Boundary Bit: Architectural Bound Checking for Buffer-Overflow Protection

Sirisara Chiamwongpaet[1] and **Krerk Piromsopa**[2], Members

ABSTRACT

We propose Boundary Bit, a new architectural bound-checking approach that detects and prevents buffer-overflow attacks. Boundary Bit extends an architecture by associating a bit to each memory entry. Software can set a (boundary) bit to delimit an object. On each memory access, the hardware will dynamically validate the object's bound using the boundary bit. With minimal hints from the compiler, our architectural design eliminates most (if not all) types of buffer-overflow attacks. These include attacks on non-control data (variables and arguments) and array-indexing errors. We evaluated the performance of Boundary Bit using simulation, and the results show that the majority of performance overheads lies in bit scanning operations. To mitigate performance overhead, we introduce a hardware bitmap to act as a cache. The results from our simulation show that the hardware bitmap can absorb most of the overhead from bit scanning, which in the best-case scenario was 30 times faster than the version that does not utilize a bitmap cache.

Keywords: Buffer overflow, Invasive software, Security kernels, Security and protection, System architectures, Unauthorized access

1. INTRODUCTION

Since the creation of the infamous MORRIS worm [35] in 1988, buffer-overflow vulnerabilities have been used by malicious worms and viruses to exploit numerous computer systems. Though it is possible to write secure code, no program is guaranteed to be free from bugs. There have been a lot of buffer-overflow vulnerabilities continuously detected and reported. For example, even the well-known operating system's libraries still suffer from buffer-overflow attacks [10]. Moreover, the well-known WannaCry ransomware attack in May 2017 exploits a buffer-overflow vulnerability in the most Microsoft Windows versions, including Microsoft Windows 10 SP1 [22].

Many solutions have been proposed [13,31]. However, they have mostly focused on control data (e.g. return addresses and function pointers). Few of them can prevent buffer overflow on arbitrary data (e.g. pointers, arrays and function arguments). We believe the existence of buffer overflow is a result of an insufficient foundation at the architectural level.

1.1 Concept

Boundary Bit provides bound checking at the architectural level by ensuring that transferred data cannot exceed the allocated capacity of buffers. While many hardware solutions (such as segmentation [25] and tag architecture [12]) exist, some of them (e.g. segmentation) are, however, not common to all architectures. In supported systems, the mechanism is usually ignored in favor of performance or compatibility. We believe a light-weight architectural solution is the key to the success of buffer-overflow protection.

To enforce bound checking without sacrificing performance, Boundary Bit associates an extra bit with each memory entry. This bit is similar to those of Secure Bit's [27] (as well as tag architecture [12]). It can do everything Secure Bit does. In addition, it can handle another whole class of attacks that Secure Bit cannot.

2. BACKGROUND: BUFFER-OVERFLOW ATTACKS

A buffer-overflow attack can be described as an attack in which a buffer is overflowed beyond its bounds into another buffer with an intent to cause malicious behavior in a program [4,29,40].

2.1 Classification by Attack Locations

- **Stack Overflows** These attacks are conducted by copying data larger than the size of an allocated buffer in the stack. As a result, the overflowed data will overwrite the return address. The eventual return instruction at the end of a function will return to execute attackers' code instead of the normal process flow. However, this stack area may contain control data and non-control data. Although the return address is the major target, this type of overflow attack can occur arbitrarily.
- **Heap Overflows** Similarly to stack-overflow attacks, heap-overflow attacks can modify data by overwriting adjacent memory. The heap memory stores function pointers and dynamically-allocated data. Such allocation is done by calling the "mal-

Manuscript received on August 28, 2019 ; revised on January 15, 2020.
Final manuscript received on January 16, 2020.
[1,2] The authors are with the Department of Computer Engineering, Chulalongkorn University, Bangkok 10330, Thailand, E-mail: sirisara.c@gmail.com and krerk.p@chula.ac.th
DOI: 10.37936/ecti-cit.2020142.212338

loc" function in C language (or a new operation in modern object-oriented programming languages). For example, a function pointer can be changed to point to attacker's code.

- **Array Indexing Errors** This type of attack is different from the other types in that it is a result of indexing beyond the boundary of an array. Thus, the attackers can theoretically write to arbitrary memory locations.

2.2 Classification Using Characteristics

There is another classification method based on characteristics, defined by [2], with some preconditions as follows.

- **Direct Executable** The target is to change the control flow of the process. This class is comparable to Stack overflows on control data. (`dir:exec = {len:buff, con:addr, con:inst, mod:radd, jmp:stack, exe:stack}`
- **Indirect Executable** The difference from "Direct Executable" is that the process state information, such as a return address, is not altered, but a function pointer is indirectly altered instead. When the function pointer is invoked, attacker's code will be executed. This class is comparable to Heap overflows on control data. `ind:exec = {len:buff, con:addr, mod:fptr, jmp:heap, exe:heap}`
- **Direct 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. This class is comparable to the Stack overflows on non-control data. `dir:data = {len:buff, con:ctrl, mod:cvar, flow:ctrl}`
- **Indirect Data** These attacks are similar to the direct data overflows. The target of indirect data buffer-overflow attacks is a pointer referring to the data that can change the execution path. This class is comparable to Heap overflows on non-control data. `ind:data = {len:buff, con:addr, mod:cptr, flow:ctrl}`

2.3 Summary of buffer-overflow attack types

From the patterns of buffer-overflow characteristics and the taxonomy of buffer-overflow solutions, the relationships between characteristics and existing solutions can be summarized as shown in Figure 1. From the table, the symbol ✓means this solution can prevent this characteristic. The symbol "?" means the solution may prevent this characteristic (depending on the implementation).

The protection solutions in the table are as follows: Segmentation [25], Integer Analysis to Determine Buffer Overflow [39], STOBO [14], Type-Assisted Buffer Overflow Detection [19], C Range Error Detector (CRED) [32], Jump Pointer [36], StackGuard [7], MemGuard [7], PointGuard [6], SmashGuard [6], Minezone RAD [37], Read-only RAD [37], Efficient Dynamic Taint Analysis Using Multicore Machines [3], HeapDefender [20], Secure Bit [27], and Secure Canary Word [29] [4]. For comparison, we also include our proposed solution, Boundary Bit, in this table.

In conclusion, a buffer-overflow protection summary table with types of buffer-overflow attacks is provided as Figure 2. The symbol ✓means this solution can prevent this attack type. The symbol "?" means this solution may prevent this attack type.

However, the tables show only types of buffer-overflow attacks that can be prevented without considering the performance or the limitations.

3. BOUNDARY BIT

Boundary Bit [5] is a hardware-assisted runtime bound-checking method that aims to prevent buffer-overflow attacks on both control and non-control data.

To enforce bound checking, Boundary Bit associates an extra bit with each byte of memory. These bits are similar to those of Secure Bit's [27], the Tag architecture [12], as well as the bound information used by various software-based bound-checking approaches [1,23,24], and are used to delimit boundaries of buffers. At runtime, these extra bits are used to check whether an access to a buffer is out-of-bounds. If an out-of-bounds access is detected, the offending program is terminated. Specifically, given a buffer at address `a`, if an attempt is made to access data at index `i`, Boundary Bit will scan bits from the address `min(a,a+i)` to `max(a,a+i) - 1` in order to determine whether an access is within the bounds. If, during the scan, Boundary Bit encounters a set bit (which signifies an end of the buffer), then it will terminate the program with an error. In case the ending address (`max(a,a+i) -1`) is less than the beginning address (`max(a,a+i)`), no scanning is required.

The following examples further illustrate how Boundary Bit works in practice.

3.1 Stack-Overflow Detection

The following function contains a potential buffer-overflow bug caused by the usage of the unsafe `strcpy()` function.

```
void func1(char *p) {
  int i;       // 4 bytes
  char b[8];   // 8 bytes
  char ch;     // 1 byte
  strcpy(b, p);
}
```

Assuming that each variable in the above function has an address in memory as shown in Figure 3a and 3b, if the length of input is 8 bytes (the maximum index of the input is 7), the system will scan bits starting at address `0x28ac58` and ending at address `0x28ac5e` (from `0x28ac58` + 7 - 1), as per the earlier

Characteristics	len:buff	con:addr	con:inst	con:ctrl	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	✓													✓

Fig.1:: Summary with buffer-overflow characteristics

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	✓	✓	✓	✓	✓	✓

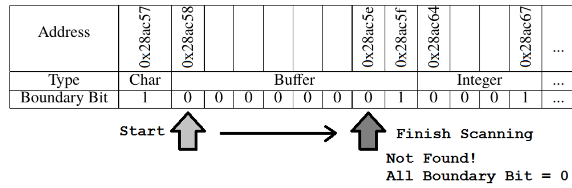
Fig.2:: Summary with types of buffer-overflow attacks

described scheme). Since there is no set bit in this range, as shown in Figure 3a, the execution will finish without any error.

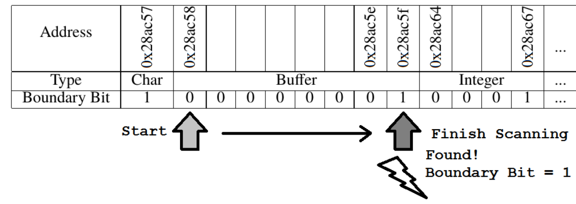
On the other hand, if the length of input is 9 bytes, the system will scan bits starting at address `0x28ac58` and ending at address `0x28ac5f`. As shown in Figure 3b, there is a set bit at address `0x28ac5f` which means that we have gone beyond the boundary of the array `b[]`. In this case, Boundary Bit will abort the execution of the function with an error.

3.2 Detection of Array-Indexing Errors

In this example, the following function contains a potential array-indexing error (bug).



(a) An example of scanning for Boundary Bit when the access is within the bounds of an object.



(b) An example of scanning for Boundary Bit when the access goes out of bound and an buffer-overflow is detected.

Fig.3:: Mechanism of Boundary Bit

```
void func2(int j) {
  int i;       // 4 bytes
  char b[8];   // 8 bytes
  char ch;     // 1 byte
  b[j] = 1;
}
```

If one assumes that the variables in the function `func2()` have the same addresses in memory as their counterparts in the function `func1()` in Section 3.1 , and the input (`j`) to the function `func2()` is 7, the system will scan bits starting at address `0x28ac58` and ending at address `0x28ac5e` (`0x28ac58` + 7 - 1). There is no set bit in this range, which is expected since this index is within the object's bounds.

If instead the input (`j`) is 8, the system will scan bits starting at address `0x28ac58` and ending at address `0x28ac5f`. In this case, there is a set bit at address `0x28ac5f` which signifies that an array-indexing error has occurred.

If the input value `j` is -1, the system will scan bits starting at address `0x28ac57` and ending at address `0x28ac57` (`0x28ac58` - 1). There is a set bit at address `0x28ac57` which signifies that an array-indexing error has occurred.

Thus, Boundary Bit can check both the upper bound and the lower bound of buffers.

For a 1-byte variable, the following function also contains a potential array-indexing error (bug).

```
void func3(int j) {
  int i;       // 4 bytes
  char b[1];   // 1 byte
  char ch;     // 1 byte
  b[j] = 1;
}
```

If the input (`j`) is 0, the system will scan bits starting at address `0x28ac58` and ending at address `0x28ac57` (`0x28ac58` - 1). In this case, there is no scanning because the ending address `0x28ac56` is less than the starting address `0x28ac57`.

If the input (`j`) is -1, the system will scan bits starting at address `0x28ac57` and ending at address `0x28ac57` (`0x28ac58` - 1). In this case, there is a set bit at address `0x28ac57` which signifies that an array-indexing error has occurred.

4. IMPLEMENTATION

There are two components of the Boundary Bit implementation: hardware runtime support, and a compiler modified to insert special instructions needed by the runtime.

4.1 Hardware Runtime Support

Two modifications to the hardware are needed to implement the runtime support of Boundary Bit:

1. A modification to the processor that adds the following 3 new primitive instructions:
   setbb – set a boundary bit of a given address
   clrbb – clear a boundary bit of a given address
   scnbb – scan a given range of addresses for a set bit and terminate the running program if a set bit is found within the range
2. Extend the memory chipset with a Boundary Bit memory interface hardware subsystem for fetching and setting boundary bits.

Figure 4a shows a rough sketch of the design of the Boundary Bit memory interface. As can be seen in our proposed design, there is no need to modify the memory subsystem to support Boundary Bit. Instead, a portion of existing memory is used to store boundary bits. Figure 4b presents the internal schematic of the Boundary Bit interface and shows how the interface partitions the memory address (in a 32-bit system) of a byte (or word) into groups of 29 high-order bits and 3 bits and using them to locate the boundary bit that is associated with that particular byte.

4.2 Compiler

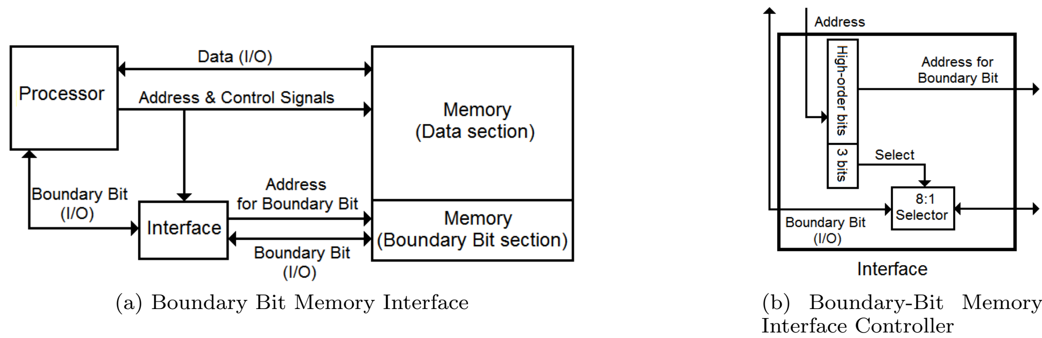
In addition to the aforementioned hardware modification, another important component of Boundary Bit is a compiler that has been modified to emit the `setbb`, `clrbb`, and `scnbb` instructions needed by the runtime during compilation.

For example, the following C code:

```
void func3(char *p) {
  int i;       // 4 bytes
  char b[8];   // 8 bytes
  char ch;     // 1 byte
  b[7] = p[0];
}
```

Which translates directly into the following assembly:

```
push  ebp
mov   ebp, esp
sub   esp, 1Ch
xor   eax, eax
```



(a) Boundary Bit Memory Interface

(b) Boundary-Bit Memory Interface Controller

Fig.4:: Memory Interface of Boundary Bit

```
; init value of ch
mov   dword ptr [ebp 14h], eax
; init value of i
mov   dword ptr [ebp 18h], eax
; init value of b[]
mov   dword ptr [ebp 4], 9DC86C9Bh
; init value of *p
mov   dword ptr [ebp 10h], ecx
cmp   dword ptr ds:[01474288h], 0
je    00000026
call  5B013C60
; get address from *p
mov   eax, dword ptr [ebp 10h]
; get value in memory of that address
movsx eax, byte ptr [eax]
; get address of b[0]
lea   edx, [ebp 0Ch]
; copy value from p[0] to b[7]
mov   byte ptr [edx+7], al
```

During the compilation, our modified compiler will insert the `setbb` instruction after each memory allocation operation in the program. Sample code is shown in the following snippet:

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

The compiler must also insert the `scnbb` instruction before each operation, such as `mov`, that writes to memory as shown in the following code snippet:

```
...
mov   eax, dword ptr [ebp 10h]
movsx eax, byte ptr [eax]
lea   edx, [ebp 0Ch]
scnbb [edx], 8              ; scan bb of b[]
mov   byte ptr [edx+7], al
...
```

Similarly, the compiler inserts the `clrbb` instructions after each memory deallocation operation.

5. OPTIMIZATION

As a consequence of the design of Boundary Bit, the larger the size of an array or struct, the longer it takes to scan for a boundary bit. To mitigate this runtime overhead, we introduce hardware bitmaps to act as a cache for storing boundary bits. With an `m-to-1` bitmap, a single bit in the bitmap cache can represent m boundary bits from a range of addresses, essentially replacing a scanning operation that has $O(n)$ runtime complexity with a table lookup operation with a constant (i.e., $O(1)$) runtime complexity, given a large enough value of m. For example, Figure 5 shows an example of a `16-to-1` bitmap, where one bit in the bitmap represents 16 boundary bits. If all 16 bits are 0, the bit in the bitmap will be 0. If at least one bit is one, the bit in the bitmap is set to 1. After that, it scans only 16 bits in the boundary bit section to find the set bit. Figure 6a shows a Boundary Bit memory interface that has been extended with a hardware bitmap that uses the boundary bit's address (which is up to 46 bits for the maximum of 2^{48} bytes in modern x86_64 architecture.) to index into the bitmap. As shown in Figure 6b, with a `16-to-1` bitmap, the boundary bit's address will be partitioned into groups of 42 bits and 4 bits, where the 42 bits portion will be used as the bitmap address while the 4 bits portion will be used to offset into the bitmap.

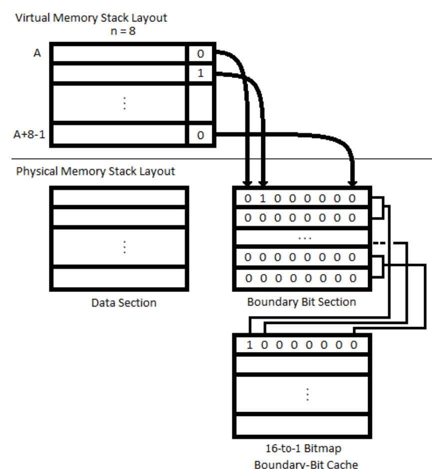
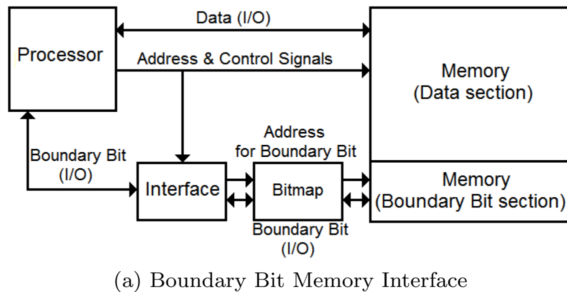
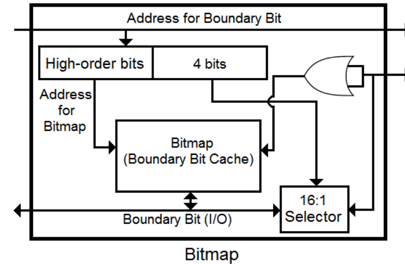


Fig.5:: Boundary Bit Bitmap Diagram



(a) Boundary Bit Memory Interface



(b) Boundary-Bit Memory Interface Controller

Fig.6:: Memory Interface of Boundary Bit with 1-Level Hardware Bitmap

Additionally, we can also implement multiple levels of bitmaps to further improve scanning speed. Figure 7 shows one possible configuration, where 2 bitmaps are layered on top of one another, with the level 1 bitmap functions as a cache for the level 2 bitmap.

The overall speed of the boundary bit technique depends on the cache. We strongly believe that cache can be redesigned and optimized to meet the performance required. Previously, Secure Bit [?] has shown that the cache can be redesign to accommodate its boundary bits. The hit and miss in the boundary bit is likely to be a subset of the standard data cache. Furthermore, additional dedicated hardware can also be added to support the boundary scan.

6. EVALUATION

We have shown in our conceptual design that the boundary bit is designed to protect against buffer overflow. A simulation was conducted to show the effect of our design on overall performance.

We only evaluated the effect on the performance because the architectural-level simulation has been validated in previous work [4]. There is no need for low-level simulation here. Though it is not shown in our paper, we believe that hardware optimization can be implemented to cover the overhead of bitmap hardware. We have initially shown in our previous work [30] that caching the bit is possible. In addition, we have a pure software (compiler) implementation that uses a helper thread to function as a bitmap subsystem as well.

Therefore, we evaluated our proposed Boundary Bit design by implementing a Boundary Bit prototype in a custom simulator we created which was written in Microsoft Visual C++. The simulator has the following performance characteristics:

- Setting a single bit takes 1 cycle
- Clearing a single bit takes 1 cycle
- Scanning for a single bit in a byte (or a word) takes 1 cycle
- Reading one byte of memory takes 1 cycle [11]
- Writing one byte of memory takes 1 cycle [11]

The simulator was hosted on a 64-bit Windows 10 machine with 8 GB of RAM. Two programs were used to evaluate the performance of the Boundary Bit prototype: Intensive Read/Write Buffer (Bubble Sort), and Random Write with inputs generated from a trace file generator. It is worth clarifying that the base program is modified with instrumentation to capture only the memory (boundary bit) activity. This is not a full architecture simulation.

Slowdown percentage is calculated as follows:

$$Slowdown(\%) = \frac{Overhead}{ReadWrite} \times 100\%$$

- $ReadWrite$ means instruction cycles from read/write memory instruction
- $Overhead$ means instruction cycles from set/clear/scan boundary bit instructions

6.1 Intensive Read/Write Buffer (Bubble Sort)

The bubble sort program is used to evaluate how the Boundary Bit prototype performs under intensive read and write scenarios.

As the results from the simulation in Tables 1 and 2 and Figure 8a show, in the case where the input consists of 10,000 elements, the Boundary Bit (BB) enabled version of the bubble sort program is 36 times slower than the base version of the program that has no protection. The majority of the overhead came from the scanning operation (`scnbb`). Fortunately, most of this overhead can be eliminated by adding a bitmap cache, with the 16-to-1 bitmap cache augmented Boundary Bit being 11 times faster than the Boundary Bit version that does not have the bitmap

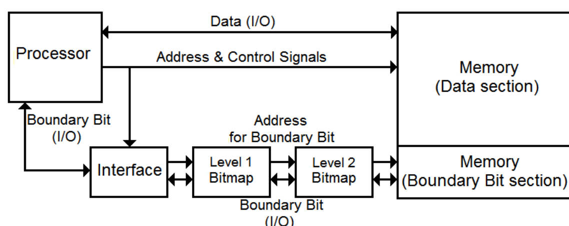
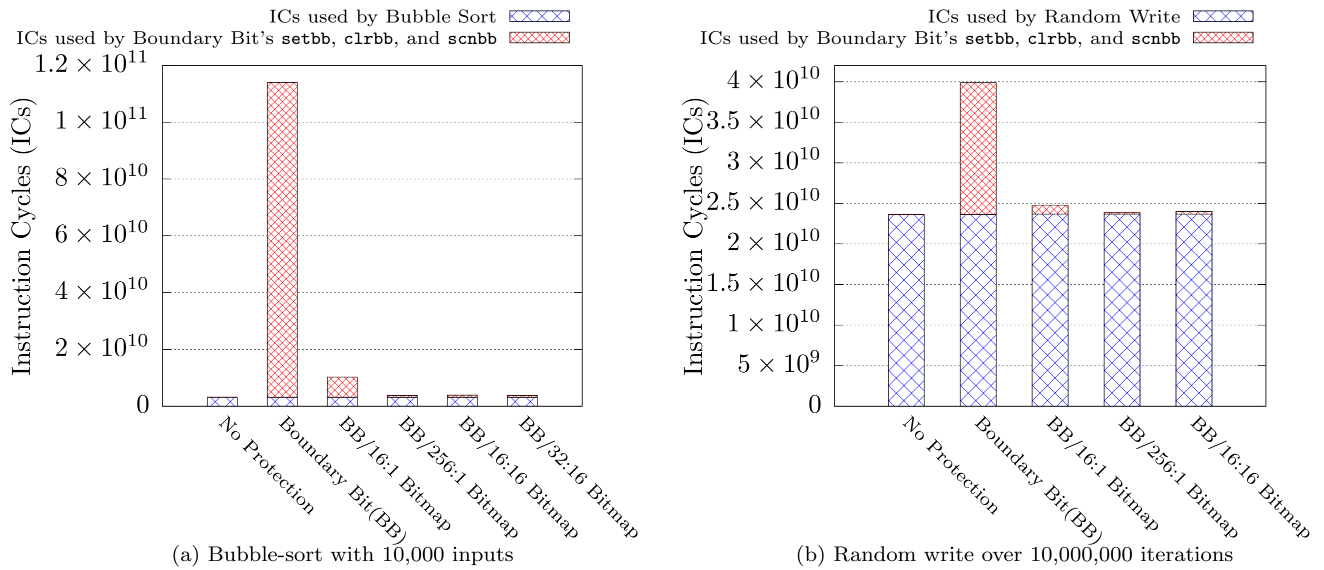


Fig.7:: Boundary-Bit Memory Interface with 2-Level Bitmap



(a) Bubble-sort with 10,000 inputs (b) Random write over 10,000,000 iterations

Fig.8:: Simulation results showing the performance overheads of Boundary Bit and the effect of using various configurations of hardware bitmaps to reduce the overhead

Table 1:: Overall Instruction Cycles in the Bubble Sort Program

Version	Cycles	Slowdown (%)
Original	3.146×10^9	-
Boundary Bit (BB)	113.991×10^9	3523.82
BB/16:1 Bitmap	10.271×10^9	226.53
BB/256:1 Bitmap	3.765×10^9	19.71
BB/16:16 Bitmap	3.890×10^9	23.67
BB/32:16 Bitmap	3.770×10^9	19.86

Table 2:: Overhead Cycles in the Bubble Sort Program

Version	BB	Bitmap	
		Level 1	Level 2
BB	110.845×10^9	-	-
BB/16:1	0.125×10^9	7.001×10^9	-
BB/256:1	0.099×10^9	0.520×10^9	-
BB/16:16	0.125×10^9	0.520×10^9	0.100×10^9
BB/32:16	0.125×10^9	0.297×10^9	0.203×10^9

cache, as can be seen in Figure 8a. Increasing the size of the bitmap or using more than one level of bitmap further reduces the overhead of Boundary Bit, with the 256-to-1 bitmap being 30 times faster than the non-bitmap version, the 2-level 16-to-1 bitmaps (16:16) being 29 times faster, and the 2-level 32-to-1 (as level 1) and 16-to-1 (as level 2) bitmaps (32:16) being 30 times faster. One of the things to note from the results is how increasing the size of the bitmap to a certain threshold and using multiple levels of bitmap can achieve similar levels of performance improvement.

6.2 Random Write

This program simulates intensive write to memory by randomly choosing to perform one of the following operations:

- write to character, integer, and double variables
- access a big array (100,000 entries)
- access a big memory block using a pointer (100,000 bytes)
- write data in various sizes to a big array (100,000 entries)

Table 3:: Overall Instruction Cycles in the Random Write Memory Program

Version	Cycles	Slowdown (%)
Original	23.651×10^9	-
Boundary Bit (BB)	39.881×10^9	68.62
BB/16:1 Bitmap	24.786×10^9	4.80
BB/256:1 Bitmap	23.838×10^9	0.79
BB/16:16 Bitmap	23.995×10^9	1.46

We ran the Random Write program 10,000,000 times for each configuration. Tables 3 and 4 and Figure 8b show the results, where the Boundary-Bit-enabled version of Random Write was roughly 1.69 times slower than the base version. Adding a 16-to-1 bitmap cache to the Boundary Bit version resulted in the 16-to-1 bitmap version being 1.60 times faster

Table 4: Overhead Cycles in the Random Write Memory Program

Version	BB	Bitmap	
		Level 1	**Level 2**
BB	16.230×10^9	-	-
BB/16:1	0.065×10^9	1.070×10^9	-
BB/256:1	0.065×10^9	0.122×10^9	-
BB/16:16	0.065×10^9	0.122×10^9	0.197×10^9

than the version that lacks the bitmap cache. Like in the case of the Bubble Sort program, increasing the size of bitmap and using more than one level of bitmap further reduces the overhead, with the 256-to-1 bitmap version being 1.67 times faster compared to the version of Boundary Bit that lacks bitmaps, and the 2-level 16-to-1 bitmaps being 1.66 times faster.

7. ANALYSIS

In this section, we will analyze the advantages and the disadvantages of Boundary Bit. In particular, we will also address the performance issue of Boundary Bit.

7.1 Advantages

First, Boundary Bit prevents many traditional buffer-overflow attacks, for example when the size of the input is larger than the size of the receiving buffer. It blocks attacks that utilize array-indexing errors.

Second, our approach uses little memory for storing metadata. Besides using less memory than most software solutions, boundary bit uses less memory for metadata compared to most hardware solutions as well. For example, for each variable, Segmentation [25] uses 3 words for metadata, one for starting address, another ending address (or limit), and the last word for current address. In contrast, Boundary Bit uses only one bit of metadata for each variable/array. Even for a large array, it uses only one bit.

Finally, the bit scanning operation can be accelerated using additional hardware so that the operation can be performed in parallel.

7.2 Disadvantages

Although the hardware-oriented approach has many strengths, it also has several weaknesses. One of which is the fact that current hardware cannot leverage Boundary Bit. However, since hardware has a life expectancy, it might be possible to replace machines with Boundary-Bit-enabled hardware when the time comes to replace obsolete hardware.

Another issue is the backward binary compatibility (transparency). As a result of the design, programmers or compilers must tell the system to set a bit at the end of any variable or buffer to mark the boundary. Thus, Boundary Bit is not completely transparent. It requires users to recompile programs using a compiler that has been modified to add the instructions needed by Boundary Bit.

7.3 Performance Analysis

Using a hardware bitmap cache can improve the efficiency of the bit scanning operation of Boundary Bit. As a result, the memory access time of bit-scanning can be greatly reduced. Moreover, the scanning can also be done in parallel with accessing associated data by modifying the processor. However, a hardware/software optimization is beyond the scope of this paper. 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 would be introduced.

7.4 Cost Analysis

To implement our boundary bit approach, we need to modify both hardware and software. In the hardware, additional memory usage is a fixed cost. In other words, the memory usage does not vary with the program size because the system allocates the block of memory for storing all boundary bits. Next, a processor must be modified to add instructions required by the boundary bit mechanism. These instructions can be made to execute in parallel. Thus, this should incur only a small performance penalty. It is a trade-off between performance and security. Lastly, the boundary bit instructions must be called in the software. This can be implemented by modifying a compiler. This is not transparent to the current software, but we believe this cost is acceptable for more security.

7.5 The Impact of Virtual Memory

In this section, we will describe how virtual memory impacts the boundary bit management when using virtual addresses in any modern systems. The information on how to manage the virtual memory is explained.

Virtual memory extends physical memory capacity by using swap space from a non-volatile direct access storage device (SSD, hard disk). It separates memory into "pages", which makes a program relocatable. The block of the Boundary Bit section can simply be 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, its boundary bits are swapped back into memory too.

7.6 Boundary Bit Protection

The boundary bit storage is not only desgined to store boundary bits, but also to protect them. Only a privileged user can modify boundary bits by using the `clrbb` instruction. To clarify, suppose an attacker wants to attack the system by bypassing the boundary bits to exploit a buffer-overflow vulnerability. Their target is to modify boundary bits stored in the memory and bitmap. The attacker may write the code for clearing all boundary bits. Then he or she will want to execute the code. However, the attacker must have already obtained the root privilege in order to execute such code. If they have that privilege, they already own the system and they do not need to clear the boundary bit.

Although the additional instructions execute in user mode, they can still prevent buffer-overflow attacks. This is possible because the hint from software will use the instructions to prevent writing beyond the boundary. Most buffer-overflow attacks occur from writing beyond the buffer. These are not privilege escalation attacks. Unless the attacker has already gained privileged mode access on the system, this is not an issue.

8. RELATED WORK : PROTECTIONS AGAINST BUFFER-OVERFLOW ATTACKS

A taxonomy of buffer-overflow protection schemes was established by Piromsopa [28, 31]. There are three broad categories: static analysis, dynamic solutions, and isolation. Static analysis schemes prevent the problem before deploying programs by parsing source code and warning the programmers of potential threats. Dynamic solutions verify the integrity of data during the execution of programs by creating meta data. Solutions in the isolation classes only limit the damage from the attacks.

Some approaches, similar to our work, are summarized in the following sections.

8.1 Bounds Checking at Runtime

Among these solutions, dynamic solutions can dynamically detect and prevent problems in run-time environments. Dynamic solutions can also be partitioned into many subclasses, such as address protection, input protection, bounds checking, and obfuscation. Boundary Bit is a member of the bounds checking subclass of dynamic solutions.

Since the 1980s, two major approaches dominate the runtime bounds checking approach. These are the pointer-based approach and the object-based approach.

8.2 Pointer-Based Approach

Early work [16, 18, 26, 38, 41] in the pointer-based approach associated bounds information with each pointer by modifying the pointer representation into what is typically called a fat pointer. While convenient, the major downside of this approach is the incompatibility between the code that uses fat pointers that the code that does not. This is a major concern because in practice it is not possible to recompile all programs and libraries to use the same pointer representation as the instrumented code.

8.3 Object-Based Approach

In this approach, each object is stored in a separate data structure and the address of an object is used to look up its associated bounds information. One of the earliest works in the object-based approach is [15], with subsequent works [9, 32] improving upon the concept by reducing overhead, increasing protection coverage, and further improving compatibility. Notable work in the object-based approach is the work by Akritidis et al. [1], which stores allocation bounds instead of precise bounds of each object. These allocation bounds are usually larger than the actual bounds of an object, while their size and alignment are constrained to facilitate efficient bounds lookups, which results in significantly lower overhead during runtime.

However, despite solving the incompatibility issue, the object-based approach has its own set of issues. One issue is the fact that it cannot detect buffer overflows in nested objects [23], such as an array inside a struct. This problem stems from the fact that in C, an address of an element inside a struct is the same as the address of the struct itself. This issue prompted the development of what we call the second generation pointer-based approach.

8.4 Pointer-Based Approach: the Second Generation

The second generation pointer-based approach aims to solve the aforementioned inability to detect overflows in nested objects inherent in the design of the object-based approach. In addition, it must avoid the incompatibility problem that plagued the old pointer-based works that utilized the fat pointer representation. It does this by borrowing the disjoint metadata concept from the object-based approach. Specifically, the second generation pointer-based approach stores the bounds information of each pointer inside a data structure, and uses the address of the pointer itself (and not the object) to look up the bounds information. This approach offers the same advantage as the object-based approach, namely compatibility between instrumented and uninstrumented code, while also being able to detect overflow in nested objects [23, 24].

8.5 Recent Architectural Approaches

Buffer-overflow attacks and memory access violation are still interesting issues, but the trade-off is also a discussed topic. Some proposed approaches are hardware solutions, because their main goal is to reduce runtime overheads in software by improving the hardware. Most of them use pipelines for parallel processing. Differences in details in implementations result from different assumptions and scope/limitations. However, they may not cover all types of buffer-overflow attacks.

For example, Low-Fat Pointers [17] and SMOV [21] modify the processor instructions for bound-checking. SMOV adds micro-operations to each stage of the secure mov instructions. Low-Fat Pointers is also a pointer-based approach implemented by adding hardware-managed tags to the pointer.

Programmable Unit for Metadata Processing (PUMP) [8] processes metadata tags, which are propagated with data or memory address, by modifying some related hardware (registers, caches, and pipelines) and deals with compilers.

A work on adaptive pipeline [34] allows a transparent solution to programmers/compilers and has very low overhead. However, it mainly focuses on stack-based buffer-overflow attacks during runtime.

8.6 Comparison

Compared to the aforementioned bounds checking approaches, Boundary Bit can be considered a hardware implementation of the object-based approach. The bounds information is stored as a single bit in an isolated region inside memory. The first obvious advantage of Boundary Bit compared to other software-oriented approaches is speed, since the hardware that handles the lookup and bounds checking operation can be made to run in parallel with other operations. A second advantage is improved security, since the region where the Boundary Bit stores its metadata is isolated by the MMU itself, making it substantially harder to tamper with. Another advantage of Boundary Bit compared to other object-based bounds checking approaches is the amount of memory required to store the bounds information. In the traditional object-based approach, bounds information usually consists of a base and a bound of an object, both of which are usually the same size as the pointer. In contrast, in Boundary Bit the bounds information of each object is just 1 bit. With that said, the obvious disadvantage of Boundary Bit is that since it is a hardware solution, it is significantly harder to develop and deploy compared to software solutions. Also, in this work we still have not addressed the inability to detect buffer overflows in nested objects that plagued many works in the object-based approach.

8.6.1 Comparison of protection coverage

In the view of the coverage of buffer-overflow attack types, most of the former buffer-overflow protection schemes cannot protect the system from all types.

For example, Segmentation [25], Type-Assisted Buffer Overflow Detection [19], C Range Error Detector (CRED) [32], Integer Analysis to Determine Buffer Overflow [39], STOBO [14] and MemGuard [7] can detect stack/heap overflows on both control and non-control data, but some cannot detect array indexing errors. Jump Pointer [36] and adaptive pipeline [34] focus on stack overflows. StackGuard [7], SmashGuard [7], Minezone RAD [37] and Read-only RAD [37] can detect stack overflows, but focus on control data only. PointGuard [6] and HeapDefender [20] focus on heap overflows.

Secure Bit [27], Canary Bit [33], Secure Canary Word [4,29], and Boundary Bit [5] form a series of related research work with each new project building upon its predecessors. Secure Bit can detect all types, but focuses on only control data, similar to Efficient Dynamic Taint Analysis Using Multicore Machines [3]. Canary Bit extends Secure Bit to focus on control data by marking control address/data. Secure Canary Word based on Secure Bit can detect most types except array indexing errors on non-control data. Boundary Bit extends Canary Bit to prevent all types, both on control data and non-control data.

Low-Fat Pointers [17] and SMOV [21] have the same goal as Boundary Bit. However, these architectural approaches mainly aim for high performance, but the coverage of detected error types is not explained in detail.

9. CONCLUSION

The main concept of Boundary-Bit is checking to ensure that transferring of data does not exceed the allocated capacity of variables or buffers. Its goal is to provide a hardware solution that protects against all types of buffer-overflow attacks, including non-control data attacks and array-indexing errors, with lower overhead than other solutions. We trade some performance and additional hardware complexity for more security. The performance of Boundary Bit can be improved by using bitmaps as boundary-bit caches.

Moreover, Boundary Bit is simple to implement. Few software modifications are required to deploy this scheme. We have demonstrated viability at the architectural level. This solution can also be implemented in a software run-time environment such as Java Virtual Machine or .NET framework.

Boundary Bit provides bound checking at the architectural level. This mechanism is able to provide protection against future buffer-overflow attacks. Given the security provided, we believe Boundary

Bit is a viable solution for protection against buffer-overflow attacks.

ACKNOWLEDGMENTS

The authors would like to thank the Graduate School of Chulalongkorn University for awarding a Chulalongkorn University Graduate Scholarship to Commemorate the 72nd Anniversary of His Majesty King Bhumibol Adulyadej, and the department of Computer Engineering, Chulalongkorn University, for providing a Graduate Scholarship.

References

[1] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *18th USENIX Security Symposium (USENIX Security '09)*, pp.1-10, USENIX, August 2009.

[2] M. Bishop, S. Engle, D. Howard and S. Whalen, "A Taxonomy of Buffer Overflow Characteristics," in *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 3, pp. 305-317, May-June 2012.

[3] M. Chabbi, S. Perianayagam, G. Andrews, and S. Debray, "Efficient dynamic taint analysis using multicore machines," *Report*, The University of Arizona, 2007.

[4] S. Chiamwongpaet and K. Piromsopa, "The implementation of Secure Canary Word for buffer-overflow protection," *2009 IEEE International Conference on Electro/Information Technology*, Windsor, ON, 2009, pp. 56-61.

[5] S. Chiamwongpaet, *Buffer-overflow Protection using Boundary Bit*, Thesis, 2017.

[6] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard^{TM}: protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th conference on USENIX Security Symposium*, vol.12, pp. 7, 1251360.USENIX Association, August 2003.

[7] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th conference on USENIX Security Symposium*, vol. 7, pp. 5, 1267554. USENIX Association, January 1998.

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

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

[10] Nicolas Economou. Microsoft windows up to 8.1 memory object win32k.sys buffer overflow. `http://www.scip.ch/en/?vuldb.11444`, 04/07/2017 2013.

[11] Agner Fog. Instruction tables. `http://www.agner.org/optimize/instructiontables.pdf`, 2017-05-02 2017.

[12] Edward F. Gehringer and J. Leslie Keedy, "Tagged architecture: how compelling are its advantages?," *SIGARCH Comput. Archit. News*, vol.13, no.3, pp.162–170, June 1985.

[13] R. Gil, *The undefined quest for full memory safety*, Thesis, 2017.

[14] E. Haugh, "Testing c programs for buffer overflow vulnerabilities," in *Proceedings of the Network and Distributed System Security Symposium*, pp. 123–130, 2003.

[15] Richard W M Jones and Paul H J Kelly, "Backwards-compatible bounds checking for arrays and pointers in c programs," in *Distributed Enterprise Applications*. HP Labs Tech Report, pp. 255–283, 1997.

[16] S. Kaufer, R. Lopez, and S. Pratap, "Saber-C — an interpreter-based programming environment for the C language," in *USENIX Association, editor, Summer USENIX Conference Proceedings*, pp. 161–171, Berkeley, CA, USA, Summer 1988. USENIX.

[17] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, G. Bioworks, and A. Dehon, "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," *CCS'13: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp.721-732, November 2013.

[18] J. L. Steffen, "Adding run-time checking to the portable c compiler," *Software: Practice and Experience*, vol.22, issue 4, pp.305–316, April 1992.

[19] Kyung-Suk Lhee and S. J. Chapin, "Buffer overflow and format string overflow vulnerabilities," *Software: Practice and Experience*, vol.33, issue 5, pp.423–460, April 2003.

[20] D. Li, 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*, pp. 851–856, 2012.

[21] A. Maia, L. Melo, F. M. Q. Pereira, O. P. V. Neto, and L. B. Oliveira, "Smov: Array bound-

check and access in a single instruction," in *2016 13th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pp. 745–751, Jan 2016.

[22] Sean Dillon (Microsoft). Microsoft windows up to server 2016 smb buffer overflow. https://vuldb.com/?id.98018,07/14/2017 2017.

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

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

[25] Elliot I. Organick. *A programmer's view of the Intel 432 system*. McGraw-Hill, Inc., 1983.

[26] H. Patil and C. Fischer, "Low-cost, concurrent checking of pointer and array accesses in c programs," *Software—Practice & Experience*, vol.27, issue 1, pp.87–110, January 1997.

[27] K. Piromsopa and R. J. Enbody, "Secure Bit: Transparent, Hardware Buffer-Overflow Protection," in *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 4, pp. 365-376, Oct.-Dec. 2006

[28] K. Piromsopa, *SECURE BIT: BUFFEROVERFLOW PROTECTION*, Thesis, 2006.

[29] K. Piromsopa and S. Chiamwongpaet, "Secure Bit Enhanced Canary: Hardware Enhanced Buffer-Overflow Protection," *2008 IFIP International Conference on Network and Parallel Computing*, Shanghai, 2008, pp. 125-131.

[30] K. Piromsopa and R. J. Enbody, "Architecting security: A secure implementation of hardware buffer-overflow protection," in *Proceedings of the Third Conference on IASTED International Conference: Advances in Computer Science and Technology, ACST'07*, pp. 17–22, USA, 2007. ACTA Press.

[31] K. Piromsopa and R. J. Enbody, "Survey of protections from buffer-overflow attacks," *Engineering Journal*, vol.5, no.2, pp.31–52, Feb. 2011.

[32] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector," in *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pp. 159–169, 2004.

[33] M. S. Kirkpatrick, *Canary bit: Extending secure bit for data pointer protection from buffer overflow attacks*, Thesis, 2007.

[34] L. K. Sah, S. A. Islam and S. Katkoori, "An Efficient Hardware-Oriented Runtime Approach for Stack-based Software Buffer Overflow Attacks," *2018 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, Hong Kong, 2018, pp. 1-6.

[35] Charles Schmidt and Tom Darby. The what, why, and how of the 1988 internet worm. http://www.snowplow.org/tom/worm/worm.html, 1988.

[36] Z. Shao, Q. Zhuge, Y. He and E. H. -. Sha, "Defending embedded systems against buffer overflow via hardware/software," *19th Annual Computer Security Applications Conference, 2003. Proceedings.*, Las Vegas, NV, USA, 2003, pp. 352-361.

[37] Tzi-Cker Chiueh and Fu-Hau Hsu, "RAD: a compile-time solution to buffer overflow attacks," *Proceedings 21st International Conference on Distributed Computing Systems*, Mesa, AZ, USA, 2001, pp. 409-417.

[38] D. W. Plater, Y. Yesha and E. K. Park, "Extensions to the c programming language for enhanced fault detection," *Software: Practice and Experience*, vol.23, issue 6, pp.617–628, June 1993.

[39] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *NETWORK AND DISTRIBUTED SYSTEM SECURITY SYMPOSIUM*, pp.3–17, 2000.

[40] Webopedia. buffer overflow. http://www.webopedia.com/TERM/B/bueroverow.html.

[41] Marvin V. Zelkowitz, Paul R. McMullin, Keith R. Merkel, and Howard J. Larsen, "Error checking with pointer variables," in *Proceedings of the 1976 Annual Conference, ACM '76*, pp. 391–395, New York, NY, USA, 1976. ACM.

Sirisara Chiamwongpaet received her B.Eng and M.Eng degrees in Computer Engineering from Chulalongkorn University in 2008 and 2010, respectively. She is now working toward her Ph.D. in Computer Engineering at Chulalongkorn University.

Krerk Piromsopa received his B.Eng. and M.Eng. degrees in computer engineering from Chulalongkorn University, Thailand, in 1998 and 2000, respectively. He has been a faculty member at this school since 2001. In 2003, he was awarded a scholarship from the Royal Thai Government for pursuing his Ph.D. degree in computer science at Michigan State University, where he received the degree in 2006. He is currently an associate professor in computer engnieering at Chulalongkorn University. His research interests are in computer security, computer architecture, distributed systems, and data analytics in healthcare. He is a member of the ECTI, the IEEE, and the IEEE Computer Society.