-of-bound errors.

6.1.2.3 Out-Of-Bound Errors When Using External Functions to Copy Buffers

The third category of tests that BoundWarden failed is the category that represents buffer overflows when using various library functions to copy data to a buffer or from one buffer to another. In this case, BoundWarden failed to properly detect and prevent buffer overflows because the external functions that are being called to copy data to and from buffers are not instrumented by BoundWarden compiler extension. In order for BoundWarden to pass the tests in this group, we can either recompile the standard library with BoundWarden compiler extension pass or use the `LD_PRELOAD` technique to override symbols in the library and redirect them to equivalent functions that has been instrumented by BoundWarden pass.

6.1.2.4 Out-Of-Bound Errors in Shared Memory

The fourth category of tests simulates buffer overflows in shared memory. We believe that the reason that BoundWarden failed this set of tests is because the current implementation of BoundWarden pass did not instrument the functions that allocated and attach shared memory into a process, namely the `shmget` and `shmat` functions. The solution would be to implement an algorithm to match and instrumented all functions that allocated and attached shared memory.

6.1.2.5 Out-Of-Bound Errors When Performing Complex Pointer Arithmetic

The fifth and final category is a test ID 277 which is a part of the test case 275–278 that evaluates BoundWarden's ability to detect buffer overflow when performing pointer arithmetic. The full code of the test ID 277 is shown below:

```
1  int main(int argc, char *argv[])
2  {
3    int i;
4    char buf[10];
5    i = 2;
6    /*  BAD  */
7    (buf + (4 * i))[2] = 'A';
8    return 0;
9  }
```

The reason as to why BoundWarden failed to pass this particular test lies in how LLVM translates the pointer arithmetic expression. Specifically, the line 7 in the above snippet is translated into the following IR:

```
1  %4 = getelementptr inbounds [10 x i8], [10 x i8]* %buf,
       i32 0, i32 0
2  %5 = load i32, i32* %i, align 4
3  %6 = mul nsw i32 4, %5
4  %7 = sext i32 %6 to i64
5  %8 = getelementptr inbounds i8, i8* %4, i64 %7
6  %9 = getelementptr inbounds i8, i8* %8, i64 2
7  store i8 65, i8* %9, align 1
```

As can be seen, the pointer arithmetic expression is translated into two `GEP` operators: one in line 5 which handles the `(buf + (4 * i))` part of the expression, and the other in line 6 handles the array indexing part of the expression (the `[2]` part). While BoundWarden pass instrumented both `GEP` operators, it failed to detect buffer overflow because the first argument of the `GEP` operator in line 6 points to an address in the middle of the array `buf`, the address of which is not stored in the bound table. Solving this issue would require extending BoundWarden with the ability to track point-

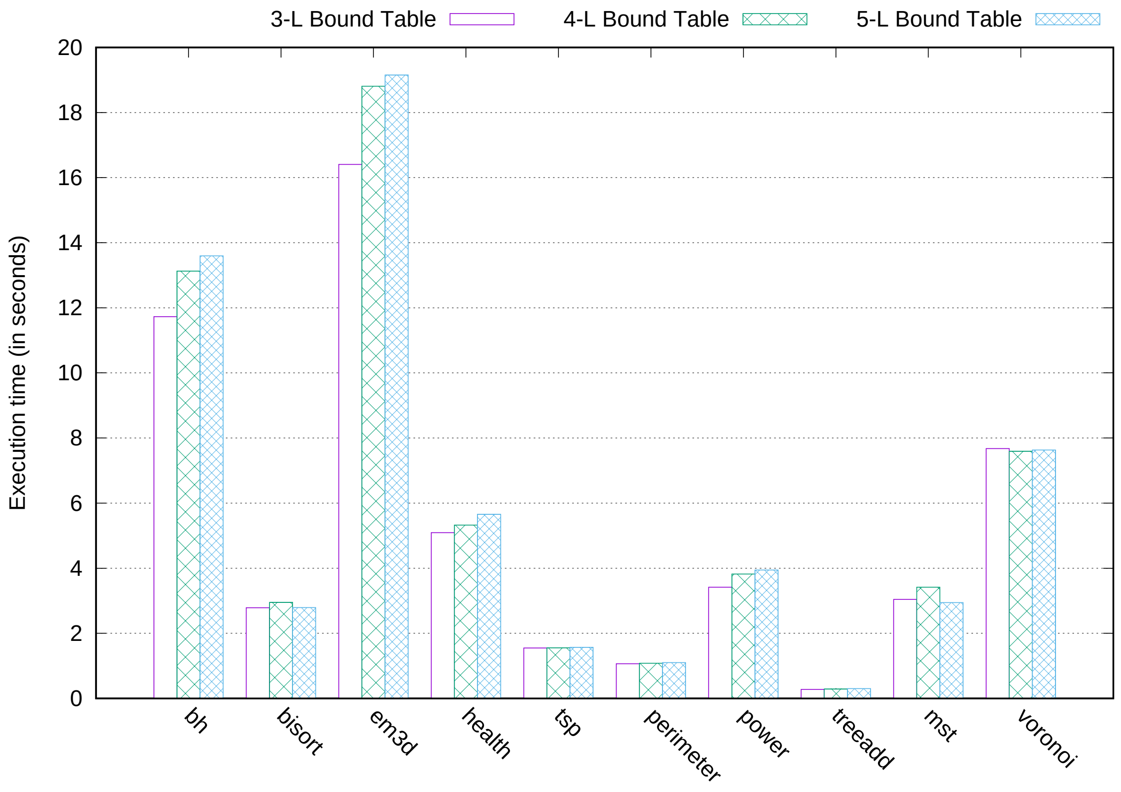


Figure 6.1: Performance of the varying sizes of the bound table (the lower the better).

ers that point to a middle of a buffer, which would turn BoundWarden into a hybrid of object-based and pointer-based approaches.

6.2 Runtime Performance

All the experiments in this and the following sections were performed on a 2012 Lenovo Thinkpad X230 with Intel Core i5-3210M CPU running at 2.50 GHz, 16 Gbyte of RAM, and an SSD. The OS is 64-bit Debian 9.9 Stretch. The compiler extension component of BoundWarden prototype was implemented as an LLVM pass against LLVM version 3.8.1, and the runtime component was implemented as a static library. All test suites and benchmark used were compiled using Clang version 3.8.1 with `-O3` optimization level.

We use Olden benchmark [33] to gauge the performance overhead of our bound checking technique. The reason why we choose to use Olden benchmark is because it is a well-known benchmark that is a part of the LLVM test suite and is also used by many other works, such as SafeDrive's Deputy [47], CCured [25, 26], SAFECode [10, 9], SoftBound [22] and Checked C [12].

We begin by performing tests to determine the optimal bound table size in Section 6.2.1, and, in Section 6.2.2, the optimal ring buffer size. We then used these parameters to run a benchmark that evaluate BoundWarden runtime performance in Section 6.2.3.

6.2.1 Determine the Optimal Bound Table Size

The first test we performed was the evaluation of the performance of the various implementations of the bound table to determine which one is the fastest. We ran each benchmark 10 times, and aggregated the results. Figure 6.1 shows the results of the experiment.

As can be seen from Figure 6.1, while the results are very close, the 3-L bound table has a slight edge over the other two, although it is also a bit more volatile (as can be seen with the higher spread of the performance data). Nevertheless, We decided to use the 3-L bound table in all the subsequent tests.

6.2.2 Determine the Optimal Ring Buffer Size

The next test that we perform is a test to determine the most optimal size of the ring buffer. As described previously, the size of the ring buffer is fixed and determined at compile time. We compare the performance of four sizes of the ring buffer: 2.4 Mbyte, 24 Mbyte, 48 Mbyte, and 240 Mbyte. Figure 6.2 shows the results of the experiment, in which we again ran each benchmark 10 times then aggregated and averaged the results. While the results are very close, the ring buffer with the size of 24 Mbyte is slightly faster than others, with runtime performance on average of 5.2 seconds. Therefore, we decided to set the size of the ring buffer to 24 Mbyte in all the tests in the rest of this work.

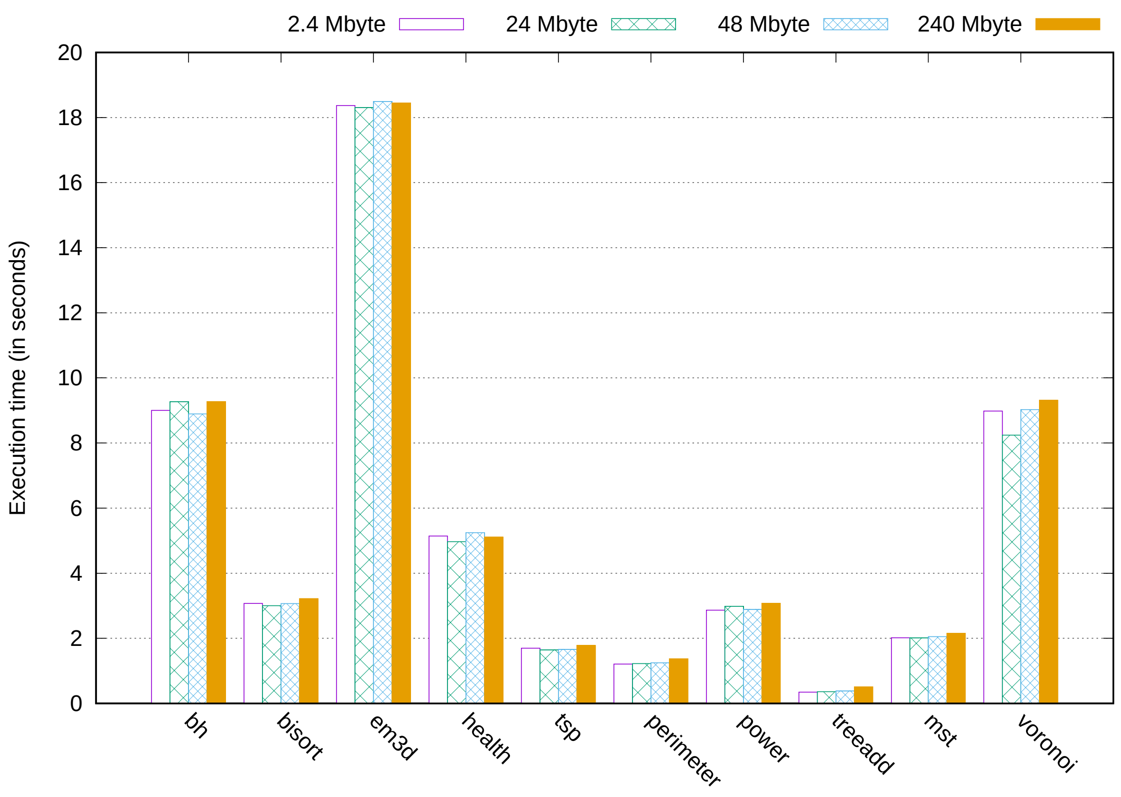


Figure 6.2: The impact of the size of ring buffer on BoundWarden performance (the lower the better).

6.2.3 Overall Runtime Performance

The last test we performed was the evaluation of the overall runtime performance of BoundWarden. We evaluated the performance of two implementations of BoundWarden: the first implementation, labeled *BoundWarden*, lacked all optimizations described in Chapter 5, and the second implementation, labeled *BoundWarden-Optimized*, was an implementation that included all the optimizations described in Chapter 5. For comparison, we also used Olden benchmark to measure the performance of SoftBound +CETS [23, 22]. We cloned SoftBound+CETS from its Git repository, and compiled and installed according to the instruction. Note that we did not enable SoftBound+CETS's LTO support.

The reason we choose to use SoftBound+CETS as a reference is because its architecture is similar to BoundWarden, though it is worth emphasizing that SoftBound+CETS enforces both spatial and temporal safeties, while BoundWarden only enforces spatial

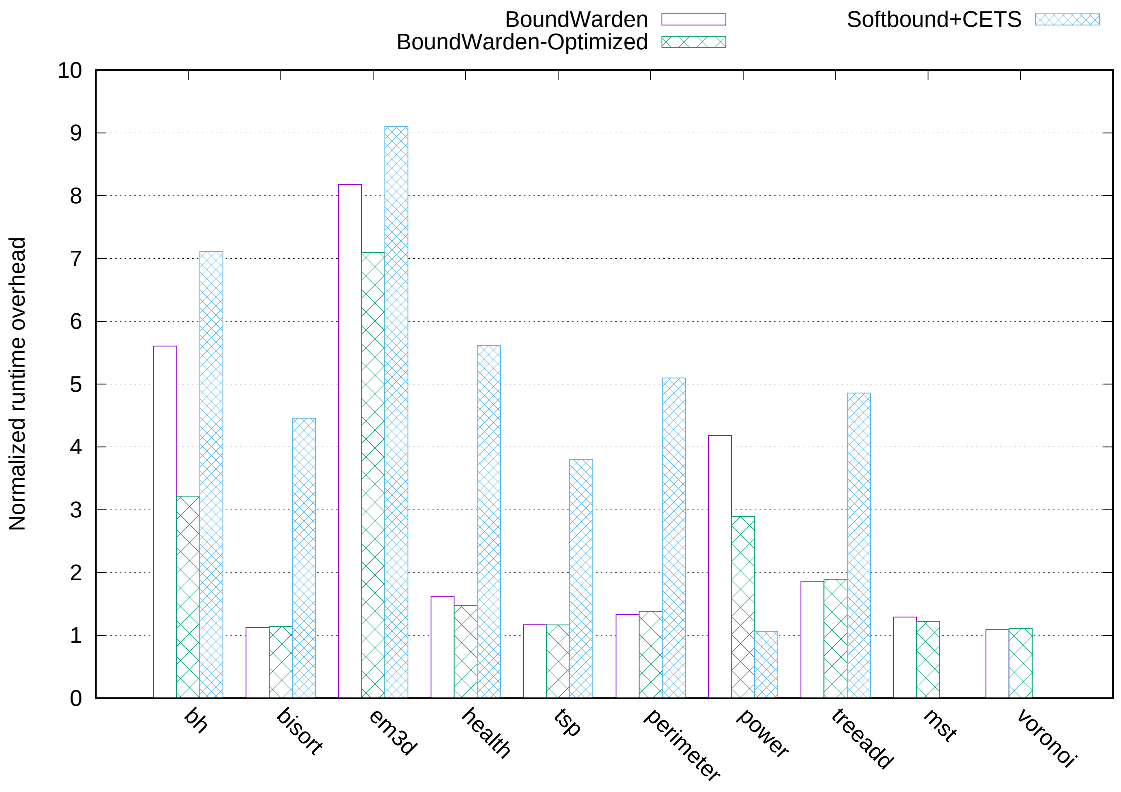


Figure 6.3: Overall runtime overhead of BoundWarden, compared against those of SoftBound+CETS (the lower the better).

memory safety. Like in the previous tests, we ran each benchmark 10 times, then averaging the results. We then normalized the results using the runtime performance data of the uninstrumented code as the base case.

When compiling the benchmark, we encountered difficulties when trying to instrument the bh application with BoundWarden, due to how the bh application allocates and manages structures. In the end, we had to made modifications to both versions of BoundWarden prototypes to workaround the issue. Testing showed that the workarounds did not affect BoundWarden's performance in other benchmarks. Also, we were unable to use SoftBound+CETS to instrument and compile mst and voronoi applications, despite disabled vectorization instructions as per the recommendation, which explains the lack of SoftBound+CETS's benchmark results from those two applications in the results. Figure 6.3 shows the results of the experiment.

As can be seen from the results, BoundWarden, especially the version which includes all the optimizations, is in general faster than SoftBound+CETS (again, it is worth repeating that SoftBound+CETS enforces spatial and temporal safeties, while Bound-Warden only enforces spatial safety, which likely contributed to SoftBound+CETS having higher runtime overhead). Specifically, on average, the version of BoundWarden without any optimization is 2.74 times slower than the uninstrumented code, while the version which includes all optimizations is 2.25 times slower than the uninstmented code. If we discard the results from the mst and voronoi applications, then the non-optimized version of BoundWarden is 3.13 times slower than the base case, while the optimized version is 2.5 times slower. In comparison, SoftBound+CETS, on average (again only considering the first eight applications from the benchmark), is 5.1 times slower than the base case.

6.3 Summary

In this chapter, we evaluate effectiveness and efficiency of BoundWarden prototype. As can be seen from the results, BoundWarden offers comprehensive protection against spatial memory violations. The current prototype of BoundWarden is capable of detecting and preventing buffer overflows in all 850 attacks that RIPE test suite generated and 94% (or 1,092 out of the total 1,164 tests) from the SARD Test Suite 89. We used the Olden benchmark to experimentally determine the optimum size of the bound table and the ring buffer, as well as to determine the overall runtime overhead of Bound-Warden, compared to SoftBound+CETS. The results showed that the latest prototype of BoundWarden is, on average, 2.25 times slower than the uninstrumented code and is around 2 times faster than SoftBound+CETS.

Interesting enough, one of the main contribution that made both versions of BoundWarden faster than SoftBound+CETS was not all the optimizations that we implemented (though they certainly contributed), but the technique that we used to detect and prevent buffer overflows in structures and unions that we described in Section 4.2.5. Recall that LLVM translates C code that access a buffer inside a structure or a union into two `GEP` operators; one that calculates the offset into structures or unions, while the other calculates the offset into buffers. Being able to differentiate between these two kinds of `GEP` operations help us cut down on the amount of `GEP` operators that must be instrumented significantly, which means less work needed to be done in general, which translates directly into faster runtime performance.

Also, from the profiling results, we discovered that the major source of the overheads in BoundWarden prototype came not from the runtime component, but from the code that were inserted into the C source code by the compiler extension component. This fact helps to back up the explanation of the beneficial effect of the algorithm that we used to insert bound checking code for buffers in composite types. It also helps direct our future research efforts, as it is clear that in order to improve the performance of BoundWarden, we need to improve the static analysis capability of its compiler extension component.

CHAPTER VII

CONCLUSION

In this dissertation, we proposed BoundWarden, a compiler and runtime system that preserves spatial memory safety for C programming language. By storing base and bound of buffers in a dedicated bound table, BoundWarden can comprehensively detect and prevent spatial memory safety violations in buffers on stack, heap, as well as data and BSS segments, without having to make any modification to the C source code. This helps preserve compatibility with existing libraries and binaries. The results from RIPE and SARD test suites showed that BoundWarden successfully detected and prevented buffer overflow and other out-of-bound errors in both test suites, with a detection rate of 100% and 94%, respectively, along with 0% false alarm and confusion rates. In order to reduce runtime overhead, BoundWarden offloads the majority of the works to a dedicated bound checking thread, which performs bound checking as well as managing metadata. The results from Olden benchmark showed that the prototype implementation of BoundWarden, with all optimizations enabled, is on average 2.25 times slower when compared to the uninstrumented code, which according to the profiling results most of the overheads came from the code that BoundWarden inserts into source code of a program.

In the future, we plan to improve the static analysis capability of BoundWarden, which in its current state is quite primitive, in order to improve performance by reducing the amount of code that are needed to be inserted into source code. We also plan to extend BoundWarden with the ability to detect and prevent temporal safety.

REFERENCES

[1] 2018CVE details [Online]. Available from: https://www.cvedetails.com/ [2018,Nov.].

[2] Akritidis, P., Costa, M., Castro, M., and Hand, S. 2009. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *18th USENIX Security Symposium (USENIX Security '09)*. : USENIX.

[3] Anderson, J. P. et al. 1972. Computer security technology planning study. Technical report, ESD-TR-73-51.

[4] Chiamwongpaet, S. and Piromsopa, K. 2009. The implementation of secure canary word for buffer-overflow protection. In *2009 IEEE International Conference on Electro/Information Technology*, pp. 56–61. : IEEE.

[5] d. Amorim, A. A., Dénès, M., Giannarakis, N., Hritcu, C., Pierce, B. C., Spector-Zabusky, A., and Tolmach, A. 2015. Micro-policies: Formally verified, tag-based security monitors. In *2015 IEEE Symposium on Security and Privacy*, pp. 813–830. :

[6] DESIGNER, S. Non-executable user stack. *http://www.openwall.com/linux/* ():

[7] Devietti, J., Blundell, C., Martin, M. M. K., and Zdancewic, S. 2008. Hardbound: Architectural support for spatial safety of the c programming language. *SIGARCH Comput. Archit. News* 36.1 (March 2008): 103–114.

[8] Dhawan, U., Hritcu, C., Rubin, R., Vasilakis, N., Chiricescu, S., Smith, J. M., Knight, T. F., Jr.., Pierce, B. C., and DeHon, A. 2015. Architectural support for software-defined metadata processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pp. 487–502. New York, NY, USA: ACM.

[9] Dhurjati, D. 2006. *Safecode: A Platform for Developing Reliable Software in Unsafe Languages*. PhD thesis, Champaign, IL, USA.

[10] Dhurjati, D. and Adve, V. 2006. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pp. 162–171. New York, NY, USA: ACM.

[11] Durden, T. 2002. Bypassing pax aslr protection. *Phrack magazine* 59.9 (2002): 9.

[12] Elliott, A. S., Ruef, A., Hicks, M., and Tarditi, D. 2018. Checked c: Making c safe by extension. In *2018 IEEE Cybersecurity Development (SecDev)*, pp. 53–60. :

[13] ISO. 2011. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland. Available from: https://www.iso.org/standard/57853.html .

[14] Jones, R. W. M. and Kelly, P. H. J. 1997. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*, number 1, pp. 13–26. : Linköping University Electronic Press; Linköpings universitet.

[15] Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. 2019. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*. : IEEE.

[16] Kratkiewicz, K. and Lippmann, R. 2006. A taxonomy of buffer overflows for evaluating static and dynamic software testing tools. In *Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics*, volume 500, p. 44. : NIST.

[17] Kwon, A., Dhawan, U., Smith, J. M., Knight, T. F., Jr.., and DeHon, A. 2013. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pp. 721–732. New York, NY, USA: ACM.

[18] L., S. J. 1992. Adding run-time checking to the portable c compiler. *Software: Practice and Experience* 22.4 (1992): 305–316.

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

[20] Lattner, C. and Adve, V. 2005. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. Chigago, Illinois: PLDI.

[21] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. 2018. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*. : USENIX.

[22] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. 2009. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pp. 245–258. New York, NY, USA: ACM.

[23] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. 2010. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pp. 31–40. New York, NY, USA: ACM.

[24] National Institute of Standards and Technology (NIST) 2019.

[25] Necula, G. C., McPeak, S., and Weimer, W. 2002. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pp. 128–139. New York, NY, USA: ACM.

[26] Necula, G. C., McPeak, S., and Weimer, W. 2002. Ccured: Type-safe retrofitting of legacy code. *SIGPLAN Not.* 37.1 (January 2002): 128–139.

[27] Oleksenko, O., Kuvaiskii, D., Bhatotia, P., Felber, P., and Fetzer, C. 2017. Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches. *CoRR* abs/1702.00719 (2017):

[28] One, A. 1996. Smashing The Stack For Fun And Profit. *Phrack* 7.49 (November 1996):

[29] Orman, H. 2003. The morris worm: a fifteen-year perspective. *IEEE Security Privacy* 99.5 (Sep. 2003): 35–43.

[30] Piromsopa, K. and Chiamwongpaet, S. 2008. Secure bit enhanced canary: Hardware enhanced buffer-overflow protection. In *2008 IFIP International Conference on Network and Parallel Computing*, pp. 125–131. : IFIP.

[31] Piromsopa, K. and Enbody, R. J. 2006. Secure bit: Transparent, hardware buffer-overflow protection. *IEEE Transactions on Dependable and Secure Computing* 3.4 (Oct 2006): 365–376.

[32] Piromsopa, K. and Enbody, R. 2011. Survey of protections from buffer-overflow attacks. *Engineering Journal* 15.2 (Feb. 2011): 31–52.

[33] Rogers, A., Carlisle, M. C., Reppy, J. H., and Hendren, L. J. 1995. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. Program. Lang. Syst.* 17.2 (March 1995): 233–263.

[34] Rosenberg, E. Test suite #89: A taxonomy of buffer overflows [Online]. Available from: https://samate.nist.gov/SARD/view.php?tsID=89 [,].

[35] Ruwase, O. and Lam, M. S. 2004. A practical dynamic buffer overflow detector. In *IN PROCEEDINGS OF THE 11TH ANNUAL NETWORK AND DISTRIBUTED SYSTEM SECURITY SYMPOSIUM*, pp. 159–169. :

[36] Saeed, A., Ahmadinia, A., and Just, M. 2016. Tag-protector: An effective and dynamic detection of out-of-bound memory accesses. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, CS2 '16, pp. 31–36. New York, NY, USA: ACM.

[37] Salamat, B., Gal, A., Jackson, T., Manivannan, K., Wagner, G., and Franz, M. 2008. Multi-variant program execution: Using multi-core systems to defuse buffer-overflow vulnerabilities. In *2008 International Conference on Complex, Intelligent and Software Intensive Systems*, pp. 843–848. :

[38] Salamat, B., Jackson, T., Gal, A., and Franz, M. 2009. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pp. 33–46. New York, NY, USA: ACM.

[39] Shacham, H. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pp. 552–561. New York, NY, USA: ACM.

[40] Szekeres, L., Payer, M., Wei, T., and Song, D. 2013. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pp. 48–62. Washington, DC, USA: IEEE Computer Society.

[41] The MITRE Corporation 2019. Common vulnerabilities and exposures (cve) [Online]. Available from: https://cve.mitre.org/ [2019,January].

[42] Tian, D., Zeng, Q., Wu, D., Liu, P., and Hu, C. 2012. Kruiser: Semi-synchronized non-blocking concurrent kernel heap buffer overflow monitoring. In *NDSS*. :

[43] W., P. D., Yelena, Y., and K., P. E. 1993. Extensions to the c programming language for enhanced fault detection. *Software: Practice and Experience* 23.6 (1993): 617–628.

[44] Wilander, J., Nikiforakis, N., Younan, Y., Kamkar, M., and Joosen, W. 2011. Ripe: runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pp. 41–50. :

[45] Zelkowitz, M. V., McMullin, P. R., Merkel, K. R., and Larsen, H. J. 1976. Error checking with pointer variables. In *Proceedings of the 1976 Annual Conference*, ACM '76, pp. 391–395. New York, NY, USA: ACM.

[46] Zeng, Q., Wu, D., and Liu, P. 2011. Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pp. 367–377. New York, NY, USA: ACM.

[47] Zhou, F., Condit, J., Anderson, Z., Bagrak, I., Ennals, R., Harren, M., Necula, G., Brewer, E., Brewer, E., and Brewer, E. 2006. Safedrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pp. 45–60. Berkeley, CA, USA: USENIX Association.

Appendix I

FAILED TESTS FROM SARD TEST SUITE 89

The following table, Table A.1, contains the full list of all 84 tests from NIST's SARD Test Suite 89, along with their description, that the first version of the prototype of BoundWarden failed to pass. As we have previously described in Chapter 6, we successfully reduced the total number of failed test cases from 84 to 72 after we applied the fix described in Section 6.1.2.1.

Test ID	Description
155-157[1]	Buffer overflows in a local initialized static `char` array
159-161[1]	Buffer overflows in a local uninitialized static `char` array
163-165	Buffer overflows in a shared memory that has been casted into a `char` array
171-173[1]	Buffer overflows in a global uninitialized static `char` array
175-177[1]	Buffer overflows in an initialized global static `char` array
179-181	Buffer overflows in a 2 dimensions array
213	Buffer overflow in a `char` array that is the first element inside a union
217	Buffer overflow in a `char` array that is the last element inside a union
219-221	Buffer overflows in a `char` array inside a struct inside an array of structs
223-225	Buffer overflows in the first `char` array inside a struct inside an array of structs
227-229	Buffer overflows in the second `char` array inside a struct inside an array of structs
231-233	Buffer overflows in a `char` array that is the first element inside

[1] Note that BoundWarden passed these 12 tests once we applied the fix described in Section 6.1.2.1.

	a struct that has 2 elements inside an array of structs
235-237	Buffer overflows in a `char` array that is the second element inside a struct that has 2 elements inside an array of structs
239-241	Buffer overflows in a `char` array that is the first element inside a union that has 2 elements inside an array of unions
243-245	Buffer overflows in a `char` array that is the second element inside a union that has 2 elements inside an array of unions
277	Buffer overflow when performing pointer arithmetic using the array name as one of the argument
291-293	Buffer overflows when using `strcpy` function to copy a string literal that is larger than the size of the buffer
295-297	Buffer overflows when using `strcpy` function to copy a local string variable that is larger than the size of the buffer
299-301	Buffer overflows when using `strncp` function to copy a local string variable that is larger then the size of the buffer
303-305	Buffer overflows when using `strncp` function to copy a local string variable that is larger then the size of the buffer, where the length of the string is stored in a variable
307-309	Buffer overflows when using `strncp` function to copy a local string variable that is larger then the size of the buffer, where the length of the string is computed inline using the addition and multiplication operations
311-313	Buffer overflows when using `strncp` function to copy a local string variable that is larger then the size of the buffer, where the length of the string is computed inline using the modular operations
315-317	Buffer overflows when using `strncp` function to copy a local string variable that is larger then the size of the buffer, where the length of the string is obtained from a function call
319-321	Buffer overflows when using `strncp` function to copy a local

	string variable that is larger then the size of the buffer, where the length of the string is obtained from a local `int` array
839-841	Buffer overflows when reading contents of a file into a buffer
843-845	Buffer overflows when using `getcwd` function to store current working directory into a buffer
1263-1265	Buffer overflows when using `memcpy` function to copy a local string variable that is larger than the sieze of the buffer
1267-1269	Buffer overflows when using `memcpy` function to copy a local string variable that is larger than the sieze of the buffer, where the size of the buffer is stored in a local variable
1271-1273	Buffer overflows when using `memcpy` function to copy a local string variable that is larger than the sieze of the buffer, where the size of the buffer is stored in a local variable and compared against a variable that contains the actual size of the buffer before being used
1275-1277	Buffer overflows when using `memcpy` function to copy a local string variable that is larger than the sieze of the buffer, where the size of the buffer is stored in a local variable and compared against the actual size of the buffer before being used

Table A.1: The 84 tests from SARD Test Suite 89 that the early prototype of BoundWarden failed to detect and prevent buffer overflow errors.

VITA

Smith Dhumbumroong was born in Bangkok, Thailand, on April, 1985. He received his BA degree in Economics from Thammasat University, Thailand, in 2007, and his MS degree in Computer Science from Chulalongkorn University, Thailand, in 2011.