การป้องกันบัฟเฟอร์โอเวอร์โฟลว์ด้วยวิธีบิตระบุขอบเขต

นางสาวสิริสรา เจียมวงศ์แพทย์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรดุษฎีบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2559
ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

BUFFER-OVERFLOW PROTECTION USING BOUNDARY BIT

Miss Sirisara Chiamwongpaet

A Dissertation Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy Program in Computer Engineering
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Academic Year 2016

Thesis Title	BUFFER-OVERFLOW PROTECTION USING BOUNDARY BIT
By	Miss Sirisara Chiamwongpaet
Field of Study	Computer Engineering
Thesis Advisor	Assistant Professor Krerk Piromsopa, Ph.D.

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

....Dean of the Faculty of Engineering

(Associate Professor Supot Teachavorasinskun, Ph.D.)

THESIS COMMITTEE

....Chairman

(Professor Prabhas Chongstitvatana, Ph.D.)

....Thesis Advisor

(Assistant Professor Krerk Piromsopa, Ph.D.)

....Examiner

(Associate Professor Kultida Rojviboonchai, Ph.D.)

....Examiner

(Assistant Professor Natawut Nupairoj, Ph.D.)

....External Examiner

(Assistant Professor Putchong Uthayopas)

สิริสรา เจียมวงศ์แพทย์ : การป้องกันบัฟเฟอร์โอเวอร์โฟลว์ด้วยวิธีบิตระบุขอบเขต (BUFFER-OVERFLOW PROTECTION USING BOUNDARY BIT) อ.ที่ปรึกษาวิทยานิพนธ์หลัก: ผศ. ดร.เกริก ภิรมย์โสภา, 85 หน้า.

การป้องกันบัฟเฟอร์โอเวอร์โฟลว์ด้วยวิธีบิตระบุขอบเขต คือการตรวจสอบขอบเขตด้วยวิธีทางสถาปัตยกรรม เพื่อป้องกันการโจมตีด้วยบัฟเฟอร์โอเวอร์โฟลว์ วิธีนี้จะเพิ่มบิตระบุขอบเขต 1 บิตของหน่วยความจำทุกแถวสำหรับการตรวจสอบขอบเขต และซอฟต์แวร์จะกำหนดค่าบิตระบุขอบเขตนี้เพื่อสร้างขอบเขตขึ้นมา เมื่อต้องการเขียนบนหน่วยความจำนี้ ฮาร์ดแวร์จะตรวจสอบขอบเขตด้วยบิตระบุขอบเขตนี้ การออกแบบสถาปัตยกรรมนี้สามารถป้องกันการโจมตีด้วยบัฟเฟอร์โอเวอร์โฟลว์ได้เกือบทุกรูปแบบ ซึ่งรวมถึงการโจมตีข้อมูลส่วนที่ไม่เป็นตัวควบคุมระบบ เช่น ตัวแปร และอาร์กิวเมนต์เป็นต้น และรูปแบบอาร์เรย์อินเด็กซิงเออเรอร์ (Array Indexing Errors) ด้วย ซอฟต์แวร์สามารถนำวิธีบิตระบุขอบเขตมาปรับใช้ได้ง่าย นอกจากนี้ วิธีบิตระบุขอบเขตมีประสิทธิภาพในการป้องกันการโจมตีด้วยบัฟเฟอร์โอเวอร์โฟลว์ โดยลดสมรรถนะของระบบน้อยมาก เนื่องจากฮาร์ดแวร์สามารถลดค่าใช้จ่ายส่วนใหญ่ในการตรวจสอบบิตลงได้ ด้วยการใช้แผนที่บิต (Bitmap) ที่มี 1 ชั้นและมีขนาดที่เหมาะสม จะดีกว่าการใช้แผนที่บิตที่มี 2 ชั้น

ภาควิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่อนิสิต	....
สาขาวิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่อ อ.ที่ปรึกษาหลัก	....
ปีการศึกษา	2559		

5471430621 : MAJOR COMPUTER ENGINEERING
KEYWORDS: INVASIVE SOFTWARE / SECURITY KERNELS / SECURITY AND PROTECTION / SYSTEM ARCHITECTURES / UNAUTHORIZED ACCESS / BUFFER OVERFLOW

SIRISARA CHIAMWONGPAET: BUFFER-OVERFLOW PROTECTION USING BOUNDARY BIT. ADVISOR: ASST. PROF. KRERK PIROMSOPA, Ph.D., 85 pp.

Boundary Bit is a new architectural bound-checking approach for preventing against buffer-overflow attacks. It adds an associated bit to each memory entry to support bound checking. To make a boundary, software can simply set a (boundary) bit. On memory writing, hardware will dynamically validate limit using the boundary bit. With a minimal hint from software (compiler), our architectural design eliminates most (if not all) types of buffer-overflow attacks, including attacks on non-control data (variables and arguments) and array-indexing errors. Software can easily support Boundary Bit with few (minor) modification. Boundary Bit is secure and efficient with few (none) performance degradation. Our implementation shows that hardware can absorbed most bit-scanning overhead by using bitmap. An 1-level bitmap with proper size is better than a 2-level bitmap.

Department:	Computer Engineering	Student's Signature	....
Field of Study:	Computer Engineering	Advisor's Signature	....
Academic Year:	2016		

ACKNOWLEDGEMENTS

I would like to thank everyone who participated in my academic accomplishments.

First, I would like to thank my advisor Asst. Prof. Krerk Piromsopa, Ph.D., Faculty of Engineering, Chulalongkorn University, for everything. He pushes me towards my dreams and always gives me very good advice when I have any problem. I would also like to thank all faculty members and friends for educating, helping and sharing a great experience with me since I studied my bachelor degree.

Second, I would like to acknowledge the Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, for giving a CP Chulalongkorn Graduate Scholarship. I would also like to acknowledge the Graduate School of Chulalongkorn University for giving a Chulalongkorn University Graduate Scholarship to Commemorate the 72nd Anniversary of His Majesty King Bhumibol Adulyadej.

Finally, I must gratitude my parents for always supporting me and believing in me.

CONTENTS

Page

THAI ABSTRACT iv

ENGLISH ABSTRACT v

ACKNOWLEDGEMENTS vi

CONTENTS vii

CHAPTER I INTRODUCTION 1

I.1 Background and Problem 1

I.2 Glossary 2

I.3 Objectives 2

CHAPTER II BUFFER-OVERFLOW ATTACKS 3

II.1 Fundamental of Buffer-Overflow Attacks 3

II.2 Types of Buffer-Overflow Attacks 4

II.2.1 Classification by Attack Locations 4

II.2.1.1 Stack Overflows 4

II.2.1.1.1 Example 1: Stack-Overflow Attacks on Control Data 5

II.2.1.1.2 Example 2: Stack-Overflow Attacks on Non-Control Data 9

II.2.1.2 Heap Overflows 10

II.2.1.3 Array Indexing Errors 11

II.2.1.3.1 "Placement New" Expression in C++ 14

II.2.2 Classification Using Characteristics 15

II.2.2.1 Direct Executable Buffer Overflows 15

II.2.2.2 Indirect Executable Buffer Overflows 15

II.2.2.3 Direct Data Buffer Overflows 16

Page

II.2.2.4 Indirect Data Buffer Overflows 16
II.2.3 Characteristics 17
II.2.3.1 Direct Executable Buffer Overflows 18
II.2.3.2 Indirect Executable Buffer Overflows 18
II.2.3.3 Direct Data Buffer Overflows 18
II.2.3.4 Indirect Data Buffer Overflows 18
CHAPTER III LITERATURE REVIEWS 19
III.1 Static Analysis 19
III.1.1 Lexical Analysis 20
III.1.2 Semantic Analysis 20
III.2 Dynamic Solution 20
III.2.1 Address Protection 20
III.2.1.1 Canary Word 21
III.2.1.2 Address Encoding 22
III.2.1.3 Copy of Address 22
III.2.1.4 Tags 22
III.2.1.5 HeapDefender [36] 22
III.2.2 Input Protection 23
III.2.2.1 Secure Bit [37] 23
III.2.2.2 Minos [38] [39] 24
III.2.2.3 Tainted Pointer [40] 25
III.2.2.4 Efficient Dynamic Taint Analysis Using Multicore Machines [41] 25
III.2.3 Bound Checking 25

Page

III.2.3.1 Hardware 25

III.2.3.2 Software 26

III.2.4 Obfuscation 26

III.2.5 Mixed Solution 26

III.2.5.1 Secure Canary Word [6] 26

III.3 Isolation 27

III.3.1 Non-Executable Memory 27

III.3.2 Sandboxing 27

III.4 Comparison and Summary 28

CHAPTER IV THE CONCEPT OF BOUNDARY BIT 32

IV.1 The Concept of Boundary Bit 32

IV.2 Examples 32

IV.2.1 Stack-Overflow Detection 33

IV.2.2 Array-Indexing-Error Detection 34

IV.2.3 One-Byte Variable 34

IV.2.3.1 Stack-Overflow Detection 35

IV.2.3.2 Array-Indexing-Error Detection 36

CHAPTER V THE IMPLEMENTATION OF BOUNDARY BIT 38

V.1 Hardware 38

V.1.1 Boundary Bits and Bitmap 38

V.1.2 Memory Architecture 40

V.1.3 Instructions for Boundary Bits 43

V.2 Software 43

Page

CHAPTER VI EVALUATION 46

VI.1 Simulation Tools 46

VI.1.1 Trace File Generation 46

VI.1.1.1 Set Boundary Bit Instruction 46

VI.1.1.2 Clear Boundary Bit Instruction 46

VI.1.1.3 Scan Boundary Bit Instruction 47

VI.1.1.4 Read Memory Instruction 47

VI.1.1.5 Write Memory Instruction 48

VI.1.2 Simulation 48

VI.1.2.1 Boundary Bit Set/Clear Management 49

VI.1.2.2 Bitmap Set/Clear Management 50

VI.1.2.3 Boundary-Bit/Bitmap Scanning 51

VI.1.3 Simulation Test Environment 52

VI.2 Evaluation Aspects 53

VI.2.1 Protection Efficiency 53

VI.2.1.1 Stack Overflows 53

VI.2.1.1.1 Stack-Overflow Attack on Control Data 53

VI.2.1.1.2 Stack-Overflow Attack on Non-Control Data 55

VI.2.1.2 Heap Overflows 56

VI.2.1.3 Array Indexing Errors 56

VI.2.1.3.1 Array-Indexing-Errors (Stack) on Control Data 57

VI.2.1.3.2 Array-Indexing-Errors (Stack) on Non-Control Data 58

VI.2.1.3.3 Array-Indexing-Errors (Heap) 59

Page

VI.2.2 Performance 60

VI.2.2.1 Variable/Argument Declaration 60

VI.2.2.2 The End of Variable/Argument Existence 61

VI.2.2.3 Variable/Argument Access 61

VI.3 Simulation Results 62

VI.3.1.1 Case 1: Intensive Read/Write Buffer (Bubble Sort) 62

VI.3.1.2 Case 2: Random Write Memory 67

VI.3.2 Discussion 72

CHAPTER VII ANALYSIS 73

VII.1 Advantages 73

VII.1.1 Prevent All Types of Buffer-Overflow Attacks 73

VII.1.2 Fixed and Low Additional Memory Usage 73

VII.1.3 Low Performance Overhead 73

VII.2 Disadvantages 74

VII.2.1 Hardware Incompatibility 74

VII.2.2 Software Incompatibility 74

VII.3 Performance Analysis 74

VII.4 Cost Analysis 75

VII.5 The Impact of Virtual Memory 76

VII.6 Boundary Bit Protection 76

CHAPTER VIII CONCLUSION 77

VIII.1 Contributions 77

VIII.2 Conclusion 77

Page

REFERENCES ... 78

VITA ... 85

CHAPTER I
INTRODUCTION

This chapter explains background knowledge and the importance of buffer-overflow attacks. We also describe objectives and scope of this research.

I.1 Background and Problem

Nowadays, cyber-attacks around the world have intensified, especially, the increase of worms and viruses that cause serious damages to systems. Mostly, malicious worms and viruses use a buffer-overflow attack to exploit system vulnerabilities in order to access the system and destroy or steal data, similar to that of the infamous MORRIS worm [1] in 1988. Even recently, on 14th April 2017, there is a report VU#676632 on IBM Lotus Domino server mailbox name stack buffer overflow [2]. Also, the buffer-overflow vulnerability was found in the file win32k.sys of the component Memory Object Handler on 11th December 2013. This affects most Microsoft Windows, including Microsoft Windows 8.1 SP0 [3]. Moreover, the well-known WannaCry ransomware attack in May 2017 exploits a buffer-overflow vulnerability in the most Microsoft Windows as well, including Microsoft Windows 10 SP1 [4].

Buffer-overflow attacks generally occur when the length of input data is bigger than the buffer size. When that happens, some data will overflow outside the buffer and overwrite memory adjacent to the buffer. The overwritten memory may contain control data or non-control data. In addition to this general method, there is another way to overwrite data that is outside the scope, named "Array-indexing Error", which will be described in the next chapter.

Although a lot of buffer-overflow solutions have been proposed, they mostly focused on control data. Few of them focus on non-control data. When control data are protected, the next target will be non-control data which is no less important than control data.

Therefore, this research proposes a new solution, called "Boundary Bit", which is expected to not only protect both control and non-control data, but also prevent array-indexing error. We aim to completely eliminate buffer-overflow attacks.

I.2 Glossary

For the sake of clarity in this dissertation, terms are defined as follows:

1. **Buffer** means an allocated memory for containing data.
2. **Control Data** means data that are generated locally by systems such as return addresses and function pointers.
3. **Non-control Data** means data declared by users such as local variables and arguments.
4. **Buffer Overflows [5]** means the condition where in the data transferred to a buffer exceeds the storage capacity of the buffer and some of the data overflows into another buffer, one that the data was not intended to go into.
5. **Buffer-overflow Attacks [6]** means an attack caused by overflowing a buffer or writing beyond data boundary with data from another domain which results in malicious or unexpected behaviors of a program. Buffer-overflow attacks can be classified into 3 types: 1) Stack Overflows 2) Heap Overflows 3) Array Indexing Errors.

I.3 Objectives

Objectives of this research are as follows:

1. To study buffer-overflow attacks and buffer-overflow protections
2. To propose a new hardware solution for buffer-overflow protection named "Boundary Bit"
3. To apply "Boundary Bit" and to evaluate its (protection) effectiveness and efficiency

CHAPTER II
BUFFER-OVERFLOW ATTACKS

This chapter describes the principle of buffer-overflow attacks including their characteristics and types. Some background knowledge about buffer-overflow attacks involved in this research can be explained as follows.

II.1 Fundamental of Buffer-Overflow Attacks

General buffer-overflow attacks have the following main steps.

1. Allocate memory by declaring an array as a buffer for storing data.
2. Input data which are bigger than buffer size to the buffer by using array-copy functions such as *strcpy()* in C programming language.
3. The effect of the previous step is that some data will overflow outside the buffer, called "Buffer Overflows”, and overwrite memory adjacent to the buffer. The overwritten memory may contain control data or non-control data.
4. If the change of value in memory harms the computer, it will cause damage to that computer.

To ease explaining, the c-language code is provided.

```
#include <stdio.h>
#include <string.h>
int main(char *p) {
      char a[8];
      int b = 0;
      strcpy(a,p);
      printf(“%d”,b);
      return 0;
}
```

The attacks can be done by the following steps.

5. Declare a string (an array of characters) as buffer, named “a” with 8-byte size, and an integer named “b” assigned as 0. This creates a stack memory layout as shown in Table 1

Table 1 Stack Layout of Allocated Memory

a								b			
-	-	-	-	-	-	-	-	0	0	0	0

6. Call *strcpy()* function with character pointer "p" as input. If pointer "p" is longer than the length of an input will be copied to buffer "a" and overflow to integer "b" as shown in Table 2

Table 2 Stack Layout after Calling strcpy()

a								b			
o	v	e	r	f	l	o	w	s	\0	0	0

7. Print the value of the integer "b". It is no longer '0' as declared.

From the example, if the value of variable "b" cannot directly be modified by programmers or users, it can be done by a buffer-overflow technique. If a variable "b" is a return address, programmers or users can change the control flow by modifying the return address. The result can be harmful.

In summary, the buffer-overflow technique allows attackers with no privilege to modify arbitrary value. However, this technique is just a preparation step for attacking the system.

II.2 Types of Buffer-Overflow Attacks

Buffer-overflow attacks can be classified using various criteria. This research categorizes them by locations and characteristics.

II.2.1 Classification by Attack Locations

With this criterion, there are 3 main types of buffer-overflow attacks as follows.

II.2.1.1 Stack Overflows

Stack-overflow attacks modify values in the stack memory of the process. The main target is a return address in the stack. Typical attacks are conducted by copying data bigger than the size of allocated buffer in the same stack. As a result, the overflowed data will overwrite the return address, which is control data of the function. The return address value can be changed to any value specified by attackers. When the function ends, it will return to execute attackers' code instead of the normal process flow.

Here are two examples of Stack-overflow attacks.

II.2.1.1.1 Example 1: Stack-Overflow Attacks on Control Data

The c-language code is given as follows:

```
#include <stdio.h>
#include <string.h>
void foo(const char* input) {
  char buf[10];
  //View stack
  printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");

  //Pass input to buf
  strcpy(buf,input);
  printf("%s\n",buf);

  printf("Now stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
}
void bar() {
  printf("Pongo hack YOU!\n");
}
int main(int argc, char *argv[]) {
  printf("Address of foo = %p\n",foo);
  printf("Address of bar = %p\n",bar);
  if(argc != 2){
    printf("Enter string as an argument!\n");
    return -1;
  }
  foo(argv[1]);
  return 0;
}
```

From the example, an attacker wants to modify the return address, which is control data, for returning to attacker's target function (in this context: calling the "bar" function). The procedure can be described as follows:

1. Find the return address of the "foo" function by using the following code and calculate distance between the return address and the declared buffer in the "foo" function.

```
printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
```

This line will show data in the stack memory of the process during runtime. The first %p will show the data at the top of the stack (lower address) in hexadecimal number and the following %p will show the second one (higher address) respectively. The program can be executed using the following command.

```
# ./stack_control 01234567890
```

The result will be:

```
Address of foo = 0x8048400
Address of bar = 0x8048444
My stack looks like:
0xbffffb98
0x401081ec
0xbffffb98
0xbffffb98
0x80484ae
0xbffffcd7
0xbffffbb8

01234567890
Now stack looks like:
0x33323130
0x37363534
0x303938
0xbffffb98
0x80484ae
0xbffffcd7
0xbffffbb8
```

From the result, the address of the "foo" function is 8048400_{16} (0x8048400 in hexadecimal). the address of the "bar" function is 8048444_{16}. The stack layout before calling strcpy() is shown in Table 3.

Table 3 Stack Layout before Calling strcpy()

Address	Content	Symbol
[E]SP+0x00	0xbffffb98	buf
[E]SP+0x04	0x401081ec	
[E]SP+0x08	0xbffffb98	
[E]SP+0x0c	0xbffffb98	-
[E]SP+0x10	0x80484ae	return address
[E]SP+0x14	0xbffffcd7	-
[E]SP+0x18	0xbffffbb8	-

Note: [E]SP (a register in the processor) stores "stack pointer" address (the top of the stack) at the time.

From Table 3, the "buf" variable is a buffer in the "foo" function. The addresses from [E]SP+0x00 to [E]SP+0x08 are the "buf" variable. However, the amount of memory allocated is always in the number of words (4 times in bytes) because 1 word is 4 bytes. Moreover, the address [E]SP+0x10 is close to the addresses of both function and the program code is always loaded on the same memory area. It can be concluded that the address [E]SP+0x10 is the return address of the "foo" function.

After calling strcpy(), the input argument was copied to the declared buffer in "foo" function. The stack layout after copying is shown in Table 4.

Table 4 Stack Layout after Calling strcpy()

Address	Content	Symbol
[E]SP+0x00	0x33323130	buf
[E]SP+0x04	0x37363534	
[E]SP+0x08	0x00303938	
[E]SP+0x0c	0xbffffb98	-
[E]SP+0x10	0x80484ae	return address
[E]SP+0x14	0xbffffcd7	-
[E]SP+0x18	0xbffffbb8	-

From Table 4, the return address is 16 bytes away from the address of the declared buffer. The stack stores data in ASCII. For instance, the ASCII of the character "0" (zero) is 30_{16} (30 in hexadecimal) or 48_{10} (48 in decimal). The data is stored in reverse order of the input argument.

2. Modify the return address to the address of "bar" function by inputting 16 characters (16 bytes) together with the reverse order of the "bar" function, which are ASCII 44_{16} 84_{16} 04_{16} 08_{16} (or 68_{10} 132_{10} 4_{10} 8_{10}). There are 2 ways to input ASCII. First, press a left "Alt" button on a keyboard + ASCII in decimal on numeric keypad. Second, press a right "Alt" button + ASCII in hexadecimal. For example, press Alt + 68 and then a character "D" will show up. To run the program, use the following command.

```
# ./stack_control 0123456789012345Dä♦◘
```

The result will be:

```
Address of foo = 0x8048400
Address of bar = 0x8048444
My stack looks like:
0xbffffba8
0x401081ec
0xbffffba8
0xbffffba8
0x80484ae
0xbffffcdf
0xbffffbc8

0123456789012345D
Now stack looks like:
0x33323130
0x37363534
0x31303938
0x35343332
0x8040044
0xbffffcdf
0xbffffbc8

Pongo hack YOU!
Segmentation fault (core dumped)
```

Although the "bar" function is not called in the code, it can be called at runtime. The stack layout after the strcpy() function is called is shown in Table 5.

Table 5 Stack Layout after Calling strcpy()

Address	Content	Symbol
[E]SP+0x00	0x33323130	buf
[E]SP+0x04	0x37363534	
[E]SP+0x08	0x31303938	
[E]SP+0x0c	0x35343332	-
[E]SP+0x10	0x8048444	bar
[E]SP+0x14	0xbffffce5	-
[E]SP+0x18	0xbffffbc8	-

From Table 5, the return address stores the address which the program will jump back after the “foo” function ends. When it is modified to the address of “bar” function, the program will run the “bar” function instead of going back to the “main” function.

II.2.1.1.2 Example 2: Stack-Overflow Attacks on Non-Control Data

Beside a return address, the target can be non-control data, such as local variables, which are stored in the stack as same as the declared buffer. The c-language code is given as follows:

```
int main(int argc,char* argv[])
{
  int data2 = 0;
  char buf[10];

  if(argc != 2)
  {
    printf("Enter an argument! e.g. 01234567890123456\n");
    return -1;
  }

  printf("buf at %p\ndata2 at %p\n",buf,data2);

  printf("Before copy\nbuf = %s\n",buf);
  printf("data2 = %d at %p\n",data2,&data2);

  strcpy(buf,argv[1]);

  printf("After copy\nbuf = %s\n",buf);
  printf("data2 = %d at %p\n",data2,&data2);

  return 0;
}
```

From the code, copying data from the input argument to the declared buffer with data bigger than this buffer, the integer variable adjacent to this buffer may be modified because local variables are stored in the same stack. If there are many variables, memory will be allocated sequentially. The last declared variable will be at the top of the stack. Thus, “data2” and “buf” variables are adjacent in the stack.

```
# ./stack_noncontrol 01234567890123456
```

The result will be:

```
buf at 0xbffffb98
data2 at 0xbffffba4
Before copy
buf = ¨ûÿ¿+„€-

data2 = 0 at 0xbffffba4
After copy
buf = 01234567890123456
data2 = 892613426 at 0xbffffba4
```

From the result, the “data2” variable is not assigned at the second time in the code, but its value can be modified at runtime. The stack layout after calling strcpy() is shown in Table 6.

Table 6 Stack Layout after Calling strcpy()

Address	Content	Symbol
buf+0x00	0x33323130	
buf+0x04	0x37363534	buf
buf+0x08	0x31303938	
buf+0x0c	0x35343332	data2

From Table 6, the value of the “data2” variable (an integer with 4-byte size) change from 0 to 35343332_{16} or 392613426_{10} because of the overflowed data from buffer.

II.2.1.2 Heap Overflows

Heap-overflow attacks are similar to stack-overflow attacks but the modified target is the heap memory instead of the stack memory of the process. The heap memory stores function pointers and dynamically-allocated data at runtime, such allocation can be done by calling “malloc” function in C language. Similar to stack-overflow attacks, heap-overflow attacks can modify data by overwriting adjacent memory as attackers wanted. For example, a function pointer is changed to point to an attacker’s code.

II.2.1.3 Array Indexing Errors

Array-indexing-error attacks are different from other types because they do not copy the bigger data than an allocated buffer like stack-overflow and heap-overflow attacks. Array-indexing-error attacks use an array variable declared in the function and index or refer to outside the boundary of this array. One example is a reference to -1 or 10 in a 10-element array. The referred target is any address in the memory that attackers need to alter the value for exploitation.

The c-language code of array-indexing-error attacks is as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int* IntVector;

void bar()
{
  printf("Pongo hack YOU!\n");
}

void InsertInt(unsigned long index, unsigned long value)
{
  printf("Address of index = %p, index = %x\n",&index,index);
  printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n");
  printf("Writing memory at %p\n",&(IntVector[index]));
  IntVector[index] = value;
  printf("Wrote successfully\n");
}
bool InitVector(int size)
{
  IntVector = (int*)malloc(sizeof(int)*size);
  printf("Address of IntVector is %p\n",IntVector);
  if(IntVector == NULL)
    return false;
  else
    return true;
}

int main(int argc, char *argv[])
{
  unsigned long index, value;

  if(argc != 3)
  {
    printf("Usage is %s [index] [value]\n");
    return -1;
  }

  printf("Address of bar = %p\n",bar);

  //Let's initialize our vector - 64 KB ought to be enough for
anyone <g>.
```

```
  if(!InitVector(0xffff))
  {
    printf("Cannot initialize vector!");
    return -1;
  }

  index = atol(argv[1]);
  value = atol(argv[2]);
  InsertInt(index, value);
  return 0;
}
```

From the code, an attacker wants to modify the return address to return to attacker's target function (in this context: the "bar" function). The procedure is as follows:

1. Find the return address of the "InsertInt" function by the following code and calculate distance between the return address and the first element of the declared array "IntVector" containing $ffff_{16}$ or 65535_{10} (4-byte) integers in the "InitVector" function.

```
# ./array_control 1234567890 9876543210
```

The result will be:

```
Address of bar = 0x8048430
Address of IntVector is 0x4010d008
Address of index = 0xbffffb68, index = 499602d2
My stack looks like:
0x401081ec
0xbffffb78
0x80485a5
0x499602d2
0x7fffffff
Writing memory at 0x6668db50
Segmentation fault (core dumped)
```

From the result, the address of the "bar" function is 8048430_{16} or 134513712_{10}. The address of the first element of the array "IntVector" is $4010d008_{16}$. The address of the "InsertInt" function argument named "index" is $bffffb68_{16}$.

When calling a function with many arguments, the arguments and the return address are pushed into stack in respective order starting from the last argument to the first argument and the return address. Note that the return address is next to the first function argument. Therefore, the address of the return address is $bffffb68_{16}$ - 4_{16} = $bffffb64_{16}$. To ease understanding, the stack layout is shown as Table 7.

Table 7 Stack Layout Example

Address	Content	Symbol
0xbffffb6c	0x401081ec	-
0xbffffb70	0xbffffb78	-
0xbffffb74	0x80485a5	return address
0xbffffb78	0x499602d2	index
0xbffffb7c	0x7fffffff	value

2. Calculate the new index to reference the return address from the following equation.

Address of the "index"th element = Address of first element + 4 × Index

Index = (Address of the "index"th element – Address of first element)) ÷ 4

So, New index = (Address of return address – Address of first element) ÷ 4

New index = $(1bffffb64_{16} - 4010d008_{16}) \div 4_{16} = 5ffbcad7_{6} = 1610336983_{10}$

Note: $1bffffb64_{16}$ and $bffffb64_{16}$ in 32-bit architecture are the same value because only 32 bits are contained. The "1" in the 33th bit is ignored.

Using "malloc" function for allocating an array in the heap memory, indexes arrange in ascending order (from low to high address). In the equation, the address of the element is calculated by adding from the the first element.

3. Modify the return address to the address of "bar" function by inputting the first argument with the new index as 1610336983 and the second argument with the address of "bar" function as 134513712. When running the program as below.

```
# ./array_control 1610336983 134513712
```

The result will be:

```
Address of bar = 0x8048430
Address of IntVector is 0x4010d008
Address of index = 0xbffffb68, index = 5ffbcad7
My stack looks like:
0x401081ec
0xbffffb78
0x80485a5
0x5ffbcadb
0x8048430
Writing memory at 0xbffffb64
```

```
Wrote successfully
Pongo hack YOU!
Segmentation fault (core dumped)
```

From the result, although the “bar” function is not called in the code, it can be called at runtime.

This type of buffer-overflow attacks can be modified any address in the memory. Unlike stack overflow, the target address is not necessary to be a higher address. Stack overflows only happen in one direction from low to high address. Thus, they can only modify addresses higher than the address of the stack.

There is another example which can be used as an array-indexing-error attack. That is “Placement New” Expression in C++.

II.2.1.3.1 “Placement New” Expression in C++

According to [7], “placement new” is an expression in C++ language shown as follows:

```
void *operator new (size_t,void *p) throw(){return p;}
void *operator new[](size_t,void *p) throw(){return p;}
```

This expression is used for allocating a dynamically-created object or an array at a given address that refers to a memory area that has already been allocated to the process. A usage example is shown as follows:

```
char *text = new char(10);
//Place newtext at the starting address of ‘‘text’’
char *newtext = new (text) char(10); //uses placement-new
```

If it is misused, it will cause security threats such as buffer-overflow attacks. There are 5 security issues of “placement new” shown as follows:

1. Allows any address allocated to the process to be used to place an object
2. Does not enforce any (compile-time/runtime) bounds checking
3. Invocation does not carry out any type-checking
4. Does not enforce any checking of alignment, it may lead to incorrect semantics, and to program termination
5. May lead to memory leaks

II.2.2 Classification Using Characteristics

Besides being classified using locations, buffer-overflow attacks can also be categorized into 4 classes, according to [8], as follows:

II.2.2.1 Direct Executable Buffer Overflows

The target of direct executable buffer-overflow attacks is to change the control flow of the process. The example of this class is stack-overflow attacks on control data (see in Example 1: Stack-Overflow Attacks on Control Data on page 5) because they alter the return address of the function in order to return to execute attacker's code instead of the normal process flow when the function ends. In this way, there are 5 preconditions defined by [8] as follows:

P1. The length of the (possibly transformed) input string is longer than that of the buffer.

P2. The input (and possibly transformed) string contains **instructions and/or addresses**.

P3. Input can change the stored **return address** without the change being countered.

P4. The program can jump to memory in the **stack**.

P5. The program can **execute** instructions stored in the stack.

II.2.2.2 Indirect Executable Buffer Overflows

Like direct executable buffer-overflow attacks, indirect executable buffer-overflow attacks change the process control flow. However, the difference is that the process state information, such as a return address, is not altered but indirect ones alter a function pointer instead. When the function pointer is invoked, attacker's code will be executed. The similar example of this class is stack-overflow attacks on non-control data (see in Example 2: Stack-Overflow Attacks on Non-Control Data on page 9) assuming that the variable "data2" in this example is a function pointer. Therefore, there are 5 preconditions defined by [8] as follows:

P6. The length of the (possibly transformed) input string is longer than that of the buffer.

P7. The input (and possibly transformed) string contains **addresses**.

P8. Input can change the value in the **function pointer** variable without being countered.

P9. The program can jump to in the **heap**.

P10. The program can **execute** instructions in the heap.

II.2.2.3 Direct Data Buffer Overflows

Data buffer-overflow attacks are different from executable buffer-overflow attacks in that no new instructions (attacker's code) are executed. Direct data buffer-overflow attacks modify some data which make the execution path change. The example of this class is stack-overflow attacks on non-control data (see in Example 2: Stack-Overflow Attacks on Non-Control Data on page 9). This example can be applied to bypass a "Login" process. Consequently, there are 4 preconditions defined by [8] as follows:

P11. The length of the (possibly transformed) input string is longer than that of the buffer.

P12. The input (and possibly transformed) string contains **data** of the type of the particular variable.

P13. The **value** stored in the particular variable can be changed without being countered.

P14. The particular variable **determines which execution path** is to be taken at a future point in the execution of the process.

II.2.2.4 Indirect Data Buffer Overflows

Almost same as direct ones, the target of indirect data buffer-overflow attacks is a pointer referring to the data that can change the execution path. Thereby, there are 4 preconditions defined by [8] as follows:

P15. The length of the (possibly transformed) input string is longer than that of the buffer.

P16. The input (and possibly transformed) string contains **addresses**.

P17. The **address** stored in the particular pointer variable can be changed without being countered.

P18. The value pointed to by the particular pointer variable **determines which execution path** is to be taken at a future point in the execution of the process.

II.2.3 Characteristics

According to [8], some preconditions are the same. As a result, they can be concluded as buffer-overflow characteristics as shown in Table 8.

Table 8 Characteristics and Preconditions [8]

Characteristics	Pseudo-Code	Preconditions
`len:buff`	`len(input) < len(buffer)`	P1, P6, P11, P15
`con:addr`	`contains(input,type(addr))`	P2, P7, P16
`con:inst`	`contains(input,type(inst))`	P2
`con:ctrl`	`contains(input,type(ctrlvar))`	P12
`mod:radd`	`modify(retnadd)`	P3
`mod:fptr`	`modify(funcptr)`	P8
`mod:cvar`	`modify(ctrlvar)`	P13
`mod:cptr`	`modify(ctrlptr)`	P17
`jmp:stack`	`jump(stack)`	P4
`jmp:heap`	`jump(heap)`	P9
`exe:stack`	`execute(stack)`	P5
`exe:heap`	`execute(heap)`	P10
`flow:ctrl`	`flow(ctrlvar)`	P14, P18

In summary, the associated sets of characteristics for 4 classes of buffer-overflow attacks are shown as follows:

II.2.3.1 Direct Executable Buffer Overflows

```
dir:exec =
  {len:buff, con:addr, con:inst, mod:radd, jmp:stack,
exe:stack}
```

This class is equivalent to Stack overflows on control data.

II.2.3.2 Indirect Executable Buffer Overflows

```
ind:exec =
  {len:buff, con:addr, mod:fptr, jmp:heap, exe:heap}
```

This class is equivalent to Heap overflows on control data.

II.2.3.3 Direct Data Buffer Overflows

```
dir:data =
  {len:buff, con:ctrl, mod:cvar, flow:ctrl}
```

This class is equivalent to Stack overflows on non-control data.

II.2.3.4 Indirect Data Buffer Overflows

```
ind:data =
  {len:buff, con:addr, mod:cptr, flow:ctrl}
```

This class is equivalent to Heap overflows on non-control data.

Note that there is no array-indexing error included, the characteristic "**out:buff**" is defined for the precondition of this attack type that there is a reference to outside the buffer boundary.

These buffer-overflow characteristics will be used in the next chapter for the summary (see in Table 13 on page 30).

CHAPTER III
LITERATURE REVIEWS

This chapter provides the overall of current approaches of buffer-overflow protection and related works.

The best solution for preventing buffer-overflow attacks is to write a correct code, such as adding if-condition to check the boundary. However, it makes applications run slower and most programmers ignore this. Sometimes they try to take care of boundary checking, but cannot find all cases. The vulnerabilities still exist and can be exploited. Therefore, researchers have proposed many buffer-overflow protection solutions shown in Figure 1.

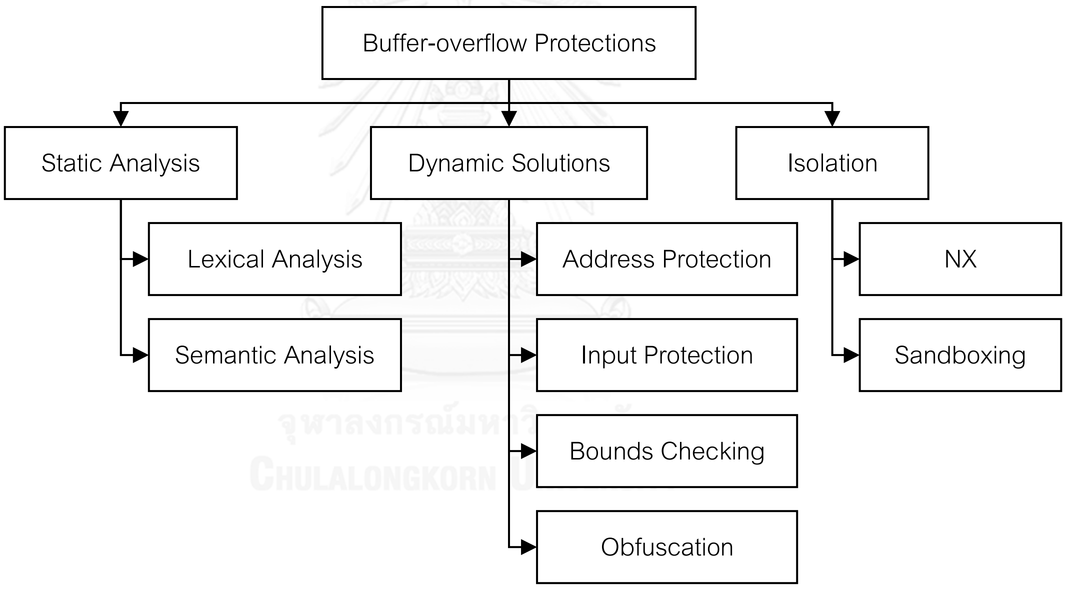

Figure 1 Buffer-Overflow Protections [9]

From the above figure, these solutions are classified into 3 categories [9] as follows.

III.1 Static Analysis

Static analysis approaches are to notify programmers to edit or to replace vulnerable functions or parts of program code that can cause buffer overflow before deployment. For instance, using strcpy() may easily make vulnerability. It should be replaced by strncpy() with bound checking.

Nonetheless, there are some limitations for user-defined functions and macros. The weak points are that they can detect only known attack patterns without runtime information. As a result, it is impossible to find all cases and may cause false alarms. Moreover, the final decision of modifying code (as suggestion from tools) depends on programmers.

This category can be divided into 2 subcategories as follows.

III.1.1 Lexical Analysis

The algorithm is to check in program code in order to find a word or a group of words which may cause buffer overflow. The examples of this approach are ITS4 [10], FlawFinder [11], RATS [12], STOBO [13] and LibSafe [14].

III.1.2 Semantic Analysis

It is different from lexical analysis. It uses parser to analyze the meaning of code instead of a word or a group of words. The examples of this approach are Splint [15] and BOON [16].

III.2 Dynamic Solution

To validate the data integrity in run-time environment, these solutions verify the metadata or data description. The verified data can be control data, such as return addresses, or non-control data, such as variables.

There are 2 major types of metadata: hardware supported and software managed.

These solutions can be divided into 4 subcategories by assumption, types of metadata, metadata management and handling routine as follows.

III.2.1 Address Protection

This subcategory is classified into many schemes with the same assumption that only protects the memory containing addresses. The reason is that addresses are critical data and should be tagged. Metadata will be created by address instruction and verified when that address is used. Each scheme applies various types of metadata. The examples are as follows.

III.2.1.1 Canary Word

The concept is to add 1 word, called Canary Word, in the memory between each address or each pointer. The hypothesis is that buffer overflows only happen in one direction. When the address is modified, its Canary Word should be modified as well. Then the value of Canary Word must be verified before using that address as shown in Figure 2.

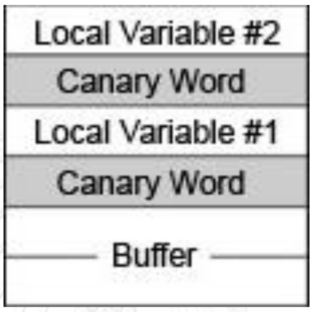

Figure 2 Stack Layout of Canary Word [6]

The weakness of this scheme is that it can easily be bypassed by retaining the same value of Canary Word after buffer-overflow attacking. It cannot confirm whether address is modified because no mechanism for protecting Canary Word itself as shown in Table 9.

Table 9 Stack Layout for Bypassing Canary Word

Before attacking	**Type**	Buffer			Canary Word	Pointer
	Value	-	-	-	0	5
After attacking	**Type**	Buffer			Canary Word	Pointer
	Value	A	A	A	0	A

The examples of this scheme are StackGuard [15, 17-20], which protects return addresses, and ProPolice [21], which protects function pointers by declaration statement.

III.2.1.2 Address Encoding

This scheme encodes addresses for the integrity of the addresses. The concept is to encode addresses before storing in the memory and decode them when reading back to the processor. The metadata of this scheme is the per-process random key for encryption. Thus, the long-term key management is the main point. However, there are some problems with array and string in C language and value assignment in the compile time.

The examples of this scheme are PointGuard [22] and Hardware Supported PointGuard [23, 24]. These solutions assume that when pointers are created, their value will not change.

III.2.1.3 Copy of Address

This scheme copies an address for the integrity of the addresses. When creating an address, copies it and stores the copy safely. Before the address is used, it should be verified with its copy.

This scheme is classified into many methods by the address copy management. The examples of this scheme are as follows: 1. StackGhost [25], using register window of SPARC processor, 2. RAS [26-28], using return address stack (the hardware for predicting return addresses in some processors), 3. Split Stack [27], SmashGuard [29], RAD Compiler [30], RAD Binary Rewrite [31], DISE [32], StackShield [33] and LibVerify [15], allocating memory as return address stack, and 4. SCACHE [34], using cache for managing a copy of return address.

III.2.1.4 Tags

Tags are for identifying type of data, such as normal data or address. The limitation is that they cannot apply on applications without modification for this scheme.

The example of this scheme is Tagged Architecture [35].

III.2.1.5 HeapDefender [36]

HeapDefender is a fine-grained instruction stream monitoring hardware defense mechanism. This hardware module is located inside of embedded processor and works in parallel with processor pipeline. Its concept is to extract a pre-defined heap memory range and check whether address of heap operations of instructions exceeds heap address range. Thus, this scheme neither modifies program nor destroys pipeline integrity. Also, it is transparent to both processor and software programmer. However, it protects embedded processors against heap overflow attacks, in other words, it protects heap memory only.

III.2.2 Input Protection

This subcategory is classified into many schemes with the same assumption that input data should not be used as control data. Therefore, there must be some differences between input data and control data. Although there is no way to recognize data as control data or non-control data in the hardware level, it can be done in the programmer/compiler level. Each scheme applies various metadata management with different implementation. The examples are as follows.

III.2.2.1 Secure Bit [37]

The concept is to add 1 bit, called Secure Bit, to each byte/word in the memory. This bit is used for identifying that input data are from outside the process via kernel. If Secure Bit is set, it contains input data which should not be used as control data, e.g. return addresses, as shown in Figure 3 and Table 10.

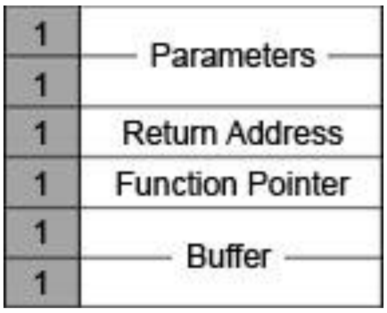

Figure 3 Stack Layout of Secure Bit

Table 10 Stack Layout for Secure Bit

Before attacking	Type	Buffer					Return Address
	Value	-	-	-	-	-	5
	Secure Bit	0	0	0	0	0	0
After attacking	Type	Buffer					Return Address
	Value	A	A	A	A	A	A
	Secure Bit	1	1	1	1	1	1

In addition, Secure Bit is transparent. Software developers do not need to edit or compile software again for applying Secure Bit. Even though the detection mechanism of buffer-overflow attacks is embedded in the hardware level, this scheme is not able to protect non-control data. For instance, there are 2 variables: **a** and **b**. First, variable **b** is assigned as 5 and the Secure Bit of variable **b** is set. When overflowing variable **a** with "AAAAA", the value of variable **b** will be 0. In this case, it cannot detect this buffer-overflow attack because the Secure Bit of variable **b** is still set as shown in Table 11.

Table 11 Stack Layout of the Undetected Case for Secure Bit

Before attacking	Type	a					b
	Value	-	-	-	-	-	5
	Secure Bit	0	0	0	0	0	1
After attacking	Type	a					b
	Value	A	A	A	A	A	\0
	Secure Bit	1	1	1	1	1	1

III.2.2.2 Minos [38] [39]

Its concept is similar to Secure Bit but the input data are from another segment, using segmentation as a boundary. The weak point is that segmentation does not exist on all systems and it is not transparent, unlike Secure Bit. As a result, this scheme will be hard to implementation.

III.2.2.3 Tainted Pointer [40]

The objective of this scheme is to prevent input data to be used as pointers. The input data are from I/O subsystem of operating system. However, sometimes input data may be used for calculating pointer, such as indexing, and there are some problems with multi-threaded program which sending address values between threads in the process. Consequently, the instruction for clearing taint values of pointers is needed and this could be another vulnerability.

III.2.2.4 Efficient Dynamic Taint Analysis Using Multicore Machines [41]

This scheme uses static binary rewriting to transform a binary to contain an original thread and a shadow thread for taint computation. These two threads are executed on different processor cores. Dynamic taint analyses start with marking the values from external sources as untrusted, i.e., tainted. Then, taint values are propagated as its rules. Finally, this scheme uses taint values to detect possible exploits.

III.2.3 Bound Checking

This subcategory is classified into many schemes with the same assumption that access to data should be within variable boundary only. The metadata is related to every block of the allocated memory and is used to limit the boundary. The examples are as follows.

III.2.3.1 Hardware

This scheme uses segmentation with base address for boundary checking. Segmentation and ring is implemented on I432 processor [42]. The weak point is that mostly operating systems avoid using segmentation by setting all memory as 1 big segment in order to work with this processor and improve efficiency.

The reason is that its processing time is more than 10-20 times as shown on VAX 11/780 [43].

III.2.3.2 Software

This scheme is backward compatible with standard C library but its weakness is high overhead making program run slower.

The examples of this scheme are as follows: 1. Array Bound Checking [44], defining that pointer value is valid only for the specific memory region but causing program slow down more than 30 times, 2. Rational PurifyPlus [45], BoundsChecker [46], SafeC [47] and Fail-Safe [48, 49], segmentation by using symbol table as segment descriptor.

III.2.4 Obfuscation

The concept is that confusion makes it harder to attack, such as Address Obfuscation [50]. The weak point is that vulnerabilities still exist.

The example of this subcategory is PAX [51] or Address Space Layout Randomization (ASLR).

III.2.5 Mixed Solution

Some solutions are mixed from above schemes.

III.2.5.1 Secure Canary Word [6]

Secure Canary Word is an architectural approach based on two existing schemes, Secure Bit (Input Protection) and Canary Word (Address Protection), for protecting against buffer-overflow attacks on non-control data (variables/arguments). Canary Word is inserted between each variable/argument to protect non-control data. Then, Secure Bit is used to protect Canary Word as control data. Its stack layout is shown in Figure 4 and Table 12. However, it cannot prevent array indexing errors on non-control data because it is no need to write canary word to bypass these attacks.

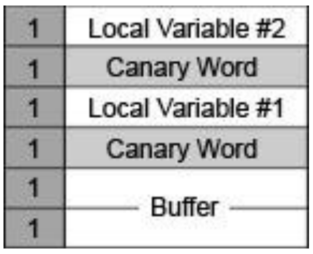

Figure 4 Stack Layout of Secure Canary Word

Table 12 Stack Layout for Secure Canary Word

Before attacking	Type	Buffer					Canary Word	Pointer
	Value	-	-	-	-	-	0	5
	Secure Bit	0	0	0	0	0	0	0
After attacking	Type	Buffer					Canary Word	Pointer
	Value	A	A	A	A	A	0	A
	Secure Bit	1	1	1	1	1	1	1

III.3 Isolation

These approaches confine attack damage instead of preventing, such as confining the application not to run outside the defined scope. This category can be divided into 2 subcategories as follows.

III.3.1 Non-Executable Memory

The assumption of this solution is from the observation that buffer-overflow attacks usually occur on Stack or memory storing data. Thus, Stack memory should not be used for storing program code. This solution, called NX, is implemented on many processors [52].

The weakness is that not all attacks are code injection. This solution prevents only code injection attacks.

III.3.2 Sandboxing

Sandboxing solution uses policy-enforcement mechanism which is to run untrusted applications in the restrict environment. For instance, application downloaded from the internet must not edit system files or call some operating-system API. Therefore, if they are attacked, the damages will be limited and not effect on the overall system. The example is Java Applet that runs in Sandbox. It cannot read or write any file in user computers.

Furthermore, this solution can be implemented on any level, such as kernel level [53], user level [54-56] and hardware-supported level, e.g. Intel LaGrande [57], TCPA [58, 59], TrustZone [60], Microsoft NGSCB [61], ChipLock [62] and Bear [58].

The success of this solution depends on a proper combination of security policy and implementation.

III.4 Comparison and Summary

From the taxonomy of buffer-overflow characteristics (section II.2.3 Characteristics on page 17) and some buffer-overflow protection solutions, the summary table can be shown as Table 13. The symbol "✓" means this solution can prevent this characteristic. The symbol "?" means this solution may prevent this characteristic.

From Table 13, which some details are based on [8], it can be summarized with types of buffer-overflow attacks as Table 14.

Segmentation, Type-Assisted Buffer Overflow Detection [49] and C Range Error Detector (CRED) [63] are range-checking solutions at runtime. These prevent len:buff characteristic and may prevent out:buff characteristic, up to the implementation. As a result, they can prevent stack/heap overflow attacks on both control and non-control data. It may also prevent array indexing error attacks on both control and non-control data.

Integer Analysis to Determine Buffer Overflow [16] and STOBO [13] check the range of the buffer and the input. Their designs are not suitable for checking the index of the buffer/array. Both prevent only len:buff characteristic. As a result, they can prevent stack/heap overflow attacks on both control and non-control data. They cannot prevent array indexing error attacks.

Jump Pointer Control [23] is a hardware/software address protection. It handles function pointers and pointer variables. It prevents mod:fptr and mod:cptr characteristics. As a result, it can prevent stack/heap overflow attacks on non-control data (indirect executable and indirect data buffer overflows). It also prevents jmp:stack characteristic by return-address bound-checking on the stack. Thus, it can prevent stack overflow attacks on control data (direct executable buffer overflows) as well.

StackGuard [20] is a return-address protection using canary word. It prevents mod:radd characteristic. As a result, it can prevent only stack overflow attacks on control data.

PointGuard [22] is similar to StackGuard. It prevents mod:fptr and mod:cptr characteristics. As a result, it can prevent heap overflow attacks on both control and non-control data (indirect executable and indirect data buffer overflows).

SmashGuard [29] is an address protection by address copying. Minezone RAD and Read-only RAD [30] are an address protection schemes using write-protected location. They prevent mod:radd characteristic. As a result, they can prevent stack overflow attacks on control data. However, they may prevent array indexing error attacks on control data, up to the implementation.

MemGuard [20] protects specific memory locations. It prevents mod:radd, mod:fptr, mod:cvar and mod:cptr characteristics. As a result, they can prevent stack/heap overflow attacks on both control and non-control data. However, they may prevent array indexing error attacks on both control and non-control data.

Secure Bit [37] and Efficient Dynamic Taint Analysis Using Multicore Machines [41] are input protection solutions. They prevent mod:radd and mod:fptr characteristics. As a result, they can prevent all attacks on control data.

HeapDefender [36] is an address protection on heap. It prevents jmp:heap characteristic. As a result, it can prevent heap overflow attacks on both control and non-control data. It may prevent len:buff and out:buff characteristics but it still no proof.

Secure Canary Word [6] is an architectural approach based on Secure Bit and Canary Word. It prevents mod:radd, mod:fptr, mod:cvar and mod:cptr characteristics. As a result, it can prevent stack/heap overflow attacks on both control and non-control data. It can also prevent array indexing error attacks on control data. However, it cannot prevent array indexing error attacks on non-control data.

Boundary Bit is our bound-checking solution. It designs to prevent both len:buff and out:buff characteristics. We believe that it can prevent all buffer-overflow attacks on both control and non-control data.

Table 13 Summary with Buffer-Overflow Characteristics

Characteristics	len: buff	con: addr	con: inst	con: ctrlr	mod: radd	mod: fptr	mod: cvar	mod: cptr	jmp: stack	jmp: heap	exe: stack	exe: heap	flow: ctrl	out: buff
Segmentation	✓													?
Integer Analysis to Determine Buffer Overflow	✓													
STOBO	✓													
Type-Assisted Buffer Overflow Detection	✓													?
C Range Error Detector (CRED)	✓													?
Jump Pointer Control						✓		✓	✓					
StackGuard					✓									
MemGuard					✓	✓	✓	✓						
PointGuard						✓		✓						
SmashGuard					✓									
Minezone RAD					✓									
Read-only RAD					✓									
Efficient Dynamic Taint Analysis Using Multicore Machines					✓	✓								
HeapDefender	?									✓				?
Secure Bit					✓	✓								✓
Secure Canary Word					✓	✓	✓	✓						
Boundary Bit	✓													✓

In conclusion, the buffer-overflow protection summary table with types of buffer-overflow attacks can be provided as Table 14. The symbol "✓" means this solution can prevent this attack type. The symbol "?" means this solution may prevent this attack type.

Table 14 Summary with Types of Buffer-Overflow Attacks

Types	Stack overflow on control data (Direct Executable)	Stack overflow on non-control data (Direct Data)	Heap overflow on control data (Indirect Executable)	Heap overflow on non-control data (Indirect Data)	Array indexing error on control data	Array indexing error on non-control data
Segmentation	✓	✓	✓	✓	?	?
Integer Analysis to Determine Buffer Overflow	✓	✓	✓	✓		
STOBO	✓	✓	✓	✓		
Type-Assisted Buffer Overflow Detection	✓	✓	✓	✓	?	?
C Range Error Detector (CRED)	✓	✓	✓	✓	?	?
Jump Pointer Control	✓		✓	✓		
StackGuard	✓					
MemGuard	✓	✓	✓	✓	?	?
PointGuard			✓	✓		
SmashGuard	✓				?	
Minezone RAD	✓				?	
Read-only RAD	✓				?	
Efficient Dynamic Taint Analysis Using Multicore Machines	✓		✓		✓	
HeapDefender			✓	✓		
Secure Bit	✓		✓		✓	
Secure Canary Word	✓	✓	✓	✓	✓	
Boundary Bit	✓	✓	✓	✓	✓	✓

However, Table 14 shows only which types of buffer-overflow attacks can be prevented but does not show the performance and the limitation of the protection solutions.

CHAPTER IV
THE CONCEPT OF BOUNDARY BIT

From the CHAPTER III, most of the buffer-overflow solutions focus on control data. They do not pay attention to array-indexing error that is also one of the buffer-overflow attacks. Therefore, this research proposes a new solution, namely Boundary Bit, to not only protect both control and non-control data, but also to prevent array-indexing error. This chapter explains the concept of Boundary Bit.

IV.1 The Concept of Boundary Bit

The main concept of Boundary Bit is bound-checking. The idea is to ensure that transferring data do not exceed the allocated capacity of variables or buffers. That is, we attempts to prevent **`len:buff`** and **`out:buff`** characteristics of buffer-overflow attacks. We aim to provide a complete solution against all types of buffer-overflow attacks.

Due to the fact that software approaches cause very high overhead, hardware approaches could be a better choice. The proposed solution uses an additional hardware bit (called Boundary Bit) associated to each byte or word in the memory for marking the end of variables or buffers. Due to memory alignment, marking the end of buffer provides a better protection.

Programmers or compilers have to tell the system to set a bit at the end of any variable or buffer to mark the boundary. At the runtime, when writing data to any variable or buffer, the system will check whether there is a boundary bit set in a given range. Assuming that the input size is **`n`** bytes (the maximum index is **`n - 1`**). If the buffer starts at the address **`st`**, the range of scanned bits will start from **`st`** to **`st + n - 2`**. This is to avoid the case of the 1-byte variable where scanning should stop before the end of the variable or buffer. To ease understanding, an example is provided.

IV.2 Examples

In this example, a c-language code is provided. In the code, a function contains a 4-byte integer, an 8-byte buffer and a 1-byte character.

```
void func(char *p) {
  int i; //4 byte
  char b[8]; //8 byte
  char ch; //1 byte
  …
}
```

From the given code, the stack memory layout while allocating memory is shown in Table 15.

Table 15 Stack Layout of Boundary Bit while Allocating Memory

Address		0x28ac57	0x28ac58						0x28ac5e	0x28ac5f	0x28ac64			0x28ac67	⋮
Before allocating	Type	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	Boundary Bit	0	0	0	0	0	0	0	0	0	0	0	0	0	…
After allocating	Type	Char	Buffer								Integer				…
	Boundary Bit	1	0	0	0	0	0	0	0	1	0	0	0	1	…

Some addresses are not continuous owing to memory alignment.

IV.2.1 Stack-Overflow Detection

This section demonstrates stack-overflow detection. In this code, a strcpy() function is given.

```
void func(char *p) {
  int i; //4 byte
  char b[8]; //8 byte
  char ch; //1 byte
  strcpy(b,p);
}
```

If the length of input is 8 bytes, the system will scan bits starting at address **0x28ac58** ending at address **(0x28ac58 + 8 – 2) = 0x28ac5e**. There is no Boundary Bit set in this range.

If the length of input is 9 bytes, the system will scan bits starting at address **0x28ac58** ending at address **(0x28ac58 + 9 – 2) = 0x28ac5f**. In this case, there is, however, a Boundary Bit set at address **`0x28ac5f`**. Thus, the attack will be detected with no false positive or false alarm.

IV.2.2 Array-Indexing-Error Detection

In this section, an array-indexing-error detection will be demonstrated. The following code shows a case where a reference is inside the boundary (no array-indexing error).

```
void func(char *p) {
  int i; //4 byte
  char b[8]; //8 byte
  char ch; //1 byte
  b[7] = p[0];
}
```

If the index is 7, **`n = 8 (n - 1 = 7)`**, the system will scan bits starting at address **`0x28ac58`** ending at address **`(0x28ac58 + 8 - 2) = 0x28ac5e`**. There is no Boundary Bit set in this range.

The following code shows a case where a reference is outside the boundary (accessing with array-indexing error).

```
void func(char *p) {
  int i; //4 byte
  char b[8]; //8 byte
  char ch; //1 byte
  b[8] = p[0];
}
```

If the index is 8, **`n = 9 (n - 1 = 8)`**, the system will scan bits starting at address **`0x28ac58`** ending at address **`(0x28ac58 + 9 - 2) = 0x28ac5f`**. In this case, there is a Boundary Bit set at address **`0x28ac5f`**. Thus, the attack will be detected with no false positive or false alarm.

IV.2.3 One-Byte Variable

In case of attacking 1-byte variables, both Stack-overflow and Array-indexing-error attacks will be explained as follows.

IV.2.3.1 Stack-Overflow Detection

Supposing that the function has a 7-byte buffer and two 1-byte characters, the c-language code is provided.

```
void func(char *p) {
  char b[7]; //7 byte
  char ch1; //1 byte
  char ch2; //1 byte
  strcpy(&ch1,p);
}
```

From the code, the stack memory layout while allocating memory is shown in the following Table 16.

Table 16 Stack Layout of Boundary Bit in case of 1-byte Variables

Address		0x28ac57	0x28ac58	0x28ac59						0x28ac5f	0x28ac60	⋮
Before allocating	Type	-	-	-	-	-	-	-	-	-	-	-
	Boundary Bit	0	0	0	0	0	0	0	0	0	0	…
After allocating	Type	Char	Char	Buffer								…
	Boundary Bit	1	1	0	0	0	0	0	0	0	1	…

If the input size is 1 byte, the system will scan bits starting at address **`0x28ac58`** ending at address **`(0x28ac58 + 1 – 2) = 0x28ac57`**. Nonetheless, **`0x28ac57`** is less than **`0x28ac58`**. It can be concluded that there is no attack shown as no false negative.

If the input size is 2 bytes, the system will scan bits starting from address **`0x28ac58`** to address **`(0x28ac58 + 2 – 2) = 0x28ac58`**. In this case, there is a Boundary Bit set at address **`0x28ac58`**. Thus, the attack will be detected with no false positive or false alarm.

IV.2.3.2 Array-Indexing-Error Detection

Supposing that the function has a 1-byte buffer and a 1-byte character, the c-language code is as follows:

```
void func(char *p) {
  char ch1; //1 byte
  char b[1]; //1 byte
  char ch2; //1 byte
  …
}
```

From the code, the stack memory layout while allocating memory is shown in the following Table 17.

Table 17 Stack Layout of Boundary Bit in case of 1-byte Buffer

Address		0x28ac57	0x28ac58	0x28ac59	⋮
Before allocating	Type	-	-	-	-
	Boundary Bit	0	0	0	…
After allocating	Type	Char	Buffer	Char	…
	Boundary Bit	1	1	1	…

In case of referencing inside the boundary (no array-indexing error), the code is as follows:

```
void func(char *p) {
  char ch1; //1 byte
  char b[1]; //1 byte
  char ch2; //1 byte
  b[0] = p[0];
}
```

If the index is 0, **`n = 1 (n – 1 = 0)`**, the system will scan bits starting at address **`0x28ac58`** ending at address **`(0x28ac58 + 1 – 2) = 0x28ac57`**. Nonetheless, **`0x28ac57`** is less than **`0x28ac58`**. It can be concluded that there is no attack shown as false negative.

In case of referring to outside the boundary with Array Indexing Error, the code is as follows:

```
void func(char *p) {
  char ch1; //1 byte
  char b[1]; //1 byte
  char ch2; //1 byte
  b[1] = p[0];
}
```

If the index is 1, **`n = 2 (n - 1 = 1)`**, the system will scan bits starting at address **`0x28ac58`** ending at address **`(0x28ac58 + 2 - 2) = 0x28ac58`**. In this case, there is a Boundary Bit set at address **`0x28ac58`**. Thus, the attack will be detected with no false positive or false alarm.

CHAPTER V
THE IMPLEMENTATION OF BOUNDARY BIT

The implementation of Boundary Bit is stated in this chapter. The guideline is divided into 2 parts: hardware and software.

V.1 Hardware

Two modifications are necessary to implement Boundary Bit.

1. An additional hardware bit associated to each byte or word in the memory will be added as boundary bit for marking the end of variables or buffers.
2. A processor will be modified by
   a) adding a new instruction to set/clear boundary bit when memory is allocated/deallocated and
   b) adding a new instruction or modified memory-written instructions with boundary bit checked.

However, the performance of bit-scanning is a concern. To efficiently check for boundary, a possible solution is to implement the boundary-bit cache (bitmap). The benefits are not only faster access time, but also various bit representation for the large amount of bit scanning. To ease understanding, we will elaborate on the details of this bitmap.

V.1.1 Boundary Bits and Bitmap

For write through cache, Store (memory-written) instruction is much slower than other instructions. To make it faster, several architectures introduce a write buffer between processor and memory. To write to memory, a processor can simply write data to the write buffer. The write buffer will then write to memory during later in background. This is useful in the pipeline processor. To avoid the saturation, the second-level (L2) cache is usually used with the write buffer as shown in Figure 5.

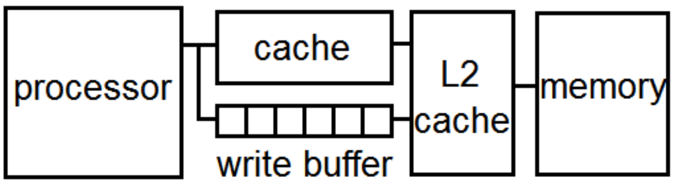

Figure 5 Write Buffer and L2 Cache Architecture

To improve the performance of boundary scan, we can add boundary-bit cache (bitmap) at the same level of L2 cache. Moreover, this also allows the bit-scanning process to be executed in parallel with the memory-written process.

The boundary-bit cache can be implemented as a bitmap to store the pack of boundary bits of the nearby memory locations. When scanning the pack of boundary bits, this behavior provides more spatial locality[1] than temporal locality[2] [64].

For the **`n-to-1`** bitmap, a bit in boundary-bit cache can represent the **`n`** boundary bits of the address **`A`** to **`A + n - 1`**. If there is a boundary bit set in the range, it will be represented as “1” in the bitmap (an OR of associated bits). For example, “01000000” bits in the Boundary Bit section can be represented as “1” in the **`8-to-1`** bitmap of boundary-bit cache, as shown in Figure 6. The bitmap can be implemented as multilevel of **`n-to-1`** bitmap to improve the scanning speed of larger memory range.

The concepts are to manage the up-to-date bitmap with a low overhead and to balance the size of n that is suitable for the scan of boundary bits.

[1] Spatial locality is a situation where a nearby reference memory is likely to be referenced in the near future.

[2] Temporal locality is a situation where a recently referenced memory is likely to be referenced again in the near future.

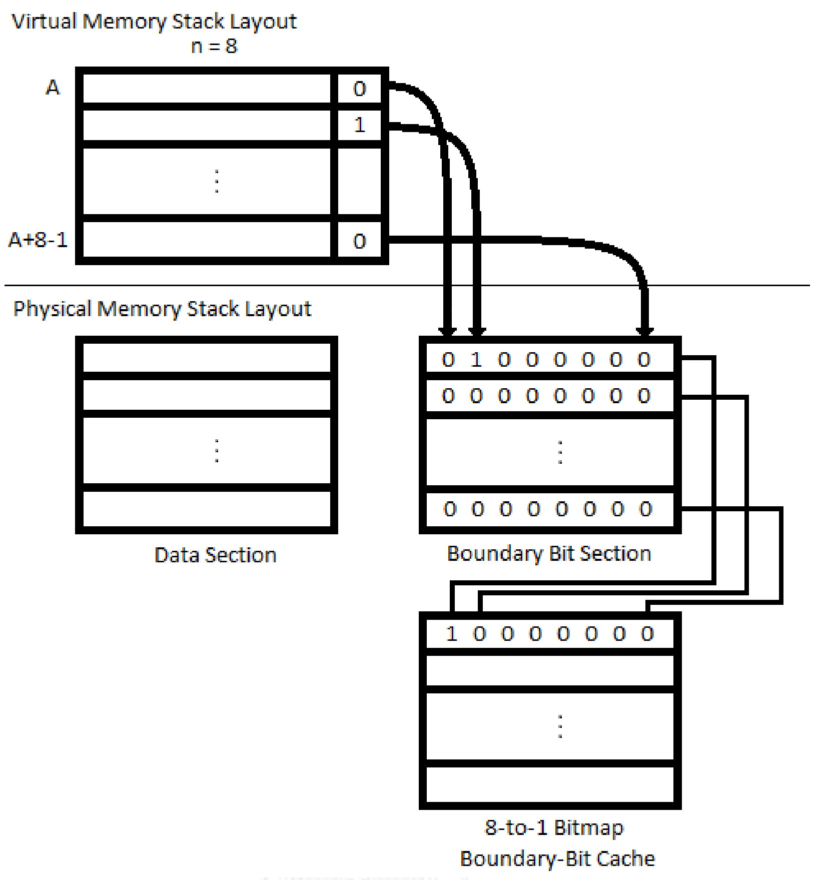

Figure 6 Boundary-Bit Bitmap Diagram

V.1.2 Memory Architecture

Our boundary bit concept from V.1.1 can be designed as shown in Figure 7, Figure 8, Figure 9 and Figure 10. To make the implementation simple, this design stores boundary bits separated from normal data in the traditional memory without adding an additional hardware bit associated to each byte or word in the memory.

In Figure 9 and Figure 10, bitmap (boundary bit cache) will reduce boundary-bit scan cycles by trading off more hardware and bitmap management cycles. Bitmap can be added more than 1 level, such as in Figure 11. However, it wastes not more than 1 cycle to manage bitmap, when setting/clearing boundary bit bitmap, because it can be parallel with boundary bit management.

Moreover, this memory architecture leads to boundary bit protection. If attackers need to modify any boundary bit, they must have a root privilege to access memory in the boundary bit section.

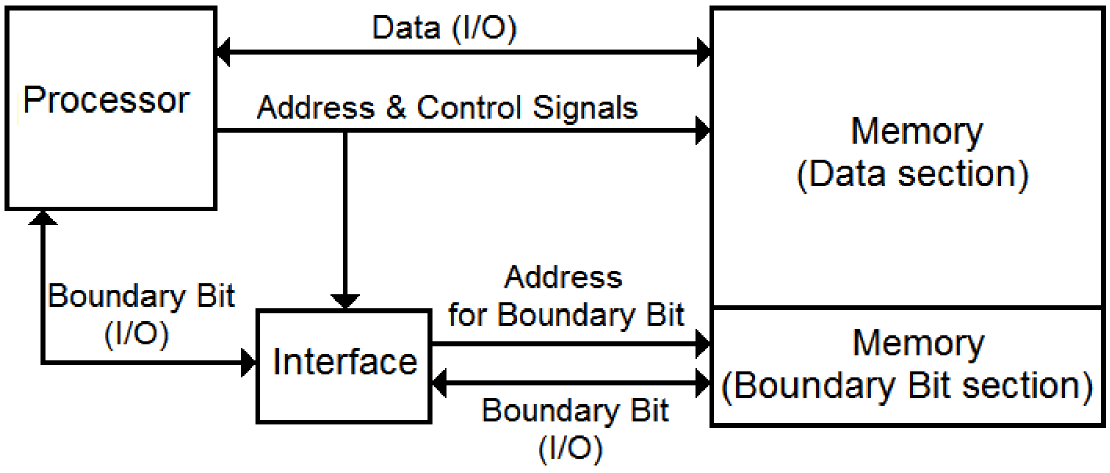

Figure 7 Boundary Bit Memory Interface without Bitmap

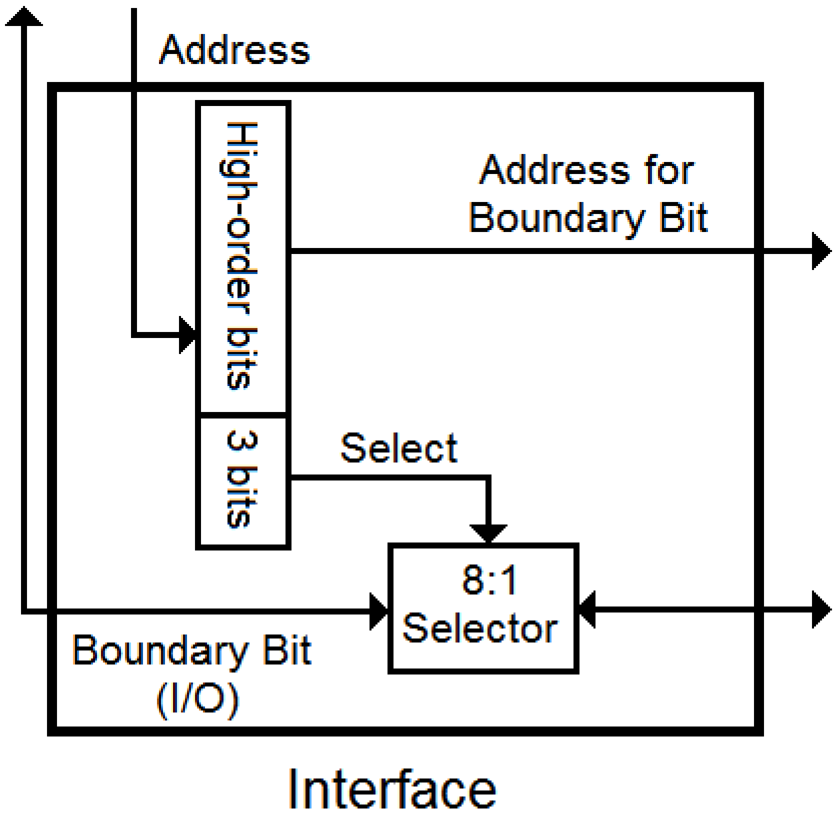

Figure 8 Boundary-Bit Memory Interface Controller

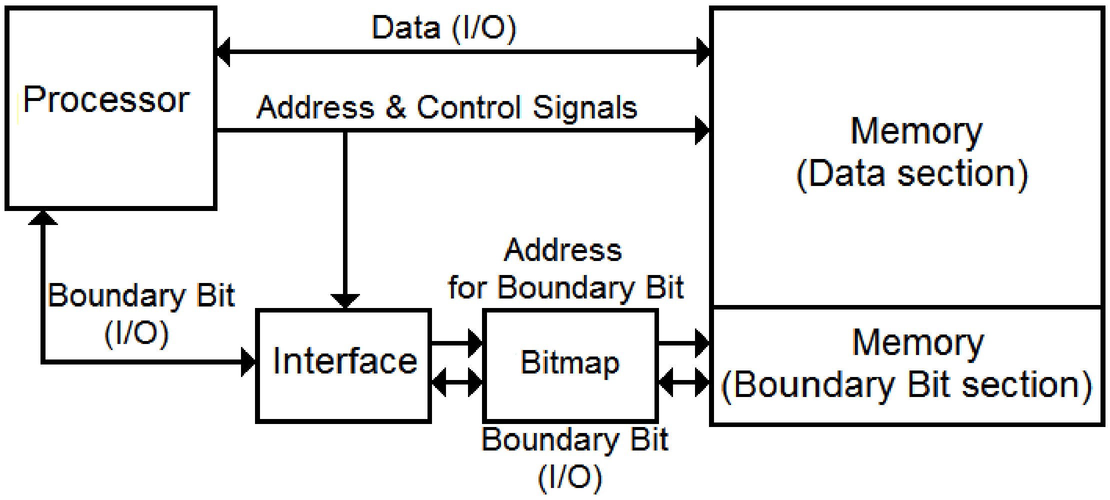

Figure 9 Boundary-Bit Memory Interface with Bitmap

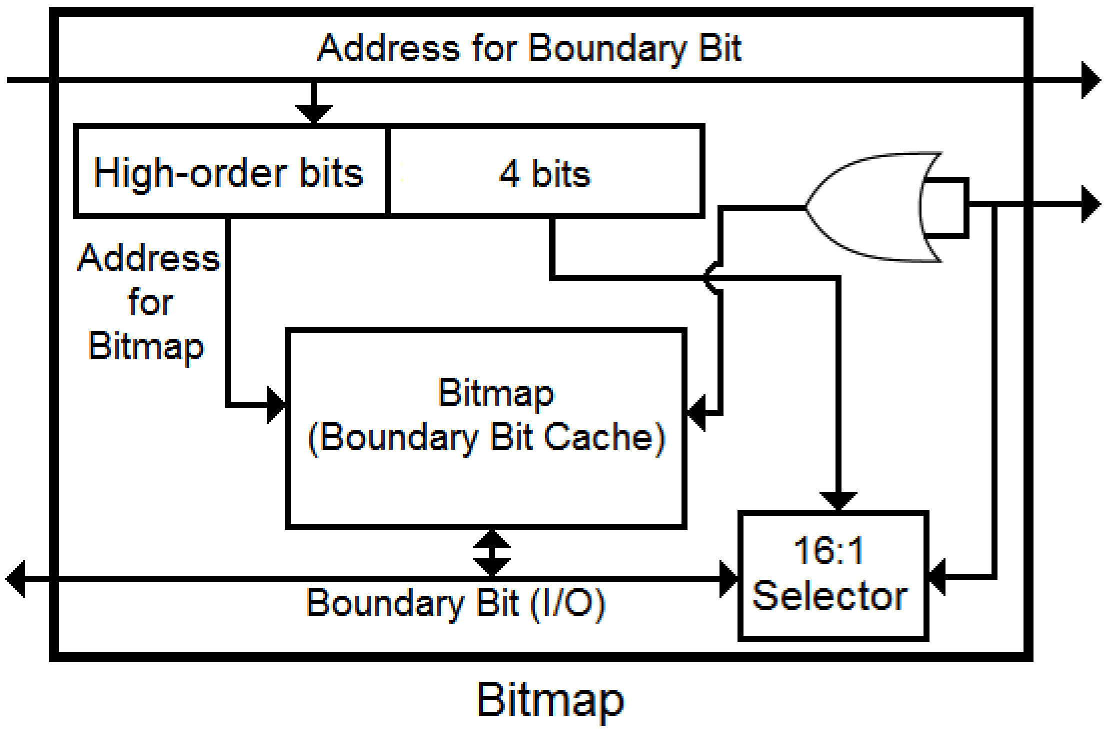

Figure 10 Bitmap Interface Controller

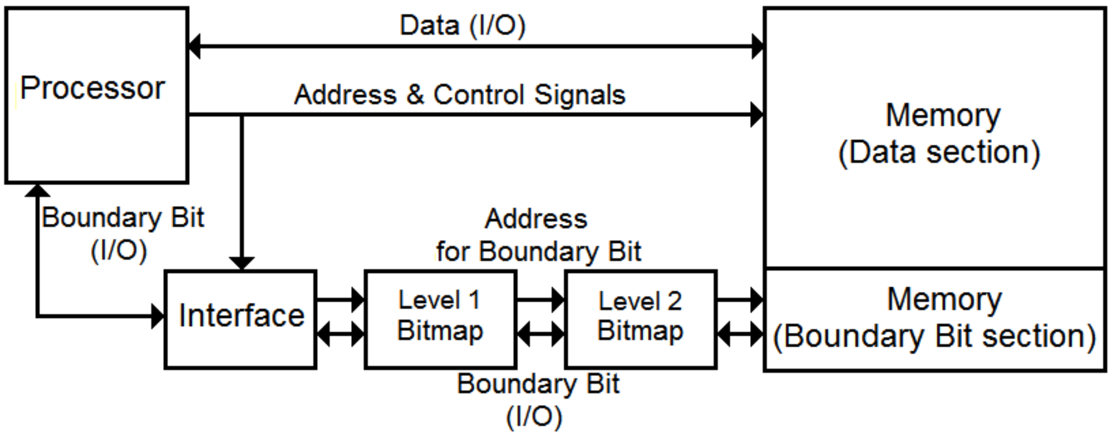

Figure 11 Boundary-Bit Memory Interface with 2-Level Bitmap

V.1.3 Instructions for Boundary Bits

The first instruction, **`setbb`**, is for setting boundary bit. It is required for memory allocation. The operand of the **`setbb`** instruction is the address of the **last byte** of each variable, argument or pointer (not the address of that). The instruction format is as follows:

```
setbb ADDRESS
```

The second instruction, **`clrbb`**, is for clearing boundary bit. It is required for memory deallocation. The operand of the **`clrbb`** instruction is the same as **`setbb`** instruction. The instruction format is as follows:

```
clrbb ADDRESS
```

The third new instruction, **`scnbb`**, is for scanning boundary bits. The first operand is the start address of each variable, argument or pointer. The second operand is the value of scanning range which is calculated from variable size or index of array Assuming that the scanning range is **`n`**, if the starting address is **`st`**, the range of scanned bits will start from **`st`** to **`st + n - 2`**. The instruction format is as follows:

```
scnbb ADDRESS,N
```

V.2 Software

Besides modifying some hardware to use boundary bit, the system must be informed to set a boundary bit at the end of any variable or buffer to mark the boundary by using the new memory-allocated instructions. When writing data to any variable or buffer, the system will check whether there is a boundary bit set in a given range by using the new memory-written instructions at the runtime. The new instructions can be called by in-line assembly code.

For example, the given c-language code will produce the following assembly code.

```
void func(char *p) {
  int i; //4 byte
  char b[8]; //8 byte
  char ch; //1 byte
  b[7] = p[0];
}
```

Assembly code (generated from Microsoft Visual Studio compiler) is shown as follows:

```
00000000 push ebp
00000001 mov ebp,esp
00000003 sub esp,1Ch
00000006 xor eax,eax
00000008 mov dword ptr [ebp-14h],eax
0000000b mov dword ptr [ebp-18h],eax
0000000e mov dword ptr [ebp-4],9DC86C9Bh
00000015 mov dword ptr [ebp-10h],ecx
00000018 cmp dword ptr ds:[01474288h],0
0000001f je 00000026
00000021 call 5B013C60
00000026 mov eax,dword ptr [ebp-10h]
00000029 movsx eax,byte ptr [eax]
0000002c lea edx,[ebp-0Ch]
0000002f mov byte ptr [edx+7],al
```

When setting boundary bits, the new instruction, **setbb**, will be added in the memory-allocated section. It must know the size of all variables, argument or pointer before allocating memory to calculate the address of their boundary bits.

Therefore, this part of the assembly code will be:

```
…
mov dword ptr [ebp-14h],eax
setbb dword ptr [ebp-14h]     ; for ch
mov dword ptr [ebp-18h],eax
setbb dword ptr [ebp-15h ]    ; for i
setbb dword ptr [ebp-5]       ; for b[]
mov dword ptr [ebp-4],9DC86C9Bh
mov dword ptr [ebp-10h],ecx
setbb dword ptr [ebp-0Dh]     ; for *p
…
```

When scanning boundary bits, the new instruction, **scnbb**, will be added before writing to memory, such as **mov** instruction with memory as destination (first operand).

Therefore, this part of the assembly code will be:

```
…
mov eax,dword ptr [ebp-10h] ; Get address of *p
movsx eax,byte ptr [eax]    ; Read value of p[0] to eax
lea edx,[ebp-0Ch]           ; Get address of b[0]
scnbb [edx],8               ; Scan Boundary Bit of b[]
mov byte ptr [edx+7],al     ; Write value from eax to b[7]
…
```

From the above example, the **mov** instruction with displacement addressing mode can be embedded with boundary-bit scanning. There is the index number of array in this instruction that can be used to calculate the scanning range.

CHAPTER VI
EVALUATION

To evaluate, we create some boundary bit simulations. Then, we evaluate boundary-bit approach in 2 aspects: protection efficiency and performance.

VI.1 Simulation Tools

Our C++ simulation tools are created on Microsoft Visual Studio 2015. There are 2 main parts of these tools.

VI.1.1 Trace File Generation

At the beginning, we generate some trace files from our benchmark manually. There are 5 instructions that we are interested:

VI.1.1.1 Set Boundary Bit Instruction

This **setbb** instruction is for setting boundary bit. It is required for memory allocation. The operand is the address of the last byte of each variable, argument or pointer (not the address of that). The instruction format is as follows:

```
setbb ADDRESS
```

For example, this instruction in the assembly code is shown as follows:

```
setbb dword ptr [ebp-5]
```

In the trace file, we represent this instruction as follows:

```
B ADDRESS
```

Note that **ADDRESS** is hexadecimal. For example, this instruction in the trace file is shown as follows:

```
B 11E76B
```

From the above code, it means to set boundary bit on address **0x11E76B**.

VI.1.1.2 Clear Boundary Bit Instruction

This **clrbb** instruction is for clearing boundary bit. It is required for memory deallocation. The operand is the address of the last byte of each variable, argument or pointer (not the address of that). The instruction format is similar to **setbb** instruction as follows:

```
clrbb ADDRESS
```

For example, this instruction in the assembly code is shown as follows:

```
clr dword ptr [ebp-5]
```

In the trace file, we represent this instruction as follows:

```
C ADDRESS
```

Note that **`ADDRESS`** is hexadecimal. For example, this instruction in the trace file is shown as follows:

```
C 11E76B
```

From the above code, it means to clear boundary bit on address **`0x11E76B`**.

VI.1.1.3 Scan Boundary Bit Instruction

The **`scnbb`** instruction is for scanning boundary bits. The first operand is the start address of each variable, argument or pointer. The second operand is the value of scanning range which is calculated from variable size or index of array Assuming that the scanning range is **`n`**, if the starting address is **`st`**, the range of scanned bits will start from **`st`** to **`st + n - 2`**. The instruction format is as follows:

```
scnbb ADDRESS,N
```

For example, this instruction in the assembly code is shown as follows:

```
scnbb [edx],128
```

In the trace file, we represent this instruction as follows:

```
S ADDRESS N
```

Note that **`ADDRESS`** and **`N`** are hexadecimal. For example, this instruction in the trace file is shown as follows:

```
S 11E770 14
```

From the above code, it means to scan boundary bits from address **`0x11E770`** to **`0x11E770 + 14 - 2 = 0x11E782`**.

VI.1.1.4 Read Memory Instruction

There are many instructions considered as read-memory instructions. For example,

- **`mov`** (move) **/ `movsx`** (move with sign-extension) instructions with memory as source (second operand)
- **`cmp`** (compare) instructions with memory

- **inc** (increment) **/** **dec** (decrement) instructions with memory
- **add** (add) **/** **sub** (subtract) **/** **mul** (multiply) **/** **div** (divide) instructions with memory as source (second operand)
- **push** instructions from memory

In the trace file, we represent this instruction as follows:

```
R ADDRESS N
```

Note that **ADDRESS** and **N** are hexadecimal. For example, this instruction in the trace file is shown as follows:

```
R 11E768 4
```

From the above code, it means to read 4-byte data from memory address **0x11E768**.

VI.1.1.5 Write Memory Instruction

There are many instructions considered as write-memory instructions. For example,

- **mov** (move) instructions with memory as destination (first operand)
- **inc** (increment) **/** **dec** (decrement) instructions with memory
- **add** (add) **/** **sub** (subtract) **/** **mul** (multiply) **/** **div** (divide) instructions with memory as destination (first operand)
- **pop** instructions into memory

In the trace file, we represent this instruction as follows:

```
W ADDRESS N
```

Note that **ADDRESS** and **N** are hexadecimal. For example, this instruction in the trace file is shown as follows:

```
W 11E768 4
```

From the above code, it means to write 4-byte data to memory address **0x11E768**.

VI.1.2 Simulation

Our architectural design is as shown in Figure 12.

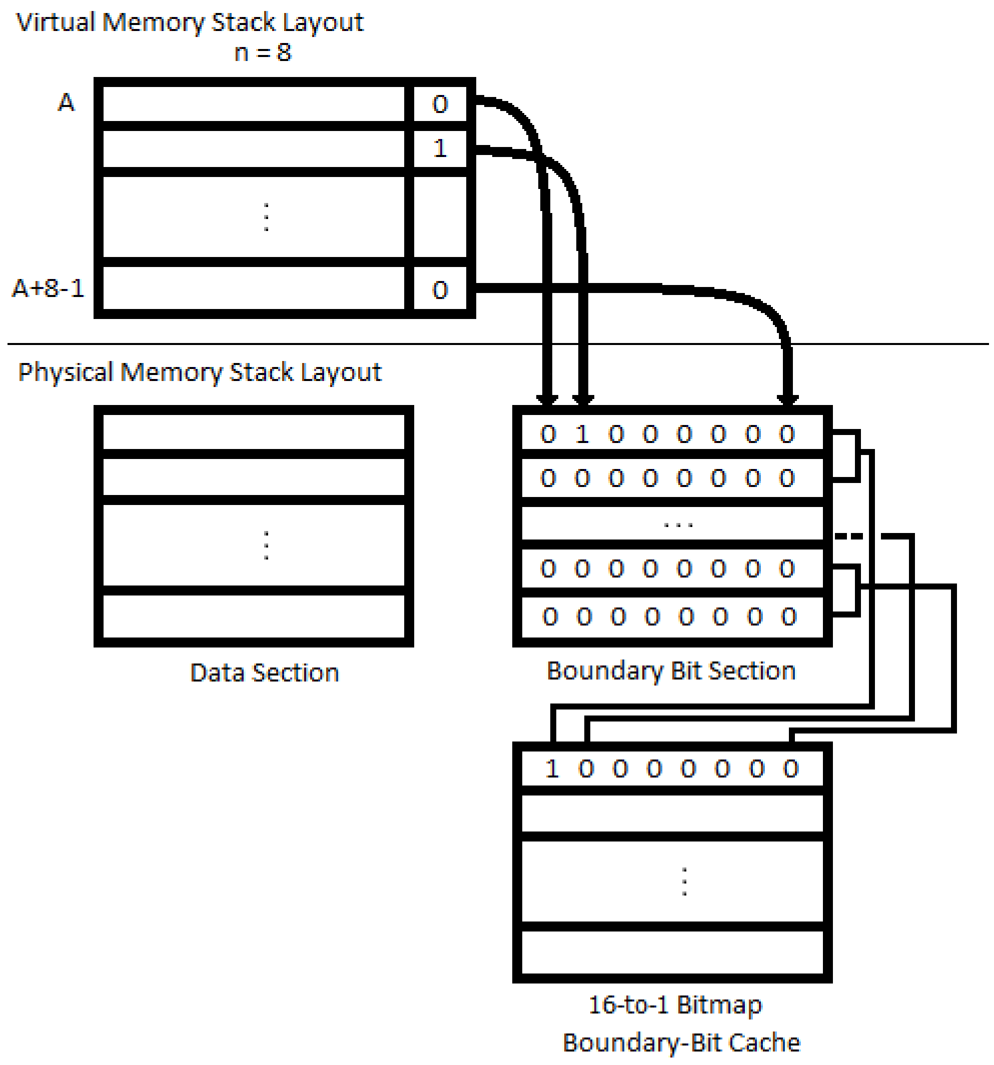

Figure 12 Architectural Design for the Test Environment

There are 3 major parts of the simulation as follows.

VI.1.2.1 Boundary Bit Set/Clear Management

When allocating memory, the system must set a boundary bit for each variable/buffer. Also, when deallocating memory, the system must clear a boundary bit for each variable/buffer. In our design, the memory that contains Boundary Bit section is separated from data section (main memory). The system needs address translation for boundary bits.

From Figure 12, the first boundary bit (high-order bit), which is set to "0", in the highest address in the boundary bit section contains the boundary bit of address **A.** That means the boundary bit of address **A** is not set. The second boundary bit, which is set to "1", in the same row contains the boundary bit of address **A + 1.** That means the boundary bit of address **A + 1** is set.

For example, if we set boundary bits at address **0x01**, **0x0E** and **0xA3**, the boundary bit section will be shown as the following Table 18.

Table 18 Boundary Bit Section Example 1

Address	Boundary Bits
0x00-0x07	0100 0000
0x08-0x0F	0000 0010
…	…
0xA0-0xA7	0001 0000

VI.1.2.2 Bitmap Set/Clear Management

Our boundary-bit cache is implemented as a 16-to-1 bitmap. After setting/clearing each boundary bit, the bitmap must be updated immediately.

From Figure 12, the first bit (high-order bit), which is set to "1", in the highest address in the bitmap represents the boundary bits of address **A** to **A + 15.** That means there is at least a boundary bit of address between address **A** to **A + 15** is set.

For example, if we set boundary bits at address **0x01**, **0x0E** and **0xA3**, the first two bytes of boundary bits, "0100 0000" and "0000 0010", which represents the boundary bits of address **0x00** to **0x0F**, will show in the first byte of 16-to-1 bitmap as "1000 0000". The boundary bit of address **0xA3** is show in the second byte of bitmap as "0010 0000" because the second byte of bitmap represents the boundary bits of address **0x80** to **0xFF**. Therefore, the 16-to-1 bitmap will be shown as the following Table 19.

Table 19 16-to-1 Bitmap Section Example 1

Address	Bitmap
0x00-0x7F	1000 0000
0x80-0xFF	0010 0000

VI.1.2.3 Boundary-Bit/Bitmap Scanning

Boundary-bit/bitmap scanning mechanism is the most delicate. We separate into 2 cases: 1) boundary bit only and 2) boundary bit and bitmap.

In the first case, each byte in the boundary bit section can contain 8 boundary bits, i.e., each byte contains the boundary bits of address **`A`** to **`A + 7.`** When scanning, it can scan every 8 addresses simultaneously. However, if the start address cannot be exactly divided by 8, the bit offset will be calculated and every boundary bit must be checked start from that bit offset. In the same way, if the end address cannot be exactly divided by 8, the system will check boundary bit set must not belong to the address that more than the end address. From Figure 12, we assume that the boundary bit section is shown in the following Table 20.

Table 20 Boundary Bit Section Example 2

Address	Boundary Bits
0x000-0x007	0001 0000
0x008-0x00F	0000 0000
0x010-0x017	0000 0000
0x018-0x01F	0001 0001
0x020-0x027	0000 0000
...	...
0x1A0-0x1A7	0000 0000
0x1A8-0x1AF	0001 0000

From above, when scanning from address **`0x004`** to **`0x01A`**, in the first byte “0001 0000”, the system will scan bits starting at bit offset = 4 (address = **`0x004`**) and ending at bit offset = 7 (address = **`0x007`**). After that, 2 bytes of boundary bits of address **`0x008`** to **`0x017`** are scanned but all are no boundary bit set. In the last byte “0001 0001”, the system will detect the boundary bit set on bit offset = 3 (address = **`0x01B`**) but it is out of scanning scope. Thus, there is no boundary bit set address **`0x004`** to **`0x01A`**.

In the second case, the system will search in the bitmap first. If there is any bit set, it will search deeply in the boundary bit section which the set bit in the bitmap is represented. From Figure 12 and Table 20, we assume that the 16-to-1 bitmap will be shown in the following Table 21.

Table 21 16-to-1 Bitmap Section Example 2

Address	Bitmap
0x000-0x07F	1100 0000
0x080-0x0FF	0000 0000
0x100-0x17F	0000 0000
0x180-0x1FF	0010 0000

From above, when scanning from address **`0x070`** to **`0x1AF`**, in the first byte "1100 0000", the system will scan bits starting at bit offset = 7 (address = **`0x070`**). After that, 2 bytes of bitmap of address **`0x080`** to **`0x17F`** are scanned but all are no boundary bit set. In the last byte "0010 0000", the system will detect the boundary bit set on bit offset = 2 (address = **`0x1A0`**). Thus, there is at least a boundary bit set between address **`0x1A0`** and **`0x1AF`**. Then, the system will search deeply in the boundary bit section as Table 20. The byte "0010 0000" is detected that the boundary bit set on bit offset = 2 (address = **`0x1AB`**). It is found that there is a boundary bit set address **`0x1AB`**.

To reduce boundary-bit scanning cycles in case of a big array, we have 2 solutions. First, we modify the 16-to-1 bitmap to 256-to-1 bitmap. Another solution is to be 2-level bitmap by adding another bitmap (16-to-1 or 32-to-1 bitmap) as shown in Figure 11 on page 43.

VI.1.3 Simulation Test Environment

The simulation is written in Microsoft Visual C++. Our test environment is conducted on 64-bit Windows 10 with 8 GB RAM. Because of the limitation of Microsoft Visual Studio 2015, we assume our test environment as follows:

1. There is 1 byte of memory per 1 boundary bit.

2. Memory size is 4 MB.
3. Boundary Bit section size is Memory size / 8 = 512 KB.
4. Boundary-bit caches are implemented into 3 cases:
    a. 1-level 16-to-1 bitmap

    16-to-1 Bitmap size is Boundary Bit section size / 16 = 32 KB

    b. 1-level 256-to-1 bitmap

    256-to-1 Bitmap size is Boundary Bit section size / 256 = 2 KB

    c. 2-level bitmap

    Level-2 16-to-1 Bitmap size is 32 KB

    Level-1 16-to-1 Bitmap size is 2 KB / 32-to-1 Bitmap size is 1 KB

VI.2 Evaluation Aspects

We focus on both protection efficiency and performance aspects.

VI.2.1 Protection Efficiency

Before running the simulation, the compiler option needs to disable "/GS (Security Check)". This setting is in the project properties > C/C++ > Code Generation > Security Check. It needs to modify to "Disable Security Check (/GS-)" [65].

The following example codes are tested. They are all detected using boundary-bit solution.

VI.2.1.1 Stack Overflows

As we mentioned in section II.2.1.1 on page 4. We test 2 types of these attacks: on control data and non-control data.

VI.2.1.1.1 Stack-Overflow Attack on Control Data

The test code is given as follows:

```
void hack() {
  Console::WriteLine("HACKED!!!");
  Console::ReadLine();
}
void func(char *p) {
  Console::WriteLine("p address = {0:X}", (int)&p);
  __asm {
    setbb dword ptr [ebp-0Ch+3h]
  }
  char b[8]; //8 bytes
  Console::WriteLine("b address = {0:X}", (int)&b);
  __asm {
    setbb dword ptr [ebp-14h+7h]
  }

  strcpy(b,p);
  __asm {
    scnbb dword ptr [ebp-14h], 1Bh
    clrbb dword ptr [ebp-0Ch+3h]
    clrbb dword ptr [ebp-14h+7h]
  }

}
int main(array<System::String ^> ^args)
{
  String^ s = Convert::ToString((int)&hack, 16);
  if (s->Length % 2 != 0)
    s = "0" + s;
  char ch[] = "00000000";
  for (int i = 0; i < s->Length; i+=2)
  {
    ch[s->Length - i - 2] = s[i];
    ch[s->Length - i - 1] = s[i+1];
  }
  String^ s2 = gcnew String(ch);
  char ch2[8] = {0,0,0,0,0,0,0,0};
  for (int i = 0; i < s->Length; i += 2)
    ch2[i/2] = (char)Convert::ToInt32(s2->Substring(i, 2),
16);
  s2 = gcnew String(ch2);
  for (int i = 0; i < 0x18; i++)
    s2 = "0" + s2;
func((char*)Marshal::StringToHGlobalAnsi(s2).ToPointer());
  return 0;
}
```

The above code will execute **hack()** function without calling it. The beginning of the code is to calculate the address of the **hack()** function and modify it as the proper input for **func()** function, as the same way in section II.2.1.1.1 on page 5. From the experiment, we know that the address of the return address of **func()** function is **0x18** bytes away from variable **b**. After the end of the **func()** function, the **hack()** function will be executed.

If it is detected, the result will be:

```
p address = 1EEEA0
b address = 1EEE98
BB FOUND! @ 1EEE9F
HACKED!!!
```

VI.2.1.1.2 Stack-Overflow Attack on Non-Control Data

The test code is given as follows:

```
void func(char *p) {
  __asm {
    setbb dword ptr [ebp-0Ch+3h]
  }

  int i = 0; //4 bytes
  __asm {
    setbb dword ptr [ebp-10h+3h]
  }

  char b[8]; //8 bytes
  __asm {
    setbb dword ptr [ebp-18h+7h]
  }
  Console::WriteLine("i address = {0:X}", (int)&i);
  Console::WriteLine("b address = {0:X}", (int)&b);

  Console::WriteLine("Before b = {0}", gcnew String(b));
  Console::WriteLine("Before i = {0:X}", i);
  strcpy(b,p);
  __asm {
    scnbb dword ptr [ebp-18h], 0Ch
  }
  Console::WriteLine("After b = {0}", gcnew String(b));
  Console::WriteLine("After i = {0:X}", i);
  __asm {
    clrbb dword ptr [ebp-0Ch+3h]
    clrbb dword ptr [ebp-10h+3h]
    clrbb dword ptr [ebp-18h+7h]
  }
}
int main(array<System::String ^> ^args)
{
  func("01234567AAAA");
  return 0;
}
```

The above code will modify the value of integer **i** without assigning it directly. From the experiment, we know that the address of integer **i** is **0x1DEE44 – 0x1DEE3C = 8** bytes away from variable **b**. Thus, we can calculate and modify it as the proper input for **func()** function, as the same way in section II.2.1.1.2 on page 9. After doing **strcpy()** function, the value of integer **i** will be modified.

If it is detected, the result will be:

```
i address = 2FEE44
b address = 2FEE3C
Before b =
Before i = 0
BB FOUND! @ 2FEE43
After b = 01234567AAAA
After i = 41414141
```

VI.2.1.2 Heap Overflows

The test code is given as follows:

```
void func(char *p) {
  Console::WriteLine("p address = {0:X}", (int)&p);
  __asm {
    setbb dword ptr [ebp-0Ch+3h]
  }

  char *b = (char *)malloc(8); //8 bytes
  __asm {
    setbb dword ptr [ebp-10h+3h]
    mov   eax, dword ptr [ebp-10h]
    setbb dword ptr [eax+7h]
  }
  Console::WriteLine("b address = {0:X}", (int)&b);
  Console::WriteLine("b = {0:X}", (int)b);

  strcpy(b, p);
  __asm {
    scnbb dword ptr [ebp-10h], 0Ah
    mov   eax, dword ptr [ebp-10h]
    clrbb dword ptr [eax+7h]
    clrbb dword ptr [ebp-0Ch+3h]
    clrbb dword ptr [ebp-10h+3h]
  }
}
int main(array<System::String ^> ^args)
{
  func("0123456789");
  return 0;
}
```

The above code will overflow the allocated memory referred by pointer **b**. When the allocated memory is written, it will be scanned and the attack is detected.

If it is detected, the result will be:

```
p address = 13F06B
b address = 13F067
b = 99848
BB FOUND! @ 9984F
```

VI.2.1.3 Array Indexing Errors

As we mentioned in section II.2.1.3 on page 11. We test 3 types of these attacks: stack (on control/non-control data) and heap.

VI.2.1.3.1 Array-Indexing-Errors (Stack) on Control Data

The test code is given as follows:

```
void hack() {
  Console::WriteLine("HACKED!!!");
  Console::ReadLine();
}
void func(char *p) {
  Console::WriteLine("p address = {0:X}", (int)&p);
  __asm {
    setbb dword ptr [ebp-0Ch+3h]
  }

  char b[8]; //8 bytes
  Console::WriteLine("b address = {0:X}", (int)&b);
  __asm {
    setbb dword ptr [ebp-14h+7h]
  }

  b[0x18] = p[0];
  __asm {
    scnbb dword ptr [ebp-14h], 19h
  }

  b[0x19] = p[1];
  __asm {
    scnbb dword ptr [ebp-14h], 1Ah
  }

  b[0x1A] = p[2];
  __asm {
    scnbb dword ptr [ebp-14h], 1Bh
  }

  b[0x1B] = p[3];
  __asm {
    scnbb dword ptr [ebp-14h], 1Ch
  }

  __asm {
    clrbb dword ptr [ebp-0Ch+3h]
    clrbb dword ptr [ebp-14h+7h]
  }

}
int main(array<System::String ^> ^args)
{
  String^ s = Convert::ToString((int)&hack, 16);
  if (s->Length % 2 != 0)
    s = "0" + s;
  char ch[8] = {0,0,0,0};
  for (int i = 0; (s->Length - (2 * i) - 2) >= 0; i++)
    ch[i] = (char)Convert::ToInt32(
            s->Substring((s->Length - (2*i) - 2), 2), 16);
  func(ch);
  return 0;
}
```

The above code will execute **`hack()`** function without calling it. The beginning of the code is to calculate the address of the **`hack()`** function and modify it as the proper input for **`func()`** function, as the same way in section II.2.1.3 on page 11. From the experiment, we know that the address of the return address of **`func()`** function is **`0x18`** bytes away from variable **`b`**. After the end of the **`func()`** function, the **`hack()`** function will be executed.

If it is detected, the result will be:

```
p address = 13ED54
b address = 13ED4C
BB FOUND! @ 13ED53
BB FOUND! @ 13ED53
BB FOUND! @ 13ED53
BB FOUND! @ 13ED53
HACKED!!!
```

VI.2.1.3.2 Array-Indexing-Errors (Stack) on Non-Control Data

The test code is given as follows:

```
void func(char *p) {
  __asm {
    setbb dword ptr [ebp-0Ch+3h]
  }

  int i = 0; //4 bytes
  __asm {
    setbb dword ptr [ebp-14h+3h]
  }

  char b[8]; //8 bytes
  __asm {
    setbb dword ptr [ebp-1Ch+7h]
  }
  Console::WriteLine("i address = {0:X}", (int)&i);
  Console::WriteLine("b address = {0:X}", (int)&b);

  Console::WriteLine("Before i = {0:X}", i);

  int diff = (int)&i - (int)&b;
  __asm {
    setbb dword ptr [ebp-10h+3h]
  }

  b[diff] = p[14];
  __asm {
    mov   eax, dword ptr [ebp-10h]
    scnbb dword ptr [ebp-1Ch], eax
  }
  Console::WriteLine("After i = {0:X}", i);
  __asm {
    clrbb dword ptr [ebp-0Ch+3h]
    clrbb dword ptr [ebp-10h+3h]
    clrbb dword ptr [ebp-14h+3h]
    clrbb dword ptr [ebp-1Ch+7h]
  }
}
int main(array<System::String ^> ^args)
{
  func("01234567890AAAA");
  return 0;
}
```

The above code will modify the value of integer **i** without assigning it directly. From the experiment, we know that the address of integer **i** is **diff = 0x3FF230 – 0x3FF228 = 8** bytes away from variable **b**. Thus, we can calculate and modify it as the proper input for **func()** function, as the same way in section II.2.1.3 on page 11. After assigning a new value to **b[diff]**, the value of integer **i** will be modified.

If it is detected, the result will be:

```
i address = 3FF230
b address = 3FF228
Before i = 0
BB FOUND! @ 3FF22F
After i = 41
```

VI.2.1.3.3 Array-Indexing-Errors (Heap)

The test code is given as follows:

```
void func(char *p) {
  Console::WriteLine("p address = {0:X}", (int)&p);
  __asm {
    setbb dword ptr [ebp-0Ch+3h]
  }

  char *b = (char *)malloc(8); //8 bytes
  __asm {
    setbb dword ptr [ebp-10h+3h]
    mov   eax, dword ptr [ebp-10h]
    setbb dword ptr [eax+7h]
  }
  Console::WriteLine("b address = {0:X}", (int)&b);
  Console::WriteLine("b = {0:X}", (int)b);

  *(b + 12) = p[0];
  __asm {
    scnbb dword ptr [ebp-10h], 0Ch
    mov   eax, dword ptr [ebp-10h]
    clrbb dword ptr [eax+7h]
    clrbb dword ptr [ebp-0Ch+3h]
    clrbb dword ptr [ebp-10h+3h]
  }
}
```

```
int main(array<System::String ^> ^args)
{
  func("0123456789");
  return 0;
}
```

The above code will access outside the allocated memory referred by pointer **b**. When accessing memory, it will be scanned and the attack is detected.

If it is detected, the result will be:

```
p address = 2FEF28
b address = 2FEF24
b = 234398
BB FOUND! @ 23439F
```

VI.2.2 Performance

For clarity, we count clock cycles of all instruction executions as mentioned in section VI.1.2 on page 48. Thus, we make the following assumptions about our test environment:

1. Boundary Bit Set/Clear management uses 1 cycle per instruction.
2. Bitmap Set/Clear management uses 1 cycle per instruction.
3. Boundary Bit and Bitmap scanning uses 1 cycle per byte of boundary bits. (Up to scanning method)
4. Read Memory instruction uses 1 cycles per byte. [66]
5. Write Memory instruction uses 2 cycles per byte. [66]

There are some examples of codes and instruction representations in the trace files.

VI.2.2.1 Variable/Argument Declaration

When declaring a variable/argument, the system will allocate the memory up to the size of variable/argument type. For example, the size of "`char`", "`int`", "`double`" for Visual C++ in Microsoft Visual Studio 2015 are 1, 4, 8 bytes respectively. To calculate the address of boundary bit set for each variable/argument, which is the last byte of each variable/argument, we must know the address and the size of this variable/argument. For instance, we declare an integer which is allocated at address **`A`**. The address of boundary bit will be **`A + size - 1 = A + 4 - 1 = A + 3`**. Thus, we represent this instruction as **`B A+3`**.

The same method also applies to a buffer/array, if a character array is allocated at address **`A`** and the array size is **`n`** bytes, the address of boundary bit will be **`A + n - 1`**.

In case of a pointer, we set a boundary bit of the pointer and a boundary bit of the memory block which the pointer points to. Normally, the pointer size is 4 bytes in the 32-bit system. If a pointer is allocated at address **`PA`**, the address of boundary bit will be **`PA + 4 - 1`**. If a memory block is allocated at address **`MA`** and the block size is **`n`** bytes, the address of boundary bit will be **`MA + n - 1`**. Therefore, the set instructions will be **`B PA+3`** and **`B MA+n-1`**.

VI.2.2.2 The End of Variable/Argument Existence

At the end of variable/argument existence, the system will deallocate the memory. The instructions will use the same address as setting a boundary bit. If an integer is allocated at address **`A`**, this clear instruction will be **`C A+3`**.

VI.2.2.3 Variable/Argument Access

There are some example codes as shown in Table 22. To reduce unnecessary scanning cycles, some write memory instructions will not scan before write memory.

The following table, some operands can be described as follows:

- **`i_address`** means the address of variable **`i`**
- **`i_size`** means the size of variable **`i`**
- **`mem_address`** means the address of allocated memory block
- **`p`** means the value of variable **`p`**

Table 22 Example Codes and Instruction in Trace Files

Example Codes	Instruction in Trace Files
`int i`	`B i_address+i_size-1`

`i++`	`R i_address i_size`
	`W i_address i_size`

`i = 1`	`W i_address i_size`

`a = b`	`R b_address b_size`
	`S a_address b_size`
	`W a_address b_size`

`a > b`	`R a_address a_size`
	`R b_address b_size`

`buf[i] = n`	`R i_address i_size`
	`R n_address n_size`
	`S buf_address (i+1)*buf[i]_size`
	`W buf[i]_address n_size`

`n = buf[i]`	`R i_address i_size`
	`R buf[i]_address buf[i]_size`
	`S n_address buf[i]_size`
	`W n_address buf[i]_size`

`p = (char*)malloc(n)`	`R n_address n_size`
	`B mem_address+n-1`
	`S p_address p_size`
	`W p_address p_size`

`*(p + i) = a`	`R a_address a_size`
	`R i_address i_size`
	`R p_address p_size`
	`S p i+1`
	`W p+i a_size`

`strncpy(arr2, arr1, n)`	`R n_address n_size`
	`R arr1_address n`
	`S arr2_address n`
	`W arr2_address n`

VI.3 Simulation Results

We create 2 cases for simulation.

VI.3.1.1 Case 1: Intensive Read/Write Buffer (Bubble Sort)

This bubble-sort program is a good simple example to show how to implement boundary bit approach with a buffer. Overall instruction cycles and slowdown of boundary-bit implementation with/without Bitmap are shown in Table 23 and Figure 13. Slowdown percentage is calculated as follows:

$$Slowdown = \left(\frac{\text{ReadWrite} + \text{Overhead}}{\text{ReadWrite}} - 1\right) \times 100\ \%$$

"ReadWrite" means instruction cycles from read/write memory instruction in VI.1.1.4 and VI.1.1.5. "Overhead" means instruction cycles from set/clear/scan boundary bit instructions in section VI.1.1.1, VI.1.1.2 and VI.1.1.3 on page 46 - 47.

Overhead cycles of boundary bits without bitmap are shown in Table 25 and Figure 14. Overhead cycles of boundary bits with 1-level 16-to-1 bitmap are shown in Table 26 and Figure 15. For optimization by changing to 1-level 256-to-1 bitmap, optimized instruction cycles are shown in Table 24. These optimized scan cycles of boundary bits are shown in Table 27. In case of 2-level bitmap, they are shown in Table 28. In case of data size = 10000, the detailed scan cycles are shown in Figure 16.

Table 23 Overall Instruction Cycles vs Data Size in the Bubble-Sort Program

Data	Original	Boundary Bit		Boundary Bit with Bitmap	
Size	Cycles	Cycles	Slow (%)	Cycles	Slow (%)
10	3624	3942	8.77	3935	8.58
100	309154	332567	38.22	427316	7.57
1000	31552131	143247583	354.00	41213399	30.62
10000	3145601035	113991054552	3523.82	10271375417	226.53

Table 24 Optimized Instruction Cycles vs Data Size in the Bubble-Sort Program

Data	256-to-1 Bitmap		2-Level 16/16 Bitmap		2-Level 32/16 Bitmap	
Size	Cycles	Slow (%)	Cycles	Slow (%)	Cycles	Slow (%)
1000	33833687	7.23	35347054	12.02	36096664	14.40
10000	3765474362	19.71	3890471569	23.67	3770446711	19.86

Note: "32/16 Bitmap" means level-1 bitmap is a 32-to-1 bitmap and level-2 bitmap is a 16-to-1 bitmap.

Table 25 Overhead Cycles without Bitmap in the Bubble-Sort Program

Data Size	Boundary Bit	
	Set/Clear	Scan
10	32	286
100	204	117958
1000	1946	111693506
10000	19794	110845433723

Table 26 Overhead Cycles with 16-to-1 Bitmap in the Bubble-Sort Program

Data Size	Boundary Bit		Bitmap	
	Set/Clear	Scan	Set/Clear	Scan
10	32	103	32	144
100	204	5137	204	17868
1000	1946	1513367	1946	8144009
10000	19794	124674311	19794	7001060483

Table 27 Overhead Cycles with 256-to-1 Bitmap in the Bubble-Sort Program

Data Size	Boundary Bit		Bitmap	
	Set/Clear	Scan	Set/Clear	Scan
1000	1946	1008244	1946	1269420
10000	19794	99720276	19794	520113463

Table 28 Scan Cycles with 2-Level Bitmap in the Bubble-Sort Program

Data Size	Boundary Bit	16/16 Bitmap		32/16 Bitmap	
		Level 1	Level 2	Level 1	Level 2
1000	1513367	1269420	1008244	1010786	2016488
10000	124674311	520113463	100043172	297417913	202713864

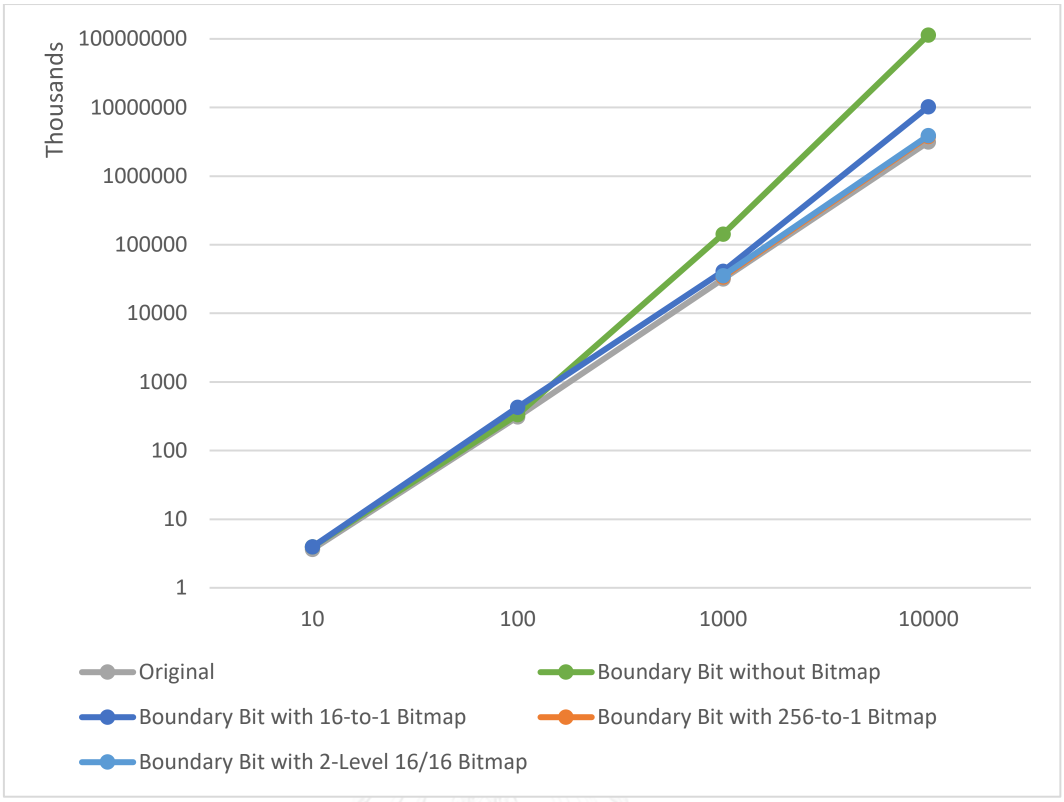

Figure 13 Overall Instruction Cycles vs Data Size in the Bubble-Sort Program

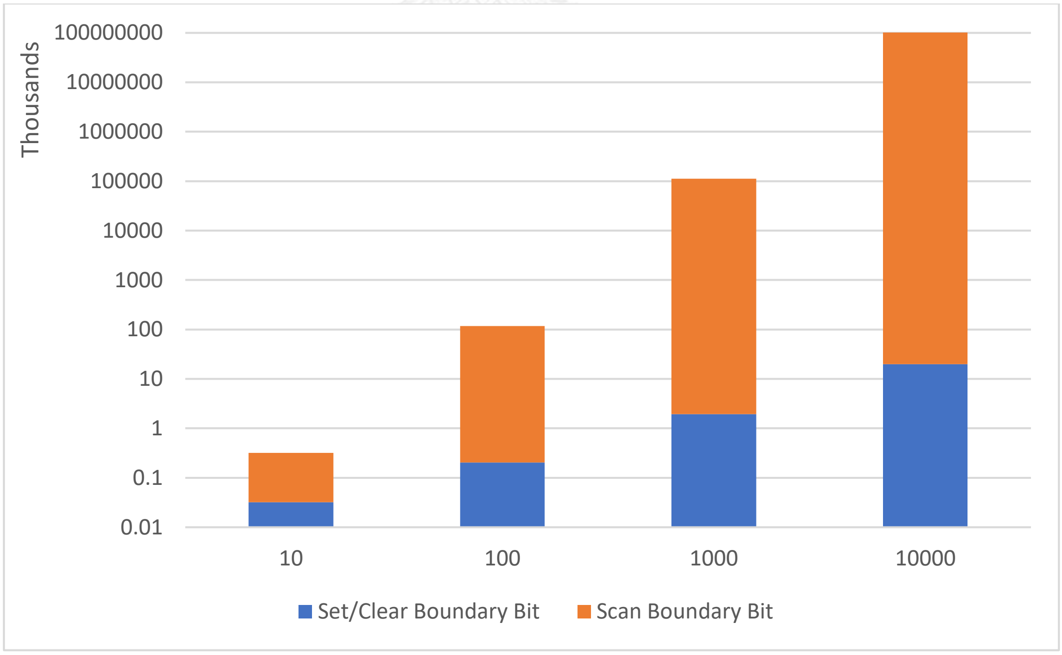

Figure 14 Overhead Cycles without Bitmap in the Bubble-Sort Program

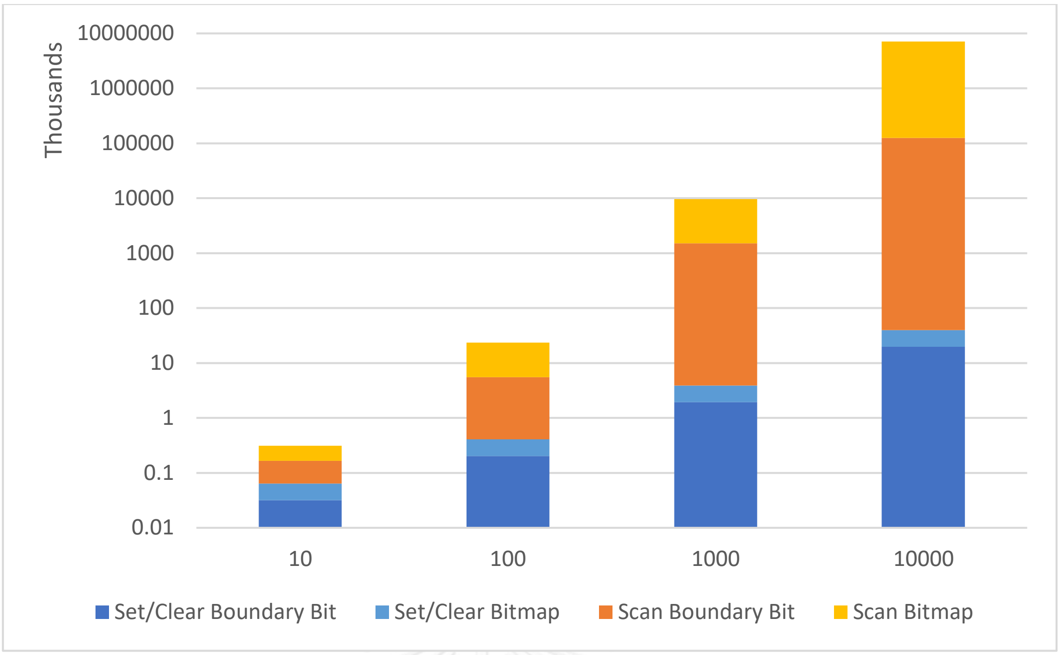

Figure 15 Overhead Cycles with 1-Level Bitmap in the Bubble-Sort Program

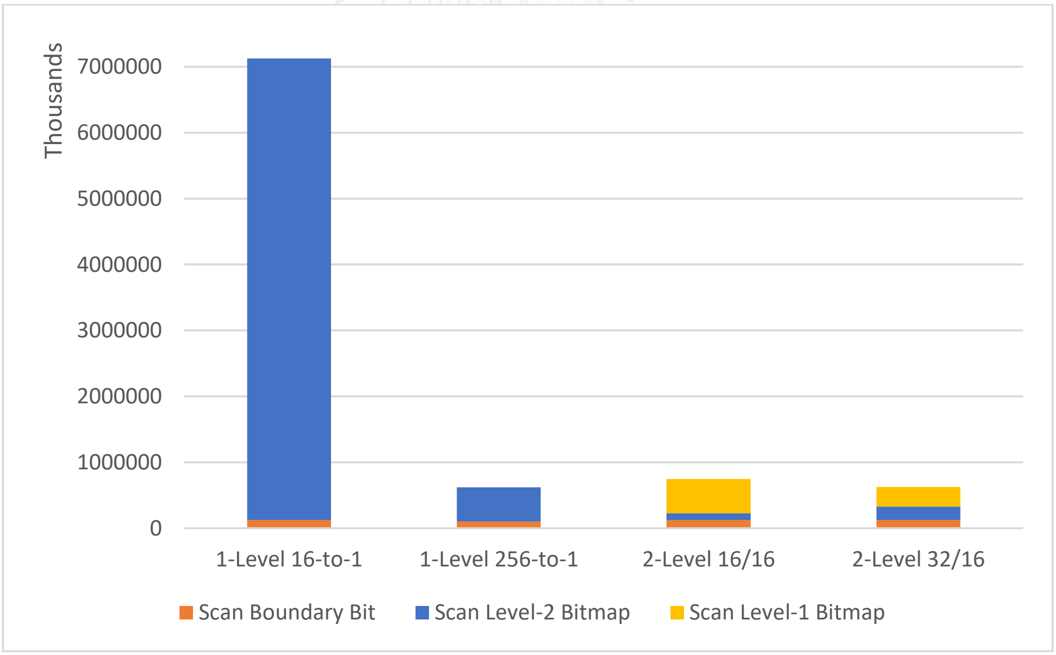

Figure 16 Scan Cycles with 1-Level and 2-Level Bitmap in the Bubble-Sort Program

As the simulation result, the smallest data size is 10. Their slowdown of boundary bit (with/without Bitmap) are about 8 %. We use this value as a baseline for comparing the optimization result. In other cases, the bigger data size makes the much more slowdown.

However, bitmap (boundary bit cache) can reduce the slowdown by reducing scan cycles. To decrease more scan cycles in case of big data size, we have 2 solutions. First, we change a 1-level bitmap from 16-to-1 bitmap to 256-to-1 bitmap. Second, we add another level of a 16-to-1 bitmap. The slowdown result shows that the first solution can decrease scan cycles more than the second solution. Thus, we try another solution by changing a level-1 bitmap from 16-to-1 bitmap to 32-to-1 bitmap. In case of data size = 1000, the slowdown result does not decrease. Even though the scan cycles of level-1 bitmap is decreased, the scan cycles of level-2 bitmap is more increased. In case of data size = 10000, the slowdown result decreases as expected.

We conclude that the optimization solution is to add the proper size of a n-to-1 bitmap. The bigger n value is suitable for a bigger array/buffer.

VI.3.1.2 Case 2: Random Write Memory

This program randomly simulates intensive write memory in many ways, such as:

- Write to character variables
- Write to integer variables
- Write to double variables
- Access a big array/buffer (100,000 entries)
- Access a big memory block using a pointer (100,000 B)
- Write data vary in size to a big array/buffer (100,000 entries)

We randomize for 10,000/100,000/1,000,000/10,000,000 times. Overall instruction cycles and slowdown of boundary-bit implementation with/without Bitmap are shown in Table 29 and Figure 17.

Overhead cycles of boundary bits without bitmap are shown in Table 31 and Figure 18. Also, overhead cycles of boundary bits with 1-level 16-to-1 bitmap are shown in Table 32 and Figure 19. For optimization by changing to 1-level 256-to-1 bitmap, and to 2-level bitmap, these optimized instruction cycles are shown in Table

30 and optimized scan cycles of boundary bits are shown in Table 33. In case of 10,000,000 times, the detailed scan cycles are shown in Figure 20.

Table 29 Overall Instruction Cycles in the Random Write Memory Program

Random Times	Original Cycles	Boundary Bit		Boundary Bit with Bitmap	
		Cycles	Slow (%)	Cycles	Slow (%)
10000	23204503	39077311	68.40	24317852	4.80
100000	234273934	396420226	69.21	245566672	4.82
1000000	2358780502	3983282338	68.87	2472176919	4.81
10000000	23650789851	39880930514	68.62	24785993009	4.80

Table 30 Optimized Instruction Cycles in the Random Write Memory Program

Random Times	256-to-1 Bitmap		2-Level 16/16 Bitmap	
	Cycles	Slow (%)	Cycles	Slow (%)
10000	23392359	0.81	23731308	2.27
100000	236190074	0.81	237300290	1.29
1000000	2377615868	0.80	2392061404	1.41
10000000	23837868276	0.79	23995229307	1.46

Table 31 Overhead Cycles without Bitmap in the Random Write Memory Program

Random Times	Boundary Bit	
	Set/Clear	Scan
10000	40086	15832722
100000	399542	161746750
1000000	3998336	1620503500
10000000	39989702	16190150961

Table 32 Overhead Cycles with Bitmap in the Random Write Memory Program

Random Times	Boundary Bit		Bitmap	
	Set/Clear	Scan	Set/Clear	Scan
10000	40086	25077	40086	1008100
100000	399542	216704	399542	10276950
1000000	3998336	2499966	3998336	102899779
10000000	39989702	25000066	39989702	1030223688

Table 33 Scan Cycles in the Random Write Memory Program

Random Times	256-to-1 Bitmap		2-Level 16/16 Bitmap		
	Boundary Bit	Bitmap	Boundary Bit	Level 1	Level 2
10000	26928	80756	25077	80756	340800
100000	300598	816458	249964	816458	1160850
1000000	2673404	8165290	2499966	8165290	14618974
10000000	25072101	82026920	25000066	82026920	157433066

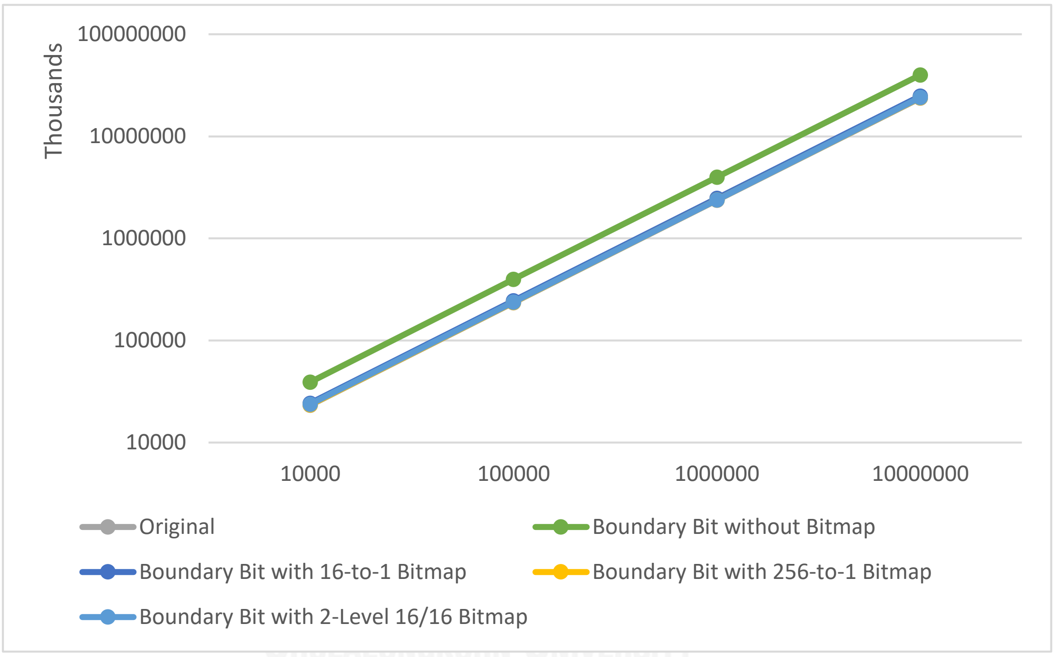

Figure 17 Overall Instruction Cycles in the Random Write Memory Program

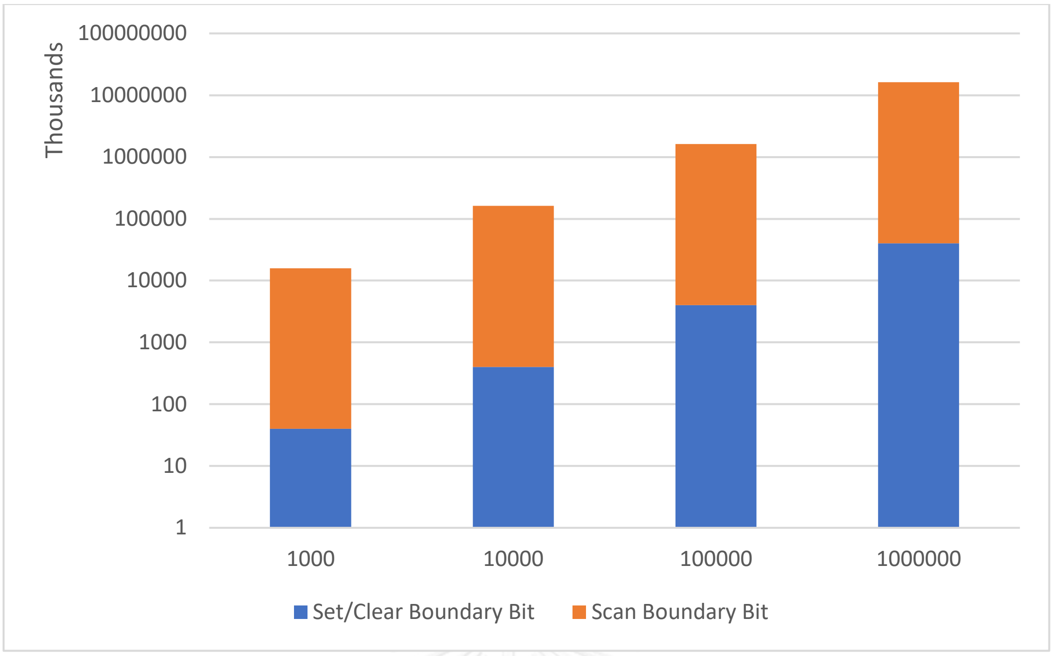

Figure 18 Overhead Cycles without Bitmap in the Random Write Memory Program

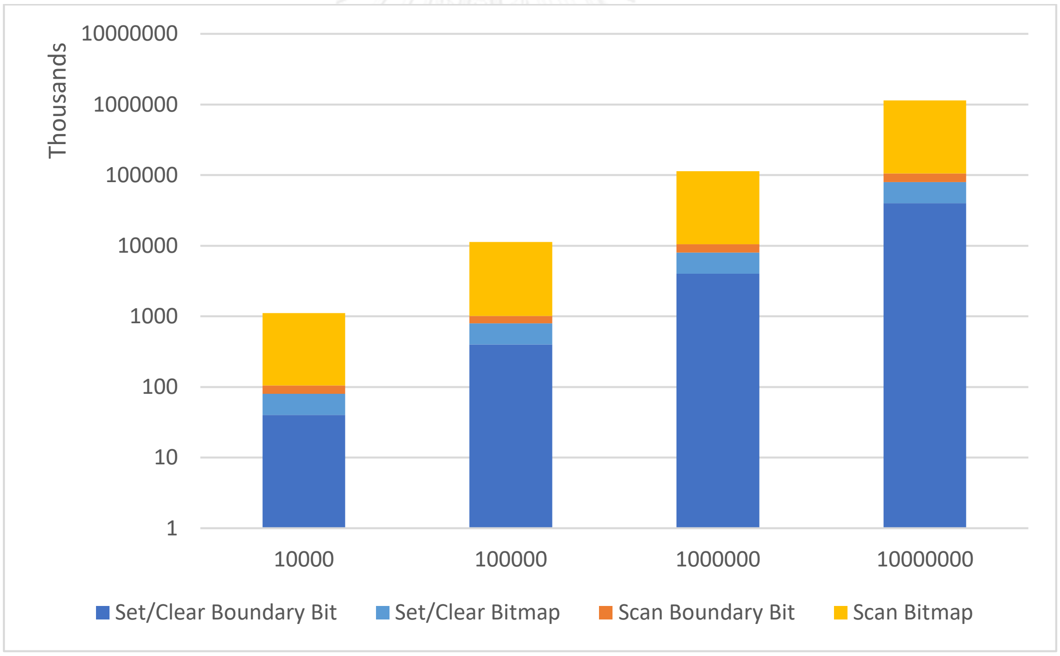

Figure 19 Overhead Cycles with 1-Level 16-to-1 Bitmap in the Random Write Memory Program

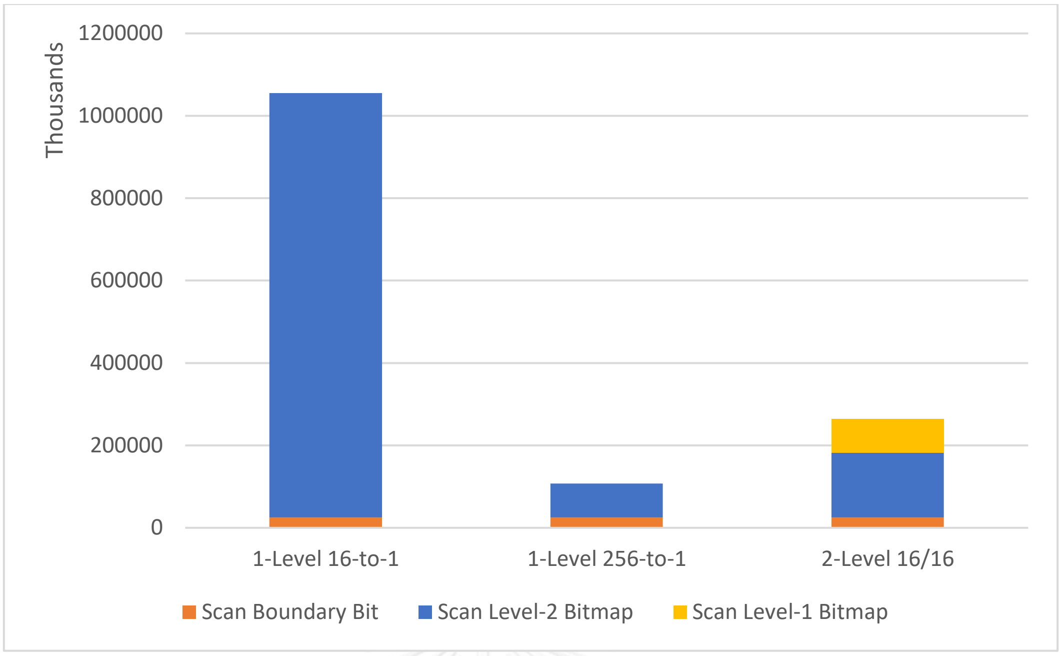

Figure 20 Scan Cycles with 1-Level and 2-Level Bitmap in the Intensive Write Memory Program

The simulation result shows that the more random times causes the more overall instruction cycles. The slowdown is 4.8 % using 1-level 16-to-1 bitmap.

If we want to reduce scan cycles, we also have 2 solutions. First, we change a 1-level bitmap from 16-to-1 bitmap to 256-to-1 bitmap. Its slowdown is reduced to 0.8 %. Second, we add another level of a 16-to-1 bitmap. Its average slowdown is reduced to 1.6 %. The slowdown result shows that the first solution can decrease scan cycles more than the second solution.

Furthermore, the average miss rate from our test is shown in Table 34.

Table 34 The Miss Rate in the Intensive Write Memory Program

Bitmap Types	Average Miss Rate	
	Level 1	Level 2
1-level 16-to-1 bitmap	0.013	
1-level 256-to-1 bitmap	0.166	
2-level 16/16 bitmap	0.321	0.095

The simulation result shows that the miss rate of 256-to-1 bitmap is higher than the miss rate of 16-to-1 bitmap. For 2-level bitmap, the miss rate of level-1 bitmap is much higher than the miss rate of 1-level 256-to-1 bitmap.

In the same way, we can conclude that the optimization solution is to add the proper size of a 1-level bitmap may be better than 2-level bitmap.

VI.3.2 Discussion

The evaluation is concluded into 2 aspects. First, in aspect of protection efficiency, our approach can protect all types of buffer-overflow attacks we tested.

Second, in aspect of performance, our assumption about instruction cycles in section VI.2.2 (on page 60) reflects the upper-bound simulation results. In other words, the overall instruction cycles may be less than our simulations because the real hardware implementation can be parallel. For example, overhead cycles for boundary bits and bitmap can be reduced to half. Moreover, during the written memory, it is no need to wait until the process finishes scanning boundary bits.

CHAPTER VII
ANALYSIS

In this chapter, we will analyze the advantages and potential disadvantages issues of boundary bit. We will also address the performance issue of boundary bit.

VII.1 Advantages

There are many advantages of our solution as follows.

VII.1.1 Prevent All Types of Buffer-Overflow Attacks

This approach prevents 2 buffer-overflow characteristics: **`len:buff`** and **`out:buff`**, which cover both legacy types and new types of buffer-overflow attacks. No matter what the target is control data or non-control data, our approach can protect all.

VII.1.2 Fixed and Low Additional Memory Usage

Additional memory usage for storing all boundary bits is fixed and it uses less memory for metadata. Besides using less memory than most software solutions, boundary bit uses less memory for metadata comparing to most hardware solutions as well. For x86 architecture, Segmentation uses additional 3 words per variable for metadata, containing starting address, ending address (or limit) and current address. However, Boundary Bit uses a fixed cost 1 bit per memory word/byte. In directly, this is equivalent to 1 bit per variable.

If there are 100 variables with 4-byte size, Segmentation consumes 3 x 100 = 300 words for metadata. Assuming that a word contains 4 bytes, segmentation would require 300 x 4 = 1200 bytes. Boundary Bit only consumes 100 x 4 / 8 = 50 bytes. In this case, comparing to Segmentation, Boundary Bit requires 24 times less memory.

VII.1.3 Low Performance Overhead

Scanning every bit does not make it slow because the hardware mechanism for scanning can be parallel. On minimal, a byte scan can cover up to 8 bits. Therefore, its performance overhead will not be too high comparing to software approach.

VII.2 Disadvantages

Although, hardware approach has many strengths, it does introduce compatibility issue.

VII.2.1 Hardware Incompatibility

To apply Boundary Bit, the existing hardware cannot be used. However, hardware has its life expectancy. When it is time to change, the new hardware with the Boundary Bit can be a good choice for enhancing security.

VII.2.2 Software Incompatibility

Because of the design, programmers or compilers must tell the system to set bit at the end of any variable or buffer for setting the boundary. Thus, Boundary Bit is not completely transparent. However, every nowadays software must be updated regularly to fix bugs and vulnerabilities. Besides, it provides a light-weight mechanism for software to set a boundary. The setting can be embedded into memory allocation.

VII.3 Performance Analysis

Hardware-software co-design and optimization is very important to improve the performance. The caching bitmap can improve the boundary-bit scanning efficiency. The memory access time of bit-scanning can be greatly reduced, although it is not in parallel. It can be modelled as follows:

$$Scan\ Cycles = Hit\ Cycles + (Miss\ Rate \times Miss\ Cycles)$$

- **Scan Cycles** is the average number of boundary-bit scanning cycles for 1 Kbytes of boundary bits.
- **Hit Cycles** is the max number of cycles when scanning in the boundary-bit cache (bitmap). It is up to the bitmap size.
- **Miss Cycles** is the max number of cycles when scanning in the boundary-bit section (in the memory) after scanning in the bitmap is insufficient.
- **Miss Rate** is the fraction of accesses which are a miss.

From our assumption in section VI.2.2 (on page 60), 1 cycle can scan 1 byte (8 bits) of boundary bits. The miss rate from our test is shown in Table 34 on page 71.

In the case of the 1-level 16-to-1 bitmap, when it hits, the hit cycle is 1024 / 16 = 64 cycles for 1 Kbytes. When it misses, the miss cycle is 16 / 8 = 2 cycles. As a result, its scan cycles will be 64 + (0.013 × 2) = 64.026 cycles.

In the case of the 1-level 256-to-1 bitmap, when it hits, the hit cycle is 1024 / 256 = 4 cycles for 1 Kbytes. When it misses, the miss cycle is 256 / 8 = 32 cycles. As a result, its scan cycles will be 4 + (0.166 × 32) = 9.312 cycles.

In the case of the 2-level 16/16 bitmap, when it hits in level 1, the hit cycle is 1024 / (16 × 16) = 4 cycles. When it misses in level 1, the miss cycle is calculated by scan cycles in level 2. When it hits in level 2, the hit cycle is 1024 / 16 = 64 cycles. When it misses in level 2, the miss cycle is 16 / 8 = 2 cycles. its level-2 scan cycles will be 64 + 0.095 × 2 = 64.19 cycles. As a result, its level-1 scan cycles will be 4 + (0.321 × (64 + 0.095 × 2)) = 24.60499 cycles.

The scan cycles of 1-level 256-to-1 bitmap is less than the scan cycles of 2-level 16/16 bitmap. Therefore, from the model and from simulation results, we suggest an optimization solution by using the proper “n” of level-1 n-to-1 bitmap as for the better performance.

Moreover, the scanning can be done in parallel with their associated data by modifying the processor. Given that there are several hardware-level parallelisms that can be implemented to hide the overhead of boundary scanning, we conclude that very little performance penalty is introduced.

VII.4 Cost Analysis

To implement our boundary bit approach, described in CHAPTER V on page 38, it needs to modify both hardware and software.

Firstly, in the hardware part, additional memory usage is a fixed cost. In other words, it does not vary on the program size because the system allocates the block of memory for storing all boundary bits.

Secondly, a processor must be modified to add more instruction set about boundary bit mechanism. That can be parallel. Thus, this cost is a little performance penalty. It is a trade-off between performance and security.

Lastly, the boundary bit instruction must be called in the software. This can be implemented by modifying a compiler. This is not transparent to the current software but this cost can be accepted for more security.

VII.5 The Impact of Virtual Memory

Addresses we mentioned before in this thesis are all physical addresses. In this issue, we will describe how virtual memory impacts the boundary bit management.

Virtual memory extends physical memory capacity by using swap space from hard disk. It separates into "pages", which makes a program relocatable.

The block of the Boundary Bit section is simply managed by following the page of the data section. When the page is swapped out from memory to hard disk, all boundary bits in the range of the page's addresses are swapped out to hard disk as well. In the same way, when the page is swapped into memory from hard disk, their boundary bits are swapped back into memory.

VII.6 Boundary Bit Protection

The boundary bit storage, from our design in section V.1.2 on page 40, is not only to store boundary bits but also to protect them. Only the root privilege can modify boundary bits, no matter using any type of buffer overflow attacks or using the boundary bit instruction such as clear boundary bit (**clrBB**).

To clarity, if an attacker wants to attack the system by bypassing the boundary bits to exploit a buffer overflow vulnerability. Their target is to modify boundary bit stored in the memory and bitmap. They may write the code for clearing all boundary bits. Then they want to execute the code. However, they must have already own the privilege to execute. If they have the privilege, that means they own the system and they do not need to hack anymore.

CHAPTER VIII
CONCLUSION

In this chapter, we provide contributions and conclusion of this research.

VIII.1 Contributions

The contributions are as follows:

1. Purpose a new hardware solution, namely "Boundary Bit", for preventing buffer-overflow attacks.
2. Analyze the effectiveness that this solution can prevent which types of buffer-overflow attacks
3. Analyze the efficiency that tradeoffs are worth to implement this solution

VIII.2 Conclusion

The underlying concept of Boundary-Bit is using bound-checking to ensure that transferring data do not exceed the allocated capacity of variables or buffers. The goal is to provide a hardware solution against all types of buffer-overflow attacks, including non-control data attacks and array-indexing errors, with the lower overhead than other software solutions.

To trade few performance and new hardware for more security, the key point is to reduce scanning cycles by using bitmap as a boundary-bit cache.

Moreover, Boundary Bit is easy to implement. It requires few software modification to deploy this scheme.

Though we have demonstrated viability at the architectural level, this solution can also be implemented in software run-time environment such as Java Virtual Machine or .NET framework.

Boundary Bit provides bound checking at the architectural level. This mechanism should provide protection against future buffer-overflow attacks. Giving the security provided, we believe Boundary Bit is a solution of buffer-overflow attacks.

REFERENCES

1. Schmidt, C. and T. Darby. *The What, Why, and How of the 1988 Internet Worm*. 1988; Available from: http://www.snowplow.org/tom/worm/worm.html.
2. Security, U.S.D.o.H. *Vulnerability Notes Database*. 04 May 2017; Available from: http://www.kb.cert.org/vuls/bypublic.
3. Economou, N. *MICROSOFT WINDOWS UP TO 8.1 MEMORY OBJECT WIN32K.SYS BUFFER OVERFLOW*. 2013 04/07/2017; Available from: http://www.scip.ch/en/?vuldb.11444.
4. Microsoft. *MICROSOFT WINDOWS UP TO SERVER 2016 SMB BUFFER OVERFLOW*. 2017 07/14/2017; Available from: https://vuldb.com/?id.98018.
5. Webopedia. *buffer overflow*. Available from: http://www.webopedia.com/TERM/B/buffer_overflow.html.
6. Chiamwongpaet, S. and K. Piromsopa. *The implementation of Secure Canary Word for buffer-overflow protection*. in *2009 IEEE International Conference on Electro/Information Technology*. 2009.
7. Kundu, A. and E. Bertino. *A New Class of Buffer Overflow Attacks*. in *2011 31st International Conference on Distributed Computing Systems*. 2011.
8. Bishop, M., et al., *A Taxonomy of Buffer Overflow Characteristics.* IEEE Transactions on Dependable and Secure Computing, 2012. **9**(3): p. 305-317.
9. Piromsopa, K. and R.J. Enbody, *Survey of Protections from Buffer-Overflow Attacks.* Engineering Journal, 2011(2): p. 31-52%V 15.
10. Viega, J., et al. *ITS4: a static vulnerability scanner for C and C++ code*. in *Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference*. 2000.
11. Wheeler, D.A. *Flawfinder*. Available from: http://www.dwheeler.com/flawfinder/.
12. *RATS*. Available from: http://www.securesw.com/rats/.

13. Haugh, E. *Testing C programs for buffer overflow vulnerabilities*. in *In Proceedings of the Network and Distributed System Security Symposium*. 2003.

14. Baratloo, A., N. Singh, and T. Tsai, *Transparent run-time defense against stack smashing attacks*, in *Proceedings of the annual conference on USENIX Annual Technical Conference*. 2000, USENIX Association: San Diego, California. p. 21-21.

15. Evans, D. and D. Larochelle, *Improving security using extensible lightweight static analysis.* IEEE Software, 2002. **19**(1): p. 42-51.

16. Wagner, D., et al. *A First Step towards Automated Detection of Buffer Overrun Vulnerabilities*. in *IN NETWORK AND DISTRIBUTED SYSTEM SECURITY SYMPOSIUM*. 2000.

17. Walthinsen, C.C.a.S.B.a.R.F.D.a.C.P.a.P.W.a.E. *Protecting Systems from Stack Smashing Attacks with StackGuard*. in *In Linux Expo*. 1999.

18. Cowan, C., et al. *Buffer overflows: attacks and defenses for the vulnerability of the decade*. in *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*. 2003.

19. Hinton, H., et al. *SAM: Security Adaptation Manager*. in *Computer Security Applications Conference, 1999. (ACSAC '99) Proceedings. 15th Annual*. 1999.

20. Cowan, C., et al., *StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks*, in *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*. 1998, USENIX Association: San Antonio, Texas. p. 5-5.

21. Etoh, H. *Gcc extension for protecting applications from stack-smashing attacks*. Available from: https://www.researchgate.net/publication/243483996_Gcc_extension_for_protecting_applications_from_stack-smashing_attacks.

22. Cowan, C., et al., *Pointguard<sup>TM</sup>: protecting pointers from buffer overflow vulnerabilities*, in *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*. 2003, USENIX Association: Washington, DC. p. 7-7.

23. Shao, Z., et al. *Defending embedded systems against buffer overflow via hardware/software*. in *19th Annual Computer Security Applications Conference, 2003. Proceedings.* 2003.

24. Tuck, N., B. Calder, and G. Varghese. *Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow*. in *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*. 2004.

25. Frantzen, M. and M. Shuey, *StackGhost: Hardware facilitated stack protection*, in *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*. 2001, USENIX Association: Washington, D.C.

26. McGregor, J.P., et al. *A processor architecture defense against buffer overflow attacks*. in *International Conference on Information Technology: Research and Education, 2003. Proceedings. ITRE2003.* 2003.

27. Xu, J., et al. *Architecture Support for Defending Against Buffer Overflow Attacks*. in *Proceedings of the Second Workshop on Evaluating and Architecting*. 2002.

28. Ye, D. and D. Kaeli, *A reliable return address stack: microarchitectural features to defeat stack smashing.* SIGARCH Comput. Archit. News, 2005. **33**(1): p. 73-80.

29. Ozdoganoglu, H., et al., *SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address.* IEEE Transactions on Computers, 2006. **55**(10): p. 1271-1285.

30. Tzi-Cker, C. and H. Fu-Hau. *RAD: a compile-time solution to buffer overflow attacks*. in *Proceedings 21st International Conference on Distributed Computing Systems*. 2001.

31. Prasad, M. and T.-c. Chiueh. *A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks*. in *Proceedings of the General Track: 2003 USENIX Annual Technical Conference*. 2003.

32. Corliss, M.L., E.C. Lewis, and A. Roth, *Using DISE to protect return addresses from attack.* SIGARCH Comput. Archit. News, 2005. **33**(1): p. 65-72.

33. Vendicator. *Stack Shield*. 08 Jan 2000; Available from: http://www.angelfire.com/sk/stackshield/.

34. Inoue, K., *Energy-security tradeoff in a secure cache architecture against buffer overflow attacks.* SIGARCH Comput. Archit. News, 2005. **33**(1): p. 81-89.

35. Gehringer, E.F. and J.L. Keedy, *Tagged architecture: how compelling are its advantages?* SIGARCH Comput. Archit. News, 1985. **13**(3): p. 162-170.

36. Li, D., Z. Liu, and Y. Zhao. *HeapDefender: A Mechanism of Defending Embedded Systems against Heap Overflow via Hardware*. in *2012 9th International Conference on Ubiquitous Intelligence and Computing and 9th International Conference on Autonomic and Trusted Computing*. 2012.

37. Piromsopa, K. and R.J. Enbody, *Secure Bit: Transparent, Hardware Buffer-Overflow Protection.* IEEE Transactions on Dependable and Secure Computing, 2006. **3**(4): p. 365-376.

38. Crandall, J.R. and F.T. Chong, *A security assessment of the minos architecture.* SIGARCH Comput. Archit. News, 2005. **33**(1): p. 48-57.

39. Crandall, J.R. and F.T. Chong. *Minos: Control Data Attack Prevention Orthogonal to Memory Model*. in *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*. 2004.

40. Chen, S., et al. *Defeating memory corruption attacks via pointer taintedness detection*. in *2005 International Conference on Dependable Systems and Networks (DSN'05)*. 2005.

41. Chabbi, M., et al., *Efficient Dynamic Taint Analysis Using Multicore Machines*. 2007, The University of Arizona.

42. Organick, E.I., *A programmer's view of the Intel 432 system*. 1983: McGraw-Hill, Inc. 418.

43. Colwell, R.P., et al., *Instruction Sets and Beyond: Computers, Complexity, and Controversy.* Computer, 1985. **18**(9): p. 8-19.

44. Jones, R.W.M. and P.H.J. Kelly. *Backwards-compatible bounds checking for arrays and pointers in C programs*. in *Distributed Enterprise Applications. HP Labs Tech Report*. 1997.

45. Corporation, I. *IBM Rational Purify & PurifyPlus Divestiture*. Available from: https://www-01.ibm.com/software/rational/products/purifyplus_divestiture/.

46. Focus, M. *DevPartner for Visual C++ BoundsChecker Suite, Named Users*. Available from: https://www.componentsource.com/product/devpartner-visual-c-boundschecker-suite-visual-studio-named-users.

47. Austin, T.M., S.E. Breach, and G.S. Sohi, *Efficient detection of all pointer and array access errors.* SIGPLAN Not., 1994. **29**(6): p. 290-301.

48. Oiwa, Y., et al., *Fail-Safe ANSI-C Compiler: An Approach to Making C Programs Secure Progress Report*, in *Software Security — Theories and Systems: Mext-NSF-JSPS International Symposium, ISSS 2002 Tokyo, Japan, November 8–10, 2002 Revised Papers*, M. Okada, et al., Editors. 2003, Springer Berlin Heidelberg: Berlin, Heidelberg. p. 133-153.

49. Lhee, K.-S. and S.J. Chapin, *Buffer overflow and format string overflow vulnerabilities.* Softw. Pract. Exper., 2003. **33**(5): p. 423-460.

50. Bhatkar, E., D.C. Duvarney, and R. Sekar. *Address obfuscation: an efficient approach to combat a broad range of memory error exploits*. in *In Proceedings of the 12th USENIX Security Symposium*. 2003.

51. Team, T.P. *Homepage of The PaX Team*. 2013.10.02 04:41 GMT; Available from: https://pax.grsecurity.net/.

52. Krazit, T. *AMD Chips Guard Against Trojan Horses*. 2004; Available from: http://www.pcworld.com/article/114328/article.html.

53. Peterson, D.S., M. Bishop, and R. Pandey, *A Flexible Containment Mechanism for Executing Untrusted Code*, in *Proceedings of the 11th USENIX Security Symposium*. 2002, USENIX Association. p. 207-225.

54. Chang, F., A. Itzkovitz, and V. Karamcheti. *User-level resource-constrained sandboxing*. in *Proceedings of the 4th conference on USENIX Windows Systems Symposium*. 2000.

55. Wahbe, R., et al., *Efficient software-based fault isolation.* SIGOPS Oper. Syst. Rev., 1993. **27**(5): p. 203-216.

56. Small, C., *A Tool For Constructing Safe Extensible C++ Systems*, in *1997 Conference on Object-Oriented Technologies and Systems*. 1997.

57. Corporation, I. *LaGrande Technology Architectural Overview*. 2003; Available from: ftp://download.intel.com/technology/security/downloads/LT_Arch_Overview.pdf.

58. MacDonald, R., et al., *Bear: An Open-Source Virtual Secure Coprocessor based on TCPA*. 2003, Dartmouth College.

59. International, A. *ACA's TCPA White Papers*. Available from: http://www.acainternational.org/tcpa/tcpa-research-and-statistics.

60. ARM. *ARM TrustZone*. Available from: https://www.arm.com/products/security-on-arm/trustzone.

61. Corporation, M. *Next-Generation Secure Computing Base*. Available from: http://www.microsoft.com/resources/ngscb/default.mspx.

62. Kgil, T., L. Falk, and T. Mudge, *ChipLock: support for secure microarchitectures.* SIGARCH Comput. Archit. News, 2005. **33**(1): p. 134-143.

63. Ruwase, O. and M.S. Lam. *A Practical Dynamic Buffer Overflow Detector*. in *In Proceedings of the 11th Annual Network and Distributed System Security Symposium*. 2004.

64. Wikipedia. *Locality of reference*. Available from: https://en.wikipedia.org/wiki/Locality_of_reference.

65. Microsoft. */GS (Buffer Security Check)*. 2015; Available from: https://msdn.microsoft.com/en-us/library/8dbf701c.aspx.

66. Fog, A. *Instruction tables*. 2017 2017-05-02; Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs]. Available from: http://www.agner.org/optimize/instruction_tables.pdf.

APPENDIX

VITA

Miss Sirisara Chiamwongpaet was born on 27th November 1986 in Bangkok, Thailand. She received the Bachelor's and Master's degrees in Computer Engineering from Chulalongkorn University in 2008 and 2010, respectively.

Her journal paper, "Boundary Bit: An Architectural Bound Checking towards Buffer-Overflow Protection", is under active reviews for publishing on IEEE Transactions on Dependable and Secure Computing (TDSC). This is her dissertation to complete doctoral degree in Computer Engineering, Chulalongkorn University.