SECURE BIT: BUFFER-OVERFLOW PROTECTION

By

Krerk Piromsopa

A DISSERTATION

Submitted to
Michigan State University
In partial fulfillment of the requirements
For the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science and Engineering

2006

ABSTRACT

SECURE BIT: BUFFER-OVERFLOW PROTECTION

By

Krerk Piromsopa

For decades, buffer-overflow attacks have remained the most persistent threat to the computer security world. The most common type of buffer-overflow attacks is an attack that changes the control flow by overflowing control data.

In this thesis, Secure Bit, architectural approach, is proposed to protect against buffer-overflow attacks on control data (return-address and function-pointer attacks in particular). Secure Bit provides a hardware bit to enforce the integrity of addresses from being modified by external data (input). Secure Bit is completely transparent to user software; providing full backward compatibility with legacy user code. It can detect and prevent all address-corrupting buffer-overflow attacks with little run-time performance penalty. Addresses passed in buffers between processes are marked insecure and control instructions using those addresses as targets will raise an exception. An important differentiating aspect of this protocol is that once an address has been marked as insecure there is no instruction to remark it as secure.

To validate Secure Bit, we first theoretically pursue a secure system with respect to buffer-overflow attacks and prove that Secure Bit provides a sufficient condition for preventing buffer-overflow attacks. Robustness and transparency are demonstrated by emulating the hardware, and booting Linux on the emulator, running application software

on that Linux, and performing known attacks. In addition to the cost analysis and issues related to the success of Secure Bit, we also suggest possible attacks that may not be protected by Secure Bit.

In addition to the proposed Secure Bit, this thesis also provides a survey of current approaches against buffer-overflow attacks. Notably, approaches are conceptually grouped into three broad categories providing a platform for studying buffer-overflow protection schemes.

For dad, the foremost inspiration of my life

ACKNOWLEDGEMENTS

I would like to express my gratitude to all who contributed to the completeness of this thesis. Royal Thai Government and Department of Computer Engineering, Chulalongkorn University, Thailand gave me the opportunity to pursue this Ph.D. in the first place. I thank to everybody at the Department of Computer Science and Engineering and Michigan State University for making East Lansing a home away from home for the past few years.

It is my honor to work with the great advisor Prof. Dr. Richard J. Enbody. He helped me with not only the research, but also with my writing. Without his principle of keeping things simple, this thesis is not possible. If I am to be a good professor, he is certainly my role model. I am deeply indebted to him.

I thank to the committee members: Dr. Anthony Wojcik, Dr. Pang-Ning Tan, and Dr. Michael Shanblatt. Their invaluable suggestions and comments helped me to improve this thesis in many ways.

Last but not least, I thank my parents, my brother, and my wife for supporting me through these years. They made my life worth living.

Preface

This thesis is divided into ten main chapters: introduction, review, theory, fundamentals, design, implementation, possible attacks, evaluation, analysis and conclusion. Though each chapter is written to be self-contained, reading the whole document in order is recommended.

Chapter 1, the introduction, presents the background of memory management from a system point of view, and the basic concept of buffer overflow. Through the chapter, readers are expected to learn the importance of protection against buffer overflow attacks.

Chapter 2, the background, reviews several approaches currently used by programmers, and architects to defeat buffer overflow. We generally try to explain a concept and point out the strength and weakness of each approach. Indirectly, we conclude that buffer overflow still needs a better solution.

Chapter 3, the theory, is intended to theoretically pursue a secure system with respect to buffer-overflow attacks. We begin by defining buffer overflows in general. Later in this chapter, we establish a sufficient condition for preventing buffer-overflow attacks and prove that it will create a secure system with respect to buffer-overflow attacks.

Chapter 4, the fundamentals of Secure Bit, proposes Secure Bit. In this chapter, readers will learn the concept of Secure Bit as an architectural approach for preserving the integrity of an address against buffer-overflow attacks.

Chapter 5, the design, is the discussion of architectural issues critical to the implementation and deployment of Secure Bit. This chapter provides sufficient concepts for adapting Secure Bit to any existing architecture.

Chapter 6, the implementation, elaborates the implementation details of Secure Bit in the BOCHS emulator and the modification necessary to the Linux kernel. Through the chapter, readers are expected to learn the impact of Secure Bit to both the architecture and the systems level.

Chapter 7, the possible attacks, is the analysis of possible methods that may be used to circumvent the protection provided by Secure Bit. We will present other types of attacks that are not protected by Secure Bit and discuss the possible solutions.

Chapter 8, the evaluation, is the assessment of protection provided by Secure Bit.

Chapter 9, the analysis, provides cost analysis and issues related to success of Secure Bit.

Chapter 10, the conclusion, summarizes the thesis and provides possible researches and applications in applying Secure Bit.

Finally, we hope to see the success of this research as an architecture solution against buffer overflow.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1 Introduction

This chapter is intended to be an introduction to buffer overflows. We begin by revisiting the memory management of processes in generic operating systems. Based on the memory management, we continue to explore buffer overflows in general, and return-address attacks and function-pointer attacks in particular. Included is an example of a multistage buffer-overflow attacks which can bypass most solutions.

1.1 Memory Management of Processes

Knowledge of how modern operating systems handle the memory of processes is required in order to understand how buffer overflows happen. From a programmer's point of view, the memory of each process is partitioned into text (code) and data. Similarly, the operating system partitions the memory into several sections (also referred to as segments). Apart from the text and the data, the operating system also allocates memory for the heap, the BSS, and the stack. This memory model is applied to both UNIX-like systems and WINDOWS systems. Figure 1 shows the model and growing direction of each segment.

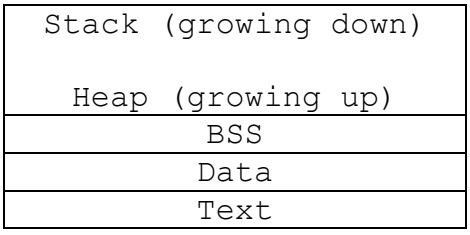

Figure 1 Memory management model of a process

The text segment is the area where the program binary is stored. The data segment consists of all initialized global data and global variables. The BSS (Block Started by Symbol) is allocated for additional space determined at compile time. This space stores

static variables and all uninitialized global variables. Compilers are responsible for generating these three segments. The heap is the space that a program can dynamically allocate at execution time. The stack is typically used for storing local variables, arguments and return addresses. A snapshot of data stored in the stack is shown in Figure 1.

While the text segment can easily be protected by labeling as read only, the others cannot. Thus, it is possible for users to arbitrarily modify data in those areas. In the next section we learn a class of attack where input is used to maliciously modify control data and cause a program to be in unexpected state. This kind of attack is known as a "buffer-overflow attack".

1.2 Buffer-Overflow Attacks

We begin by establishing the fundamentals of buffer-overflow vulnerabilities and attacks. Later we present variations on buffer-overflow attacks.

1.3 Fundamental of Buffer-Overflow Attacks

Several researchers ([12, 15, 23, 27, 38, 46]) have positioned that there are two necessary conditions for buffer-overflow attacks to be successful: (1) injecting malicious code and (2) redirecting the program control flow to execute that code. Although it is common, injecting malicious code is not necessary since the code can be resident code found in shared libraries. For example, jumping to resident shell code while in privileged mode is sufficient. Therefore, we claim that only the second condition is truly necessary.

We first take a quick look at buffer overflow vulnerabilities and how they were first exploited. A buffer overflow is an anomalous condition where a program somehow writes data beyond the allocated end of a buffer in memory [76, 81]. The eventual result is an access to unexpected data [56, 11]. A buffer-overflow attack is a buffer overflow that overwrites critical data and results in malicious or unexpected behavior. Examples range from denial of services to gaining privileged control over the target system [56, 11]. In all cases, the malicious data in an attack are input data, such as data from another domain (e.g., user inputs, network packets, data passed from another process), that was not verified. It is possible for buffer overflow to take place entirely within a process, but such a condition usually results in segmentation fault rather than an attack.

With this observation, we define a ***buffer-overflow attack*** to be

> ***an attack caused by overflowing a buffer with data from another domain which results in malicious or unexpected behavior of a program.***

This concept is not new. For example, Howard and LeBlanc state in their book "**All input is evil until proven otherwise**" [**32**] Accordingly, an intuitive solution against buffer-overflow attacks is an ability to detect and validate input, especially input which is eventually used for control.

Using the concept that inputs are the origin of attacks, a key component for detecting and validating the input is metadata. Metadata is additional information on the properties of data. It can be one or a combination of: type descriptor, guard value, encoding key, redundancy copy of data, tagged value, or even programming logic. This concept is also

not new. It was suggested that metadata is a key to preventing buffer-overflow attacks in the 2003 article [29]. In this chapter we will use the variety of metadata and its management as a way of classifying buffer-overflow prevention schemes.

Note that metadata can also be used for other purposes, e.g. tagged architectures such as Symbolics [47] or tagged operating systems [28]. However, the main focus of this thesis is on the use of metadata for buffer-overflow attack prevention.

In summary, buffer-overflow attacks occur when a malformed input is being used to overflow a buffer causing a malicious or unexpected result. Some metadata is necessary for prevention. In Chapter 2, we will present a variety of schemes for handling buffer-overflow attacks. Most of them use metadata.

1.4 Sample Attacks and Variations

There are two main targets of buffer-overflow attacks: control data and local variables. In the vast majority of attacks, control data is the target so prevention schemes have focused on control data. Control data can be divided into several types: return addresses, function pointers, and branch slots. Return addresses have been the primary target since their location can easily be guessed. More advanced buffer-overflow attacks target other control data. Some literature refers to attacks on return addresses as first-generation attacks, and those on function pointers as second-generation attacks [11].

The first step-by-step description of how to construct a buffer-overflow attack was written by Elias Levy (a.k.a. Aleph One) in 1996 [51]. In his paper, the term “Stack

Smashing" is probably first used to refer to the plastering of stack with shell code and its address to set up an eventual overflow of a return address on the stack.

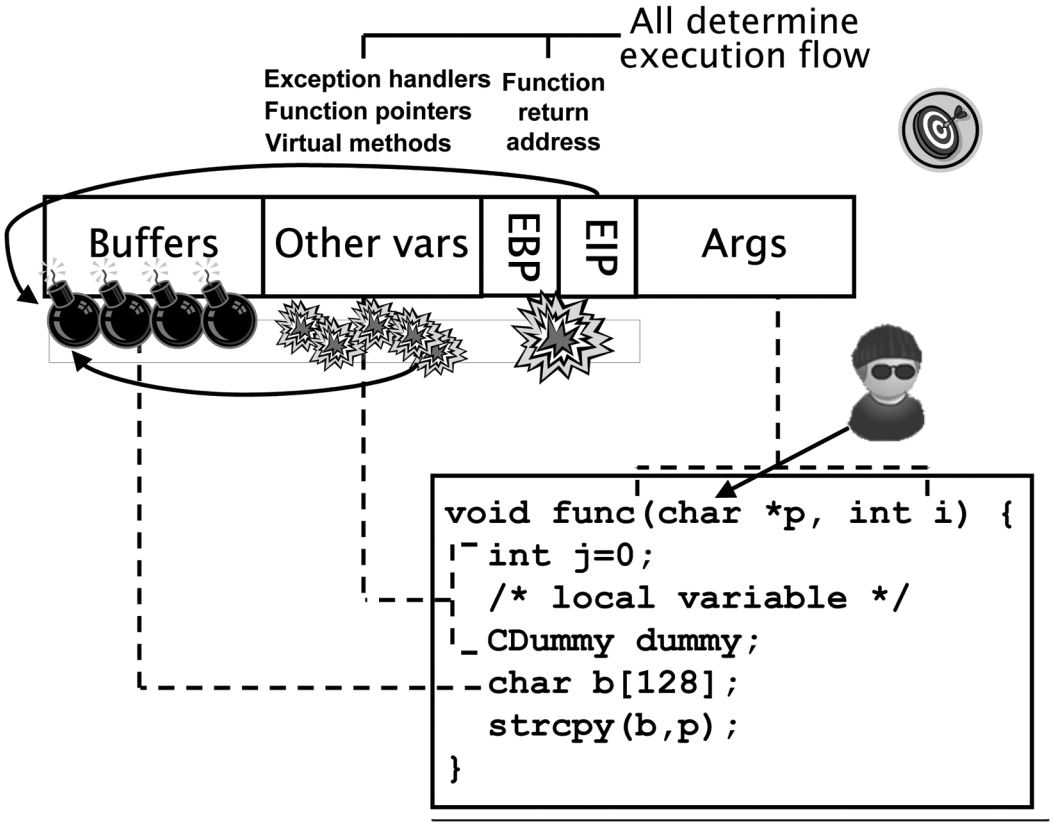

Figure 2 Stack Smashing

Figure 2 shows a stack-smashing example. In this example, attackers (shown in the black hat) will pass a buffer containing malicious code (e.g. shell) and multiple copies of the address of the target buffer as an argument to the vulnerable program (through parameter *p*). A buffer manipulation function (e.g. strcpy in this example) will overflow the function's return address with the address of the target buffer (which contains malicious code). The eventual result is that the return instruction will use the address of the target buffer as a return address and return the program flow to execute that malicious code. Additional attack vectors are provided but not used in the code such as function pointer *fp*.

As outlined above, it is important to note that the critical component of this attack is the modification of the return address (replaced by the address of the target buffer).

To illustrate an advanced buffer-overflow attack, we provide a multistage buffer-overflow attack (a.k.a. Hannibal Exploit [19]) that can bypass most software buffer-overflow solutions. Fundamental to a multistage buffer-overflow attack is that there exists a vulnerable pointer to a buffer. That is, there is a user-writeable buffer sufficiently near a useful pointer. First, the pointer is modified (by overflowing) to point to a specific location (e.g. a jump slot or a function pointer). In the second stage of the attack, an input is stored at the pointer's target. These two steps allow attackers to create a pointer to any location (first stage) and overwrite the pointer's target with a desired value (second stage). The next time some program jumps to that target, it will be redirected based on the value inserted by the attacker. In particular the program will be redirected to the attacker's malicious code. For example, if the program is running in privileged mode and the pointer points to shell code, the attacker will have created a privileged shell allowing free reign. Figure 3 is an example of such a vulnerable program.

Before examining the code, let's review how a jump table is used. Consider the slot in the table for the pointer to the *printf* executable. A call to *printf* indexes to that slot in the table and then jumps to the *printf* executable.

To attack this type of program, the buffer-overflow is done in two stages. First, the *ptr* pointer is overflowed to point to a desired memory location (1), e.g. the *printf* slot in the

jump table. In particular, *argv[1]* controlled by the attacker will contain the address of the *printf* slot in the jump table. The strcpy routine will copy *argv[1]* into *buffer,* but overflows to overwrite *ptr* with the *printf* address slot. In the figure, we see that the pointer *ptr* originally pointed to 'buffer' (arc labeled 'Before') but now points to the jump slot (arc labeled 'After). Now that *ptr* points to the *printf* slot in the jump table, we need to insert a desired value into that slot.

Suppose, for illustration, that we also have determined the address of resident shell code (we will call it *residentcode*). Using our modified *ptr* we will overwrite the jump table slot with the *residentcode* address. We use the second strcpy call (2) to write *argv[2]* (also controlled by the attacker and whose value is *residentcode*) into the target of *ptr* which now points to the *printf* entry in the jump table. The result of that second strcpy call is that we have placed the address *residentcode* (resident shell code) into the *printf* slot of the jump table. The attacker has achieved his goal. Now when a program calls *printf,* control passes as usual to the *printf* entry in the jump table, but now the attacker has redirected control to *residentcode,* the address of the shell code. Instead of *printf* a shell will be started. If the program which called *printf* was operating in privileged mode, the attacker will have succeeded in creating a privileged shell with full system access. (See [57] for more details). As we will see below, this multi-stage attack gets around most buffer-overflow protection schemes. Less obvious is that we can use a similar approach to circumvent some software solutions to buffer-overflow attacks by modifying a handling vector which can allow us to bypass the buffer-overflow handling routine.

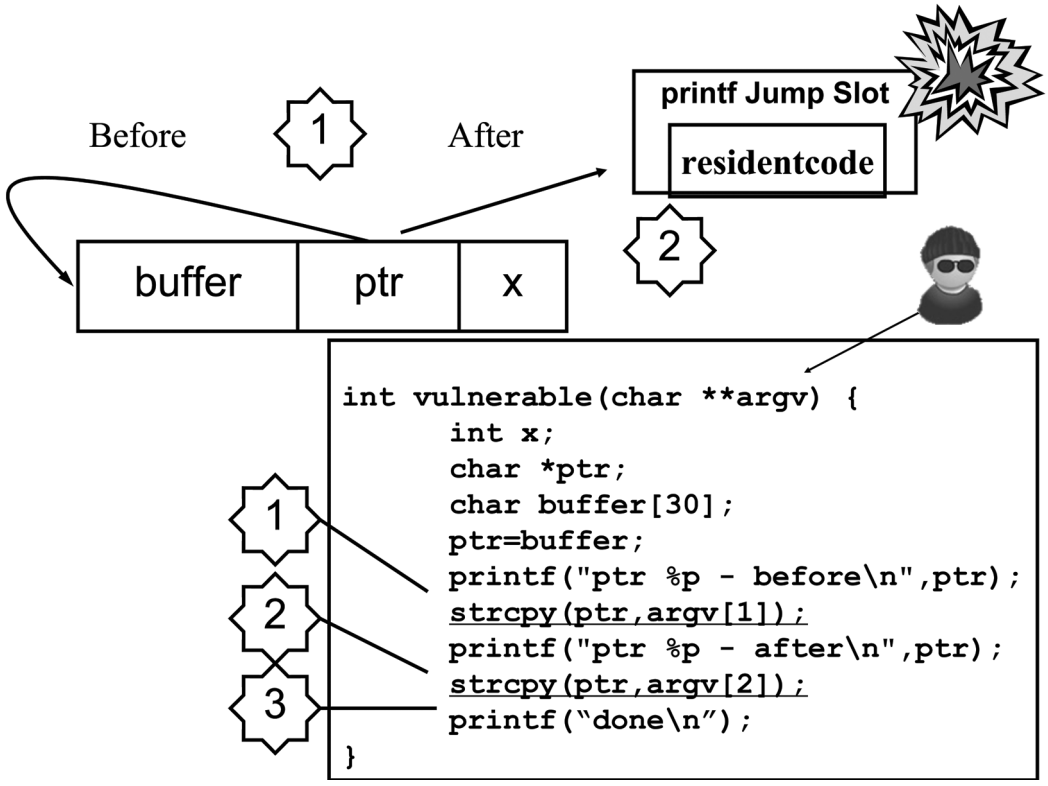

Figure 3 An example of a vulnerable program

Another variation which most schemes cannot protect is an attack on generic pointers. By modifying both the source and destination pointers, copying from one arbitrary memory location to another is possible. Figure 4 illustrates an attack based on attacking pointers. In this figure, the first ***strcpy*** will allow attackers to overflow buffer ***b*** so the ***src*** and ***des*** pointers are replaced with two pointers of the attacker's choice: in this case, valid control data and target address respectively. The second ***strcpy*** will then copy from the ***src*** location to the ***des*** location. Using this approach, it is possible to modify any memory entry with a known local entry while avoiding most protection mechanisms. For example, an encoded function pointer can be used to attack another function pointer (e.g. jump slot).

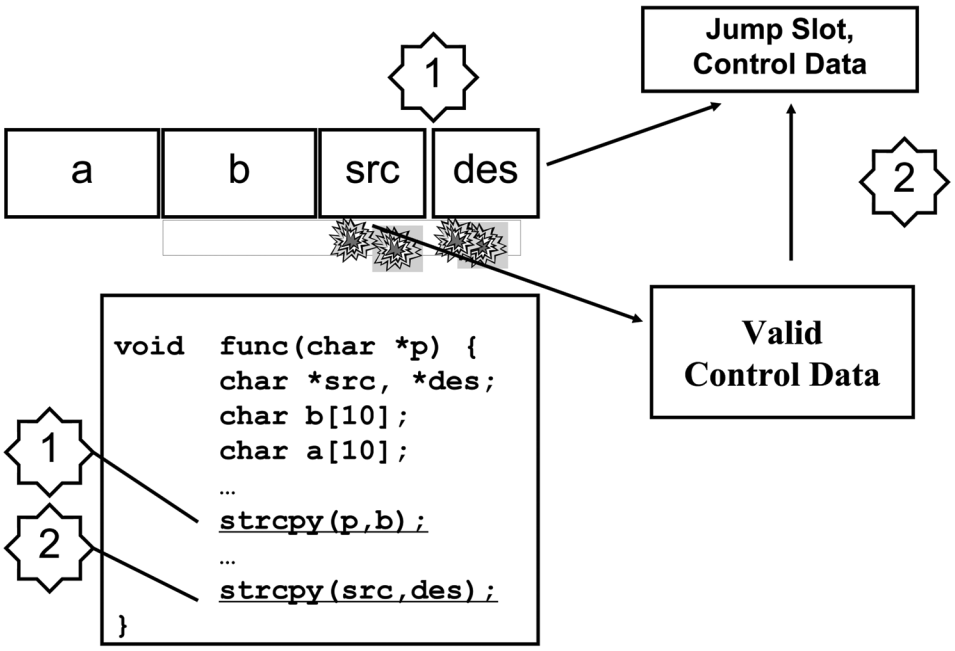

Figure 4 Buffer-Overflow Attacks on Pointers

A related attack is worth noting here: printf vulnerabilities [63]. Malformed formatting instructions can allow arbitrary memory to be overwritten. It is not a buffer-overflow attack, but with the ability to overwrite arbitrary memory the attack then proceeds like many buffer-overflow attacks by attacking control data. Some buffer-overflow schemes can prevent some of those attacks. However, any variable can also be attacked, and no buffer-overflow scheme protects arbitrary data. Fortunately, simple static analysis of source code can identify printf vulnerabilities. As we will see below, static analysis is not sufficient for buffer overflows.

Recently "integer overflows" (known more generically as "integer arithmetic" attacks) have emerged as a variation [5]. In these attacks guard data which protects buffers is attacked. After defeating the guard data some buffer-overflow attack is used. Therefore,

these attacks can be considered as a variation of a buffer-overflow attack which the more robust schemes can protect against.

An attack on local variables is exemplified by the classic password attack from 1987 paper [80]. Basically, a variable is being overflowed allowing an arbitrary password to be validated resulting in root or administrator access. No control data is involved so no existing buffer-overflow protection schemes protect against this type of attack.

1.5 Summary

Through the chapter, we have learned the fundamentals of buffer-overflow attacks that can occur in any user accessible area. In the big picture, a buffer-overflow attack is the effect of modifying critical data by overflowing a buffer. The critical data are control data and generic variables. Most attacks target control data. When control data is the target, attackers are then able to redirect the control flow to execute the unexpected code.

Chapter 2 Reviews

This chapter is the review of current approaches against buffer-overflow attacks. To elucidate the wide variety of approaches, the methods are categorized to form a taxonomy (in Figure 7). For each class in the taxonomy, we briefly discuss the mechanisms and the (potential) problems.

2.1 Introduction

Although they date back to the infamous MORRIS worm of 1988 [62], buffer-overflow attacks remain the most common type of attacks. Though skilled programmers should write code without buffer overflows, no program is guaranteed free from bugs so it cannot be considered completely secure against buffer-overflow attacks. The persistence of buffer-overflow vulnerabilities speaks to the difficulty of eliminating them. In addition, as buffer overflow vulnerabilities are eliminated in operating systems, they are being found and exploited in applications. When applications are run with root or administrator privileges, the impact of a buffer overflow is equally devastating.

In an effort to avoid relying on individual programming skill, a number of researchers have proposed a variety of methods to protect systems from buffer-overflow attacks. Most of them are not able to provide complete protection. For example, some only prevent the original stack-smashing attack, but can be circumvented by more recent attacks.

2.2 Protection schemes

In Chapter 1, we have established a basis for understanding buffer-overflow attacks so we can now examine current approaches to protect against them. We partition the approaches into three broad categories: static analysis, dynamic solutions, and isolation. Static analysis tries to fix functions that are vulnerable to buffer-overflow attacks. Dynamic approaches monitor or protect data that is either a target or the source of buffer-overflow attacks. Isolation seeks to limit the damage of attacks.

2.2.1 Static Analysis

Given a set of known vulnerabilities, we can simply avoid buffer-overflow attacks by fixing the vulnerable functions. Since this approach works on source code it is called "Static Analysis."

The main idea of "Static analysis" is to find and solve the problem before deploying the program. To do so, we first analyze the source code or disassembly of the program by looking for code with a predefined signature. For example, it is well known that the "strcpy" function in C can be vulnerable to buffer-overflow attacks. Knowing this fact, we can create a signature to search for any use of the "strcpy" function, warn the programmer of the vulnerability and suggest solutions (e.g. using the bound-checking version "strncpy"). A generic static analysis algorithm is presented in Figure 5.

```
// Static Analysis

1. Open pre-defined signatures
2. Search target source code for signature code
3. if(found(signature))
         replace with less vulnerable code, inform programmer
```

Figure 5 Algorithm of Static Analysis

2.2.2 Dynamic Solutions

Knowing which data are critical to attacks, we can prevent attacks by validating the integrity of that data. As mentioned above, the data of interest are control data such as (but not limited to) return addresses. We name these "Dynamic Solutions" because data are dynamically managed and verified in the run-time environment. The generic algorithm is shown in Figure 6.

```
// Dynamic Buffer-Overflow Protection
// During run-time execution

Tag control data
If (accessing_control_data && legal_tag)
  Continue
Else
  raise exception
```

Figure 6 Algorithm of Dynamic Solutions

In order to validate (and identify) the integrity of data, it is necessary to have a tag (metadata) associated with the data. This "metadata" will allow us to check the integrity of data by validating the tag. To differentiate the approaches we will compare four components of dynamic solutions:

- Underlying assumptions of the approach
- Creation of metadata
- Validation of metadata
- Handling of invalid data

We provide several examples to illustrate the components.

Example 1,

In this first example, buffer overflows are foiled by preventing any input (including buffers) from being used as control data. Metadata is used to tag data as input. When any

data is used by control instructions (e.g. jump, call, or return), the instructions must verify that it is not input (by verifying the metadata). If it is input, an exception is raised.

From this example, we can derive the main concepts as:

Assumption: Input is untrustworthy for use as control data.

Creation: All functions that send or receive data from another process must set the metadata tag of data as untrustworthy input.

Validation: Control instructions must validate that the associated control data is not tagged.

Handling: Exception is raised.

Example 2,

In this second example, return addresses can only be created by call instructions. When a return address is created, it must be tagged by the call instruction. Other instructions must clear the metadata tag. A return instruction must verify that the address has a valid tag (not modified by other instruction) before executing the return. If the return address is invalid, an exception is raised.

From this example, we can derive the main concepts as:

Assumption: The validity of return addresses must be maintained.

Creation: Call instruction must set the tag of the return address.

Validation: Return instruction must validate that the return address is tagged.

Handling: Exception is raised.

The two examples are different in that their solutions are based on different assumptions. However, they share the same idea that a tag (or metadata) is necessarily attached to critical data. They differ in that the first example tags input in general while the second example specifically tags return addresses.

Dynamic methods can also be differentiated in how the metadata is stored. In general, there are two types of metadata: hardware supported and software managed metadata. Hardware solutions require modification of either the hardware organization, the instruction set architecture or both. For software managed metadata, the metadata is (mostly) managed by software as normal data. In general, prologue and epilogue are inserted to the program for the creation, validation and handling of metadata.

Using these components we can broadly classify dynamic solutions into four groups:

- Address Protection
- Input Protection
- Bounds Checking
- Obfuscation

The address protection schemes such as Example 2 share the assumption that addresses (e.g. return address) are critical data and must be tagged. In these schemes the metadata is created by functions that create the address (e.g. call instruction), and verified by the many instructions that use the address (e.g. return instruction). The schemes within this group are differentiated by the types of metadata they use.

The input protection schemes such as Example 1 assume that external data are untrustworthy and should not be used as internal control data. The underlining concept is that "All input is evil until proven otherwise" [32]. In most cases, metadata are tightly coupled to the data in hardware (e.g. tagged memory). Data from external sources are tagged so it can be recognized, if there is an attempt to use it as control data. The schemes in this group differ in the management of metadata.

Rather than tagging data, bounds checking schemes explicitly bound buffers to prevent overflow. In this case, the metadata is associated with every block of allocated data and is used to bound accesses.

Instead of protecting the data directly, obfuscation schemes reorganize memory to obscure memory making malicious manipulation of memory through buffer overflows more difficult. These schemes assume that attackers rely on a certain snapshot of addresses to overflow the critical data. If the snapshot is random or difficult to guess, an attack is more difficult

2.2.3 Isolation

Isolation schemes limit the damage from attacks rather than preventing attacks. As a result their protection is not limited to buffer-overflow attacks. This category includes sandboxing and confinement schemes.

Isolation schemes isolate the attacker either to eliminate an attack vector or to contain damage after a successful attack. Preventing the execution of code in stack memory

isolates the stack from the attacker. Alternatively, limiting the memory of a process can isolate a compromised process. NX nonexecutable memory is an example of the former while sandboxing is an example of the latter.

As with dynamic solutions, isolation can be implemented purely in software or with support from hardware.

2.2.4 Summary of Protection schemes

In this section, we introduced a classification of buffer overflow protection into three broad categories: fixing the function, protecting the data, or limiting the effects. In this paper, we refer to these schemes as Static Analysis, Dynamic Solutions and Isolation respectively. For illustration, we draw a taxonomy in Figure 7, and will continue to review various proposed solutions in the next section.

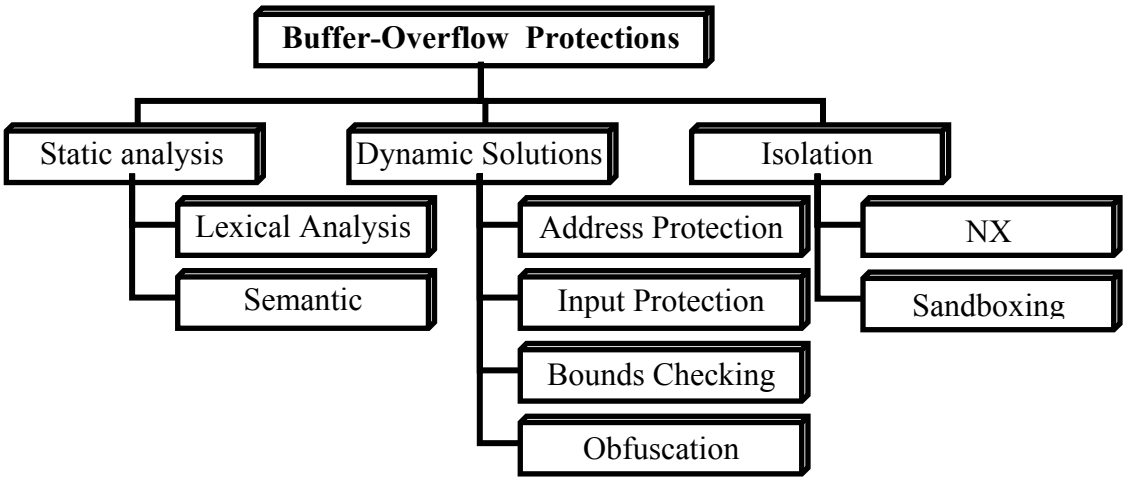

Figure 7 Taxonomy of solutions against buffer-overflow attacks

2.3 Static Analysis

A number of tools examine source code to detect for possible buffer overflow vulnerabilities. The method can be a simple string matching algorithm, a lexical analyzer, or a scanner (parser). Static Analysis allows a programmer to prevent the problem before deploying the program. However, the static analysis has no runtime information. As a result, it might not be able to evaluate all possible problems and may generate false alarms. No matter how good the static analysis tool is, a programmer, eventually, has to make a final decision in correcting the program logic.

2.3.1 Lexical Analysis

ITS4: Researchers at Reliable Software Technologies developed a static analysis tool for detecting security vulnerabilities in C and C++ and named it *It's the Software Stupid!* (ITS4 [**74**]). The tool uses lexical analysis and matches tokens with known vulnerable functions in a pre-defined database. It is useful for highlighting potential security problems as code is written. The report generated from ITS4 includes a brief description of the problem, a brief description of how to solve the problem, and the level of severity.

FlawFinder and RATS: FlawFinder [**25**] and RATS [**61**] are similar tools that scan source code using lexical analysis. When the developers of each team noticed that they were developing similar tools, they decided to combine the two tools into one in the future. Basically, FlawFinder uses the same method as that of ITS4. However, the report generated from FlawFinder is associated with the values of the parameters of the function. For example, variable strings are considered more risky than constant strings. In addition to C, RATS also supports Perl, PHP and Python source code.

STOBO: This Systematic Testing Of Buffer Overflow (STOBO [30]) is an interesting static analysis tool for C programs. It uses profiling to generate its report. This tool will insert special functions into the original program to keep track of each variable and memory allocation. Running the modified program will provide dynamic analysis from the runtime environment.

LibSafe: LibSafe [2] is the implementation of static analysis in the run time environment. It is a safe implementation of a library that is forced to be loaded before the standard C library. Since the library is accessed in the order of loading, Libsafe is able to intercept the known vulnerable functions (e.g. strcpy, strcat).

Dynamic Tainted Analysis: Dynamic Tainted Analysis [50] is a monitor tool modified for examining the arguments and results of each system call, and determines whether data should be marked as input. The current implementation is in Valgrind, a Linux tool for detecting memory management problems. Memory is emulated for tracking the information flow (using a tainted bit associated to each byte). System calls are monitored and recording. According to the policy, some system call may generate the tainted data. All arithmetic instructions are captured to propagate the tainted. This tool is useful in that it uses dynamic information for fixing the program.

Potential Problems:

- **False alarms**
- **Does not handle buffer overflow in user-defined functions and macros**

2.3.2 Semantic Analysis

Splint: Splint [24] is a tool for statically checking C programs for security vulnerabilities and coding mistakes. The name and some of its functionality originates from a popular static analysis tool for C called Lint released in the seventies. Splint uses a parser to perform semantic analysis. This means that the tool has a better chance of differentiating between correct and incorrect use of functions than those based on lexical analysis.

Boon: BOON [75] is another tool for statically checking C programs for security vulnerabilities. Boon treats the C String as an abstract data type, and models buffers with two integer ranges. These two ideas provide a framework for the analysis.

Potential Problems:

- **False alarms**
- **Does not handle buffer overflow in user-defined functions and macros**

2.4 Dynamic Solutions

Several dynamic solutions have been introduced. Each method varies in assumption, choices of metadata and management scheme, or handling routines. We will review them in the following order: Address Protection, Input Protection, Bounds Checking, and Obfuscation. For each method, we will pinpoint the choice of metadata and the management of the metadata.

2.4.1 Address Protection

There are several similar methods that share the assumption of protecting the address. However, they use different types of metadata. These variations include: a canary value,

tagged memory, and hardware stack. Though some solutions in this class have become obsolete as attacks have matured, they are a good initial step in preventing buffer-overflow attacks.

2.4.1.1 Canary Words

Assuming that corrupting an address will also corrupt the adjacent data, the validity of address can be verified by validating special adjacent metadata (the canary word)[1]. The general mechanism can be described as:

1. Place a canary word next to the address (e.g. return address) when the address is created (e.g. by a call instruction).
2. Verify the canary word before using the address (e.g. before a return instruction uses a return address).

StackGuard [15, 17, 18, 31]

Assumption: Return addresses must not be modified after creation

Metadata: Canary word adjacent to the return address

Creation: Compiler injects prologue to the header of every function for placing a canary word next to the return address.

Validation: Compiler injects epilogue at the end of every function for validating the canary word.

Handling: Compiler injects an error handling routine.

---

[1] A canary word is similar to a canary in a mine: if the canary dies, that indicates a problem.

Potential Problems:

- Insufficient assumption: buffer-overflow attacks have changed in response to this type of protection to now attack other addresses.
- Insufficient protection for the canary word itself and the return address (See [7] for more details.)

ProPolice [23]

ProPolice supports two mechanisms. The first one is return address protection similar to StackGuard. In addition to the return address protection, ProPolice also assumes that buffer overflow occurs only in one direction, and reorders the declaration statement to protect against function pointer attacks. Since an overflow only goes in one direction, declaration reordering can prevent the function pointer from being overflowed. Figure 8 shows the sample of the reordering process. However, this can only protect some variables from being overflowed, but not other variables that are still in the overflow direction.

Original Code	Reorder Code

Int bar() {
void (* funct2)();
char buff[80];
…..
}
	int bar() {
	char buff[80];
	void (* funct2)();
	…..
	}

Figure 8 Sample of code reordered by IBM ProPolice

2.4.1.2 Address Encode

Knowing that encryption can help preserving the integrity of data, the same concept is applied to protect the integrity of addresses (e.g. function pointers or return addresses). In this type of protection, the metadata is the key used to encode the data. The general mechanism can be described as:

1. Encode an address with a pre-defined key before storing it to the memory.

2. Decode the address on dereferencing (loading back to the processor).

PointGuard [16] and Hardware Supported PointGuard [64, 71]

Assumption: Pointer must not be modified after creation.

Metadata: Per-process random key for encrypting pointers (generated by software or hardware supported hash table)

Creation: Compiler injects code for encoding pointer (with or without hardware supported instruction)

Validation: Compiler injects code for decoding pointer on dereferencing. Pointer to outside process boundaries is flagged as malicious.

Handling: Exception is raised

Potential Problems:

- (Possibly) insufficient randomness of encryption key
- Issues with arrays and compile-time assignment of pointers. Since a pointer is encoded with a per-process key that is generated when the program is loaded, it is not possible to assign a value to the pointer before the key is determined. In addition, languages (such as C, C++) may handle a string as a pointer of characters (array). It is not clear whether the string pointer should be encoded.
- Shared libraries and interprocess communication. While each process is aware of its key, the shared library is shared. Thus, passing a pointer to the shared library requires sharing a key between the main program and the shared library. Similarly, passing a pointer to another process also requires special handling for the key.

2.4.1.3 Copy of address

A simple method for preserving the integrity of an address is to preserve another copy of the address. The mechanism can be summarized as follow:

1. Create a safe copy of the address (e.g. return address) when it is created.
2. Verify the (return) address against the safe copy before using it (e.g. on return).

StackGhost [27]

Assumption: Return addresses must not be modified after creation.

Metadata: A copy of return addresses stored in the register window of Sun SPARC processor.

Creation: Hardware supported instructions automatically create a copy of the return address in the register.

Validation: Operating system validates return address with the copy in the register window

Handling: Operating system (OpenBSD) patch of handling routine to handle the new exception.

Potential Problems:

- Insufficient assumption: overflow attacks have changed in response to this type of protection to now attack other addresses.
- Dynamic update of return addresses (e.g. non-LIFO control flow).
- Overflow of register window.
- Architecture specific.

RAS [44, 78, 79]

Assumption: Return address must not be modified after creation.

Metadata: A copy of return addresses stored in the Return Address Stack (RAS)—a hardware accelerator technique in some processors

Creation: Hardware supported instructions automatically create an entry of a return address in Return Address Stack (RAS)

Validation: Compiler injects prologue for validating the return address against the copy before the return instruction.

Handling: Exception is raised.

Potential Problems:

- Insufficient assumption: overflow attacks have changed in response to this type of protection to now attack other addresses.
- Dynamic update of return addresses (e.g. non-LIFO control flow) .
- Overflow of return address stack.
- Architecture specific.

Split Stack [78]

Assumption: Return address must not be modified after creation.

Metadata: A special memory area for storing the return address (control stack) separately from the regular data stack. Note that this can be implemented as either software or hardware.

Creation: Software: compiler injects epilogue to call instruction for storing the return address in a control stack (software). Hardware: the semantics of the call instruction are modified to store the address.

Validation: Software: compiler injects prologue to return instruction for restoring the return address from the control stack,, or Hardware: the semantics of the return instruction are modified.

Handling: Compiler injects handling routine (software) or exception is raised (hardware).

Potential Problems:

- Insufficient assumption: overflow attacks have changed in response to this type of protection to now attack other addresses.
- Dynamic update of return addresses (e.g. non-LIFO control flow).
- Breaking of system programs that assume the layout of the stack frame.

SmashGuard [53]

Assumption: Return address must not be modified after creation.

Metadata: A hardware stack for storing a copy of return address.

Creation: The semantics of the call instruction are modified to push a copy of return address onto the hardware stack.

Validation: The semantics of the return instruction are modified to pop and compare the two return addresses.

Handling: Exception is raised.

Potential Problems:

- Insufficient assumption: overflow attacks have changed in response to this type of protection to now attack other addresses.
- Dynamic update of return addresses (e.g. non-LIFO control flow). In the paper, the authors extend functionality of stack to cover some aspects of this issue.
- Breaking of system programs that assume the layout of the stack frame.

RAD Compiler [12], RAD Binary Rewrite [59] and DISE [14]

RAD Compiler, RAD Binary Rewrite and DISE share the same concept of storing a redundant copy of return addresses for preserving the integrity of the return address. However, the mechanisms for injecting the protection code are slightly different. In RAD Compiler [**12**], the compiler is responsible for embedding the protection code into the program. In RAD Binary Rewrite [**59**], the loader is responsible for injecting the protection code using binary rewrite. For DISE [**14**], the hardware-assisted implementation, dynamic instruction stream editing, will expand the instructions on the fly. Due to the similarity, we will only elaborate the details of RAD Binary Rewrite [59].

RAD Binary Rewrite

Assumption: Return address must not be modified after creation.

Metadata: A special memory area for storing a redundant copy of return addresses (Return Address Repository, RAR) which is protected by the *mprotect()* system call to mark the area as read-only [12].

Creation: Loader performs binary rewrite to inject prologue to the header of every function for storing a copy of the return address in RAR.

Validation: Loader performs binary rewrite to inject epilogue to the end of every function for validating the return address with a redundant copy.

Handling: Loader performs binary rewrite to inject a new exception handing routine.

Potential Problems:

- Insufficient assumption: overflow attacks have changed in response to this type of protection to now attack other addresses.
- Dynamic update of return addresses (e.g. non-LIFO control flow).
- Performance issue of the *mprotect()* system call.

StackShield [73]

There are three mechanisms proposed in StackShield: Global Ret Stack, Ret Range Check, and Function Pointer Protection. The first two mechanisms are similar concepts, but differ slightly in implementation. We will present the first two together and the pointer protection separately.

Global Ret Stack and Ret Range Check

Assumption: Return address must not be modified after creation.

Metadata: A special memory area for storing a redundant copy of return addresses (Global Array).

Creation: Compiler injects prologue into the header of every function for copying the return address to the Global Array.

Validation: Compiler injects epilogue into the end of every function for copying the return address from the Global Array back to the original location without checking (Global Ret Stack) or for validating the return address with a redundant copy (Ret Range Check).

Handling: Compiler injected exception handling routine.

Function Pointer Protection

Assumption: Function Pointer must only point to a location in the text (code) segment.

Metadata: The range of the code segment.

Creation: Loader creates the code segment.

Validation: Compiler injects prologue to each function pointer for validating that its value must be in the valid range.

Handling: Compiler injected exception handling routine.

Potential Problems:

- Insufficient assumption: overflow attacks have changed in response to this type of protection to now attack other addresses.
- Insufficient protection for the metadata itself and the return address (See [7] for more details.)
- Dynamic update of return addresses (e.g. non-LIFO control flow).

LibVerify [2]

LibVerify is a type of dynamic solutions that is implemented via binary rewrite. Figure 9 shows the mechanism.

Assumption: Return address must not be modified after creation.

Metadata: A special memory area for storing a redundant copy of return addresses.

Creation: Loader performs binary rewrite to create a copy of secure code by injecting a prologue for storing return addresses in the special memory area.

Validation: Loader performs binary rewrite to inject epilogue for verifying the return address.

Handling: Loader performs binary rewire to inject handling routine.

Potential Problems:

- Insufficient assumption: overflow attacks have changed in response to this type of protection to now attack other addresses.
- Dynamic update of return addresses (e.g. non-LIFO control flow).

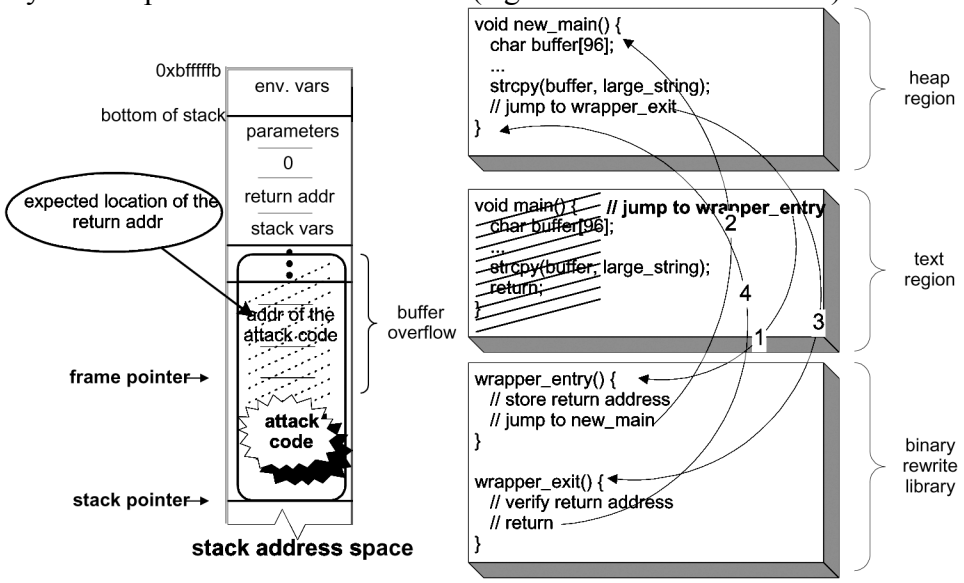

Figure 9 LibVerify. (From [2])

SCACHE [35].

Assumption: Return address must not be modified after creation.

Metadata: Cache memory.

Creation: Hardware creates replica cache line for each return address store.

Validation: Hardware compares the return address loaded from stack with the replica in the cache line.

Handling: Exception is raised.

Potential Problems:

- Insufficient assumption: overflow attacks have changed in response to this type of protection to now attack other addresses.
- Complexity of cache replacement algorithm.
- Dynamic update of return addresses (e.g. non-LIFO control flow).

2.4.1.4 Tags

Tagged Architecture [28]

Tagged memories go back at least thirty years with many proposed architectures and a few which made it to market. For our purposes, tagged-memory architectures provide three main functions. The first supports language and the Symbolics [47] Lisp computer, which achieved a relatively short-lived commercial success, is a classic example. In that computer, the tags were used to efficiently keep track of dynamic types. The second type uses tags to support capabilities. Capabilities reach back forty years, and their popularity has increased recently with increased interest in security exemplified by capability-based operating systems such as EROS [65]. The tags support the capabilities which control access. The third is best exemplified by the IBM System/38 [21] database computer (the

predecessor of the IBM AS/400) which used a tag to protect pointers. Of those, it is the IBM System/38 that is relevant to this survey.

Here, we will present the concepts embedded in the IBM System/38 as a representative of protection against buffer-overflow attacks provided by tagged architecture in general.

IBM System/38 [21]

Assumption: Return address and function pointer must not be modified after creation.

Metadata: A bit associated with each word (or byte) of memory.

Creation: Call instruction and a special instruction set the tag of a return address and a function pointer respectively.

Validation: Return instruction and branch instruction validate the tag bit of a return address and a function pointer.

Handling: Exception is raised.

Potential Problems:

- Compatibility
- Performance

2.4.2 Input Protection

Some methods assume that input data should be treated differently from local data. The idea is that input should not be used as control data. We will review three methods, Minos [19, 20], Tainted Pointer [4], and Dynamic Flow Tracking [67] that share the same assumption, but different implementations. Minos views data across segments as input. Tainted Pointer considers data passed from the operating system as input. Dynamic Flow

Tracking relies on operating systems for marking input. Note that Secure Bit treats data passing between processes through the kernel as input.

Minos [19, 20]

Assumption: Input (low-integrity data) must not be used as control data.

Metadata: A bit associated to each word (or byte) of memory.

Creation: A reserved bit in a descriptor is used to indicate whether data is moving across segments (rings) and marks the data as low integrity (set the tag) or carries the tag over. All 8-bit and 16-bit load operations and floating point, MMX, BCD, and similar extensions mark the data as low integrity. In addition, data that is accessed after the ctime established by the kernel and some system calls (e.g. exec, pread) will force the data to be low integrity.

Validation: Jump, Call or Return instructions validate that control data are low integrity.

Handling: Exception is raised.

Potential Problems:

- Possible issues with multi-threading program (e.g. Sun Java JVM). To solve the issue, some programs may have to be executed in a "compatibility mode" which has no protection.

Tainted Pointer [10]

Assumption: Input must not be used as control data.

Metadata: A bit associated to each byte of memory (tainted bit).

Creation: The current implementation is in a hardware simulator (SimpleScalar) which modifies SimpleScalar I/O functions (SYS_READ and SYS_RECV) to mark every byte

coming into a process as tainted. ALU instructions are modified to propagate the taintedness. (Note that compare instructions will untaint every byte in the operands of the instruction.)

Validation: Jump instructions validate the pointer.

Handling: Exception is raised

Potential Problems:

- Since the current implementation is in a hardware simulator some operating system issues are not yet resolved. For example, there are potential issues with multi-threading program (e.g. Sun Java JVM).
- Issues with compare instruction. Since the compare instruction can be used to untaint data that opens a vector for attack.

Dynamic Flow Tracking [67]

Assumption: Input must not be used as control data.

Metadata: A bit associated to each byte of memory (tainted bit).

Creation: The current implementation is in a hardware simulator (SimpleScalar) which modifies SimpleScalar I/O functions (SYS_READ and SYS_RECV) to mark every byte coming into a process as tainted. ALU instructions are modified to propagate the taintedness.

Validation: Jump instructions validate the pointer.

Handling: Exception is raised

Potential Problems:

- Since the current implementation is in a hardware simulator some operating system issues are not yet resolved. For example, there are potential issues with multi-threading program (e.g. Sun Java JVM).

2.4.3 Bounds Checking

In Bounds Checking, the methods assume that all accesses to data must be done within the boundary of that variable. We will review two main approaches: software and hardware approaches. For the software, the implementation can be a modification to the compiler of current languages or virtual machine solutions (e.g. type-safe programming languages).

Array Bounds Checking [37]

Researchers from Imperial College, London [**37**] created a backward compatible bounds checking in C. The method does not change the representation of a pointer. Thus, it is compatible with the standard C library. For every pointer, a base pointer is defined. A pointer value is valid for only one memory region. Checking whether the reference is in the same region as the one referred by a base pointer enforces bounds checking. This method is useful to prevent buffer overflow attacks. However, with the overhead of a symbol table used to keep track of each pointer, it experiences more than a 30 times slowdown in a pointer-intensive program. As a result, this tool is ideal for debugging, but may not be suitable for most applications. Similar tools include ***Rational PurifyPlus*** [**60**], and ***BoundsChecker*** [6]. Conceptually, we can view this approach as a software implementation of segmentation which uses an entry in the symbol table as a segment descriptor.

Assumption: Accessing a memory area must be in a boundary.

Metadata: A descriptor attached to each pointer and memory allocation.

Creation: Compiler generates descriptor for each pointer and allocation.

Validation: Compiler injects prologue for checking pointer and array boundary.

Handling: Handling routine

Potential Problems:

- Performance issue (30X).
- Compatibility with legacy code (with respect to the presentation of pointer and array).
- Nested structure of memory.

Segmentation: Limited hardware protection has existed in various processors for many years. Among them, segmentation is a novel one. Segmentation is primarily used as a mechanism to support the relocation of memory. In the early implementation of segmentation, a base register is required for each memory access. IA-32 and I432 [52] also adopt the idea and associate segmentation with base address, boundary check, and rings. Explicitly declaring and referring every buffer with base and boundary, segmentation can protect against buffer-overflow attacks. A drawback of segmentation is the extra storage for storing segment descriptors. In IA-32, every memory access (in protected mode) requires a base and limit. However, most operating systems bypass segmentation by setting one large segment for whole memory in order to maintain portability and gain better performance. I432 was a CISC architecture that is designed with security awareness. Based on the paradigm of the ADA programming language, it checks every data boundary and forces every function call to create a new domain (segment). Since I432 instructions are bit encoded, ranging from six to 321 bits,

computation took 10 to 20 times as long as the contemporary VAX 11/780 [13]. Consequently, I432 was a commercial failure.

A similar concept can also apply to a function pointer. For example, one of the 1960s architecture, ICL 2900 series systems [28], had a native hardware 'pointer' type (a.k.a. descriptor) that included in it the size of the object pointed to. The hardware would check that any dereferences were not out of bounds.

Assumption: Accessing a memory area must be within a boundary.

Metadata: Hardware supported descriptor registers.

Creation: Compiler generates descriptor for each memory allocation via supported management from the operating system.

Validation: Hardware supported bound checking though a descriptor.

Handling: Exception is raised.

Potential Problems:

- Performance issue
- Nested structure of memory

Type-safe programming languages and interpreted languages: In the type-safe programming languages and interpreted languages (e.g. Java, .Net), the metadata and bounds checking process is embedded into the virtual machine. Correct bounds checking can prevent buffer overflows. However, the virtual machine itself is written in C/C++ so somehow the virtual machine has to interact with underlying components which use type-

unsafe languages. Type-safe languages decrease the probability of being attacked, but can still be exploited by buffer overflows. For example, there have been buffer-overflow attacks in Java [22, 68], Perl [72], etc. There also exists a type-safe C: CCured [48].

Assumption: Accessing a memory area must be within a boundary.

Metadata: A descriptor attached to each variable.

Creation: Virtual machine generates a descriptor for each memory allocation.

Validation: Virtual machine validates every single memory access.

Handling: Virtual machine handles the exception.

Potential Problems:

- Performance issue
- Complexity of Garbage Collector (if applicable)
- Compatibility with legacy code

2.4.4 Obfuscation

When there is no appropriate solution, confusion and increased difficulty can be used as a protection mechanism. However, it should not be used alone when other methods are applicable [8].

Address Obfuscation

Conceptually, Address Obfuscation [3] reorganizes the memory area of each process to make it difficult for attackers. Facing a changed the memory alignment, malicious users will encounter difficulty in overwriting the expected addresses. A compiler is modified to randomize the base addresses of each memory segment, the distances between each pair

of data items, and permutes the order of variables/routines. A similar mechanism is implemented in the Address Space Layout Randomization (ASLR) schema of the ***PAX*** project [54]. In the big picture, this method cannot protect against skilled attackers who obtain the binary of the program. Since the randomness does not occur during execution, a dissembler tool will unveil the necessary information.

Assumption: Buffer-overflow attacks assume a specific layout of memory snapshot.

Metadata: Offset added to each variable for accessing each data.

Creation: Compiler generates offset value for each variable.

Validation: None

Handling: None

Potential Problems:

- Randomness of the offset value

2.4.5 Isolation

There are two main ideas behind isolation. The first is to limit the execution of code that may result from buffer-overflow attacks. Another idea is to sandbox the whole process from accessing certain system resources based on a predefined policy. Apart from these two main ideas, variations include the isolation of executable code from being installed or modified in the run-time environment.

Non-Executable Memory

Another common technique is non-executable partitions of memory such as pages or segments. Many non-X86 processors such as SPARC support non-executable memory, and AMD has recently added a similar feature named "NX" [41]. Non-executable memory prevents code in the buffer on the stack from being executed, effectively protecting against a class of buffer overflow attacks that results in the execution of code on the stack. The Solar Designer group [66] and INGO [34] also proposed a patch to the Linux kernel to make a ***non-executable stack***. However, the integrity of the return address is not protected—leaving the system vulnerable to attacks using the address of either a resident shell or code in the heap. In certain cases, such as signal handler return on Linux, the system requires an executable stack in order to function properly. Moreover, any LISP-like functional language requires an executable stack in their normal operation (aka. trampoline). As a result, this method only protects against a narrow range of attacks.

SPEF: Alternatively, researchers from Microsoft and the University of California at Los Angeles have developed a Secure Program Execution Framework [40] (SPEF). Instead of protecting the data, the method protects the code. The method aims at making a system difficult to inject malicious code. SPEF is a platform that consists of architectural mechanisms and compilation tools. The installation of a program requires both encryption and transformation. As a result, injecting the malicious code is not simple and requires a special process. This method prevents the injection of malicious code. Obviously, it is still possible to overflow the buffer and modify the return address or the function pointer to point to a known address. Based on 3DES and domain ordering hardware, SPEF should experience performance difficulties and may not be feasible for

general applications. Another implementation that shares the same concept but uses instruction block signature is [46]

Instruction-Set Randomization: Similar to SPEF, Instruction-Set randomization [38] introduces difficulty in injecting the malicious code. The General idea is to randomize the coding of instructions by XORing them with a key. The authors propose a per-process key schema which makes it difficult for a dynamic-linked library. As a result, the method supports only static-linked libraries. The technique can also be applied to scripting languages by adding a random number at the end of each instruction and modifying the virtual machine to validate the number in each instruction. In [38], an example of random Perl is presented. The drawbacks are the lack of support for dynamic-linked libraries, the requirement of special hardware, and the limitation of using polymorphic and self-modifying code. Like SPEF, which only prevents the injection of malicious code, overflowing with a known address is possible.

Sandboxing: Sandboxing is a policy-enforcement mechanism. Since buffer-overflow occurs when information is passed from one domain to another domain, sandboxing a process intuitively cannot prevent such attacks. With appropriate policy rules, it is, however, possible to limit the damage of buffer-overflow attacks. Sandboxing can be done at several levels: kernel level [55], user level [9], or even hardware-supported sandboxing (e.g. Intel LaGrande [36], TCPA [43, 70], TrustZone [1], Microsoft NGSCB [45], ChipLock [39], Bear [43].) Like tagged memory, there exists a very fine-grained approach to memory management (e.g. MMP [77]), but such approaches can be

successful for buffer-overflow protection only if a perfect combination of a security policy and an implementation exists. We believe that it is complementary to other techniques rather than a replacement.

2.5 Analysis

With the knowledge of buffer overflow attacks in hand, we will raise issues critical to protections against buffer-overflow attacks. Those issues are: Common Pitfalls, Performance, Compatibility (Transparency), and Deployment and Cost.

2.5.1 Pitfalls

Buffer-overflow prevention, like many security efforts, has been an "arms-control race" in that attacks have evolved to counter prevention schemes which in turn require increased sophistication in prevention. In hindsight, we can look back and see why earlier efforts failed. Two themes have emerged:

Insufficient assumptions: Some approaches only provide protection against a subset of buffer-overflow attacks. This is due to the maturing of buffer-overflow attacks that have shifted their target from one address to another. For example, the best known type of buffer-overflow attack modifies return-addresses. As soon as developers tried to protect the return address, function-pointer attacks became popular. Another example is trusted code. It can be wrong to assume that trustworthy code is 100% safe from buffer-overflow attacks. Compromising the signed code or the signing mechanism allows execution of malicious 'trusted' code.

Insufficient protection of metadata: Metadata is necessary to assist in protecting critical data. Ideally, metadata must not be controllable by attackers. If attackers can control the metadata, they can successfully create buffer-overflow attacks by modifying both data and metadata. For example, if a key or canary can deterministically be reproduced, attacking an encrypted data or guarded canary is possible. (More examples in [7, 42, 49])

2.5.2 Performance

In the trade off between performance and security, performance has always received priority. This is best exemplified by segmentation (e.g. I-432). An appropriate utilization of segmentation is a perfect tool, if one is willing to explicitly declare each variable with a base and limit (including those of integer and floating-point variables, since buffer overflow can also result from type casting). However, the tremendous overhead of the symbol table (or the segment-descriptor table) for such an approach is unacceptable.

2.5.3 Compatibility (Transparency)

Given a large number of existing programs and libraries, backward compatibility can be a significant requirement. A good product can fail in the marketplace, if it breaks too many things. This paradigm also applies to computers. Here are some compatibility issues critical for solutions against buffer-overflow attacks.

Data representation is a critical problem, if the data representation of the prevention scheme is not interoperable with legacy software libraries. An example is the pointer (and array) representation of PointGuard

Non LIFO control flows include signal handling, trampoline (a type of software optimization where code is dynamically generating and running on top of the stack frame), and far and near call optimization [57]. In most cases, methods that try to create a redundant copy of data of the stack frame will fail to support the dynamic stack management of non-LIFO control flows. Examples of these methods include: Separated Stack, RAS, RAD, etc.

Binary Compatibility is perhaps the most difficult goal in implementing solutions against buffer-overflow attacks. To support binary compatibility, the solution must maintain not only the same data representation, but it also has to maintain the programming model, the communication protocol (e.g. call and return) and the syntax of every instruction. Most solutions presented here fail to achieve this goal. However, some code modifications may be necessary in order to provide protection. For example, a minor modification such as a patch to the operating system may provide protection while preserving binary compatibility for all user code. Secure Bit, which embedded the mechanism in the hardware, provides binary compatibility for user code. Approaches which use binary rewrite by modifying the program loader also can provide binary compatibility.

2.5.4 Deployment and Cost

Deployment may only be critical to those solutions that require hardware support. If hardware modifications are beyond the acceptable ratio of investment to benefit, it may fail commercially. Another issue is cost. In general, cost analysis can be viewed as performance penalty (time), and space. Usually these two conditions have an inverse

relationship with each other. Risk assessment is possibly the major player here. However, such a study is beyond the scope of this paper.

2.6 Conclusions

In this chapter, we have learned that collectively protection against buffer-overflow attacks on control data is based on an ability to detect and validate control data. Intuitively from this observation, some metadata is necessary to distinguish between input and local data.

Among the proposed solutions, there are only three themes: preventing **functions** from overflowing data, protecting target data or tagging **data** that may cause buffer overflow, and limiting the **effects** caused by buffer-overflow attacks. These themes are embedded in Static Analysis, Dynamic Solutions, and Isolation respectively.

In the big picture, every solution which protects against buffer-overflow attacks comes at some cost. Also, as attacks have matured some solutions have proven insufficient for providing full protection against all classes of attacks. We conclude that no solution is perfect in every aspect.

Chapter 3 Buffer-Overflow Protection: The Theory

This chapter theoretically pursues a secure system with respect to buffer-overflow attacks. We begin by defining buffer overflows in general, and buffer-overflow attacks on control data attacks in particular. Later in this chapter, we establish a sufficient condition for preventing buffer-overflow attacks and prove that it will create a secure system with respect to buffer-overflow attacks.

It is worth clarifying that materials in this Chapter specifically focus on buffer-overflow attacks on control data. However, there exists an attack on variables (non-control data) which is not explicitly covered in this material.

3.1 Buffer Overflow

A definition of buffer overflow is presented in Definition 1 (from the Webopedia Computer Dictionary [76]).

Definition 1:

The condition wherein 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.

Since buffers can only hold a specific amount of data, when that capacity has been reached the data has to flow somewhere else, typically into another buffer, which can corrupt data that is already contained in that buffer.

Exploiting buffer overflow can lead to a serious system security breach (buffer-overflow attacks) when necessary conditions are met. The seriousness of buffer-overflow attacks ranges from writing into another variable, another processes memory (segmentation fault), or redirecting the program flow to execute malicious or unexpected code. Based on the definition of buffer overflow, Definition 2 defines the buffer-overflow attacks.

Definition 2:

A **buffer-overflow attack on control data** is an attack that (possibly implicitly) uses memory-manipulating operations to overflow a buffer which results in the modification of an address to point to malicious or unexpected code.

In general, a buffer-overflow attack is an attack on any data (including variables and addresses). To avoid confusion, the term "buffer-overflow attacks" is used to refer to attacks on control data.

Observation: An analysis of buffer-overflow attacks indicates that a buffer of a process is always overflowed with a buffer passed from another domain (machine, process)—hence its malicious nature.

Initially, the attacked address was a return address, but later other control data (e.g. function pointers, jump table) were attacked. In either case, the eventual access of that address (e.g. by a return or function call or jump) will redirect the program control flow

to execute the malicious or unexpected code. If the address was modified by something other than a buffer overflow, it is a race condition, a Trojan horse, or other type of attacks.

3.2 Prevention

This section discusses the necessary conditions for preventing buffer-overflow attacks on control data.

Postulate 1:

In buffer-overflow attacks on control data, the generic buffer/memory-manipulating operations are used by the vulnerable routine to overflow the address (e.g. a return address or a function pointer).

From Definition 2, we observe that preserving the integrity of the address is a sufficient condition to prevent this class of buffer-overflow attacks. To clarify, Definition 3 shows the meaning of the integrity of an address in this context.

Definition 3:

Maintaining the integrity of an address means that the address has not been modified by overflowing with a buffer passed from another domain.

Consider the implication of Definition 3 in light of our "Observation" about Definition 2 which noted the importance of attacks working across domains (machines, processes): *in order to preserve the integrity of the address (e.g. a return address or a function pointer),*

an address cannot be created from data passed across domains (e.g. machines, processes) via buffer overflow.

To maintain its integrity, the address created locally can be signed when it is created and is validated by associated instructions (e.g. return, call, and jump instructions) before they are completely executed. Implicitly, a signature represents some metadata associated with the address. Necessarily, the signature must not be passed across domains; and the integrity of it must be preserved. If the signature could be passed across domains or maliciously modified, a valid address could be used for attacking a system. If we assume that a signature only exists locally, the last condition is enforced when a buffer is passed across a network/hardware device where the signature cannot be passed. Nonetheless, several approaches failed to provide sufficient protection for metadata (signature) and resulted in a new vector of attack where both address and signature are modified.

If local data and data passed from another domain can be differentiated, we can detect buffer-overflow attacks on control data. Thus we may reverse the signature by signing data that passed across domains and leave the local data unsigned. This scheme provides better backward-compatibility since no modification is required for legacy processes.

With these definitions, Theorem 1, and its corollary are introduced. The corollary is the key to the entire framework presented in this paper since it defines a sufficient condition for buffer-overflow attacks.

Theorem 1:

Modifying an address by replacing ("overflowing") it using a buffer passed from another domain is a necessary condition for buffer-overflow attack on control data.

Restatement: If there is to be a buffer-overflow attack on control data, an address must be modified using a buffer passed from another domain.

Proof:

Theorem 1 follows directly from Definition 1, and Definition 2.

QED

Corollary 1.1:

Preserving the integrity of an address is a sufficient condition for preventing a buffer-overflow attack.

Restatement: If the integrity of an address is preserved, that is a sufficient condition for preventing a buffer-overflow attack.

Proof:

From Theorem 1, "If there is to be a buffer-overflow attack, an address must be modified by manipulating a buffer from another domain." The contrapositive of that statement is "If an address cannot be modified (or such modification can be detected), then a buffer-

overflow attack is not possible." We know that the contrapositive of a true statement is true.

QED

Intuitively, from Definition 2, the attack is the ability to redirect the program flow to execute malicious or unexpected code. To achieve this goal, the address must be modified. If the address cannot be modified, the buffer-overflow attack fails. If modification of the address can be recognized, the buffer-overflow attack can be recognized and stopped. On the other hand, if the address can be validated, execution can proceed safely.

3.3 Summary

The buffer-overflow attack is a problem that was indirectly addressed in several ways. Among several variations, buffer-overflow attacks on control data required overflowing addresses (return addresses and function pointers) with a buffer passed from another domain (machine, and process). From the chapter, we conclude that ***"a necessary condition for preventing buffer-overflow attacks is the preservation of the integrity of addresses across domains***".

Chapter 4 Fundamentals of Secure Bit

Fundamentally, a bit (semantic meaning or metadata) is augmented to every memory byte (or word) to protect the integrity of addresses. This semantic meaning is used to distinguish local data and data from another domain (input).

In this chapter, a general concept of Secure Bit is given. Later, we prove that the mechanism creates sufficient conditions for a secure system with respected to buffer-overflow attacks on control data.

4.1 General Mechanisms

A Secure Bit is added to every memory location (and register). This bit is handled by the memory manipulating instructions as part of a regular memory word (moved along with the associated word). Except for passing words in buffers between processes, such operations have to mark the Secure Bit at the destination (either a register, or a memory location). Words in buffers passed between processes get their Secure Bit set. Call, return, and jump instructions check the Secure Bit; and if the Secure Bit is set, the processor issues an interrupt or fault signal. Figure 10 shows a memory snapshot with Secure Bit in (a) local operations, (b) passing buffers, and (c) executing control instructions.

By setting the Secure Bit in a buffer passing across domains, control instructions can easily detect that an address (a return address or a function pointer) was modified by a buffer passing from another domain—there is a buffer-overflow. For later reference, we

establish the condition of setting the Secure Bit when manipulating data across domains as Protocol 1.

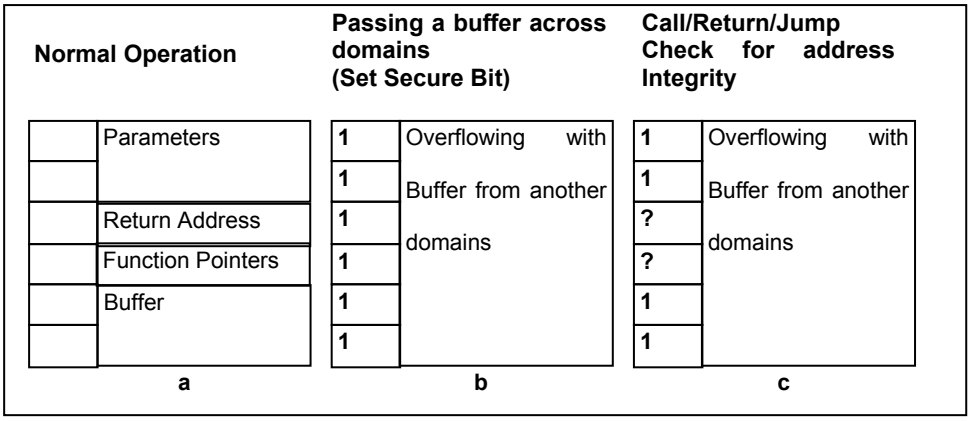

Figure 10 Memory Snap shot with Secure Bit (a) normal operation. (b) Passing a buffer across domains. (c) Related instructions validate the address

Protocol 1: Passing a buffer across domains (devices, machines, and processes) always sets the Secure Bit.

Restatement: All input will have the Secure Bit set.

We will establish that Protocol 1 is a sufficient condition for preventing buffer-overflow attacks in the next section.

To distinguish between normal memory manipulation (within the domain) and passing a buffer across domains, a mode of operation (sbit_write) is introduced to a processor. Manipulating memory will always set the Secure Bit when the sbit_write mode is set. Normal memory manipulating operation (when the sbit_write mode is clear) will carry

the Secure Bit along with the associated memory words. With the presence of this mode, the kernel (or a process) can switch the mode of operation when handling the buffer from another domain. This scheme allows us to provide backward compatibility to all legacy code—mode changes are only necessary within an operating system so Secure Bit is transparent to all other code. We will elaborate the mechanism and necessary conditions for enforcing Protocol 1 in next section (4.3).

4.2 Formalization of Concept

To claim that a system can enforce the integrity of the addresses and result in a secure system, a validation will be discussed. Assuming that a computer system can be represented as a finite-state automaton, we can define a secure system.

Definition 4: A security policy is a statement that partitions the states of the system into a set of authorized, or secure, states and a set of unauthorized or insecure, states. (A definition from [4])

In the case of buffer-overflow attacks, the security policy is simply the statement:

"Overflowing a buffer cannot create a valid address

(e.g. a return address or a function pointer)"

which follows from Corollary 1.1. Before going further, we first define a secure system.

Definition 5: A secure system is a system that starts in an authorized state and cannot enter an unauthorized state. (A definition from [4])

Lemma 2: A system which preserves the integrity of an address (e.g. a return addresses or a function pointer) is a secure system with respect to buffer-overflow attacks.

Restatement: A system that does not use input as control data is a secure system with respect to buffer-overflow attacks on control data.

Proof: Assume that a system is partitioned into two states: normal operation and buffer-overflow attack. By the definition of a buffer-overflow attack (Definition 2), only overwriting the address (e.g. a return address or a function pointer) with an address passed as a buffer (input) to vulnerable programs will result in the state of buffer-overflow attack. By the definition of maintaining the integrity of the address (Definition 3), if such overflowing can be recognized and prevented, the system will not result in the state of buffer-overflow attacks. With respect to Definition 5, our system cannot enter an unauthorized state and is considered to be a secure system.

QED

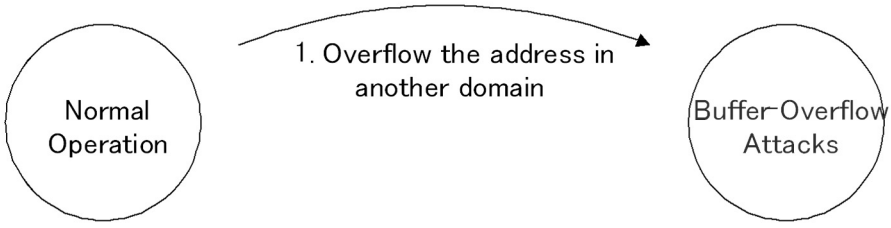

Figure 11 State-transition diagram of buffer-overflow attacks

We will show that the enforcement of Protocol 1 results in a secure system with respect to buffer-overflow attacks.

Lemma 3: Secure Bit and Protocol 1 can preserve the integrity of an address, and result in a secure system with respect to buffer-overflow attacks.

Proof: With the presence of the Secure Bit and Protocol 1, we can detect that an address (e.g. a return address or a function pointer) is overflowed by a buffer passed from another domain (including input). If we can detect that an address is modified by a buffer from another domain, we can preserve the integrity of the address. This follows directly from Definition 3. Thus Secure Bit preserves the integrity of the address and is a secure system with respect to buffer-overflow attacks. This follows directly from Theorem 2

QED

4.3 Protocol Enforcement

This section discusses modifications necessary to enforce Protocol 1. We first elaborate the definition of passing data across domains in our context. From this definition, we provide a guideline for necessary modifications required to enforce such a protocol.

To ease understanding the definition of passing data across domains, we first introduce the term "threat surface" found in Threat Modeling [69]. By invoking threat modeling into the life cycle of software engineering, the "threat surface" is defined as all possible inputs crossing from the software interface. In this context, a domain is a boundary with respect to the current process, and passing data across domains means interfacing the

software with other components – it is “threat surface”. Intuitively, passing data across domains includes passing a buffer between processes (regular IPC), reading and writing from I/O devices, passing a command-line argument to a new process, sending and receiving data from a network socket, etc. Though it sounds somewhat complex, the fundamental concept is that every buffer that is used to interface between the software and outside components is data passing across domains and is therefore suspect.

With this fundamental concept in hand, we can easily analyze the threat surface of software (the O.S. in our case) and modify the surface to operate in sbit_write mode: set the Secure Bit to every memory word passing across domains. The set of access points between processes (the threat surface) is easily identified because they operate across different segments or pages (or any other future type of memory protection structure). In this case, the buffers on the threat surface always pass though the kernel. Thus, we can simply enforce Protocol 1 by modifying the modules that move data across domains to operate in the sbit_write mode.

In certain cases, the kernel may want to pass control data to a user process without setting the Secure Bit (e.g. signal handling and inter-process communication). However, those data are not considered as an input from untrusted environments. Put another way: the kernel is allowed to pass trusted control data to a process without resulting in an attack (if we can validate that data [32]).

4.4 Summary

By augmenting a bit to each byte (or word) of memory, we can implement Secure Bit an provide transparent protection against buffer-overflow attacks on control data. This additional bit is managed by the kernel via sbit_write mode, and provides the semantics for differentiating between local and external data. With the presence of these conditions, the semantics of instructions can be modified to enforce the protocol of preventing external data from being used as control data.

Chapter 5 Design

This chapter discusses issues of design critical to the implementation and deployment of Secure Bit. Those issues include Memory Architecture, Instruction Set Architecture and System Software.

5.1 Memory Architecture

In our design, a pin is added to the memory chip to ensure the integrity of the Secure Bit. This pin is associated with operations that modify the Secure Bit. It functions as both input and output at the same time. In a similar way, processors also require a pin to access this Secure Bit. Figure 12 shows the system block diagram when Secure Bit is present. Secure Bit (shown in *) is the only line added.

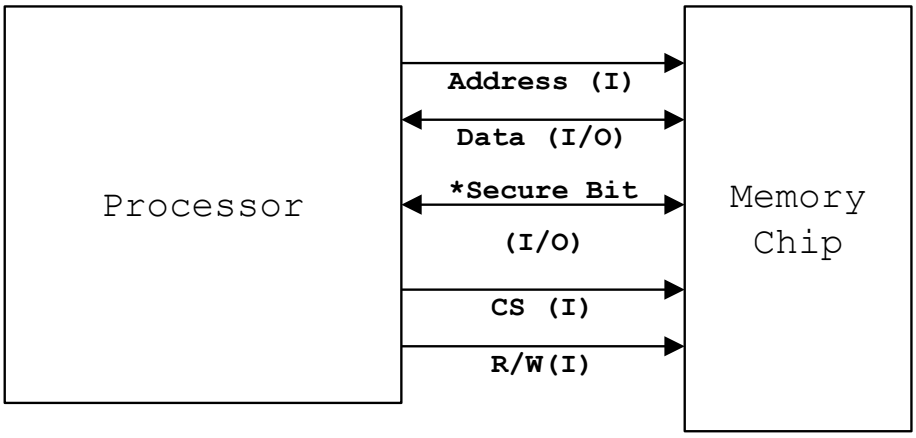

Figure 12 System block diagram

We must ensure that only our integrity-preserving instructions have control over the Secure Bit and that other instructions that write data to memory will always copy the bit. To ensure that every access to memory using other instructions will always copy the bit, the semantics of the write instruction have to be modified. In a generic load-store (RISC)

processor, there will be one write instruction. However, the IA32 architecture has several write instructions (e.g. MOV, MOVS, LEA, etc.). As a result, the semantics of these instructions have to be modified. In order to set or clear the bit, the processor can simply send "1" or "0" to the secure pin. When cache is present, this can be done by issuing the value to the Secure-Bit line of the cache bus leaving the memory controller to handle main memory updates. In term of cache latency, our SimpleScalar implementation [26] has unveiled that there is no hidden cost.

Another issue is that the Secure Bit creates a data bus with an odd number of lines. Moreover, there is no off-the-shelf memory controller or memory chip that currently supports Secure Bit. We will handle this issue by exploring several possibilities.

5.1.1 Memory Organization Modification

Like ECC memory, a memory chip can be added to a RAM card for storing the Secure Bit. The processor, chipsets and caches (all levels) have to be modified to manage this additional memory. Modifications are trivial, but widespread: a path for one Secure Bit per word is needed from the memory chips to the processor execution unit. This path includes: the memory chip itself, CACHE memory, chipsets, and processor. The benefit of this approach is simple organization and better performance. Costs are buried in the hardware. In fact, some researchers [19] suggest that the costs are covered in three days with respect to MOORE's law.

5.1.2 Interleaving Memory

Though the memory architecture is similar to that of a parity bit, as a transitional approach (which might be a better approach anyway), we can interleave data storage and semantic storage (Secure Bit) using a separate interface. While data lines have a byte boundary, the semantic line has a bit boundary. This separation allows us to create a simple memory controller that will convert a bit interface of the Secure Bit to access legacy, byte-boundary memory chips. Since addressing 8 bits requires three bits, the address of the Secure Bit memory (on the byte-boundary memory chips) is simply obtained by interleaving the last three bits of the address for selecting a particular bit from a byte. Figure 13 shows block diagram of the memory interfaces and the circuit of the controller.

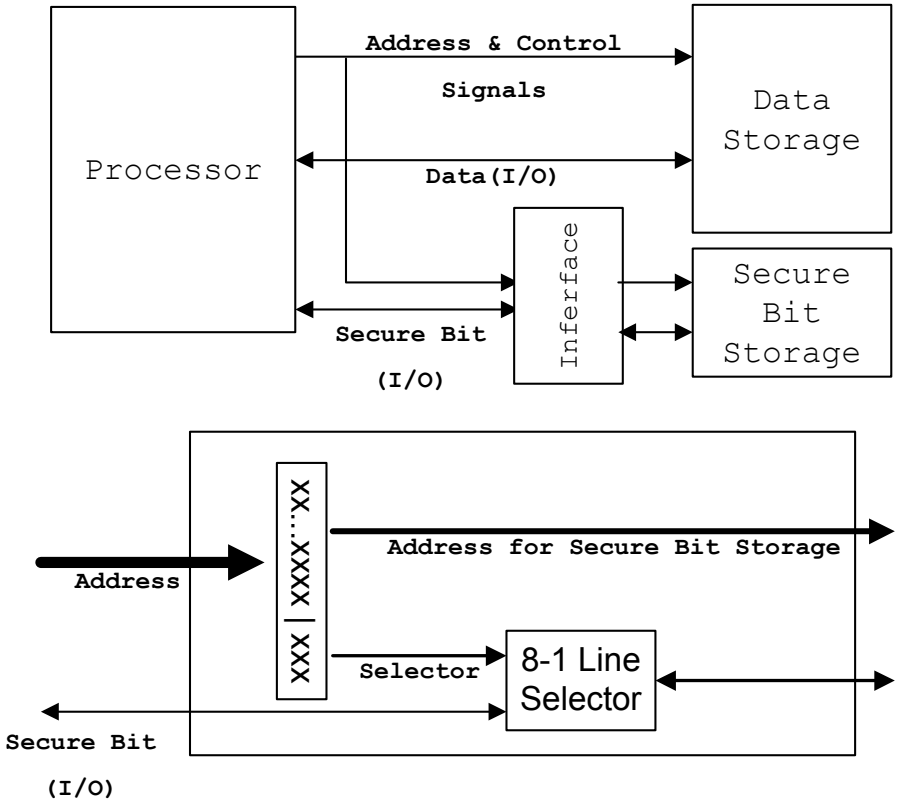

Figure 13 Interleaving Memory Interface and Secure Bit controller

This approach allows us to implement Secure Bit on top of standard byte-boundary memory chip. However, an additional bus line is still need for Secure Bit.

5.1.3 Secure Bit Relocation (Shared Memory)

Alternatively, Secure Bit can be implemented as a processor-only solution and leave chipsets and memory interfaces unmodified. The idea is to allocate a part of the main memory for storing Secure Bit. An additional base register is introduced to a processor for use as a base address for this memory area. The mechanism is similar to that of simple memory relocation using a base register. Figure 14 shows the address translation scheme.

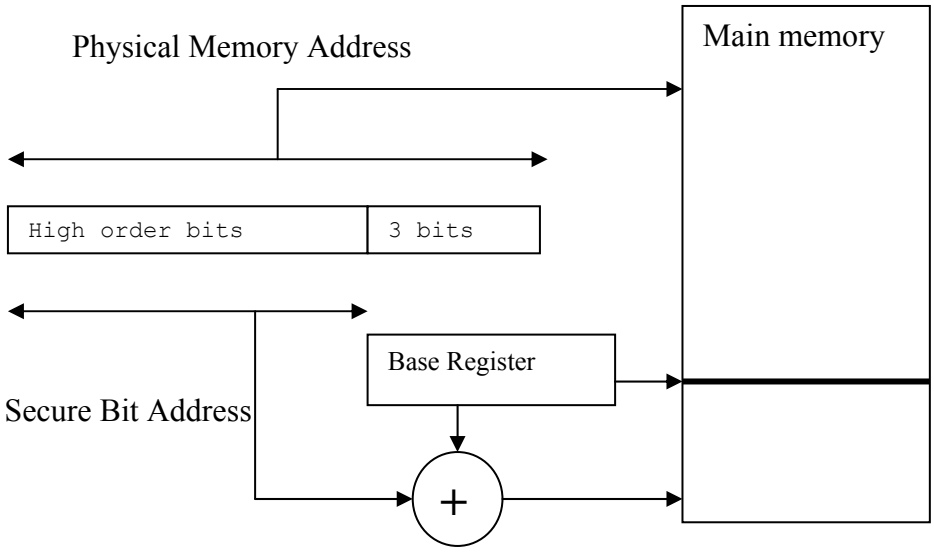

Figure 14 Address translation scheme for Secure Bit Relocation

In this translation scheme, base register, adder and line selector circuits have to be added to a processor chip. This hardware will allow Secure Bit to be stored in a regular byte boundary memory. If desired, there can be separate L1 caches for Secure Bit and Data/Code. However, it is possible to share the bus lines between levels of the memory hierarchy.

The benefits of this approach are that only modifications in a Processor are necessary. In fact, the scheme also provides an efficient way for managing Virtual Memory. This is due to the fact that the bitmap can easily be accessed by the swapper routine. However, sharing the memory means less usable memory (31/32 of physical memory for data and 1/32 for Secure Bit), and less bandwidth for main memory due to bus sharing. We believe that hardware optimization can solve some parts of this issue. Nonetheless, the details are beyond the scope of this thesis.

5.2 Instruction Set Architecture

To implement Secure Bit we needed to change the semantics of several instructions and add an sbit_write mode to the architecture. With several combinations of addressing mode, boundary access, processors' mode and address translation, the full details are somewhat lengthy, but can be summarized as follows.

- The sbit_write flag is added to the EFLAGS register.
- The semantics of the RETURN instruction are modified to validate the Secure Bit of the return address and raise the protection flag when the Secure Bit is invalidated.
- The semantics of the CALL and JUMP instruction are modified to validate the Secure Bit of the address/register that holds the target address when the target is an indirect value (a function pointer), and raise the protection flag when the Secure Bit is invalidated
- Other instructions that access memory are modified to carry the Secure Bit along with the memory word when the sbit_write mode is cleared, and to set the Secure Bit at the destination when the sbit_write mode is set.

- Operations (e.g. shift, arithmetic, or logical) with an insecure operand have an insecure result (Secure Bit is set). An immediate operand is considered to be secure (Secure Bit is cleared).

Although the summary above is for the IA-32 there is nothing about a RISC ISA which would prevent a similar implementation. In fact, our hardware simulation (using SimpleScalar [83]) indirectly supports that assertion (See [26] for more details).

In an attempt to elaborate the design of Secure Bit, we will investigate two types of instructions and related data paths (and components). Those types are: Arithmetic and Logical instructions, and Control instructions.

5.2.1 Arithmetic and Logical Instructions

In order to maintain the integrity of addresses, we must ensure that not only external data are prevented from being used as control data, but that any derivative of external data is also prohibited. This condition can be enforced by extending the arithmetic unit of the processor to carry Secure Bit along with every operation.

A typical processor contains three types of arithmetic and logical operations: shift operations, logical operations, and arithmetic instructions. In most cases, shift instructions (as well as rotate instructions) only involve a register or a word of memory. However, they sometimes shift or rotate across two registers (or words). When two or more operands are involved, the Secure Bits of all bytes of both operands are ORed together. If logical ‘1’ obtained, the result of the Secure Bit is set. Similarly, the Secure

Bit generated from logical and arithmetic operations is the OR of the Secure Bits of all operands.

With the presence of sbit_write mode, the Secure Bit of the results must be forced to be '1' regardless of the operations. Altogether, the ALU of the processor can be augmented for managing Secure Bit using simple OR logic. A possible logic diagram is presented in Figure 15. When the operand contains more than a byte of data, the Secure Bit of each operand is the OR of all Secure Bits in every byte. This way, we can easily enforce protocol one.

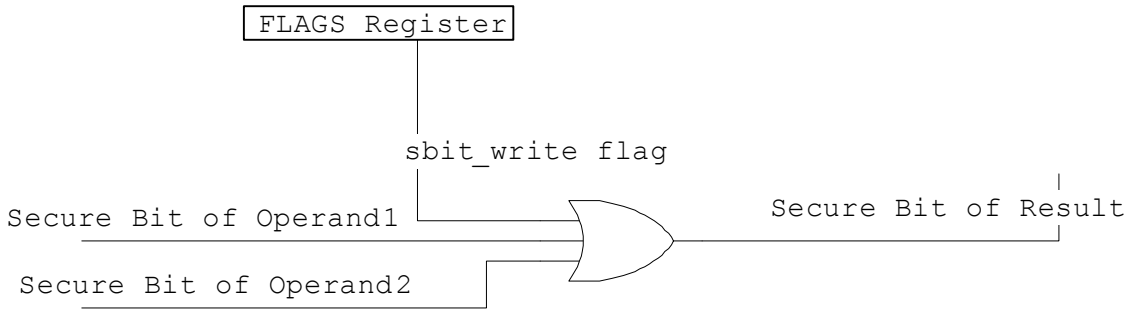

Figure 15 Operations of Secure Bit

5.2.2 Control Instructions

Control instructions such as call, jump, and branch, are instructions that change the control flow such as call, jump, and branch. Depending on the architecture, a call instruction can be implemented as a branch-and-link instruction where the previous instruction pointer is stored in a pre-defined register. However, we are only concerned with the use of the address associated with those instructions.

A control instruction may use an immediate operand for specifying the target address or use a value stored in memory (or storage) as a target. Since the Secure Bit of an immediate value is '0', such a case is considered to be secure. Thus, we only have to validate that no external data should be assigned to the instruction pointer (also referred to as the program counter). This can be enforced by adding a Secure Bit to the instruction pointer. If the instruction pointer is loaded with external data, the Secure Bit will be set to '1'. The processor can simply verify this bit and generate the hardware exception.

5.3 Operating System

There are two issues concerning the management of Secure Bit in the operating system: enforcing the Secure Bit protocol, and virtual memory. In order to enforce the protocol, the kernel must be modified to mange buffers passing between domains in sbit_write mode. With the presence of virtual memory, the kernel must be modified to allocate additional space for storing Secure Bits. We address both issues.

5.3.1 Domains and Buffer Manipulation

To facilitate the management of Secure Bit, it is useful to define domains or boundaries with respect to external data. Modern processors usually partition privileges into at least two modes (also referred as rings): supervisor mode (ring 0) and user mode (ring 3). Each section (page or segment) of memory is then associated with a mode using the applicable memory management mechanism, such as paging or segmentation. Switching to execute code in different modes requires special mechanisms (e.g. call gate or trap). The typical implementation of an operating system would put the kernel in supervisor mode and processes in user mode. In this way, the kernel can allocate resources for each

user process. Thus, all data flowing in and out of the process directly invokes the kernel. The first obvious boundary here is the boundary between kernel and processes.

With sbit_write mode, the kernel also has the flexibility of defining the boundary between processes in the same mode. Considering the threat model, data that a program does not have control over is consider to be external data. This is the second boundary.

It is necessary to note that sometimes a process may be constructed as a group of lightweight processes (threads). Since all threads have freedom to share all data (including control data) among each other, they are considered to be in the same domain.

Given the boundaries discussed above, enforcing the protocol is simply enforcing all functions that copy data to be operated in sbit_write mode between the kernel and user processes, except for the cases when the kernel must pass control data for the process to handle an exception signal and when data is passing between threads.

5.3.2 Virtual Memory

With the presence of virtual memory, a page of memory can be swapped to the swap space (where it is protected by existing protection mechanisms). As a result, a Secure Bit has to be swapped securely in and out of main memory. There are at least three ways to do that securely.

Modifying disk firmware to handle Secure Bit much like parity or ECC bits is conceptually the easiest, but practically expensive.

Alternatively, a separate portion of swap space can be allocated to Secure Bits. A chip-set modification or new instruction can move the Secure Bit to disk. (A separate bitmap is likely an efficient way to store the bits—512 bytes for 4K bytes of memory.) Note that the space for storing the Secure Bit is only required for the Stack and the Heap. The Text section does not need any swap space for the Secure Bit. The paging routine will be modified to render the bitmap of the Secure Bit on swap out, and use the sbit_write mode to restore the Secure Bit on swap in.

A third technique implements Secure Bit on top of byte-boundary memory with assistance from cache and address translation mechanisms (see section 5.1.3). This approach not only allows Secure Bit to be deployed without modifying the memory, but it also allows Secure Bit to be swapped without any special operations.

Since the swapper in the kernel is far removed from the threat surface where buffers are passed from the outside, attacking these routines with a buffer overflow attack is impossible. Similarly, malicious modification of the swap space is protected (e.g. by the operating system, by the file permission). The latter protection is considered to be safe from buffer-overflow attacks.

Though execution time will be increased for swapping a process, the complexity is relatively small compared to other methods. Furthermore, a hardware/software

optimization plays a role here. Regardless of the size of a generic hard drive today, the additional space is not an issue.

5.4 Summary

The major concerns in the design of Secure Bit are: Memory Interface, Instruction Set Architecture, and System Software. The memory Interface can be implemented using several approaches. Each has its strength and weakness. The proper design of a memory interface using the relocation scheme would yield better performance and facilitate the memory management unit in the operating system. The semantics of the impacted instructions are easily enforced with trivial modifications to Arithmetic and Logical Unit (ALU), instruction pointer, and exception handling units. Trivial modifications to the kernel for managing buffers in sbit_write mode are sufficient for enforcing the protocol.

Overall, Secure Bit can be straightforwardly added to any processor. The specific requirement only applies to the memory interface, ALU, instruction pointer, and exception handling units. However, such modifications may touch several parts of the processor.

Chapter 6 Implementation

This chapter discusses the implementation details of Secure Bit in the BOCHS [82] emulator and modifications necessary to the Linux kernel. Readers are expected to have background in the IA-32 architecture and knowledge of the architecture of a Unix-style operating system.

6.1 BOCHS emulator

BOCHS [82], the IA-32 emulator, is constructed from various components. Each component is written in C++. The main components are the CPU, I/O, and Memory objects. The I/O object has several sub-objects representing block devices and I/O subsystems. Similarly, the memory object has several sub-objects. Each sub-object inherits methods from its ancestor. For example, the BIOS has the interface of the memory object, but the lower engine is mapped to a file. As a result, introducing a new sub-component to the emulator is done by deriving a new object from the main object. In addition to these objects, debugger code is embedded into several parts for controlling and monitoring the emulator. The general organization is presented in Figure 16

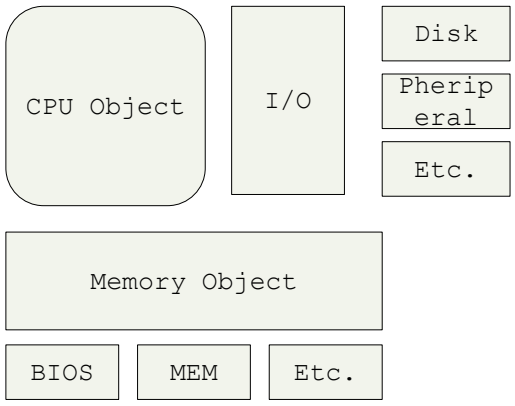

Figure 16 BOCHS Components

To implement Secure Bit, we have to add a bit to a memory byte (or word), customize the CPU and introduce a new device. This section is dedicated to the discussion of these implementation issues.

6.2 Memory

Secure Bit requires that a bit is added to each memory byte. To manage this bit, we intuitively have to allocate additional space for storing Secure Bits. In addition, the interfacing routines must be modified for interfacing with instructions. We will address them respectively.

6.2.1 Memory allocation

Since the emulator is running on top of a byte boundary memory, we have to allocate and map Secure Bit on top of legacy memory. This concept is similar to the one proposed in Section 5.1.2. In the emulator, the space for storing the Secure Bit is the size of physical memory divided by 8 (1 byte for storing 8 Secure Bits). The snapshot of this code is shown in Figure 17.

```
// for secure bit (KPR)
  Bit64u test_mask_s = (alignment - 1)/8;
  size_t bytes_s = bytes/8;
  actual_vector_s = new Bit8u [(bytes_s+test_mask_s)];
// round address forward to nearest multiple of alignment.  Alignment
// MUST BE a power of two for this to work.
  Bit64u  masked_s  =  ((Bit64u)((actual_vector_s  +  test_mask_s)))  &
~test_mask;
  vector_s = (Bit8u *)masked_s;
// sanity check: no lost bits during pointer conversion
  BX_ASSERT (sizeof(masked_s) >= sizeof(vector_s));
// sanity check: after realignment, everything fits in allocated space
  BX_ASSERT (vector_s+bytes_s <= actual_vector_s+bytes_s+test_mask_s);
  BX_INFO (("allocated memory at %p. after alignment, vector=%p for
secure bit",actual_vector_s,vector_s));
```

Figure 17 Memory Allocation for storing Secure Bit

In the memory object, BOCHS uses two vectors, ***vector*** and ***actual_vector***, for managing the memory of the emulated machine. To add Secure Bit, two canonical vectors, ***vector_s*** and ***actual_vector_s***, are added, and one eighth the size of ***actual_vector*** is allocated to ***actual_vector_s***. Note that the ***actual_vector*** is the word aligned version of ***vector***. The mentioned variables are declared in "/memory/memory.h", and the related code is located in function "***alloc_vector_aligned***" of file "/memory/misc_mem.cc". These vectors simplify the association and addressing of data and its assocated Secure Bit.

6.2.2 Memory interface

Conceptually, we have learned that accessing a Secure Bit can be done on top of the byte-boundary memory. Ignoring the segmentation and the mixing of the byte word double-word memory model of the IA-32, a naive design using a line selector should look similar to the diagram in Figure 13.

Due to the complexity of addressing modes in IA-32, the implementation of Figure 13 in the emulator results in a hierarchy of code for manipulating and reading the Secure Bit. However, segmentation and paging are managed by the processor, and the same address can be used to access both data and its Secure Bit. Thus, the mapping is only applicable to the accessing of physical memory. We will omit the details and show the read and write access to Secure Bit in Figure 18 (a) and (b) respectively. This code is located in "/memory/memory.cc".

```
// Set/Clear Secure Bit by KPR
Bit32u a20addr_s;
Bit8u sbyte;
for (int i=0;i<len ;i++ )
{
a20addr_s=(a20addr+i)>>3;
sbyte=(a20addr+i)&0x00000007;
sbyte=1 << sbyte;
if (*sbit==1)
{ // set
vector_s[a20addr_s]|=
      (sbyte&0xff);
} else { // clear
vector_s[a20addr_s]&=
      ~(sbyte&0xff);
}
      sbyte=sbyte<<1;
}
```

A. writePhysicalPage

```
// Read Secure Bit by KPR
Bit32u a20addr_s;
Bit8u sbyte;
Bit8u sread;
sread=0x00;
for (int i=0;i<len ;i++ )
{
a20addr_s=(a20addr+i)>>3;
sbyte=(a20addr+i) & 0x00000007;
sbyte=1 << sbyte;
sread|=(vector_s[a20addr_s]
      &sbyte);
sbyte=sbyte<<1;
}

*sbit=sread;
```

B. readPhysicalPage

Figure 18 Code for (a) manipulating Secure Bit (b) reading Secure Bit

To facilitate our modification to Secure Bit, it is helpful to provide some overloaded functions that maintain the same interface (number of parameters). While some instructions, such as processor control instructions (processor mode and vm8086), do not require any access to Secure Bit, the interfacing functions do. Rather than using one interface, overloading functions provide several interfaces to the same mechanism. Those functions that do not take Secure Bit as a parameter would call the regular Secure Bit aware functions and leave the Secure Bit unmodified. Figure 19 lists all interfaces to physical memory.

Corresponding to these interfaces, the linear memory interfaces (before translating from linear address to physical address) and higher levels are either modified or overloaded to use one of these interfaces. This scheme allows minimal modification to the processor. The modifications to higher level memory access routines are detailed in “/cpu/access.cc”.

```
/// Overload Functions
/// For Secure Bit (KPR)
///
/// Read Data and Secure Bit
BX_MEM_SMF void    readPhysicalPage(BX_CPU_C *cpu, Bit32u addr,
      unsigned len, void *data, int *sbit) BX_CPP_AttrRegparmN(3);
/// Write Data and Secure Bit
BX_MEM_SMF void    writePhysicalPage(BX_CPU_C *cpu, Bit32u addr,
      unsigned len, void *data, int *sbit) BX_CPP_AttrRegparmN(3);
/// Write Data (with optional Secure Bit)
/// if ignore=0, leave the Secure Bit unmodified
BX_MEM_SMF void    writePhysicalPage(BX_CPU_C *cpu, Bit32u addr,
      unsigned    len,    void    *data,    int    *sbit,int    ignore)
      BX_CPP_AttrRegparmN(3);
///
/// End (KPR)
///
/// Read Data, ignore Secure Bit
BX_MEM_SMF void    readPhysicalPage(BX_CPU_C *cpu, Bit32u addr,
      unsigned len, void *data) BX_CPP_AttrRegparmN(3);
/// Write Data, ignore Secure Bit
BX_MEM_SMF void    writePhysicalPage(BX_CPU_C *cpu, Bit32u addr,
      unsigned len, void *data) BX_CPP_AttrRegparmN(3);
```

Figure 19 List of overloading functions for accessing physical memory

For debugging purpose, we also modify the *dbg_fetch_mem* function which is used for memory dumping. Basically, this function is used by the monitor sub-system of the emulator for dumping the memory.

6.3 Instruction Set Architecture

To enforce the Secure Bit, the semantics of several instructions have to be modified. All data operations have to carry and maintain the Secure Bit (except the clear instruction). Data manipulation instructions (e.g. move) have to set Secure Bit to '1' when operated in sbit_write mode, and control instructions must validate the Secure Bit of all target addresses. We will consider each set of instructions seperately.

6.3.1 Operations

The IA-32 contains operations that can be grouped as shift operations, logical operations, and arithmetic operations. Since BOCHS implemented a function for each operation and addressing mode (approximately 180 combinations), the details are somewhat lengthy. To smooth the progress of our modifications, we introduce a macro for each operation in Figure 20. For each operation, the Secure Bit of every operand is fetched into corresponding variables and calculated using the relevant macro.

```
// Secure Bit operation for each type of ALU instruction
#define SBIT_SHX(sbit1) (sbit1 ==0)?0:1
#define SBIT_ROX(sbit1) (sbit1 ==0)?0:1
#define SBIT_XOR(sbit1,sbit2) (sbit1|sbit2)==0?0:1
#define SBIT_AND(sbit1,sbit2) (sbit1|sbit2)==0?0:1
#define SBIT_OR(sbit1,sbit2)  (sbit1|sbit2)==0?0:1
#define SBIT_NOT(sbit1) (sbit1 ==0)?0:1
#define SBIT_ADD(sbit1,sbit2) (sbit1|sbit2)==0?0:1
#define SBIT_SUB(sbit1,sbit2) (sbit1|sbit2)==0?0:1
#define SBIT_MUL(sbit1,sbit2) (sbit1|sbit2)==0?0:1    // and DIV
```

Figure 20 Macros for operations on Secure Bit

In the current implementations, we leave MMX and other vector operations unmodified, sicne most of these instructions are sometimes implemented in the co-processor. We believe that these instructions are rarely used in manipulating control data, leaving the Secure Bit of these operands unmodified should not cause any problem. However, applying the same concepts to these operations should be trivial.

6.3.2 Data Manipulation

In an attempt to avoid adding any new instruction, we choose a reserved bit in the EFLAGS register to create a mode. To allow this mode bit to be accessible by all widths of instructions, the fifth bit is selected. This choice allows the processor to use existing instructions for managing the sbit_write mode. Switching the mode can be done easily by

flipping a bit in the EFLAGS register. Given this setup, we create a macro "sbit_mode" to refer to this flag and use it to manage the Secure Bit in move instructions (including string instructions and similar). Basically, the instructions are modified to read the Secure Bit, calculate the result, and store it in the destination. The statement for calculating the Secure Bit of a destination is exemplified in Figure 21.

```
sbit=(sbit_mode)? 1:sbit;
```

Figure 21 Data manipulation using sbit_write mode

6.3.3 Control Data

The IA-32 architecture uses two registers (word) for constructing the linear address of an instruction: code segment (ECS) and instruction pointer (EIP). Code segment is the base register, and instruction pointer is the offset. We simply enforce our protocol by validating the Secure Bit of a target address assigned to both registers. The modification can be summarized as: for every assignment to a code segment and instruction pointer, if the Secure Bit is set to '1', raise the exception. This statement can be translated to code, Figure 22.

```
  // Validate call target
  if (sbit != 0) {
        BX_INFO(("call_ew: sbit of target is not secure"));
#ifdef HAS_SBIT_EXCEPTION
                exception(BX_GP_EXCEPTION, 0, 0);
#endif
  }
```

Figure 22 Validation of Secure Bit in control instructions

6.4 Linux

To understand the modifications to the kernel, we first explain the organization of Linux kernel in general. Linux uses a monolithic kernel where all device drivers and file systems are included in the kernel with the memory management unit, process

management unit, and scheduler. Some parts of the kernel are architecture specific. Table 1 presents the directory structure of the Linux kernel. This organization will serve as the basis of our threat interface.

Name	Description
Arch	Architecture specific functions
Drivers	Device drivers
Fs	File system
Include	Generic header
Init	Kernel startup routine
Ipc	System V inter-process communication
Kernel	Generic kernel
Lib	Generic library
Modules	Kernel modules
Net	Network library

Table 1 Directory structure of Linux Kernel

To enforce the protection mechanism, we modify the part of the Linux kernel that passes a buffer across domains (copying data between a process and a kernel or a common threat interface of every process). With this modification, we cover a variety of communication: a buffer of a network socket, command-line arguments, a buffer passing from a process to kernel and vice versa, and a buffer passing between processes. These routines are usually implemented as preprocessing macros in one file "/include/asm-i386/uaccess.h" which refer to functions in the file "/arch/i386/lib/usercopy.c". The modifications can be summarized by listing the set of routines to operate in sbit_write mode:

- copy_to_user
- copy_from_user

Figure 23 shows the organization of the Linux kernel and routines for handling buffers across domains.

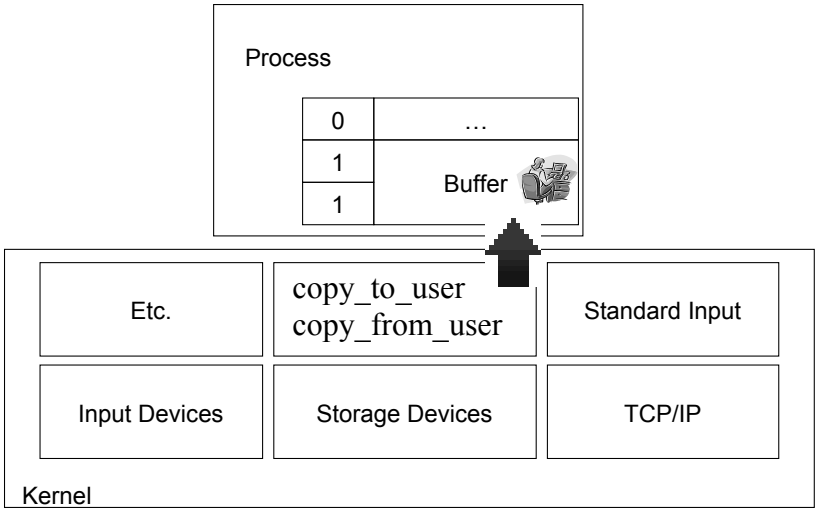

Figure 23 Linux Kernel Organization

Since sbit_write mode is implemented as a bit in the EFLAGS register, switching the mode can be easily done by flipping a bit in the register. To facilitate the modification, two macros, SET_SBITMODE() and CLR_SBITMODE(), are created, Figure 24. The SET_SBITMODE macro is inserted into the top, and the CLR_SBITMODE macro is inserted into the end of the copy functions.

```
// For Secure Bit 2 by Krerk Piromsopa
#define SET_SBITMODE() \
        asm volatile(                                                   \
                "       pushl   %eax\n" \
                "       lahf\n" \
                "       orb     $0x20, %ah\n"   \
                "       sahf\n" \
                "       popl    %eax"   )

#define CLR_SBITMODE() \
        asm volatile(                                                   \
                "       pushl   %eax\n" \
                "       lahf\n" \
                "       andb    $0xdf, %ah\n"   \
                "       sahf\n" \
                "       popl    %eax"   )
```

Figure 24 Macros for managing sbit_write mode (in "/include/asm-i386/uaccess.h")

As mentioned earlier in Section 4.3, the kernel has the ability to pass validated control data to a process without setting the Secure Bit (using other functions). If that data is

trustworthy (not an input from other processes or devices), the system will not move to the state of a buffer-overflow attack. Thus, we list the same functions that do not operate in sbit_write mode as follows:

- copy_to_user_s
- copy_from_user_s

Routines that are allowed to pass trusted data to the user (using copy_to_user_s and copy_from_user_s) are several kernel and filesystem routines. The kernel routines are located in "/kernel" and "/arch/i386/kernel". Those routines that are allowed to manage trusted user data are routines that mange signal handling, ptrace, description table, capability and scheduler. The file system routines that can store trusted user data are located in in "/fs/pipe.c" and in "/fs/binfmt_elf.c", which are the pipe communication (read and write) used by pthread and elf-headers creation.

6.5 Summary

The implementation of Secure Bit into an existing architecture is a straightforward process. However, the implementation complexity depends on the complexity of the architecture. In this chapter, the modifications necessary for enforcing Secure Bit at both the architecture and the systems level using IA-32 and Linux as an example were presented. The same concepts should be easily adapted to any operating system or architecture.

Chapter 7 Possible Attacks on Secure Bit

For any proposed security scheme, one must always consider ways around it. Two possible attack vectors exist. One is on the Secure Bit itself; the other is on the protocol.

- Attack the Secure Bit itself: Since no mechanism exists to clear a Secure Bit associated with a word, any attempt to mark an untrustworthy piece of data as trustworthy cannot succeed. Furthermore, since operations using untrustworthy data result in untrustworthy data, it is not possible to use some combination of operations to result in a trustworthy piece of data which was corrupted by untrustworthy data. One might consider attacking a Secure Bit which has been swapped to disk, but existing protection mechanisms provide sufficient protection.

- Attack the Protocol: The other possibility is an attack on the protocol itself. Most of the protocol is implemented in hardware so that part is protected from software attacks. The remaining part lies in the modifications to the operating system. However, operating system binaries should be in protected memory, e.g. read-only, so this part of the protocol is also protected.

If Secure Bit itself resists attacks, what related attacks or weaknesses exist? There are two of interest. We have stated that Secure Bit is not designed to protect against attacks on non-control data. We will first consider at a buffer-overflow attack of non-control data which can be used to modify control. Then we will look at an unsafe style of compilation which may cause Secure Bit to generate a false positive.

7.1 Attacks on non-Control data

With the presence of Secure Bit, using data passed from another domain (or its derived result) as a function pointer or a return address is prohibited. Since an external attack is not possible, only an internal attack is a possibility (assigning local data or a constant value). Such an internal attack must actively circumvent protections to maliciously attack itself or is an effect from another attack. However, it is not an attack on control data, but a "process suicide" (programming error) or other type of attack. With respect to the definition of buffer-overflow attacks on control data (definition 2) only using data passed from another domain (including input) as control data is considered a buffer-overflow attack.

Figure 25 presents a sample case of an attack on non-control data where the vulnerability might be applicable.

```
int b() {
      char *src,*dest;
      char buff[10];

      printf("Input string:.\n");
      gets(buff); // Overflow *src, *dest
      ...
      strcpy(src,dest); // Copy src to dest
}

int a() {
      …
      b();
      …
}

int main (int argc,char *argv[]) {
        a();
}
```

Figure 25 Sample Buffer-Overflow attack on non-control data

In this example, main calls function "a" which then calls the vulnerable function "b". Within "b" the user inputs `buff` which can overflow to both overwrite *`src` to point to the return address of a previous call (e.g. "`a()`") and overwrite *`dest` to point to the target address (e.g. return address of "`b()`" or "`main()`"). Note that this overflow is possible only if all optimization is turned off so that neither `src` nor `dest` is in a register. Under these circumstances it is possible to change the control flow without replacing control data with external data—only internal data is used. Note that the damage in this example is to create an infinite loop or to crash the program, effectively a denial of service to the process.

While most internal data targets will be benign, one can imagine malicious possibilities, even if they are a bit far-fetched. For example, if for some reason a programmer created a function pointer to shell and had both a vulnerable copy routine and no optimization; one could copy that shell pointer elsewhere to allow a shell call someplace different than the programmer intended. Note that the desired *privileged-elevated* shell is not possible with this attack. Alternatively, (again with a vulnerable copy routine and no optimization) if one had function pointers to both an authorization "accept" function and a "reject" function, one might be able to redirect program flow to subvert an authorization routine to the "accept" function when the "reject" function was expected.

Though these attacks sound probabilistically low, they are not impossible. We simply have to eliminate at least one critical condition. There are three possible methods. The first is preventing a raw address from being stored directly in the program. A second

possibility is securing the target address from being modified (e.g. GOT and function pointers). The final possibility is to validate that both the source and destination pointers have not been maliciously modified.

Rather than storing an address directly into the GOT table or function pointer, we may choose to store an encoded version of an address or to store a relative address. Even a trivial encoding such as XOR with some constant would be sufficient. However, this approach does not prevent a copy between locations that share the same encoding scheme or key used to encrypt the address (e.g. between function pointers or entries in the GOT).

Rather than making the useful address useless, we can protect the target from being modified. In the case of GOT, we can protect the GOT from being a target by declaring it as read only after the shared library is configured. Nonetheless, we cannot apply the same idea to protect function pointers or return addresses in general.

Alternatively, we can validate (assert) the source and destination pointers before running the "strcpy(..)" function. If the source and destination pointers can be validated, the attack can be prevented. However, a false alarm may be generated when a pointer is the arithmetic result of input where its Secure Bit is legally set.

Thus, we propose extending Secure Bit to protect against buffer-overflows of non-control data. In addition to the broader protection provided, this specific attack can be prevented

by preserving the integrity of the source and destination pointers from outside modification. However, this is outside the scope of this thesis.

7.2 False Positives

Programs should not use input to construct control data. If a program uses input as control data, Secure Bit will raise an exception. Raising such an exception is reasonable, but compilers sometimes choose to generate control data from inputs for optimizations. The obvious example is the switch statement in the C programming language. In such a case, a compiler can use a part of the input to create a target address. This method sometimes yields faster switching and smaller code.

The switch statement can be implemented by using compare instructions. This approach avoids the unsafe approach of using input as control data. For example, code generated from the GNU C compiler always uses the safe approach of implementing switch statements with compare instructions.

7.3 Summary

Secure Bit provides protection against buffer-overflow attacks on control data. However, there exists an attack on pointers that may allow local data to be constructed as control data. This is not a direct attack on control data and requires another mechanism for protection. Nonetheless, a solution is beyond the scope of this thesis.

In addition, Secure Bit may prevent some optimized code from running when input is used to construct an address. In this situation, the program (or compilers) may have to be

modified. Though the case is rarely applicable and can be avoided, it prevents Secure Bit from providing 100% backward compatibility to legacy user binaries.

Chapter 8 Evaluation

Our assumption is that Secure Bit will protect against buffer overflow without degrading performance. Though it requires more memory, adding more memory is a one-time cost compared to the indefinite cost of future buffer-overflow attacks. We evaluate our work with respect to several hardware and software issues as follows.

- An ability to boot Linux on a Secure-Bit-enabled machine demonstrates the completeness of our implementation in its ability to handle software of the complexity of a real operating system.
- An ability to run a legacy application on a Secure-Bit-enabled machine demonstrates the transparency and compatibility of Secure Bit from an application point of view.
- An ability to prevent an unmodified buffer-overflow-vulnerable program from being exploited validates the effectiveness in preventing buffer-overflow attacks.
- The instructions that have to be modified in order to support Secure Bit indirectly demonstrate the necessary semantics for preventing buffer overflow attacks in hardware.

We will evaluate these issues individually.

8.1 Booting Linux

We have booted Linux on the emulator to demonstrate the transparency of Secure Bit as well as its compatibility. With Linux running, we mount existing buffer-overflow attacks, and demonstrate how Secure Bit will trap them. Since our architecture does not modify

the syntax of any instruction, we are able to boot an unmodified version of Linux (RedHat 6.2 [84]) in the emulator

To provide the protection, the kernel is modified to manage buffers passing between kernel and process in sbit_write mode (See section 6.4 for more details). The Secure-Bit-enabled Linux is able to boot in the emulator.

8.2 Compatibility

The Secure Bit protection mechanism is compatible with legacy binaries. In fact, we have successfully compiled and run several serious benchmarks, such as GCC, in the emulator without any modification. One notable application is Apache. With Apache running we were able to set up a Web server which was useful for both testing compatibility as well as resistance to attacks.

The first version of our Secure-Bit-enabled Linux failed to run any Java program in SUN JVM. The same program ran correctly on Kaffe [85], a free implementation of Java Runtime Environment. The difference is that SUN JVM uses pthread libraries. POSIX thread (pthread) libraries, a standards based lightweight process API for C/C++, use the pipe system call for passing control data between threads. Since threads belong to the same process, passing control data within a process does not violate the Secure Bit policy. Thus, the kernel was modified to correctly implement the policy. The Pipe system call is modified to allow data to flow between threads without flagging the Secure Bit. This modification allows Secure Bit to provide compatibility to thread-aware software.

To appreciate the compatibility of Secure Bit, here are some benchmarks (some are from SPEC CPU2000 [86]) that we have run on the emulator.

- gzip (SPEC CPU2000): GNU zip is a popular data compression program. It uses Lempel-Ziv coding (LZ77) as its compression algorithm. An ability to run this application indicates that the system is able to handle that complex algorithm.
- bzip2 (SPEC CPU2000): bzip2 compresses files using the Burrows-Wheeler block-sorting text compression algorithm and Huffman coding. Compression is generally considerably better than that achieved by more conventional LZ77/LZ78-based compressors, and approaches the performance of the PPM family of statistical compressors. Running this program simply demonstrates an ability to handle a slightly more complex piece of code.
- gcc (SPEC CPU2000): This gcc is based on gcc version 2.91.66. We used it to manually build the Linux box (kernel and some applications). Since gcc exercises a wide variety of data structures, it serves as a good benchmark for validating compatibility.
- Perl and Shell scripts: Perl is an interpreted language optimized for scanning, extracting, and printing text. As a part of building our server, we have encountered several Perl scripts and shell scripts. The ability to run these scripts shows that Secure Bit does not introduce any incompatibility with popular scripting languages.
- OpenSSL: As a part of building a working server, we also build OpenSSL version 0.9.5. OpenSSL provides a full-strength general purpose cryptography

library. The library is used to manage the cryptography for Apache and OpenSSH. This demonstrates the compatibility in handling the complexity of cryptography.

- Apache with mod_ssl: Apache version 1.3.12 and mod_ssl version 2.6.6 are selected for use in our Linux box. Since this version is vulnerable to the SLAPPER worm, it also allows us to test the protection provided by Secure Bit.. One notable characteristic of Apache is that it is a multithreading application. Apache always forks a new thread for handling an individual HTTP request. Running Apache shows that Secure Bit can handle the complexity of real, multithreaded server application (including SSL).
- Telnetd and WUFTPD: These daemons are turned on by default in RedHat 6.2. Before installing OpenSSH, this was our default for transferring data and accessing the box remotely. One characteristic of ftp protocol is that two ports are used for the communication (one for control and one for data). These applications demonstrate ability in handling legacy network applications (and protocols).
- OpenSSH: As a replacement for telnet and ftp, we successfully built and installed SSHD server and SSH client (including sftp and scp). They are our method for remotely accessing the Secure Bit box. Embedded in an ability to run OpenSSH is the ability to handle encrypted client-server applications.
- Java Virtual Machine: There exist several implementations of Java Virtual Machine. We have tested Sun JVM and Kaffe. By default, Java Virtual Machine includes garbage collector and uses several threads for managing the Virtual

Machine. An ability to run these applications shows that Secure Bit can handle the complex Virtual Machine and lightweight processes (threads).

It is not practical to test all applications, but this list indicates that Secure Bit is compatible with a variety of existing software.

8.3 Mounting Attack

Testing a full suite of well-known buffer-overflow attacks is unrealistic on an emulator. Instead, we have created a simple program that is vulnerable to buffer-overflow attacks, and tried to mount several attacks on it. In addition to normal return-address attacks and function-pointer attacks, we also mount a multistage buffer-overflow attack that overflows the Global Offset Table (as described in Section 1.4). In fact, we managed to install a vulnerable version of Apache mod_ssl in the emulator and remotely mount the SLAPPER [33] worm attack. Resisting SLAPPER is important because its multistage attack defeats other buffer-overflow protection mechanisms [7]. Without any modification, all attempts failed to compromise the Secure Bit.

To understand the attacks protected by Secure Bit, we have created a test suite for modeling buffer-overflow attacks on control data. This suite is sufficient for representing variations of stack smashing, return-address attacks, function-pointer attacks, and GOT attacks.

Test 1: Stack smashing and return-address attacks

In a stack smashing attack, a buffer overflow replaces the return address of the calling function stored in the stack frame. Figure 26 shows a simple stack smashing vulnerable

program used in our test. With appropriate arguments, this return address of the main function will be overflowed (by the strcpy() function).

```
void concat_arguments(int argc, char**argv) {
  char buf[20];
  char *p = buf;
  int i;

  for(i=1;i<argc;i++) {
       strcpy(p, argv[i]);    /* The damage occurs here. */
       p+=strlen(argv[i]);
       if(i+1 != argc) {
        *p++ = ' ';
       }
  }
  printf("%s\n", buf);
}

int main(int argc, char **argv) {
  concat_arguments(argc, argv);
  printf("%p\n", &concat_arguments);
}
```

Figure 26 Example of stack smashing venerability

To facilitate the testing process, a wrapper program was created to explode this example by passing a lengthy argument to the program. Assuming that the address of ***concat_arguments()*** function is 0x080484 (obtained from *objdump* utility) and the return address is 44 bytes relative to the ***buff***, we can construct a buffer-overflow attack to create an infinite call to ***concat_arguments()*** by overflowing the return address with the address of ***concat_arguments()***. The details can be found in Figure 27

Without Secure Bit, the program is expected to loop indefinitely. However, Secure Bit stops this program by raising a segmentation fault signal. Monitoring the console, we can observe the protection provided by Secure Bit (see Table 2).

```
int main(int argc,char **argv) {
  char *buf = (char *) malloc(sizeof(char)*1024);
  char **arr = (char **)malloc(sizeof(char *)*3);
  unsigned char tmp;
  int i=0,j=44;
/* fill the buffer until we reach the offset */
  for(i=0;i<j;i++) buf[i]='x';
/* target address */
  buf[j]  = 0x37;  buf[j+1]= 0x84;  buf[j+2]= 0x04;  buf[j+3]= 0x08;
  buf[j+4]= 0x02; /* for argc */
  buf[j+5]= 0x00; /* string terminator */
/* arguments for execv() */
  arr[0]="./test";  arr[1]=buf;  arr[2]='\0';

  execv("./test",arr);  /* call the vulnerable program */
}
```

Figure 27 Wrapper program for exploding stack smashing

A. Application Console	$./wrapper 0x8049744 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx7 Segmentation fault
B. Emulator Console	Retnear32: sbit of EIP is not secure [8048437]

Table 2 Results from stack smashing test

Test 2: function-pointer attacks, and GOT attacks

Another obvious target for buffer-overflow attacks on control data is a function pointer. Though function pointers can be found arbitrarily, a particularly tricky attack is on the entries in the Global Offset Table (GOT). This attack is important because it defeats most other buffer-overflow protection schemes [7]. An example of such a vulnerability is shown in Figure 28. In this example, the function-pointer entry of ***printf()*** function located in the GOT will be modified to point to ***residentcode()*** function.

```
int residentcode() {
  /* We are in trouble */
  execl("/bin/sh","/bin/sh",0x00);
}

int vulnerable(char **argv) {
  int x;
  char *ptr;
  char buffer[30];
  ptr=buffer;
  printf("ptr %p - before\n",ptr);
  strcpy(ptr,argv[1]);  /* overflow ptr */
  printf("ptr %p - after\n",ptr);
  strcpy(ptr,argv[2]);  /* overflow the target */
}

int main (int argc,char *argv[]) {
  printf("Sample program.\n");
  vulnerable(argv);
  printf("Program exits normally.\n");
}
```

Figure 28 Example of GOT vulnerability

Suppose that the address of ***residentcode()*** function is 0x8048454 and the GOT entry of ***printf()*** function is located at 0x8049730 (obtained from *objdump* utility). Exploding this program can be accomplished by first overflowing the ***ptr*** pointer to point to the entry of ***printf()*** function. Overflowing this entry can bind the ***printf()*** function to any code. In this case, the ***residentcode()*** function is the target. The wrapper program for exploiting such vulnerability is shown in Figure 29.

Without any protection mechanism, a shell session would be expected. However, Secure Bit stops the second call to ***printf()*** function from executing the ***residentcode()*** by raising segmentation fault signal. A snapshot of the console is shown in Table 3.

```
int main(int argc,char **argv) {
  int *iptr;
  char *buf1 = (char *)malloc(sizeof(char)*46);
  char buf2[5]="Addr";
  char **arr = (char **)malloc(sizeof(char *)*4);
  memset(buf1,'x',0x20);
  iptr=(int *) buf1;
  iptr+=(0x20 / sizeof(int));
/* printf entry in the GOT */
  *iptr=0x08049730;  buf1[0x24]='\0';
/* address of residentcode() */
  iptr=(int *)buf2;
  *iptr=0x08048454;
/* arguments for execv() */
  arr[0]="./vul";  arr[1]=buf1;  arr[2]=buf2;  arr[3]='\0';

  execv(arr[0],arr);
}
```

Figure 29 Wrapper program for exploding GOT example

A. Application Console	Sample program. ptr 0xbffffac0 - before ptr 0x8049730 - after Segmentation fault
B. Emulator Console	jmp_ed: sbit of target is not secure

Table 3 Results from GOT test

8.4 Instruction Set Architecture

Instructions that are modified to handle Secure Bit can be divided into 3 groups: Control instructions, ALU instructions, and Move instructions. Control instructions are modified to validate the Secure Bit of target addresses. ALU instructions are modified to propagate the Secure Bit. Depending on the sbit_write mode, Move instructions are modified to either carry the either carry the Secure Bit or set the bit to '1'. Such modifications are

trivial but touch several instructions. From the architecture point of view (see Chapter 5), these modifications however involve few data paths and logic for handling the Secure Bit.

8.5 Summary

We qualitatively conclude that Secure Bit provides protection against buffer-overflow attacks on control data with minimal modifications to the processor and operating system.

Chapter 9 Analysis

In this chapter, we analyze several cost aspects of Secure Bit. These aspects are: backward compatibility, deployment, space (hardware requirement), performance, and power consumption.

9.1 Backward Compatibility

The Secure Bit protection mechanism is compatible with legacy binaries. In fact, we have successfully run several serious benchmarks, such as GCC and Apache, in the emulator without any modification. Only the kernel modules required small modifications. We conclude that Secure Bit provides backward compatibility to user binaries and that the protection mechanism is transparent to the user.

9.2 Deployment

Though the Secure Bit requires new hardware, the amount of investment is relatively small compared to new, security-enhanced hardware such as Intel LaGrande [36] (with Microsoft NGSCB [45]). In fact, Secure Bit is complimentary to LaGrande by enhancing it to provide buffer-overflow protection, so Secure Bit could be added onto LaGrande. An intermediate implementation path is also available because Secure Bit could be implemented as middleware. Yet another alternate path is possible by implementing Secure Bit only on the processor with a special memory mapping scheme and cache, thus freeing memory from modification.

9.3 Space

The one-time cost of Secure Bit is comparable to that of a parity bit: having n words of memory, an additional n bits are needed. From a memory point of view, this Secure Bit is simply another data bit. From a processor point of view, it is only controlled by selected instructions. The Secure Bit need not be kept on disk, but the memory bus will have to handle it. The details of this cost are critical to the success of Secure Bit, and our investigation of an implementation at the processor level with SimpleScalar addresses this point—there are no hidden costs.

9.4 Performance

Accessing Secure Bit adds no additional latency to accessing other data since Secure Bit access can be done in parallel with accessing its associated data. The few instructions added to the kernel for switching the sbit_write mode add a delay that is too small to measure. Only a trivial additional penalty is added for rendering the Secure Bit when swapping the page in and out from swapping spaces. However, a hardware/software optimization can play a role here, but that study is beyond our scope. Thus, we can conclude that no significant performance penalty is introduced.

9.5 Power Consumption

With Secure Bit in place, more electric current is apparently required to drive the extra line. An extra line occurs in several places including registers, data paths, data buses, and memory chips. Since electric current only flows when logic switches, power consumption is highly depended on the memory trace. However, we believe that the power consumed by Secure Bit is comparable to that of ECC or parity memory. In fact, it is minimal

compared to the benefits provided by Secure Bit. Nonetheless, such a detailed study would be valuable, but is beyond the scope of this thesis.

9.6 Cost Analysis

Up to this point, we have to trade off between investment and security. One may argue that security provided by Secure Bit is coming at a high price. We do agree that Secure Bit requires modifications to the processor, chipset, and memory hierarchy. However, the cost should be resonable.

Except for instruction semantics, all modifications are similar to adding ECC or parity bits to memory words. To appreciate memory modifications, Figure 30 shows memory cards with and without ECC bits. In the latter case, an extra memory chip is included on the card along with a smaller chip to run error-correcting algorithms. Secure Bit could use leftover bits in the ECC memory, or use a similar dedicated chip.

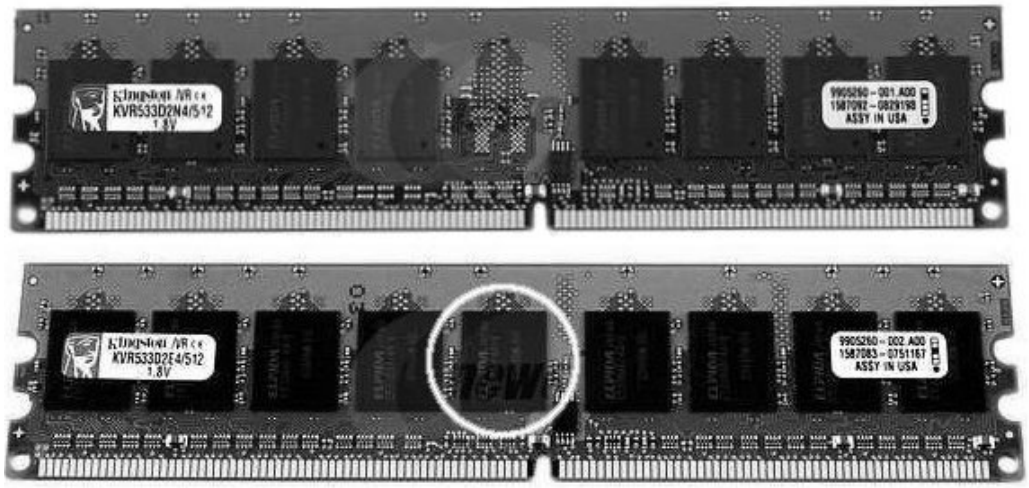

Figure 30 Top: DRAM card without ECC;

Bottom: DRAM card with an extra memory chip for ECC

Considering that processor implementations are constantly evolved, adding Secure Bit to the processor would be trivial. In fact, some researchers [19] suggest that the costs are covered in three days with respect to Moore's law.

Chapter 10 Conclusion

This chapter summarizes the contributions of this thesis, the significance of Secure Bit, its short-comings and possible future research.

10.1 Contributions

This thesis provides several original concepts critical to the success of buffer-overflow protection. These concepts are:

- Survey and classification of current approaches against buffer-overflow attacks: This classification provides a foundation for observing and analyzing pitfalls and identifying components critical to the practicality of solutions.
- Theory of buffer overflow and its counterparts: From our observations, we proposed a theory for preventing address-corrupting buffer-overflow attacks and supported this idea by proving that it provides a secure system with respect to buffer-overflow attacks. This theory serves as a framework for validating a solution against buffer-overflow attacks in that the integrity of address and metadata (if applicable) must be preserved.
- Secure Bit: The protocol and implementation of Secure Bit have provided a case study for buffer-overflow protection where metadata is encapsulated in hardware and unavailable to users, programmers, and attackers.

In addition, we contributed to the classification of buffer-overflow attacks itself. Attacks are classified into generic buffer-overflow attacks, multistage buffer-overflow attacks, and pointer attacks (a.k.a. arbitrary copy). However, this classification is not entirely original, but is based the collected wisdom of others.

10.2 Secure Bit

The necessary and sufficient condition for an address-corrupting buffer-overflow attack is the corruption of an address. Based on that condition, we proposed and demonstrated Secure Bit: a hardware buffer-overflow-attack prevention scheme. Secure Bit is transparent to user software so it provides backward compatibility. In addition, it protects against first-generation stack smashing, against the more recent function-pointer attacks and Global Offset Table attacks.

Based on the principle of protecting the integrity of an address, the mechanism should provide protection against future buffer-overflow attacks. Although this thesis is limited to control-modifying attacks, we may be able to apply the concept of the Secure Bit to prevent a type of buffer-overflow attack that modifies a data pointer to leak confidential information (e.g. password protected data in an object) by preserving the integrity of a data pointer (modifying the syntax of every instruction that accesses memory in indirect mode to validate the Secure Bit).

Our booting of Linux on an emulation of Secure Bit demonstrates the transparency and the robustness of our approach. Running real software such as GCC, JVM and Apache on that Linux plus staging attacks on the emulation further demonstrates its effectiveness. Finally, a hardware simulation [26] demonstrates Secure Bit's viability at the register-transfer level.

10.3 Future Research

Like other solutions, Secure Bit also fails to protect from buffer-overflow attacks on non-control data where attackers can arbitrarily copy data from one location to another. After completely eliminating buffer-overflow attacks on control data from the field, the next critical target is the buffer-overflow attack on data in general. However, unlike the control data, variables are sometimes derived from input. That characteristic prevents us from transparently providing the same protection.

We propose applying Secure Bit to protect data by providing the ability for the user to manage additional metadata for differentiating between legal use of input and illegal overflow.

10.4 Conclusion

In addition to Secure Bit, this thesis provides insight for step-by-step study of buffer overflow, buffer-overflow attacks, and buffer-overflow protection. While we hope to see the success of Secure Bit as an architecture solution against buffer overflow, we expect this research to be a useful resource for studying buffer-overflow attacks.

APPENDICES

Appendix A: Secure Bit 1: the Origin

We propose a new, minimalist, architectural approach, Secure Bit, to protect against buffer overflow and function-pointer attacks. Secure Bit is completely transparent to software, and has no (little) run-time performance penalty. The goal of Secure Bit is to provide hardware support to protect against current and future generations of buffer-overflow attacks by protecting the integrity of addresses.

Fundamentally, we add one bit to every memory word to protect the integrity of addresses by adding semantic meaning to each word of the memory. This semantic meaning will be used to distinguish local data and data from another domain.

Protection Against Return Address Attacks

We begin with a somewhat simplistic description of Secure Bit to provide an overview. We add a Secure Bit to every memory location to mark a memory location as a return address. The bit is set by a call statement, and its validity is checked (and cleared) by a return statement. If only the call statement can set the bit and any write clears the bit, the integrity of the return address will be preserved with respect to buffer overflow attacks.

The one-time cost of Secure Bit is comparable to that of a parity bit: having n words of memory, additional n bits are needed. From a memory point of view, this Secure Bit is another data bit. From a processor point of view, it is only controlled by selected instructions. The Secure Bit need not be kept on disk, but the memory bus will have to

handle it. The details of this cost are critical to the success of Secure Bit, and our investigation of an implementation at the processor level plans to address this point.

In contract to other approaches that change the system memory management scheme or inject prologue and epilogue around vulnerable operations, we protect against the buffer overflow attacks by marking (locking) return addresses. The semantics of call and return instructions are modified to manage the Secure Bit. Modifying memory using other instructions will always clear the Secure Bit.

Normally, the call instruction writes a return address on the stack. The return instruction reads the return addresses from the stack and changes the instruction pointer to that particular address. In our design, these semantics are modified to include Secure Bit management. When call writes the return address, it will also set the Secure Bit of that memory location. On return, the return instruction will check if the Secure Bit is set before changing the program counter. If the Secure Bit is not set, the processor will raise a protection failure signal. The summary of the semantics is shown in Figure 31.

At the execution of the call instruction, the return address and Secure Bit are stored to the memory (stack) at the same time. If malicious code tries to overwrite the return address on the stack, the ordinary write instruction will clear the Secure Bit. On executing the return instruction, the processor first checks the Secure Bit. Since any modification to memory from other operations always clears the Secure Bit, the malicious user is

effectively not permitted to change the return address. Figure 32 shows stack snapshots of each step of operation.

Call Instruction	Return Instruction
Save current PC to stack *Set the Secure Bit for that address of stack Increase Stack Pointer.	Read the return address from top of stack *Clear the Secure Bit. *If the Secure Bit of entry on top of stack is not marked, Issue General Protection fault signal/interrupt. Decrease Stack Pointer Jump to return address

* - Semantic added to support secure memory architecture.

Figure 31 Semantic of call and return function

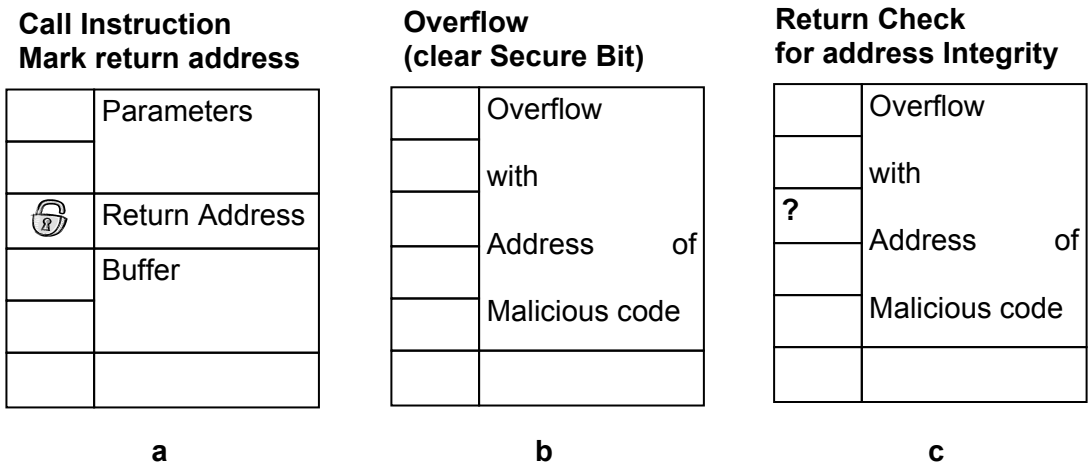

Figure 32 Stack snapshot with Secure Bit (a) After call instruction, (b) After buffer overrun, and (c) During return instruction.

Protection Against Function Pointer Attacks

In addition to return address protection, we can also apply the Secure Bit to the protection of function pointers. In fact, we believe that in protecting function pointers, we can protect all addresses that target valid entry points of a program (the target addresses of jump instructions)—a more general approach than simply protecting function pointers. To provide this protection, jump instructions are modified to check for a valid entry point before changing the program counter to the target address (similar to our "return"-instruction semantics). If the address is not a valid entry point, a general protection fault signal is raised. Management of the Secure Bit in this situation is more complicated, but the loader provides an excellent place for management since it is aware of all entry points. We introduce a special write instruction that writes (the address) and sets the bit. Since this instruction is only designed to serve the loader program, it can be placed in supervisor mode. Figure 33 shows a memory snapshot of loader and call/jump instruction semantics.

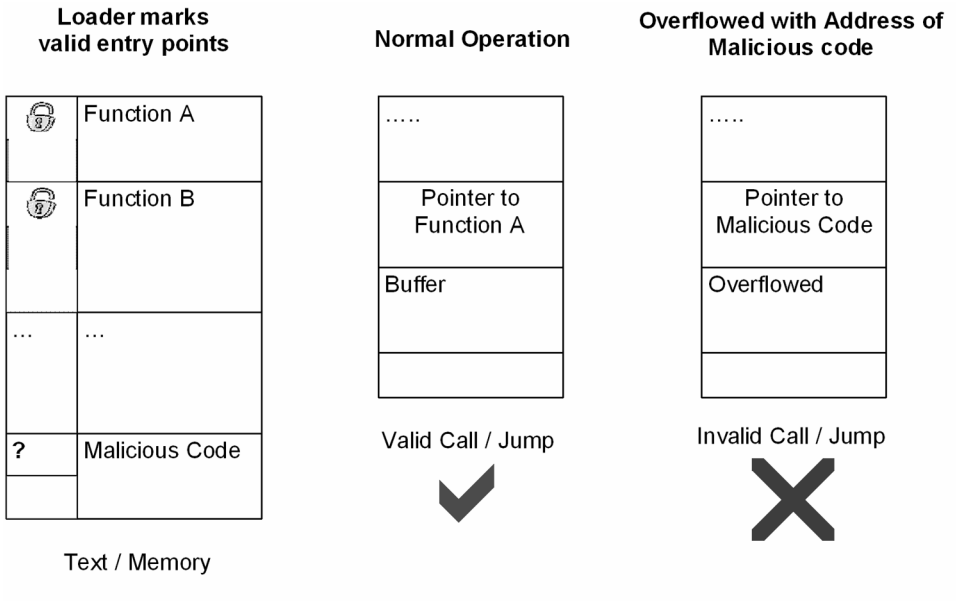

Figure 33 Function pointer protection using Secure Bit.

In order to target malicious code to be executed by modifying an address, the malicious user also has to set the Secure Bit. As a privileged instruction the write is not available to the user, and since this privileged write instruction does not exist in any part of the program/library for normal operations other overflow attacks to fool this instruction will not be available. As a result, it is impossible to inject the malicious code to the data area and set it as a valid entry point at the same time within the normal overflow process. With this approach the obvious next point of attack is the loader. The loader can be made as secure as the kernel, that is, if an attacker could modify the loader, they could modify the kernel which would obviate attacking the loader. Alternatively, one could use a certificate to secure the loader. With this method, the run-time system could check the validity of the loader before loading programs.

Appendix B: Non-LIFO control flow

Non-LIFO control flow can pose a problem for all buffer-overflow protection schemes since it skips return addresses on the stack. If software always uses LIFO control flow, hardware solutions such as SRAS [6, 40] would provide a substantial amount of protection without recompilation of legacy code. However, non-LIFO procedure control flow exists. Handling of non-LIFO control flow is necessary and nontrivial. Our study in chapter 2 found that several techniques cannot handle this issue. In several cases, a tool has to be explicitly turned off in order to execute the portions of code with non-LIFO control. Several tools ignore discussing the issue and leave it unresolved. With no absolute answer, our analysis indicates that they cannot handle non-LIFO control flow.

Non-LIFO control appears as an optimization of some compilers or an implementation of specific languages such as exception handling in C++. Figure 3.4 is an example of IA-32 code optimization (for size) where "near call" and "far call" share the same return (RETF). Basically, the "far call" is a call between segments where both code segment (CS) and instruction pointer (IP) are pushed onto stack. In order to share the same return (RETF), the entry point of "near call" is modified to construct an appropriate stack frame.

```
NEAR_ENTRY:
            POP   AX    ; POP instruction pointer (IP)
                        ; from the top of stack into
                        ; accumulator (AX)
            PUSH  CS    ; PUSH CS
            PUSH  AX    ; PUSH IP back onto stack
FAR_ENTRY:

            RETF        ; POP IP and CS off stack
```

Figure 34 Sample IA-32 optimization of near call and far call for size

The critical aspect that breaks buffer-overflow protection schemes, and the integrity of Secure Bit in particular, is that the construction of the return address is not a product of call instruction. In the optimization example above, the return address constructed from the "near entry" results from the pushing of "code segment" and "instruction pointer" registers. If the properties of the Secure Bit are added to the push of these two registers, the problem can be solved. In addition to the call instruction, we modify the "PUSH CS" instruction to mark the Secure Bit. For the IP, the actual IP is created by the call instruction. However, it is manipulated through the accumulator in the non-LIFO optimization. As a result, a bit is added to the accumulator for handling the Secure Bit. The "POP AX" instruction is slightly modified to save the value of the Secure Bit to an additional bit in the register. Similarly, the "PUSH AX" instruction is modified to store the value of the Secure Bit back onto the stack. Caution is needed ensure that the mechanism will not be misused to render a new type of buffer overflow attack.

Additional examples of non-LIFO control appear in Objective C, in the use of trampolines, and with the Longjmp instruction. In all cases, if the integrity of addresses are preserved, the order of control will not matter. Observe how Secure Bit handles non-LIFO control. If only a very limited set of instructions can set a Secure Bit, control flow may be executed in any order (including non-LIFO), but the return addresses will always be a valid return address.

Appendix C: TCPA, Intel LaGrande, and Microsoft NGSCB

Rather than providing a solution against the attacks, many vendors provide an add-on hardware module to enhance security. Usually, it is a hardware-accelerated-encryption unit. The main idea is to provide a mechanism for trusting the computer or a piece of software by using encryption (PKI), saving the vulnerable data (key) in the hardware. In addition to TCPA, a strong partition of processes is embedded into the hardware. The additional mechanism can be viewed as an implementation of sandboxing in hardware. With an appropriate software-driven policy, sandboxing can confine the damage from buffer overflow attacks. This section first describes the details of TCPA as background. Given Intel's LaGrande (the hardware technology building on top of TCPA), Microsoft's NGSCB (the software technology built on top of the hardware) is introduced. The section should provide sufficient background to understand that these technologies are not enough for buffer overflow protection.

Recently, a group of vendors formed the Trusted Computing Platform Alliance (TCPA [43, 70]), an organization working on creating a standard for a "trusted PC." The group announced the Trusted Platform Module (TPM) a specification of hardware with encryption power and secures registers to save certain secure data. On top of that, Trusted Computing Group, the successor organization to TCPA, has released the TCG API, a software API that interoperates with TPM hardware. The sample application of TPM is checking if the system is in a safe stage before installing or processing a transaction. Currently, there are several ongoing projects that use TPM to protect more dynamic

system components. Examples here are Intel's LaGrande Technology and Microsoft NGSCB [45] Technology.

Intel's LaGrande Technology (LT) is a set of enhanced hardware components designed to help protect sensitive information from software-based attacks. An overview of LaGrande Technology [36] unveils a number of key capabilities that claims to deliver the protections to the IA-32 platform. The capabilities include: Protected Execution, Sealed storage, Protected Input, Protected graphics, Attestation, and Protected Launch. To achieve these capabilities, several extensions are placed on the processor, chipset, keyboard, mouse, graphics device, and TPM device. All together, they provide a domain manager, the mechanism used to manage domain separation, as an intermediate layer between operating system and hardware (Figure 35).

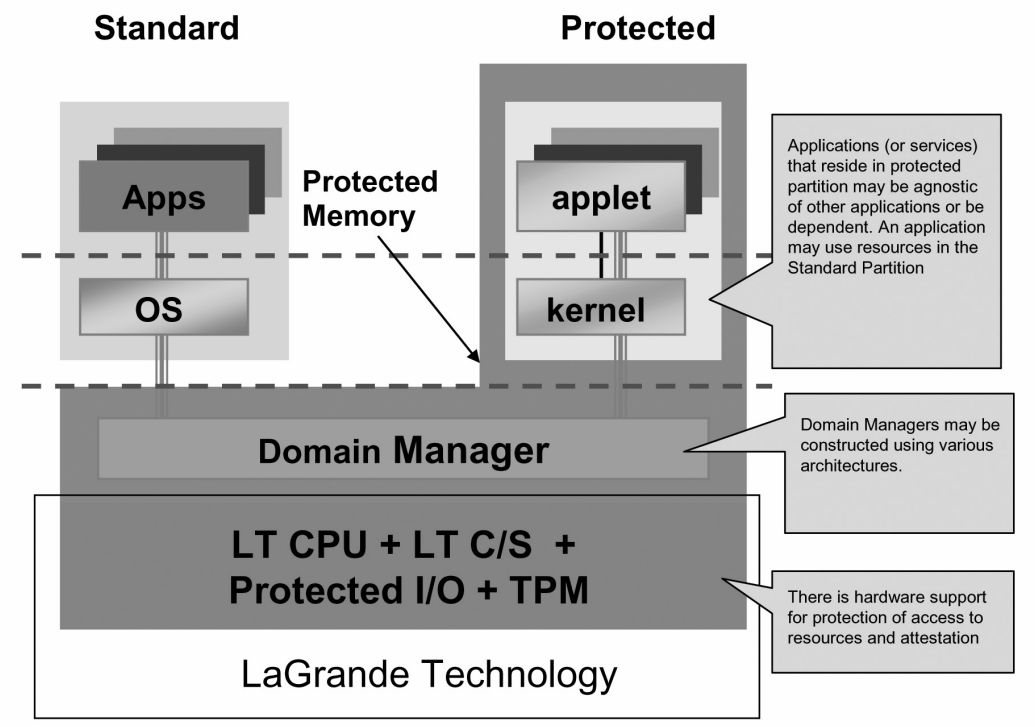

Figure 35 Intel LaGrande Architecture (from [36])

In this architecture, applications can be written to make use of both partitions, standard and protected. Much of the application code may still reside in the standard partition, and services written to handle sensitive information, would move to a module in the protected domain. The document also tells the fact that LT's domain separation is a memory paging protocol that protects the data from viewing or modification by unauthorized applications. Hardware mechanisms are used to prevent the data from direct memory access (DMA), and device probes. However, the applications in the protected domain may use resources in the standard partition. There is also similar technology named TrustZone by ARM [49]

Microsoft's Next-Generation Secure Computing Base (NGSCB) technologies [48] are designed to help provide better system integrity, information security, and privacy, by offering a foundation to help ensure that privacy- and security-sensitive hardware and software which can interact with greater integrity. The main features of NGSCB are: Strong Process Isolation, Sealed Storage, Secure Path to and from User, and Attestation. In this architecture, new hardware and software components are introduced. The nexus is a new component acting as the kernel of the protected software stack. TPM 1.2 is expected to serve as the Security Support Component (SCC), a hardware module that can perform certain cryptographic operations and securely store cryptographic keys that are used by the nexus. Process Isolation can limit the damage caused by buffer overflow. In general, a NGSCB-enabled computer requires a new hardware component similar to that provided by the Intel LaGrande Technology.

An issue of protection against buffer overflow is raised here. Buffer overflow happens in poorly written code. Passing information from one function to another function (regardless of domain) can in principle result in buffer overflow. LT's Domain manager cannot protect against buffer overflow attacks, and noticeably does not claim to. It may be able to enforce bound checking when information is passed from one domain to another, but still the mechanism cannot enforce information passing between functions in particular domains. LaGrande Technology is based on encryption, but encryption is not a solution to prevent buffer overflow. For example, SSL and SSH, which are encrypted communication protocols, are still exploitable by buffer overflow attacks. LaGrande Technology is another ring (-1) with hardware support for encryption and a secure interface to devices. It only protects applications from wire-trapping and extracting data from the devices (including memory and storage). While elaborate details have not been made public, domain manager might be implemented as a segmentation mechanism on top of the paging mechanism in order to maintain backward compatibility. Segmentation can be used to limit (or protect against) buffer overflow if a subroutine is forced to create a new segment every time it is called (similar to the mechanism found in I432 as mentioned earlier).

Though Intel's LaGrande Technology and Microsoft's NGSCB cannot protect against buffer overflow, they are perfect tools to enforce digital copyright management and to protect security sensitive data. Our analysis shows that the technology is useful only to limit the damage caused by buffer overflow. The protection provided by the Secure Bit can significantly enhance both technologies.

BIBILOGRAPHY

1. ARM. 2004. TrustZone Technology.

2. Baratloo, A., Singh, N., and Tsai, T. 2000. Transparent Run-Time Defense against Stack Smashing Attacks. In *Proc. of the USENIX Annual Technical Conf.* (2000)

3. Bhatkar, S., Duvarney, D. C., and Sekar, R. 2003. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proc. of the 12th USENIX Security Symposium*.

4. Bishop, M. 2002. Computer Security, Addison-Wesley, (Dec. 2002)

5. Blexim. 2002. Basic Integer Overflow. http://www.phrack.org/phrack/60/p60-0x0a.txt

6. Compuware DevPartner, DevPartner for Visual C++ BoundsChecker Suite. Available: *http://www.compuware.com/products/devpartner/bounds.htm*

7. Bulba and Kil3e. 2002. Bypassing stackguard and stackshield. *Phrack Magazine*, 10(56)

8. Cannon, J. C. 2004. Privacy: What Developers and IT Professionals Should Know, Addison Wesley Professional

9. Chang, F. Itzkovitz, A. and Karamcheti, V. 2000. User-level Resource-constrained Sandboxing. *USENIX Windows System Symposium*

10. Chen, S., Xu, J., Nakka, N., Kalbarczyk, Z., Iyer, R. K. 2005. Defeating Memory Corruption Attacks via Pointer Taintedness Detection, in Proc. Of *IEEE International Conf. on Dependable Systems and Networks (DSN)*, Yokohama, Japan, June 28 - July 1, 2005

11. Chien, E. and Szor, P. 2002. Blended Attacks Exploits, Vulnerabilities and Buffer-Overflow Techniques in Computer Viruses. In *Proc. of Virus Bulletin Conf*.

12. Chiueh, T., Hsu, F. 2001. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *Intl. Conf. on Distributed Computing Systems*.

13. Colwell, R. P., ET AL. 1985. Instruction Sets and Beyond: Computers, Complexity and Controversy. *IEEE Computer*.

14. Corliss, M. L., Lewis, E. C., and Roth, A. 2005. Using DISE to Protect Return Addresses from Attack. *ACM SIGARCH*, Vol 33. No. 1

15. Cowan, C., Beattie, S., Day, R. F., PU, C., Wagle, P., and Walthinsen, E. 1999. Protecting Systems from Stack Smashing Attacks with StackGuard. *the Linux Expo*, Raleigh, NC

16. Cowan, C., Beattie, S., Johansen J., and Wagle, P. 2003. PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proc. of the 12th USENIX Security Symposium*.

17. Cowan, C., Pu,C., Maier, D., Hinton, H., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *the proc. of the 7th USENIX Security Symposium*

18. Cowan, C., Wagle, P., Pu, C., Beattie, S., and Walpole J. 2000. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. *DARPA Information Survivability Conf. and Expo (DISCEX*).

19. Crandall, J.R. and Chong. F.T. 2004. Minos: Control Data Attack Prevention Orthogonal to Memory Model. *Intl. Sym. on Microarchitecture*.

20. Crandall, J.R. and Chong. F.T. 2005. A Security Assessment of the Minos Architecture. *ACM SIGARCH*, Vol 33. No. 1

21. Dahlby, S.H. Henry, G.G. Reynolds, D.N. and Taylor, P.T. 1982. Chapter 32. The IBM System/38: A High-Level Machine. Computer Structures: Principles and Examples.

22. Dean, D., Felten, E. W., and Wallach, D. S. 1996. Java Security: From HotJava to Netscape and Beyond. In *Proc. of the IEEE Symposium on Security and Privacy*, Oakland, CA

23. Etoh, J. 2000. GCC extension for protecting applications from stack-smashing attacks. Available: http://www.trl.ibm.com/projects/security/ssp/

24. Evans, D. and Larochelle, D. 2002. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*

25. Davis, N. 2001. Clean Up Your Code with Flawfinder. published on Linux DevCenter. Available: *http://www.linuxdevcenter.com/pub/a/linux/2001/05/29/insecurities.html*

26. Fletcher, M., Piromsopa, K., and Enbody R. 2005. Simulating Hardware-level Buffer-overflow Protection. Technical Reports #MSU-CSE-05-9, Department of Computer Science and Engineering, Michigan State University (2005)

27. Frantzen, M. and Shuey. M. 2000. StackGhost: Hardware facilitated stack protection. In *Proc. of the 10th USENIX Security Symposium*

28. Genhringer, E. F. and Keedy, J. L. 1985 Tagged architecture: how compelling are its advantages?. *Intl. symposium on Computer architecture*, pp. 162-170

29. Glew, A. 2003. "Segments, Capabilities, and Buffer Overrun Attacks," Computer Architecture NEWS, ACM SIG Computer Architecture Vol.31, No.4 - September 2003, pp. 26 – 31

30. Haugh, E. and Bishop, M. 2003. Testing C Programs for Buffer Overflow Vulnerabilities. In *Proc. of the 2003 Symposium on Networked and Distributed System Security (SNDSS 2003)* (Feb. 2003)

31. Hinton, H., Cowan, C., Delcambre, L., and Bowerds, S. 1999. SAM: Security Adaptation Manager In *Proc. of the Annual Security Applications Conference (ACSAC)*.

32. Howard, M. and Leblanc, D. 1965. Chapter 10:All Input Is Evil!. Writing Secure Code, Microsoft Press, 2nd ed.(1965)

33. Hsiangren, S. 2002. Apache/mod_ssl (slapper) Worm. GIAC Certified Incident Handler.

34. Ingo. 2004. Exec Shield, new Linux security feature.

35. Inoue, K. 2005. Energy-Security Tradeoff in a Secure Cache Architecture against Buffer Overflow Attacks. *ACM SIGARCH*, Vol 33. No. 1

36. Intel Corporation. 2003. LaGrande Technology Architectural Overview.

37. Jones, R. W. M. and Kelly, P.H.J. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In The 3rd Intl. Workshop on Automated Debugging.

38. Kc, G. S., Keromytis, A. D. and Prevelakis, V. 2003. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proc. of the 10th ACM Conf. on Comp. and Comm. Security*

39. Kgil, T., Falk, L., and Mudge, T. 2005 ChipLock: Support for Secure Microarchitectures. *ACM SIGARCH*, Vol 33. No. 1

40. Kirovski, D. Drinic, M. and Potkonjak, M. 2002. Enabling Trusted Software Integrity. ACM Intl. Conf. on Architectural Support for Programming Languages and Operating Systems

41. Krazit, T. 2004. PCWorld - News - AMD Chips Guard Against Trojan Horses. IDG News Service.

42. Litchfield, D. 2003. Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server. NGSSoftware

43. Macdonald, R., Smith, S. W., Marchesini, J. and Wild, O. 2003. Bear: An Open-Source Virtual Secure Coprocessor based on TCPA. *Tech. Report TR2003-471*, Department of Computer Science, Dartmouth College.

44. Mcgregor, J. P., Karig, D. K., Shi, Z., and Lee, R. B. 2003. A Processor Architecture Defense against Buffer Overflow Attacks. In *Proc. of the IEEE Intl. Conf. on Information Tech.: Research and Education (ITRE 2003)*, 243-250.

45. Microsoft Corporation. 2004. The Next-Generation Secure Computing Base: An Overview.

46. Milenkovie, M., Milenkovic, A., and Jovanov, E. 2005. Using Instruction Block Signatures to Counter Code Injection Attacks. *ACM SIGARCH*, Vol 33. No. 1

47. Moon, D. A. 1987. Symbolics architecture. Computer archive Volume 20, Issue 1 (January 1987) IEEE Computer Society Press Los Alamitos, CA, USA, 43 – 52.

48. Necula, G. C. Mcpeak, S. and Weimer, W. 2002. CCured: Type-Safe Retrofitting to Legacy Code. In *The Proc. of the Principles of Programming Languages*

49. Newsham, T. 1997. BugTraq Archive: Re: StackGuard: Automatic Protection From Stack-smashing Attacks

50. Newsome, J., and Song, D. 2005. Dynamic Taint Analysis: Automatic Detection and Generation of Software Exploit Attacks. In *NDSS* (Feb, 2005)

51. One, A. 1996. Smashing stack for fun and benefit, *Phrack Magazine, 49(7)*

52. Organick, E. 1983. A programmer's View of the Intel 432 System, McGraw-Hill

53. Ozdoganoglu, H., Vijaykumar, T.N., Brodley, C.E., Jalote, A. and Kuperman, B. A. 2003. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. Tech Report (TR-ECE 03-13), Department of Electrical and Computer Engineering, Purdue University.

54. PAX TEAM. 2003. Documentation for the PaX project.

55. Peterson, D. S. Bishop, M. and Pandey, R. 2002. Flexible Containment Mechanism for Executing Untrusted Code. In *Proc. of the 11th USENIX UNIX Security Symposium*

56. Pincus, J. and Baker, B. 2004. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns, IEEE Security & Privacy, Vol. 2, No. 4, July/August 2004, pp. 20 - 27

57. Piromsopa, K. and Enbody, R. 2004. Buffer Overflow: Fundamental. Technical Reports #MSU-SE-04-47, Department of Computer Science and Engineering, Michigan State University

58. Piromsopa, K. and Enbody, R. 2005. Secure Bit2 : Transparent, Hardware Buffer-Overflow Protection. Technical Reports #MSU-CSE-05-9, Department of Computer Science and Engineering, Michigan State University (2005)

59. Prasad, M. and Chiueh, T. 2003. A Binary Rewriting Defense against Stack based Buffer Overflow Attacks. In *Usenix Annual Technical Conference*, General Track

60. Rational PurifyPlus. *IBM Rational Software*

61. RATS, Available: http://www.securesw.com/rats/

62. Schmidt, C., and Darby, T. The What, Why, and How of the 1988 Internet Worm, Available: http://www.snowplow.org/tom/worm/worm.html

63. Shankar, U. Talway, K. Foster, J.S. and Wagner, D. 2001. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proc. of the 10th USENIX Security Symposium*

64. Shao, Z., Zhuge, Q., He, Y., Sha, E. H.-M. 2004. Defending Embedded Systems Against Buffer Overflow via Hardware/Software. In *Proc. of the 20th Annual Computer Security Applications Conference*, Tucson, Arizona (Dec. 6-10, 2004)

65. Shapiro, J. S. 1997. EROS: A Capability System, Department of Computer and Information Science Technical Report MS-CIS-97-04, University of Pennsylvania

66. Solar Designer. 2002. Linux kernel patch from the Openwall Project (Non-Executable User Stack). Available: *http://www.openwall.com/*

67. Suh, G., Lee, J., and Devadas, S. 2004. Secure program execution via dynamic information flow tracking. In *ASPLOS XI* (Oct, 2004.)

68. Sun Alert Notification. 2004. Document ID 57643: Netscape NSS Library Vulnerability Affects Sun Java Enterprise System.

69. Swiderski, F and Snyder, W. 2004. Threat Modeling, Microsoft Press (2004)

70. Trusted Computing Platform Alliance. 2004. TCPA IT White paper.

71. Tuck, N., Calder, B. and Varghese, G. 2004. Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow. In *Proc. of the 37th Intl. Symposium on Microarchitecture*

72. U.S. Department of Energy Computer incident Advisory Capability. 2004. O-130: Perl and ActivePerl win32_stat Buffer Overflow, Available: *http://www.ciac.org/ciac/bulletins/o-130.shtml*

73. Vendicator. 2000. Stack Shield technical info file v0.7.

74. Viega, J., Bloch, J.T., Kohno, Y, and Mcgraw, G. 2000. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *Proc. of the 16th Annual Computer Security Applications Conference*.

75. Wagner, D. Foster, J. S. Brewer, E. A. and Aiken, A. 2001. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proc. of the 10th USENIX Security Symposium*.

76. Webopedia Computer Dictionary. What is buffer overflow?, http://www.webopedia.com/TERM/B/buffer_overflow.html

77. Witchel, E., Cates, J. and Asanovic, K. 2002. Mondrian memory protection. In *ASPLOS-X*, Oct 2002.

78. Xu, J., Kalbarczyk, Z., Patel, S., and Iyer, R. K. 2002. Architecture Support for Defending Against Buffer Overflow Attacks. In *Workshop on Evaluating and Architecting Systems for Dependability.*

79. Ye, D., Kaeli, D. 2005. A Reliable Return Address Stack: Microarchitectural Features to Defeat Stack Smashing. *ACM SIGARCH*, Vol 33. No. 1

80. Young W. D. 1987. Coding for a Believable Specification to Implementation Mapping. *IEEE Symposium on Security and Privacy* 1987: pp. 140-149.

81. Wikipedia, the free encyclopedia, "Buffer Overflow", Available: http://en.wikipedia.org/wiki/Buffer_overflow

82. BOCHS, The Open Source IA-32 Emulation Project, Available: http://bochs.sourceforge.net/

83. Austin, T., Larson, E., and Ernst, D. 2002. SimpleScalar: an infrastructure for computer system modeling. *IEEE Computer, vol. 35, no.2, Feb. 2002.*

84. RedHat Linux, Available: http://www.redhat.com/

85. Parmelan. E. G. 2001. Porting Kaffe to a new platform. Available: http://www.kaffe.org/doc/port-kaffe/port-kaffe-0.2.html

86. SPEC CPU2000. Available: http://www.spec.org/cpu/