Memory Access Validation Scheme

Against

Payload Injection Attacks

BY

DONGKYUN AHN
B.S.,Yonsei University, Republic of Korea, 1998
M.S., Yonsei University, Republic of Korea, 2000

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2013

Chicago, Illinois

Defense Committee:

Zhichun Zhu, Chair And Advisor
Ashfaq Khokhar
Shantanu Dutt
Wenjing Rao
Venkat Venkatakrishnan, Computer Science

This thesis is dedicated to my parents Kyunghong Ahn and Kyungja Kim,

without whom it would never been completed.

ACKNOWLEDGMENTS

First and foremost, I am truly grateful to my former advisor, Dr. Gyungho Lee, for privilege of studying under his advisory. I deeply appreciate his support, dedication, patience, and unconditional support during my early period in UIC. His insightful vision on computer architecture made me a prepared computer engineer.

I would like to thank to Dr. Zhichun Zhu, for accepting me as her Ph.D student and guiding me to finish my degree. As an advisor and a professor, she gave me invaluable help during my staying in UIC.

I would also like to thank my thesis committee - Dr. Ashfaq Khokhar, Dr. Shantanu Dutt, Dr. Wenjing Rao, and Dr. V.N. Vekatakrishnan - for their resolute support and assistance. Their guidance in all areas helped me accomplish my research goals.

Lastly, I am deeply grateful to my parents. My parents made sacrifices to ensure that I get the best education and quality of living in America. Their unconditional love and support was one of the main reasons in the success of this work. I would never be able to express my gratitude to my parents in words.

DKA

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Continued)

SUMMARY

As more individuals and organizations become more dependent on computers and the Internet to create, manage, and share their resources efficiently, computer security issues are of increasing significance in every corner of our life. Due to insecure programming environments and structural limitations at the hardware level, many vulnerabilities are still being discovered. This thesis explores various issues concerning vulnerabilities and protection measures, including attack vectors and how protection measures are designed to address security threats.

This thesis addresses threats from payload injection attacks at the architectural level by leveraging existing hardware techniques. The first work utilizes the address translation for virtual memory system. With a TLB (Translation Lookaside Buffer) that is usually split between data (DTLB) and instructions (ITLB) as found in virtual memory system of modern processors, a simple protection can be developed based on an observation that activating an injected code causes a DTLB hit under ITLB miss with dirty bit set in the hit TLB entry. To evaluate our idea, we have revised the address translation function in Bochs x86 simulator and conducted code injection attacks available over the Internet. The experimental results with two simulators show that the proposed protection can detect all the code injection attacks tested.

The second work pursues more fine-grained protection against sophisticated attacks like return-oriented-programming attacks by leveraging existing hardware techniques. Two widely adopted hardware techniques – the cache structure and the branch predictor – increase performance in modern microprocessors by exploiting expected or predicted circumstances and

events. As malicious payloads are prone to induce unprecedented or unexpected circumstances at control flow redirection, validating those circumstances or events at the associated handlers could be utilized in countering payload injection attacks.

In order to utilize these components for protection, this thesis clarifies practical issues in distinguishing legitimate miss events from those caused by malicious attacks and integrating supporting mechanisms into multi-tasking environments. Based on the observation and discussion, we propose a memory-access validation scheme against payload injection attacks. This scheme consists of two parts – the validation unit and taint-status data. The validation unit handles queries from other processor components, namely the cache structure and the branch predictor, and validates suspicious control flow redirections by referring to the active taint-status data set. Experimental results with two simulators show that the proposed validation scheme is able to detect simulated payload injection attacks under negligible to moderate performance degradation.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Since their debut, computers and the Internet have become an indispensable part of the infrastructure of modern society. Most organizations including government agencies, corporations and educational institutions depend on computers and the Internet more and more in order to create, manage, and share their resources effectively. Computer security is crucial in modern society because any compromise of integrity or reliability can result in a devastating outcome, and the damage can be extremely costly – especially in a networked environment. For example, the infamous *Conficker* worm, which targets the Microsoft Windows operating system, disables several security-related services in infected systems and creates congestion on a local network by flooding packets. This worm also utilizes the networking infrastructure for its own benefit – to propagate itself to other machines online or to upgrade itself. Due to its advanced circumvention technique, this worm is unusually difficult to detect and to eradicate, and the estimated economic cost from this worm was about $ 9.1 million in 2009 (1).

Security vulnerabilities in software allow adversaries to compromise and to take control of target systems for malicious activities like stealing classified information stored in an electronic format or coordinating distributed denial of service attacks with multiple infected systems to bring down another target system. Although most attack vectors rely on specific properties of

the target systems and often are not transferable from one target platform to another, attackers are able to infect a large number of vulnerable systems because these systems frequently share software platforms such as operating systems or libraries, or each of them provides the same address space for common resources.

Some attacks try to execute their own software module on the target remotely. In an uncompromised operating system environment, only binary executables installed by the operating system or the administrator are allowed to be loaded and executed with binary loaders. However, through software vulnerabilities like buffer overflows, malicious parties are able to inject and execute their own code without the operating system's authorization. This problem becomes even more threatening if the adversaries are able to acquire the supervisor privilege of the operating system through brute-force or dictionary-based attacks. As the supervisor privilege gives unlimited access to a system's resources, an adversary with this privilege can stealthily install or inject their code and make the processor execute it without being detected by anti-malware software.

According to the National Vulnerability Database, vulnerabilities related to code injection attacks accounted for 16 percent of the total vulnerability reports in 2009. Although those vulnerabilities can be eliminated or mitigated with various detection approaches and/or patches at the software level, the overall number of vulnerabilities reported is increasing. This is because a large number of source codes are still written in weakly typed languages like C/C++, and binary executables generated from those languages are deployed every day.

1.2 Protection and related features

Protection against code injection attacks can be categorized as two types – static analysis during compile time or runtime checks and prevention. Although proposed detection and protection based on these approaches are helpful in thwarting some code injection attacks, they cannot provide an ultimate solution for several reasons: (1) use of unsafe low-level programming languages for software components that require high-performance; (2) limitations in static and dynamic analysis; and (3) inherent assumptions made about modern computer systems.

In order to eliminate software vulnerabilities in programming, many high-level programming languages, like Java and C#, have adopted various safety features, which have eliminated numerous security problems. However, there are still many software packages written in unsafe low-level languages such as C/C++ or assembly languages, for several reasons: (1) compiler or library dependency on legacy software modules and libraries; (2) the high performance required for system-critical software modules such as an operating system kernel or device driver modules. Generally, writing safe and secure programs using such low-level languages is difficult and programmers are prone to introducing bugs and vulnerabilities, which are very difficult to detect during compile time due to the lack of runtime information.

For these unsafe language programming environments, static analysis and dynamic runtime checks are proposed, some of which are applied in commodity products. A modern static analyzer can find programming errors or vulnerabilities before generating a binary executable. However, their detection performance is limited due to the lack of runtime information such as target addresses for indirect branches, and they are prone to finding false positives, making

them hard to use in practice. Dynamic runtime checks can partially complement the limitations of the static analysis approach – however, sophisticated algorithms to detect anomalies usually incur significant performance overheads. In addition, both approaches require prerequisite work, mainly for dealing with source codes.

Each item of software in a computer system is composed and executed based on certain assumptions established with regard to the target platform's design and implementation. For example, the x86 architecture supports four levels of privilege in its operation mode – zero to three – where the lower number has more privileges. However, most operating systems utilize only two of them – zero, which is the most privileged, and three, and it is assumed that processes running with the privilege level zero are reliable and uncompromised. Unfortunately, adversaries are able to stealthily acquire the privileged running level by exploiting these assumptions, and malicious activities performed with the hijacked privilege are permitted as if they were legitimate operations.

1.3 Work focus

Based on the protection against payload injection attacks and related features, we define our research scope for these attacks and the protection against them as follows:

What we do NOT propose: This thesis does not propose any protection feature that prevents payload injection attacks exploiting vulnerabilities or lexical or semantic errors in high-level runtime environments such as JavaScript engines or SQL engines. Also, we do not aim to counter malicious activities that try to sneak out information through physical media such as covert channels or by wiretapping the system bus. We do not consider malicious attacks that

compromise the integrity of existing software modules such as the CMOS BIOS. Finally, we are not interested in countering socially organized attacks such as distributed denial-of-service (DDoS) attacks on the target machine.

What we do propose: This thesis proposes a protection scheme against payload injection attacks at the processor architecture level. In this regard, a payload is a foreign object that could be executed as a viable machine code after being implanted during runtime or that is able to redirect the control flow of a process that shares the same address space as the payload itself in an unexpected way.

We are interested in leveraging existing processor components and integrating a protection scheme into multi-tasking environments. For this research, we scrutinized the general memory architecture and hardware optimization in terms of handling unexpected circumstances and foreign objects. We also investigated the resource management of multi-tasking environments so as to support existing platforms – from the architecture level to the user application level. Based on this thorough investigation, we tried to propose a protection scheme that requires neither fundamental changes in the existing processor architecture nor considerable modifications to the software level – either at the OS kernel or the application level.

1.4 Dissertation overview

In order to propose a protection measure to counter payload injection attacks at the architectural level, this thesis discusses various issues regarding payload injection attacks – from vulnerabilities to countermeasures. We analyze payload injection attacks from architectural standpoints and clarify practical issues in leveraging architectural components and inherent

events so as to address limitations found in software-based protections. Based on the discussion and analysis, this work develops a protection scheme against payload injection attacks at the memory architecture level and evaluates reliability, effectiveness, and performance impact of the proposed scheme.

In Chapter 2, we explore various aspects of payload injection attacks – security vulnerabilities, various attack vectors exploiting those vulnerabilities, and countermeasures against them. We discuss various aspects of these issues – the architectural standpoint, programming environments, runtime environments, etc.

Chapter 3 analyzes payload injection attacks from the architectural standpoint – especially in terms of hardware-based acceleration techniques like cache structures and discusses practical issues and requirements in exploiting these techniques for protection.

Chapter 4 proposes a protection approach against shell code injection attacks. The proposed approach exploits a TLB structure and I/O operations to detect instruction fetch attempts toward arbitrarily modified memory pages. We evaluate effectiveness and performance impact with simulators.

In Chapter 5, we propose our memory-access validation scheme based on the discussions presented in Chapter 3. For this scheme, we develop a data structure to store taint-status data and design a validation unit that answers queries from other hardware components by referring to taint-status data contained in main memory. We also propose integration approaches and a caching structure so as to alleviate the performance overhead. The experimental results show

that our validation scheme is able to counter simulated payload injection attacks with a modest performance impact.

Chapter 6 concludes this thesis. We summarize this work and discuss open issues related to the proposed scheme.

1.5 Contribution

This thesis makes the following key contributions in preventing payload injection attacks in multi-tasking environments at the processor architectural level:

1. **Investigation of payload-injection attacks.** This thesis explores various issues for payload-injection attacks – vulnerabilities, attack vectors, and countermeasures. We discuss these issues for simple types of payload-injection attack to advanced types targeting runtime environments like web browsers. This investigation analyzes payload-injection attacks not only in the software domain but also the architectural circumstances that allow malicious attacks.

2. **Analysis of payload-injection attacks from an architectural standpoint.** This thesis analyzes payload-injection attacks from an architectural standpoint. We find that the miss handlers of the instruction cache and the return-address stack (RAS) could be utilized to counter payload-injection attacks. A protection scheme exploiting these handlers has advantages in countering payload-injection attacks because of its transparency to the software level.

3. **Hardware-friendly data structure.** This work proposes a data structure for auxiliary data that contains information on unreliable foreign objects. This data structure can be

seamlessly integrated into existing multi-tasking environments and memory architectures and guarantees constant access time to its instances.

4. **Architectural integration approaches for memory-access validation.** In order to create a memory-access validation scheme against payload-injection attacks, we developed architectural integration approaches that exploit inherent events of the processor such as cache misses. We also propose a caching structure to mitigate performance degradation from accessing instances of the proposed data structure.

5. **Experimental validation.** In order to evaluate the effectiveness and performance impact of the proposed scheme, we built simulation environments based on two simulators. We ran extensive tests on both simulators – a shell code injection benchmark, the GCC compilation, and the SPEC2K benchmark.

CHAPTER 2

BACKGROUND

2.1 Payload injection attack

2.1.1 Payload injection attack overview

A payload injection attack is an attack strategy that exploits a vulnerability in a computer system to execute an arbitrary code maliciously crafted by an adversary.

In the modern computing environment, an application consists of procedures and data referred to and modified by the procedures. Each procedure is programmed to accomplish its tasks through various arithmetic operations, data movement, and other procedures. Some of the procedures in an application are loaded into the same address space when the application is prepared for execution, and others are linked to external binary(ies), called a "library," and are invoked on the fly. Regardless of whether a procedure is provided internally or externally, every procedure exchanges data with other procedures or hardware devices to accomplish what it is designed to do. During an exchange, arguments and return values are passed between procedures, and some of the exchanged values require a storage space larger than the size of a memory word – for example, 4 bytes. In order to handle this multiple-word memory requirement, a portion of consecutive memory space is allocated in the form of array and utilized for data processing/exchanging. Such memory spaces are referred to as a "buffer."

While procedures in an application process data and return the result to other procedures, buffer spaces can contain various types of data – such as integers, characters, floating-point numbers, and even machine code. Meanwhile, the control flow in each procedure is dynamically changed depending on various factors, like the outcomes of conditional branches or function pointer values. In buffer use and control flow, the processor hardware blindly executes instructions pointed to by the program counter under the following assumptions: (1) the instructions are organized in a way such that the buffer spaces are accessed only for their original purpose; and (2) the control flow is always bounded within a specified address range.

However, these two assumptions can be easily violated due to software bugs caused by human error or hardware limitations – especially if a software program was composed using a weakly typed programming language like C/C++. One example is when a buffer space allocated for strings contains a machine code because of (1) a programmer's naive assumption that only alphanumeric characters or unexecutable binary sequences will be entered as input; and (2) the lack of a type-restriction enforcement on memory at the hardware level. These circumstances allow intruders to place a binary sequence of arbitrary values into a buffer space unsupervised by the operating system or runtime environment.

An example of a control flow compromise is a corruption of the control flow data – especially for indirect branches referring to a register or a memory word. Unlike a direct branch instruction, which has only two target addresses – taken or not taken determined by the result of an arithmetic comparison, an indirect branch can redirect control flow to an arbitrary location with the target address saved in a register or main memory. As the target address is determined

at runtime, indirect branches provide flexibility so that programmers can redirect the control flow to different locations with one instruction. As indirect branches can divert the control flow to an arbitrary location at the machine code level, software developers try to carefully manage such memory words for these instructions while programming, and compilers may show errors or warning messages for their improper use during compile time. However, such indirect branch information can become corrupted at runtime through other programming errors such as an *out-of-bound* pointer arithmetic operation, and the control flow may be redirected to an unexpected location. For example, if an initialization procedure mistakenly overwrites a function pointer with zero, an illegal reference exception is triggered at its de-reference, as a zero value in pointer arithmetic means `NULL`.

In summary, payload injection attacks are accomplished through sequential intended violations in the execution of arbitrary machine codes as follows, and Steps 2 and 4 are related to the violations explained above.

1. An adversary crafts a malicious payload to be placed in a writable region of a victim program and figures out a vulnerability of the program, which allows the adversary to overwrite the control flow data with an arbitrary value.
2. Through a legitimate input operation such as a string input, the adversary places the payload in a writable memory region of the victim program.
3. During or after payload placement, the adversary tries to compromise vulnerable control flow data such as a return address in a stack.

4. When the corrupted control flow data is de-referenced for an indirect branch, control flow is diverted to the location chosen by the adversary.

While problematic programming practices can result in violating the assumptions and these violations provide opportunities for malicious intrusions, other features of memory systems in modern computing systems allow payload injection attacks as well.

Shared address space: In most modern computer systems, code and data referenced by the code share the same virtual address space. This property allows a payload injected as data by an adversary to be later accessed as an instruction fetch without any hardware intervention between the accesses. However, this access sequence is also followed by legitimate software loading/execution procedures in processors based on the von Neumann architecture. In order to secure the integrity of software programs running on a computer system and to prevent execution of arbitrary codes, modern operating systems are designed to execute only legitimately loaded codes and to protect these by managing various properties like the "read-only" attribute in virtual memory systems.

Flat memory model: The flat memory model refers to a memory-addressing paradigm in which the processor can linearly address all of the available memory locations without having to resort to any sort of memory segmentation. Although some architectures include a non-flat memory organization like the segmentation unit in the x86 processor, many operating systems use this non-flat memory model in a limited way – as a flat memory model. This is because other memory management mechanisms, like memory paging, can replace it effectively while simplifying the overall management. In addition, because of its hardware dependency, such a

non-flat model is not considered to be a viable management scheme especially since portability is important in modern operating systems.

The following section describes attack vectors used in payload injection attacks, and the countermeasures used to guard against them.

2.1.2 Stack overflows

The simplest form of a payload injection attack is to craft a malicious payload exploiting the buffer overflow vulnerability and to place the payload in a buffer allocated in the stack region.

The stack region consists of a set of stack frames, each of which is allocated for one function call at runtime. In most microprocessors, one or two registers are associated with a stack frame in use; in the x86 architecture, `ESP` (i.e. the stack pointer register) points to the top of the stack, and `EBP` points to the bottom of the current stack frame. Whenever a subroutine is called with a `call` instruction, the processor hardware decreases the stack pointer register and stores a return address in the memory word pointed to by the register. Then the stack frame information for the caller routine is stored in the stack, and registers related to the stack are decreased to make a new stack frame whose size is determined by the invoked subroutine. Instructions in the subroutine refer to these registers as a base for register-indirect addressing. Upon finishing the subroutine, control flow goes back to the next instruction of the `call` instruction by reading the return address from the stack, and the stack frame information for the caller routine is restored to related registers.

The buffer overflow vulnerability is a software bug in which an input stream whose size is larger than that of the allocated buffer space corrupts memory words next to the buffer.

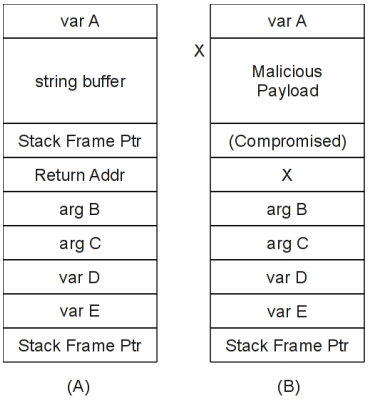


Figure 1: Changes in a stack frame during a stack overflow
(A) A vulnerable stack layout in which the return address (`Return Addr`) is located close to a string buffer. (B) The stack layout after a stack overflow. The string buffer is filled with a malicious payload and the return address is overwritten with the starting address of the string buffer – "`X`."

When programmers use a weakly typed language like C/C++ to write procedures utilizing buffers, they occasionally miss the boundary check for a data move operation. For example, the `strcpy` function copies a character string from one location to another until the "end-of-string" character, which is usually the `NULL` character, is encountered in the source string and it does not consider the size of the destination buffer. Therefore, if a programmer uses this function without a boundary check to copy a string from one place to another, the memory region following the destination buffer can be compromised, because this function does not consider the size of the destination. In summary, as described previously, the processor executes instructions blindly so that *out-of-size* data movement can corrupt memory words next to the buffer.

The stack region is the most frequent victim of attacks exploiting this vulnerability because without a boundary check, data written into the stack can overwrite the return address. By

supplying an excessive amount of input data to an application subject to this vulnerability, an adversary can compromise a memory word containing a return address so that, when the corrupted return address is de-referenced, the control flow is redirected to another location never anticipated by the programmer. Generally speaking, there are two types of payload injection attack that exploit stack buffers – the first type puts the machine code to be executed into the payload and the other type crafts the payload with machine code addresses and parameters. The latter attack vector was devised to circumvent countermeasures against the first type. This section discusses only the first type and the latter type is discussed in Section 2.1.6 in detail.

A maliciously intended payload for simple stack overflow attacks consists of (1) a machine code that organizes arguments in the stack or registers and invokes existing procedures in the victimized system and (2) a memory word containing the starting address of the code. The distance between the starting address of the machine code and the memory word for the return address can be calculated from the source code or by exploiting other vulnerabilities like the format string, which is explained in Section 2.1.4. By placing this payload in the stack of the vulnerable application, adversaries can execute their own code without the loading process being handled by the operating system. As this attack vector directly corrupts a memory word referenced for control flow, it is often referred to as a "single-stage" buffer overflow attack as illustrated in Figure 1.

While the single-stage buffer overflow attack overwrites a return address in the stack region by overflowing directly, the multi-stage attack compromises the return address indirectly through a pointer variable located close to an input buffer in the stack region. The attack code

is crafted to corrupt only a memory word used as a pointer variable next to the buffer while not touching other memory words between the pointer and the return address. By overwriting the pointer variable with the location of control flow data, the attacker can redirect the control flow to injected code when the corrupted memory word is de-referenced. In other words, the pointer is exploited as a "bridge" to compromise a target word through a buffer overflow. This attack vector is designed to circumvent countermeasures against the early form of buffer overflow attacks, which are based on a single-stage buffer overflow.

This attack form has been widely adopted in various in-the-wild attacks – such as L10n Worms, Slapper Worms, Sasser Worms (2), and Code Red (3), and Conficker (1). Although many forms of protection have been proposed, many applications are still exposed to the vulnerability exploited by this attack vector because of their programming environments – like weakly-typed language like C/C++ or assembly languages.

2.1.3 Compromising a function pointer table

Not only the stack region, but also other data regions can contain control flow data, and they can be exploited to execute an injected code as well. An example of such a region is one containing a function pointer table used to invoke procedures provided from libraries. Most operating systems provide various services to user-space applications – from simple ones like the current date/time to complicated ones like controlling network communication. Procedures for these services are stored in shared libraries so they can be reused, for memory space efficiency, and for the centralized management of shared resources. In order to support this feature dynamically, operating systems load and link these libraries upon receiving an invocation at

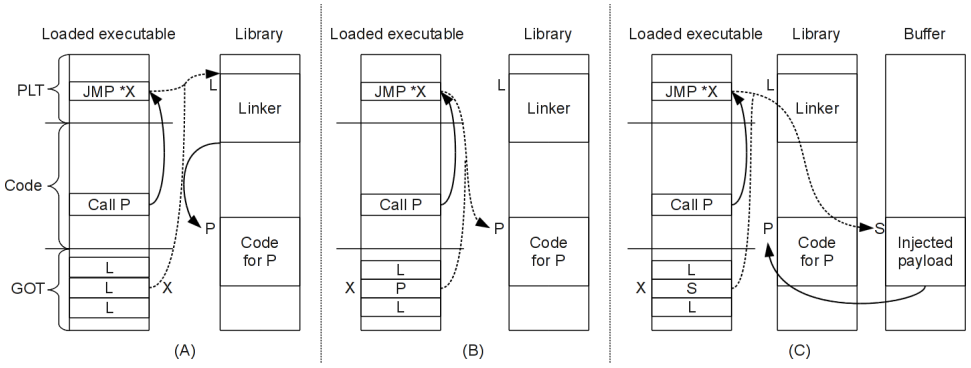


Figure 2: Dynamic linking in the Linux operating system
(A) Initial status of the memory layout when a binary executable is loaded. When control flow goes through the linker, the `GOT` for procedure `P` is updated with the runtime-decided location of its actual location – `P`. Therefore, the next time the same procedure is invoked, control flow goes to the procedure without going through the linker procedure, (B).

runtime, and this process is referred to as "dynamic-linking." This linking process requires a function pointer table mapping its elements to the actual addresses of the corresponding procedures. In the case of the Linux operating system, this table is called the "Global Offset Table," or the GOT. In addition to the GOT there is another table called the "Procedure Linkage Table," or the PLT, which contains a set of indirect function-call instructions referring to the GOT. While the PLT is stored in a code section that is read-only, the GOT is stored in a writable data section. A similar linking feature using indirect branches is employed in the Windows operating system family, and the corresponding table is referred to as the "Import Address Table" (IAT).

During the initial loading of a binary executable, the GOT is filled with the entry address of the dynamic linker, which is responsible for updating function pointers in the GOT – (A)

in Figure 2. When an element of the GOT is referenced by the loaded executable to invoke a procedure in a library, the dynamic linker is executed and it overwrites the element with the actual address of the requested procedure. When a previously requested procedure is invoked again, the procedure in the loaded library is executed without going through the dynamic linker – (B) in Figure 2. Although the locations of loaded libraries (`P` in Figure 2) are determined at runtime, a userland application can invoke functions with an indirect branch instruction that refers to the function pointer value in the GOT. It is obvious that this table should not be modified by any software modules other than the dynamic linker. However, with a sophisticated attack code, an attacker can locate the table and modify its pointer element to redirect the control flow to an injected code whenever the compromised element is de-referenced. This attack vector is adopted by code injection attacks targeting the IAT in the Windows operating system as well (4).

Not only the function pointer table for dynamic linking but also other tables containing control flow data can be exploited – for example, a virtual function table, also known as a `vtable`. This type of table is extensively utilized in object-oriented programming environments such as C++. In such an environment, various types of classes can be declared under an inheritance hierarchy. In a hierarchy, a derived class inherits methods from the base class. Some of these methods are "virtual," which means the actual procedure called for the method is determined at runtime, since it depends on the class of the method invoked. In the implementation, a function pointer table containing addresses of related procedures is associated with the instances of each class. Every object instance locates a function pointer in this table with a designated

pointer (the virtual table pointer) and the proper offset contained for each instance. Through this table-pointer-instance coordination, the same method indicated by the same pointer field can be linked to different procedures at runtime.

There are two vulnerable points in this coordination – one is the virtual table pointer, and the other is the virtual function table itself. An adversary can make a fake virtual function table and compromise a virtual table pointer to point to the table so that the control flow goes to an injected code. Corruption of the virtual function table is similar to an attack compromising a function pointer table used for dynamic linking.

In general, every function pointer that can be overwritten through any kind of vulnerability can be exploited by a payload injection attack. For example, if a buffer-overflow attack corrupts a function pointer next to a buffer, the control flow is redirected to another location when it is de-referenced. Moreover, even if the target location of an indirect jump cannot be restored precisely due to randomization in the address space, an adversary can abuse a vulnerable function pointer by overwriting it with a random value so as to execute the injected code. An adversary can craft and execute an attack script that repeats payload injections containing a random value to be referenced as function pointer variable until the random value hits the starting address of the injected payload. An example of this type of attack vector is described in Section 2.1.5.

To complete an attack based on this vulnerability, an attacker may employ a second attack vector to locate and overwrite the table. One example of such coordination is the use of the format string vulnerability described in the next section.

```
printf("%s : %d - %f - %x @ %p\n",
       pString,               /* %s */
       count,                 /* %d */
       fraction,              /* %f */
       bit_mask,              /* %x */
       address);              /* %p */
```

Figure 3: String formatting example using the printf function

2.1.4 Exploiting the format string vulnerability

While the stack-based buffer overflow relies on the stack layout of the victim application to execute the injected code, format string exploitation utilizes a bug in the programming syntax. In the C programming language, a special directive "%" is used to format a string in string output functions, such as `printf` and `sprintf`. However, this directive can be exploited to disclose the stack layout and its elements to an output terminal like telnet, and an adversary can craft a payload that redirects the control flow to an arbitrary location where a shell code is stored with the internal information on stack usage.

Figure 3 is an example of a string output function. The first argument is a string referenced for formatting with "%" directives, and the remaining arguments are the variables associated with each ("%"-character) pair in sequence. These arguments are stored in the stack prior to invocation so that the `printf` function can refer to them. The character following the directive indicates how the data is to be displayed. For example, "%s" is a format command meaning that the string pointed to by the corresponding pointer in the stack (in this example, `pString`)

is to be printed out until a `NULL` character is encountered. Unlike the "%s" format directive, other "%" codes are instructions to read the memory word(s) directly from the stack and to print out its value in the format indicated by the character following the "%" directive. The following list shows some examples of these formatting codes (Figure 3):

- `%d` – decimal number
- `%f` – floating point number
- `%x` – hexadecimal number in lower case
- `%p` – virtual address format, equivalent to "`0x%x`"

While the `printf` function is parsing the format string passed as the first argument, the function fetches one value from the stack and prints out the value according to the corresponding format directive whenever a "%" directive is encountered. This operation is based on the assumption that the stack frame in use contains all of the arguments for the format string – but does not count the number of arguments provided in the source code. In other words, even if some arguments after the format string are omitted, the source code can still be compiled[1] and executed. This means that if the number of directives and arguments programmed in the source code do not match, the output would not be as the programmer expected – moreover, an adversary can exploit this.

Assume that a programmer writes an output function such as the following:

---

[1]Recent compilers may show warning messages

```
printf(buffer); /* buggy */
```

At first glance, it seems that this line would work without any problem – just print out the string pointed to by the `buffer` argument. However, if the string contains more than one "%" directive, this output code would reveal the data stored in an arbitrary location as well as those in the stack region because, as described above, it is assumed that all of the arguments for formatting directives are properly contained in the stack frame in use. An *out-of-bound* reading like this will certainly result in information leakage.

Exploiting this information leakage is limited and another attack vector is required to redirect the control flow to where an adversary wants. To make things worse, the same `printf` function has a special directive that can write an arbitrary value into any location – "%n." This directive assumes that the corresponding argument is an integer pointer and writes the number of bytes formatted so far into the memory word associated with the pointer. One may ask how an arbitrary number can be written through this directive if the number is determined by the format string. This problem can be overcome with another directive – "%au," which controls the number of bytes formatted so far. With a well-crafted format string, an adversary can gain the write privilege to any writable memory region and redirect control flow to wherever he/she wants.

The infamous *wu-ftpd* attack is one example, which exploited this vulnerability. Instead of typing in a legitimate input argument for the vulnerable command (`SITE EXEC`), the attacker used a string like the following line to find out values contained in the stack:

```
> SITE EXEC %x %x %x %x
```

When this input string is parsed, four values including the address of the `ftpd` process, are printed out to the shell terminal. With these values, the attacker can inject their own payload into the victim's address space and execute it by corrupting control flow data – such as a return address.

Another example of this vulnerability is demonstrated by an application programmed in the Perl language. In (5), the author mentions that with a carefully crafted input string with special directives, an adversary can make the victim computer consume a large amount of memory, modify variables arbitrarily, or alter intended outputs. It is also stated that fixing the interpreter problem will not completely eliminate the impact of the format string vulnerability but only reduce it.

2.1.5 Heap-spraying

Since there are many forms of protection to prevent attacks that victimize the stack region, intruders have shifted their target to the heap region, in which objects are allocated and de-allocated at runtime. Injecting and executing a machine code in the heap is more difficult than exploiting the stack. This is because the addresses of objects in the heap are less inferable and predictable than those in the stack region, and the randomization of the address space reduces the likelihood of exploitation from attacks targeting the heap. In order to overcome these limitations, attackers have developed several attack methods, one of which is heap spraying (6). Heap spraying was used occasionally in early 2000. However, it has become widespread in attacking web browsers and executing arbitrary code since 2005 – especially victimizing Internet Explorer from Microsoft.

Unlike other code injection attacks, which place a malicious payload in a writable region and execute the payload by precisely locating it, the heap spraying attack spreads an object containing a malicious payload in the heap through a legitimate procedure invocation. After populating the heap region with clones of the object, the intruder corrupts a function pointer in the victim application with an arbitrary address. The attack vector which compromises a virtual function pointer is described in Section 2.1.3. If this arbitrary address happens to fall in the address ranges assigned to sprayed objects, the payload in the object is executed when the corrupted function pointer is de-referenced for control flow redirection.

It is quite obvious that the success of heap spraying relies on the probability of an address hit. In order to increase this probability, an adversary fills the object with ineffective instructions such as `NOP` and ends it with the malicious payload. Technically, any instruction can be included as an ineffective portion of this object as long as it does not affect the operation of the payload. A structure such as this is referred to as a `NOP` sled as the control flow slides into the payload when the program counter hits one of the `NOP`s. In addition to this structure, the attacker makes the overall spread size very large to increase the probability of a hit. For example, if the spread size is 256 MB, then there is only a 4-bit entropy in a 32-bit architecture ($= 32 - log_2(256 \times 2^{20})$) for attackers to predict the location of the cloned objects (7).

Heap spraying needs a victim application that allows an attacker to populate the heap region with arbitrary data without much effort. Of the many programming environments, scripting languages meet these criteria – they provide the programming interface in which users can assign objects dynamically in the heap region using a simple syntax. In particular, JavaScript

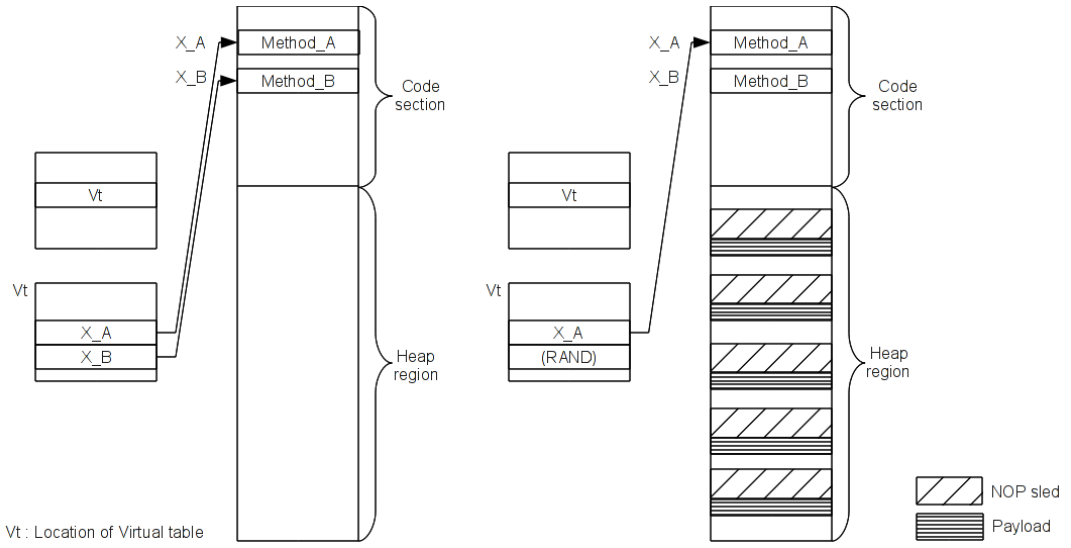


Figure 4: Heap spraying attack: Before and after
if the (RAND) value hits one of the `NOP` sleds (45 degree hatched regions), control flow passes to the payload (horizontally hatched regions).

can simplify the steps for generating a malicious object containing a payload and spraying it. It is widely known that the JavaScript engine is employed in many GUI-based web browsers, therefore they may be exposed to this attack vector. Moreover, it has been found that not only web browsers but also other commodity software with the same parsing engine can be vulnerable to heap spraying. One noticeable "high"-risk example is the vulnerability found in the Adobe PDF viewer with the JavaScript engine enabled (8).

2.1.6 Return-to-libc and return-oriented programming

A return-to-libc attack is a variation of the stack overflow attack, which executes a pre-existing procedure in a library without injecting a machine code into a writable region. It is designed to circumvent protection based on the non-executable property (Section 2.2.5). As

explained in Section 2.1.2, a stack region is used to allocate local variables and to pass arguments to subroutines. In normal calling conventions for subprocedures, arguments are saved in the current stack frame and the target subprocedure is invoked through a `call` instruction. A subroutine assumes that arguments are already contained in the stack frame and they can be referenced through register-indirect addressing. Based on this convention, it is possible to exploit any pre-existing procedure without injecting a payload to be executed if the adversary is able to: (1) identify the location of the procedure(s) to be exploited; and (2) put arguments for the procedure in a stack frame of the victimized process. Basically, a payload crafted for a `return-to-libc` attack contains the entry point of the exploited procedure like `system()` and arguments for the procedure, such as "`/bin/sh`," and this payload is injected into the victimized stack frame via a buffer overflow. When this corrupted value is referenced for the `return` instruction, the control flow is diverted to the target procedure, which is usually located in a C library. Since a subroutine usually does not care how the control flow reached it, compromising a return address is good enough to execute a pre-existing procedure.

Not only at the procedure level, but also at the machine code level, existing instructions can be exploited to form a valid attack. "Return-oriented programming" (ROP) exploits control of stack frames to execute carefully selected machine codes located prior to `return` instructions (9). While `return-to-libc` attacks could be prevented with randomization-based protection like ASLR (Section 2.2.4) and rely on the content of the exploited library, return-oriented programming fundamentally addresses both limitations by collecting existing machine codes in the address space of the victimized process before an attack and executing them.

In general, a payload for an ROP attack is organized with *gadgets*. Each gadget consists of pointers to instruction sequences ending with `return` and immediate data words referenced by the instructions pointed to. A gadget is crafted for a basic operation like load, store, or addition. These pointers are referenced by `return`s and immediate data words are accessed by the instructions pointed to like `pop`. By putting a collection of gadgets into a payload, an adversary is able to synthesize a viable attack vector out of raw instructions stored in the address space shared by the injected payload. As mentioned in (10), the easiest way to place gadgets into the address space of the victimized process is to overflow stack – i.e. stack overflow. When a function tries to fetch a return address from the compromised stack, the ROP attack is executed.

2.2 Protection

Computer security engineers have proposed numerous protection measures against various types of payload injection attacks, and some of them are widely employed in commodity software. This section discusses several protection approaches including some theoretical models.

2.2.1 Stack protection

As described in Section 2.1.2, the stack is the area most vulnerable to payload injection attacks so that various protection schemes have been proposed for it. This section discusses countermeasures against payload injection attacks that exploit this stack overflow vulnerability.

Overflow detection: The most straightforward solution to the buffer overflow vulnerability is to check the integrity of control flow data at its de-reference. In StackGuard (11), an additional piece of data called a "canary" is placed between a buffer and a memory word

containing a return address in the stack layout in order to detect an overflow. This protection is based on the observation that "overflowing" in the stack corrupts not only a return address but also other memory words possibly located between the buffer and the memory word containing the return address. The StackGuard prevents the control flow from being diverted as follows: when a new stack frame is allocated, a canary word is inserted between a buffer and the memory word containing the return address. Before a `return` instruction is executed, an epilogue code checks the canary word to see if the protected return address is compromised or not. If the canary word has a different value from what it had at the time of stack frame allocation, an exception is triggered so that the operating system terminates the infected application or process (Figure 5). If not, program execution continues and the `return` instruction reads the return address. The canary word can have a randomized value so that a sophisticated attack trying to circumvent StackGuard using a static canary value can be thwarted.

As both the prologue code for inserting canaries and the epilogue code for verifying their integrity are generated by a compiler, no source code modification is required – only re-compilation is necessary. However, because StackGuard protects only return addresses in the stack region, other control flow data such as function pointers in the stack or heap region remain vulnerable to sophisticated attacks. In addition, a multi-stage buffer overflow attack is able to locate the return address and to overwrite it precisely without touching the canary, thereby circumventing StackGuard.

Program counter encryption/decryption: Although StackGuard can prevent control flow redirection by verifying the integrity of a canary next to a memory word containing the

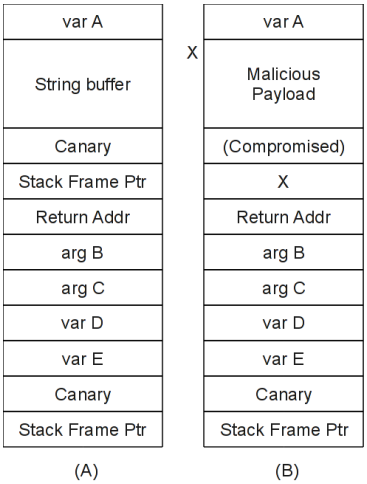


Figure 5: Changes in a stack frame protected by StackGuard
(A) Stack layout generated by a compiler patched for StackGuard. As shown in (B), the canary value is corrupted during a payload injection attack and the overflow can be detected by software.

return address, it does not actually protect the control flow data itself and therefore circumvention techniques such as a multi-stage overflow attack are effective. Lee and Tyagi (12) introduced a protection scheme by encoding return addresses. In their work, when a `call` instruction is executed to invoke a subroutine, the return address is encrypted before being pushed onto the stack whose top is pointed to by the stack pointer register. It is decrypted before being restored to the program counter for the `return` instruction. A similar scheme was proposed in a hardware implementation by Frantzen (13) and employed in the SPARC processor. In a system protected by these randomization-based countermeasures, an adversary may be able to place their payload in the stack region and corrupt a return address. However, as the overwritten return address has not been properly encrypted, the decrypted value of the

return address will point not to the injected payload but to a random location, and the injected code cannot be executed. Unfortunately, such a random jump may result in an undesirable situation – for example, the execution of another procedure located in the same address space, with arbitrary values in the stack frame in use unintentionally referenced as arguments, which could be harmful. In addition, while the return addresses in the stack can be easily identified with the stack pointer register on the fly, other control flow data cannot be protected because it is hard to identify them.

Additional stack frame allocation: Another protection method for a stack region is to save return addresses in a memory region invisible to applications. These saved addresses are reliable return addresses. This protection can be implemented in both hardware and software. In a hardware implementation by Park et al. (14), the Return Address Stack (RAS) is enhanced to be able to hold unlimited entries while a conventional RAS is implemented to store only the most recently stored return addresses with a circular queue structure. In order to store an arbitrary number of return addresses, the RAS is equipped with spill/fill procedure support for overflow/underflows in conjunction with the operating system. Additionally, sophisticated mechanisms to handle non-local jumps and miss-predicted speculative executions are included because these operations will cause false alarms under the present RAS mechanism.

StackShield (15) is proposed as a software approach to give this type of protection. With StackShield, a prologue code moves the return address to a different area when a subroutine is called, and an epilogue code restores the return address upon completing the subroutine.

StackShield only protects return addresses and applications using this protection need to be recompiled.

2.2.2 Compiler-based protection

Compiler-based protection prevents various attacks at the source code or library level. CCured (16) by Necula transforms an unsafe C source code into a trusted one whose memory safety is guaranteed. In order to ensure safety, memory operations that may create an error, such as an out-of-bound reading, are detected and eliminated during compilation and runtime checks are inserted where such off-line analysis is insufficient. Cyclon (17) by Grossman is a dialect of C, and it is designed to preclude attacks exploiting pitfalls in C. This programming language disallows several syntactical expressions and pointer arithmetic, and replaces them with other expressions with similar or equivalent functionality. Because of these changes, a programmer may need to inspect and modify the source code of the application to be protected. ProPolice (18) by Etho is an enhanced version of StackGuard. This protection not only utilizes canaries, but also rearranges local variables and local array(s) in the stack to make it difficult to corrupt control flow data via an overflow. ProPolice has been widely adopted in Linux-based operating systems as well as the BSD operating system family. However, its protection coverage is limited within a stack, therefore other forms of protection are required in conjunction with ProPolice to guard against attack vectors exploiting other vulnerabilities. Protection performed at the library level (19)(20) substitutes vulnerable functions for safe ones so that vulnerabilities, like buffer overflows from unsafe procedures and format strings, are ineffective at the library level. Because vulnerabilities are handled after the control flow goes

into the procedure contained in a library, this protection requires neither modification of source code nor recompilation of an application.

2.2.3 Control-flow integrity

Control-flow integrity (CFI), which was proposed by Martín et al. (21), forces software to follow a path in a *Control-Flow Graph* (CFG) determined ahead of time. The CFI approach labels vulnerable indirect branch instructions and their target locations with IDs. There are associated ID checks. Before invocation of an indirect branch, the ID check compares the ID hard-coded prior to the branch with the ID marked in the target location. This comparison verifies that the control flow redirection conforms to the CFG. If the two IDs match, program execution continues to the target location. Otherwise, the ID check invokes a thwarting procedure. Because the ID checks are written with the same instruction set as the protected program, no modification is required to either the OS kernel or the processor hardware.

Although the CFI implementation labels only machine code, nearly half of the labeled codes are referenced as data by an ID check before an instruction fetch. This is because the ID check accesses the target location as data to find the ID marked in the location before control flow redirection. Because of the memory hierarchy, at least one data cache line must serve this ID check but an ID occupies only a small portion of one cache line (4 bytes out of 32 or 64 bytes). This inefficient data cache utilization could result in high data cache miss rates.

2.2.4 Randomization in address space

Address space layout randomization (ASLR) makes addresses of various objects, stack frames, and procedures less predictable so that attacks based on knowledge of absolute ad-

dresses cannot compromise critical data or divert the control flow into an injected code. Except for absolute addresses, which are decided at compile time, the whole address space of an application is changed every time the application is loaded and executed. These runtime changes help in thwarting attacks that assume that a victim application's address space is static. However, such randomization can only reduce the likelihood of local exploitation – not totally eliminate threats from such exploitations. Shacham et al. (22) investigate the effectiveness of ASLR in a realistic environment, and they find that, due to the limited number of bits available for randomization in a 32-bit architecture, a simple brute-force attack derived from a standard buffer overflow attack can compromise Apache running on a Linux PaX ASLR system.

2.2.5 Non-executable protection

If a memory area is flagged as non-executable then instruction fetches from it are not allowed. Protection measures utilizing this property include hardware extensions or software patches, and many microprocessor vendors provide this protection as a hardware extension under different names- - "NX" for AMD, "XD" for Intel, and "XN" for ARM. In hardware-based protection, memory pages pointed to by translation entries whose "non-executable" property has been set are prohibited from being accessed for instruction fetches. This protection is based on an observation that an adversary places his/her own payload in a writable region such as a stack or heap and executes the payload by redirecting the control flow into the payload. If a process or an application tries to access memory pages with the "non-executable" property for instruction fetch, the processor triggers an exception so that the operating system can handle the violation. In order to utilize this protection, a specified bit field in the control register of the protected

system has to be set, and a programmer or operating system has to explicitly declare that the protected page(s) is not-executable when a new page is allocated. This protection has been helpful in thwarting various code injection attacks.

However, this protection can be circumvented or disabled by sophisticated attacks. As discussed in Section 2.1.6, *return-to-libc* can execute pre-existing procedures without injecting a machine code, and return-oriented programming can exploit existing machine code at much more fine granularity. In the Windows operating system family, memory pages protected by this protection[1] can be arbitrarily changed to "executable" if the procedure controlling the corresponding attribute is invoked (1).

2.2.6 Virtualization of Harvard architecture

Riley et al. (23) present an architectural approach to preventing code injection attacks by virtualizing the Harvard architecture on the x86 architecture. Their approach is based on the observation that code injection attacks exploit the memory architecture of modern computers - the von Neumann architecture – wherein code and data are both addressable within the same address space. Unlike the von Neumann architecture, the Harvard architecture employs a split memory model, enforced at the hardware level, for code and data; therefore it could be suitable for prevention of code injection attacks. In order to virtualize a Harvard architecture in which code and data have their own separate address spaces on the x86 architecture, this approach assigns two sets of physical memory pages for a binary executable during the loading procedure

---

[1]Data Execution Prevention(DEP)

for the binary executable – one for instruction fetches and the other for data accesses. At runtime, it exploits the existing Translation Lookaside Buffer (TLB) to get a physical address for a virtual address according to whether the translation request is an instruction fetch or data access. Additionally, the authors developed this protection to have several attack response modes – break, observe, and forensic mode – so that this protection can be utilized for forensic analysis.

It was shown that this protection can prevent code injection attacks from both benchmarks and real-world examples while overall system performance is acceptable.

2.2.7 Static analysis of foreign objects

Basically, some payloads injected by code injection attacks can be regarded as foreign objects. Runtime environments utilize foreign objects in various ways: as temporary data locally stored for fast fetching or processing, or as a binary executable such as a plug-in component. If these objects can be or are designed to be executed verbatim, the corresponding runtime environments must prevent the arbitrary execution of unreliable foreign objects. In this section, we discuss two examples for such protection, and both of them are for web-browsing environments.

Heap-spraying detector with object interpretation: To guard against the heap-spraying attack described in Section 2.1.5, Ratanaworabhan et al. (24) presented "NOZZLE," a detector for heap-spraying attacks, which monitors heap activity and reports spraying attempts as they occur. In order to detect malicious attempts to exploit the heap region for heap-spraying attacks, the authors use a two-level detection scheme: local and global. Local detection is performed at the per-object level, and global detection monitors global heap health metrics. At the

per-object level, this protection performs a lightweight interpretation of heap-allocated objects, treating them as code. The interpretation includes approaches for analysis – disassembly and building a control flow graph – while focusing on detecting ineffective instruction streams like the "NOP" sled, because they are frequently employed in heap-spraying attacks. Such runtime interpretations are utilized in various applications (25)(26)(27).

Unfortunately, because of the density of the x86 instruction set, even a simple binary sequence composed of alphanumeric characters may look like executable code, so that interpreters are prone to raising false alarms for harmless character strings. In order to mitigate this problem, the global heap health metric is monitored in parallel with the per-object interpretation. As described in Section 2.1.5, an adversary tries to spray as many objects containing a malicious payload as possible to maximize the likelihood of success of the heap-spraying attack. Because the spraying results in a sharp increase in global heap usage, the heap health metric is an indicator for a possible heap-spraying attack. The authors show that this coordination between a local object-based analysis and a global heap health metric can detect a heap-spraying attack under acceptable performance degradation.

Sandbox for untrusted x86 native code: NaCl (Native Client System), introduced by Yee et al. (28), enforces several constraints on a x86 binary executable generated as a plug-in component for a web browser. A static analyzer confirms that these constraints have been followed before executing it. As existing plug-in components running on web browsers are handicapped in an application field that requires computational performance, the authors proposed a plug-in-based runtime environment supporting native x86 machine code. For such a runtime

environment, they proposed a software sandbox for secure execution of foreign objects. Because software modules running as plug-in components are usually provided as foreign objects, the NaCl runtime environment treats such modules as untrusted binaries and imposes several constraints on their execution rather than relying on non-technical measures for secure execution. These constraints include reliable disassembly for static analysis and 32-bit alignment of instructions for control flow integrity, etc. The static disassembler is employed as a validation component to verify whether the untrusted software module conforms to the constraints before execution. This pre-execution procedure prevents arbitrary execution of an untrusted software module and secures control flow integrity.

In order to secure control flow integrity, the disassembler statically computes the target of each direct branch, and confirms that the target is a valid instruction satisfying another constraint for valid instructions. For indirect branches, NaCl limits the virtually addressable range of the loaded software module using the x86 processor's segmentation, sized to a multiple of 4 KB. In addition to this address range limit, 32-bit alignment is enforced for indirect branches by clearing the lower 5 bits of the indirect branch target address.

The authors assessed the performance of the NaCl runtime environment with various benchmarks – from SPEC2K to a widely known 3D game (Quake) – running on web browsers. The experimental results show that the sizes of the executable binaries increased slightly due to the 32-bit alignment, and performance varied depending on each benchmark. However, except for several benchmarks from SPEC2K, most of the benchmarks performed as well as when running in a native environment.

2.2.8 Randomization of instruction set

Kc et al. (29) proposed a randomization of the instruction set (machine code) to counter code injection attacks, also known as Instruction Set Randomization (ISR). For this work, the authors created a runtime environment in which new instruction sets are generated for each process with randomization within one system. Under such an environment, a binary executable is generated with encoded instructions whose decoding key is contained in its header. When the encoded binary executable is loaded into a code section for execution, its decoding key is copied into a special purpose register with a special privileged instruction. When running the encoded executable, the decoding hardware in the processor de-randomizes the instruction stream into a standard one using the decoding key to allow normal execution. This de-randomization procedure triggers exceptions, thereby prohibiting "foreign data" from being executed as legitimate code. Such exceptions may be caused by an illegal instruction generated by de-randomization or by a segmentation fault from referring to an undefined memory range. For an adversary to circumvent this protection and accomplish a code injection attack by producing a properly encoded instruction stream, he/she has to know the exact decoding key as well as the algorithm used for encryption.

Although this approach is effective in stopping code injection attacks from placing machine code in a victim system's memory space, it has several limitations. For example, non-control data attacks, which change control flow, are not detected because no foreign data is injected and executed during the attack. In addition, the performance overhead can be prohibitive due to per-instruction de-randomization and a lack of hardware support (29)(30).

2.2.9 Randomization of the system call interface

Jiang et al (7) proposed a protection scheme based on randomization of the system call interface. As the semantics of modern operating systems prevents processes from having any outside effects without invoking a system call, malicious payloads almost always use system call instructions. Therefore, preventing exploitation of the system call interface can be a good countermeasure against code injection attacks, and the authors implemented protection based on this observation by applying randomization of the interface as a proof-of-concept. Unlike ISR, the proposed approach randomizes only system call instructions by dynamically encoding the system call numbers in a userland application interface on the fly and decoding the encoded number in the system call handler. In order to enhance the existing ASLR, the authors also added randomization to both the loading of libraries and their invocation. When library modules are loaded for a new process, this method inspects the modules and relocates them to random addresses. When a library API function is called based on its string name, the name is randomized/de-randomized as well by intercepting related interface functions.

This method of protection was evaluated with various real code injection attacks and found to be effective in thwarting them. Because this protection randomizes both the system call interface and a library API without regard to any specific type of vulnerability or its exploitation, it is effective against zero-day exploitation as well. This capability was verified with real exploitation attacks, which were able to inject and execute arbitrary code. In spite of having several advantages, this method requires modification of a number of operating system components – such as library loaders and the system call handler.

2.2.10 Information flow tracking

While other protection approaches try to prevent or detect existing attack vectors at designated monitoring points – like the system call interface or stack frames – information flow tracking (IFT) gives more comprehensive protection by tracing *unsafe* data on the fly. Generally, this technique labels spurious data from unreliable I/O such as network interface cards as "tainted" data, and traces the propagation of such data using tags (labels). When spurious data, or data arithmetically derived from such data, is referenced for control flow redirection at runtime, the processor hardware generates a trap (or exception) to let the operating system know about the malicious activity and handle it. Several approaches have been proposed for IFT implementation: an architectural approach, a software-based approach, and a virtualization-based approach.

The architectural approach (31)(32) adds one-bit taint status information to every storage element of a target architecture (registers, cache lines, and memory words in main memory) and propagates the taint status through the designated data channel throughout the architecture. While this approach is effective in the automatic detection and protection of attacks and is efficient in both space and performance overheads, it mandates non-trivial changes in the target processor architecture as well as the operating system supporting the protection feature.

The software-based approach (33) (34) utilizes dynamic instrumentation of machine code to trace propagation of spurious data without modifying existing software modules. TaintCheck by Newsome et al. (33) uses dynamic taint analysis using the Valgrind emulator (35). By implanting the taint-tracing code in the instruction set used by the emulator, the authors

were able to detect various attacks without modifying existing machine code though under a significant performance impact. This approach was proven to be useful in an automatic signature generation system. This system can be used for semantic analysis based on signature generation.

Low-overhead practical information flow tracking (LIFT) by Qin et al. (34) also employs a software-based approach. Unlike TaintCheck, the authors devised several optimization approaches to alleviate the runtime overhead due to dynamic instrumentation, and these optimizations were so effective that the performance overhead decreased by a factor of 5-12 times compared to TaintCheck.

The virtualization technique also can be utilized in an IFT implementation. Practical information flow tracking (PIFT) by Ermolinskiy uses a novel security architecture and a set of associated mechanisms for IFT. Unlike the other IFT methods discussed above, PIFT focuses on detecting information leakage by tracing sensitive data in a virtualization environment. The author takes advantage of spare CPU cores to track information asynchronously and in parallel with the instruction stream executed in a protected guest system. PIFT supports real-time tracking of the information flow in graphical desktop environments.

2.3 Hardware-based protection and software-based protection

Previous sections discussed various attack vectors and protection measures. Broadly speaking, protection measures can be classified into two types – hardware-based and software-based. This section compares two aspects of two approaches and explains our choice in proposing a protection measure.

2.3.1 Required modifications

Generally, most software-based protection requires re-programming or recompilation of target binaries. Even if a protection measure does not require re-programming source codes, it may require recompilation of the generated binary – as seen in ProPolice (Section 2.2.2) and StackGuard (Section 2.2.1).

One exception that does not require software modification is software-based IFTs. As discussed in Section 2.2.10, software-based IFTs dynamically instrument machine code to trace propagation of spurious data without manipulating the operation of baseline software modules. Both instrumentation and execution of instrumented codes are highly likely to incur performance degradation.

Hardware-based protection requires insignificant to non-trivial changes in existing platforms depending on their protection granularities. Insignificant modification examples are instruction-set randomization and program-counter encryption. As these types of protection encrypt and decrypt their target objects – instructions and program counters – only at specified events, only simple encryption/decryption algorithms and key management mechanisms are required, which are already available.

The most intrusive modifications appear to be hardware-based IFT approaches. This is because taint bits must be augmented into every storage element of the target platform and propagation mechanisms must propagate and clear taint statuses of those storage elements for instructions that are related to taint propagation. These modifications pertain to not only structural organization but also internal operations because taint bits (or tags) must be

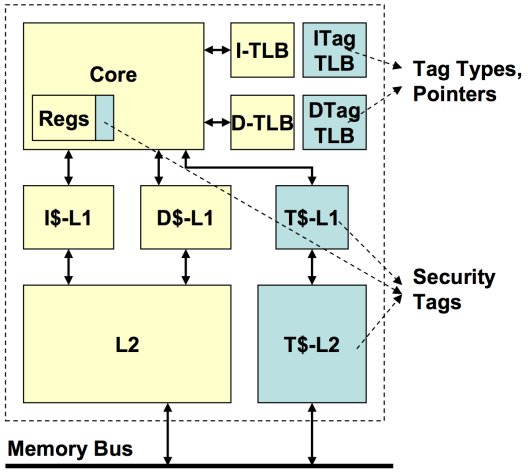


Figure 6: On-chip structures to manage security tags
Dark (blue) boxes represent additional structures (from Suh et al.(31))

synchronized (either in lock steps or at specified events like system calls) with instructions and affected memory elements (Figure 6 from Suh et al. (31)).

However, one of the hindrances for hardware-based modification is that the processor itself must be modified and tested to give reliable operation of affected existing components and features in both the hardware and software domains. For example, program-counter encryption (12) will be incompatible with existing non-local jumps like `setjmp/longjmp` because some implementations of those jumps manipulate return addresses for jumps violating the last-in-first-out call stack behavior.

2.3.2 Performance overhead

The performance overhead is a critical issue in adopting a protection scheme for target platforms. Furthermore, some approaches can be implemented in both hardware and software,

and there is trade-off between them. For example, vendors for x86/x86_64 processors use a NX-based protection feature in most of their products. The performance penalty from adopting hardware-based NX protection is nearly zero because virtual-physical address translations (during which NX-based protection is applied) are performed in parallel with the L1 cache lookup and because optimization techniques applied in address translation – like indexing/tagging of cache lines – can minimize the performance overhead from address translations.

Performance impacts from software-based protection vary depending on protection targets. Stack protection, as used by ProPolice and StackGuard, incurs a negligible performance overhead because changing the layout of memory words in a stack does not affect runtime operations (ProPolice) and executing small code snippets – such as a function prologue and epilogue – does not require much computing power.

On the other hand, if software-based protection were to inspect a target application's behavior closely, it is highly likely that the protection would incur significant overhead. This is because *a protection context* normally consists of auxiliary code that is not directly related to the outcomes of its target application and is inlined into its target code. A good example is instruction-level software-based IFT. As discussed in Section 2.2.10, software-based IFT approaches utilize instrumentation infrastructure to track propagation of unreliable data. Because a target code must be instrumented before execution, binary translation procedures inevitably incur performance degradation. Figure 7 from Qin et al. (34) shows three examples of instruction instrumentation. The instrumentation shown in bold font is an original instruction from the target program. The unbold instructions are instrumented before the bolded instructions

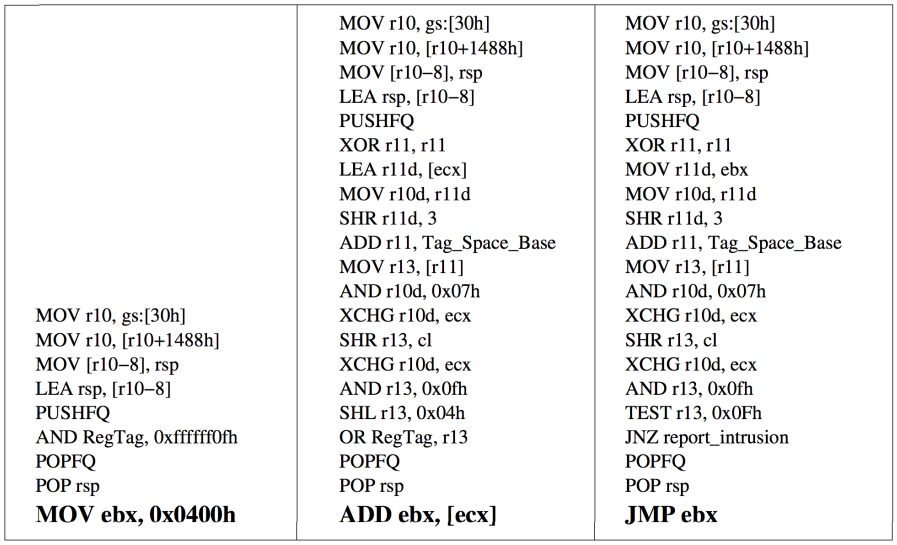


Figure 7: An example of information flow tracking for LIFT-basic
From Qin et al.(34)

perform information flow tracking. An addition to the second column requires six additional memory accesses, one of which refers to the other segment. All of these memory dereferences will make the L1 data cache busier. Furthermore, the increased code size will inevitably trigger frequent instruction cache misses.

Although some optimization approaches like code caching are able to reduce instrumentation overheads, inherent overheads from generated code cannot be mitigated. One noteworthy approach for reducing the execution overhead is to omit taint propagation analysis for procedures not handling spurious data. However, the benefits from such optimizations are only observed in specific applications while other applications – especially general benchmarks – still suffer from significant degradation.

Moreover, hardware-based NX protection barely incurs a performance overhead, as already discussed. In fact, hardware-based NX protection requires only a simple implementation. Only a few conditional branches are required in a TLB/paging unit implementation for the detection mechanism and the thwarting process is carried out by the OS kernel. Other hardware-based protection methods, like randomizing instructions (29) or the program-counter (12) approach, are even simpler than NX-based approaches.

Even if a hardware-based IFT requires complicated operations, the performance overhead could be alleviated with optimization techniques. Hardware-based IFT approaches are designed to minimize the runtime overhead with optimization techniques like employing taint caches (31) or de-coupling taint propagation from instruction execution (36). Those works showed a negligible performance overhead in most target applications. It is quite obvious that those approaches require non-trivial and/or intrusive changes in existing hardware architecture – such as a multi-level taint cache structure or a dedicated co-processor.

This thesis pursues hardware-based approaches to countering malicious attacks at the architectural level. We chose the hardware-based approach because we believe that existing architectural components and inherent events could be utilized to counter malicious attacks while minimizing the required modifications in both the hardware and software domains.

This thesis proposes two hardware-based countermeasures that exploit architectural components and inherent events. The first approach utilizes a TLB structure and monitors I/O activities in order to counter shell-code injection attacks. The second approach pursues a more

detailed approach than the first by exploiting more architectural components and inherent events.

CHAPTER 3

PAYLOAD INJECTION ATTACKS FROM ARCHITECTURAL STANDPOINTS

In spite of extensive research on payload injection attacks, malicious parties have been successful in circumventing the protection measures proposed by those works as discussed in previous sections, and many vulnerabilities are still being discovered and announced by the security research community. This chapter discusses payload injection attacks from the architectural viewpoint, how to utilize this observation, and what technical issues are expected in utilization.

3.1 Shell-code injection attacks and virtual address translation

3.1.1 Linear address translation in the x86 architecture

Modern processors have virtual memory systems, which give the OS and applications a seamless virtual address space for physical memory even though this may have less physical space than the virtually addressable range. In this section, we use the x86 architecture as an example of virtual address translation.

In the x86 architecture, the processor translates a logical address to a physical one through a segmentation unit (memory management unit, MMU) followed by a paging unit (page table and TLB – Translation Look-aside buffer). The address translated by the segmentation unit is called the linear address, which is the input for the paging unit.

The OS kernel organizes the paging translation structure in a multi-level hierarchical table saved in DRAM – the page directory and page table – for the processor to traverse when a linear address is translated into a physical one, and this traversing is called the "page table walk". Every element – Page Directory Entry (PDE) and Page Table Entry (PTE) – of this structure contains an access control attribute initialized and managed by the OS kernel and the processor gathers these attributes to verify whether address translation is legitimate or not. In order to maximize translation throughput, translated entries are cached in the TLB.

Like other caches, a TLB lookup can fail and a page table walk takes place. The processor performs a page table walk to see if there are valid and accessible paging structure entries for the requested linear address. If the processor finds matching ones, it brings them into the TLB and the TLB access is tried again. If not, the processor triggers a page fault exception for the OS kernel to handle.

Microprocessors implement TLB miss handling either in hardware (as in x86 and ARM processors) or in software (as used by IBM, SUN and MIPS) for looking up a page table structure to fill up a missed TLB entry. In software-managed TLB, a TLB exception is triggered when there is no valid entry for a requested address and the OS kernel traverses the page structure to fill the missed TLB entry. On the other hand, on microprocessors with a hardware-managed TLB, the OS kernel is not aware of a TLB miss as this situation is handled by the hardware.

3.1.2 Address translation and shell-code injection attacks

Regardless of whether a machine code is legitimate or not, simple data move instructions are invoked to write the code into a writable region in main memory, and this operation leaves

"dirty" attributes (such as a bit vector) in the corresponding DTLB entries. The paging unit changes this attribute, which the OS kernel uses for coherency management between main memory and a storage peripheral like a hard disk drive, and mirrors this attribute change to translation entries saved in main memory for coherency of the translation information.

If this code is loaded by an OS kernel, the kernel invokes a post-loading procedure for the code before execution, and this procedure includes attribute changes in the translation entries (in main memory) such as: (1) clearing the "dirty" bit; (2) setting the "read-only" attribute to secure the integrity of the code; and (3) initializing the required privilege to access the code. Optionally, the OS kernel could invalidate the translation entries in the TLBs, for example with the `INVLPG` instruction in the x86 architecture, because of the discrepancies in attribute bit fields between the translation entries in the TLBs and those in main memory.

On the other hand, if a code is loaded without the kernel's supervision, in other words arbitrarily injected, the post-loading procedure, including the "dirty" bit clearing and TLB entry invalidation, is omitted so that the injection leaves a trace in the DTLB – a valid entry with the "dirty" bit set.

This observation infers that translation entries for arbitrarily written codes would not be initialized as entries for legitimate machine codes. For example, the "dirty" bit(s) of the translation entries for a malicious payload will remain uncleared until those entries are referenced for an instruction fetch. At first glance, exploiting this observation could be the basis of a viable protection approach against shell-code injection attacks at the memory architecture level. How-

ever, there are two types of legitimate code that look nearly the same as shell-code injection attacks in terms of address translation. The next section discusses this issue.

3.1.3 Runtime code and address translation

While translation entries for a virtual-physical address translation mechanism can be utilized to detect a code injection attack, there is one problem – how to deal with a binary code legitimately written at runtime.[1] In terms of memory address translations for placement of machine code and an instruction fetch, there is no difference between code injection attacks and several runtime codes: binary data is stored in a writable memory region on the fly, and later the program counter is updated to its starting address without the post-loading procedure – such as clearing dirty bits – being handled by the OS kernel.

Runtime codes are used to manage the computer resources dynamically or to maximize performance in a runtime environment. One example of dynamic resource management is device driver software for hardware devices. As there are a myriad of hardware devices that can be connected to one computer system, including all of the device drivers for those devices in the monolithic kernel is generally not a good solution. In the Linux operating system, the device drivers can be built as external software modules, and they can be dynamically loaded and unloaded at runtime.

Another example of this runtime code is a code snippet generated by JIT compilers like JVM in the Java runtime environment (37), TraceMonkey in the Firefox web browser (38) from the

---

[1] We refer to this type of binary executable as "runtime code" hereafter.

Mozilla Foundation, or the V8 engine in the Chrome web browser from Google (39). In those runtime environments, programs to be executed are compiled into or composed in a standardized portable format, like `class` or `jsp` files, in order to provide portability over various architectures. This portability is accomplished by interpreting those programs into native machine code before execution at runtime. In addition, by keeping the translated codes that have been or seem to be invoked frequently and executing them later rather than re-translating them, the performance of those runtime environments can be significantly improved.

In terms of address translation, these runtime codes look almost the same as shell-code injection attacks, therefore referring only to address translation entries will trigger false alarms for such legitimate codes. This means that we need to refer to architectural or operational circumstances other than address translation entries to prevent false alarms.

3.2 Payloads as foreign objects and architectural components

Referring to address translation information for countering shell-code injection attacks has several advantages, including a small impact on performance, because virtual address translations are mostly handled by designated architectural components like the paging unit or TLB. In terms of effectiveness, this approach is similar to NX-based protection (Section 2.2.5). However, since malicious parties have been successful in circumventing NX-based protection (Section 2.1.6), it is highly likely that the same circumvention techniques will be able to bypass protection approaches based on the observations in the previous section. This section analyzes payload injection attacks from various architectural standpoints in order to develop a more sophisticated and fine-grained approach for countering advanced attack vectors.

3.2.1 Payload injection attacks as unexpected/unprecedented events

Generally speaking, payloads crafted for malicious attacks can be seen as foreign objects placed in the victimized process's address space at runtime. This infers that the processor's execution contexts directed by such payloads are highly likely to encounter unprecedented or unexpected events from the architectural point of view. This observation is adopted from Shi's work (40), which focuses on validating branch behavior by referring to branch histories. This thesis takes an architectural approach with regard to the virtual memory system and foreign objects.

Processor vendors adopt various acceleration techniques in their processor products to maximize performance at the architecture level. Two representative examples are cache structure and the branch predictor. Both architectural components take advantage of *previous* events or circumstances – caches exploit temporal locality and spatial locality and branch predictors refer to previous branch outcomes for speculative execution. These two optimizations are critical in achieving high performance, and CPU designers spend considerable time in improving their performance.

We believe that good performance by those components – high cache hit rates and low branch misprediction rates – are not only essential in maximizing a processor's performance but are also useful in preventing malicious attacks. As discussed above, payloads injected by adversaries are highly likely to cause unprecedented events or circumstances, and the cache controller and the branch predictor would invoke their handlers accordingly during execution of the payload to resolve such events. Our claim is that malicious attacks can be detected at

the architectural level by exploiting those handlers. We expect two advantages from utilizing the cache structure and branch predictor handlers.

One advantage is that these handlers are transparent to software modules. In most microprocessors, cache misses and branch mispredictions are inherent and transparent to attackers as well as benign software modules and therefore there is no way to manipulate or exploit them. In most ISAs, only a few instructions are provided for cache line invalidation and software modules cannot arbitrarily load a cache line into the L1 instruction cache without accessing the cache line for an instruction fetch. There is no way for software modules to arbitrarily manipulate branch predictor outcomes in order to fill the processor's pipeline with instructions based on the crafted outcomes.

The other advantage is that the execution context that is expected or predicted would not be influenced by exploiting the cache structure and branch predictor handlers. In other words, the cache lines in the L1 instruction cache would not trigger cache misses therefore the processor can execute instructions fetched from the L1 instruction cache. For the branch predictor, correct branch predictions allow the processor pipeline to continue to consume instructions decoded for speculative execution.

Obviously, these observations do not mean that every cache miss and every branch misprediction are due to a malicious attack. We need a sophisticated mechanism to distinguish benign events from ones caused by malicious attacks.

3.2.2 Cache miss handler

As is widely known, most microprocessors have a cache structure and the main memory is accessed only through the cache structure and its own cache organization – such as the unified cache for the low-level cache, or the shared cache among the processor cores. In this section, we use the following cache structure and control instructions for our discussion. This model is referenced from the Intel Core architecture, which is commercially available as of this writing. For simplicity, other cache optimization techniques like victim caches (41) or processor-specific instructions like the `PREFETCHh` instruction (42) are not considered in this thesis.

Architectural parameters	**Value**
Cache line size	64 Bytes
L1 Instruction cache	32KB, 4-way set assoc., LRU
L1 Data cache	32KB, 8-way set assoc., LRU
L2 Unified cache	256KB, 8-way set assoc., LRU

TABLE I: CACHE STRUCTURE

As discussed in Section 1.3, this thesis considers payload injection attacks targeting low-level semantics at the machine code level. In this regard, the miss handler of the L1 instruction cache would be our first validation point.

The processor core fetches instructions only from its L1 instruction cache, and the miss handler of the cache is responsible for handling missed cache lines. This implies that we can take

advantage of the L1 instruction cache miss handler to prevent some types of payload injection attack – such as injecting shell codes into a stack region (Section 2.1.2). In other words, when the miss handler of the L1 cache fetches the missed cache line from the L2 unified cache, the handler could validate the integrity of the cache line. Because this validation procedure would be implemented as an extended procedure of the cache miss handler, this approach could be easily integrated into the existing memory hierarchy. Furthermore, depending on the validation mechanism, this procedure could be executed in *parallel* with the cache miss handler. In the meantime, once a cache line is validated and loaded into the L1 cache, the processor core can fetch instructions from the loaded (and validated) cache lines without any additional procedures.

One plausible argument against this approach concerns the exploitation of existing cache lines in the L1 cache. As mentioned earlier, it is impossible for a software module to arbitrarily put certain cache lines into the L1 instruction cache without accessing those lines for an instruction fetch. This means that the only way to mount a viable attack with existing cache lines is to craft a payload that accesses only the instructions stored in the L1 instruction cache. We argue that this attack vector is highly unlikely because of several operational limitations: (a) instruction executions are frequently disrupted by interrupts and exceptions, and the processor core flushes cache lines before and/or after resolving those events; (b) cache lines are frequently removed from the L1 cache based on replacement policies like LRU, therefore how long a cache line would remain in the cache is unknown; (c) at the software level it is impossible to know what cache lines are stored in the L1 cache.

One thing to be aware of with regard to this utilization approach is that, of the cache miss handlers, we can use only an instruction cache miss handler to invoke a validation procedure. As shown in Table I, we assume there is a unified L2 cache in our cache structure; therefore we have only one instruction cache miss handler. In general, a unified cache is larger than a high-level instruction cache and a data cache combined. This means that an instruction cache miss at a high-level cache could be served with a cache line of the low-level unified cache. In this case, the miss handler of the unified cache would not be invoked for a cache line containing a malicious payload. If we want to utilize a unified L2 cache for access validation, we need an additional procedure for the cache controller; this procedure must distinguish hit-miss statuses based on the purpose of the memory access and issue a validation query accordingly.

Utilizing the L1 data cache for countering payload injection attacks has one major problem. At first glance, preventing malicious memory accesses – like overwriting a function pointer variable in main memory – seems viable. However, there are countless memory writes during a normal process run, and it is nearly impossible to distinguish memory accesses by attacks from regular memory accesses within an execution context or a procedure at the machine code level. In other words, the legitimacy of a memory access cannot be judged unless there is an *explicit policy* at the memory hierarchy level. A protection approach enforcing such a policy in the form of data flow integrity (DFI) was proposed by Miguel et al (43). The DFI approach is specifically designed as a software instrumentation feature; therefore, we do not consider this approach in this thesis.

In summary, our claim in this section is that the L1 instruction cache miss handler could be utilized to counter payload injection attacks and such an approach would be implementable by exploiting its miss handler. On the other hand, utilizing the L1 data cache in the same way as the L1 instruction cache is only feasible with an explicit policy to enforce data writing – like data flow integrity.

3.2.3 Branch predictor

Modern microprocessors utilize branch predictors for speculative execution. By executing instructions in advance and keeping the processor's pipeline busy, processor performance can be significantly improved.

As branch predictors produce branch predictions based on previous outcomes, it is highly likely that a branch prediction hit indicates that the execution path to be explored has already been visited, therefore, it is safe to execute. On the other hand, a misprediction does not necessarily mean that the actual branch outcome was caused by a malicious attack. Other than suspicious activities, there are several sources of misprediction in conditional branch predictors.

First, most conditional branch predictors require training periods. When a branch predictor has few branch histories, it cannot make accurate branch predictions; therefore, mispredictions are unavoidable during training periods. This is frequently observed after hardware events like context switches (44). Second, hardware branch predictors have capacity and aliasing problems – in the branch history register and pattern history tables. If a branch outcome correlates with a branch history that has already shifted out, the predictor is highly likely to generate inaccurate

predictions. The final source is the randomness of the input data. The more a branch outcome relies on direct input data values, the more the predictor makes inaccurate predictions.

While most microprocessors are equipped with conditional branch predictors, indirect branch predictors are adopted in some microprocessors. Generally speaking, indirect branches are hard to predict because they are able to lead the control flow into arbitrary locations in a given address space, unlike conditional branches that have only two predetermined target addresses – taken or not-taken.

The latest research on indirect branch prediction (45) proposes an improved branch predictor, which has hit rates ranging from 30.0 to 83.3 percent (SPEC Benchmarks) and from 44.6 percent to 61.2 percent (Java applications). It is obvious that an improved indirect branch predictor will be helpful in reducing performance degradation from mispredictions as well as power consumption. Nonetheless, a branch predictor with hit rates like 83.3 percent does not appear to be a good candidate for an independent part of a protection suite because of the post-processing required for mispredictions.

Regardless of the branch predictor type, if a protection mechanism considers a branch misprediction as a symptom of an attack, the processor core should regard the event as an insolvable problem and pass it to the OS kernel. As with NX-based protection handled by the page fault exception handler, an attack or suspicious activity detected by the processor hardware must be handled through exception handling. This means that every branch misprediction would trigger the exception handler, and this approach has two problems.

One is the performance overhead from triggering an exception and subsequent recovery. Basically, asynchronous events like interrupts and exceptions disrupt processor execution flow. During an exception trigger, the processor core discards instructions contained in the pipeline and forces the control flow into the designated exception handler. As discussed earlier, branch predictors are highly likely to generate mispredictions after context switches. Even if the processor core allows a grace period in which the core does not trigger exceptions from branch mispredictions until the branch predictor is fully trained, exceptions from branch mispredictions are inevitable because of other circumstances like aliasing. Furthermore, if no evidence of an attack is found for a suspicious branch outcome, all of those exception triggers and handling will become CPU clock waste.

The other problem is that the user application must be reprogrammed to handle messages about suspicious branch outcomes sent from the OS kernel. In general, the OS kernel contains limited information about user applications – it mostly has information about resource management like memory usage and scheduling, and does not have internal information about them, like basic block layouts.

These circumstances enforce the application that had a suspicious branch outcome to verify the outcome based on the message about the branch outcome transferred from the OS kernel and its internal information. Upon receiving the message, the application is supposed to take the snapshot of the execution context – including registers and memory blocks – and parse the message and scrutinize those snapshots to determine whether or not it is under attack. This procedure looks very much like program debugging, which is very slow in program execution.

User applications that need to be protected must be programmed with procedures handling those messages. Again, all of these handlings, signaling, and processing will become CPU clock waste if there is no evidence of an attack.

Meanwhile, the RAS prediction mechanism can be viewed in a different way. Unlike conditional branch predictors or indirect branch predictors, the RAS branch predictor contains predetermined target addresses and each `return` instruction has one pair of target addresses – one in the RAS and the other in the stack element pointed to by the stack register. If we could incorporate a hardware-based feature that determines whether a return address associated with the stack register is compromised or not, the processor core would be able to prevent or thwart stack-compromising attacks like *return-oriented-programming* or *return-to-libc* without having a validation or verification procedure in the software domain.

3.2.4 Legitimate miss events

It is quite clear that not every instruction cache miss and every RAS misprediction is due to a payload injection attack. In this section, we briefly discuss the circumstances under which legitimate misses could occur.

Instruction cache misses may occur in various situations: (a) the processor core simply tries to fetch instructions that have never been executed before; (b) the cache line may have been replaced with another cache line; or (c) critical events like context switches flush the all of the cache lines and most of the instruction fetch attempts after a flush are highly likely to trigger instruction cache misses. These operational circumstances suggest utilizing the L1 instruction cache miss handler is an inefficient approach. We address this concern with two observations.

First, in our simulations with a fully featured multi-tasking environment, the L1 instruction cache showed very low miss rates – from 1.7 percent to 3.2 percent. This implies that the L1 instruction cache miss handler would not be frequently invoked; therefore, the performance impact from utilizing the L1 instruction cache miss handler could be insignificant. Second, we have a reasonable timeframe between issuing a cache line address to the lower level cache and an instruction fetch from the missed cache line (Intel Core architecture with 45nm technology – L1 hit latency = 4 clock cycles, L2 fetch = 42 clock cycles). We can take advantage of this timeframe to validate the legitimacy of the missed cache line.

There are two cases in which RAS misses are triggered for legitimate return addresses in an active stack. One is the circular buffer underflow, and the other is the `longjmp`. In current microprocessors, the RAS is implemented with a circular buffer. Because circular buffers have a limited number of entries and the RAS entries are discarded at various events like context switches or mispredictions, the RAS may have underflows. Following a RAS underflow, the branch predictor must trigger the RAS miss handler to fetch return addresses from the active stack frame.

Unlike the RAS buffer underflow that is caused by a structural limitation, `longjmp` is caused by stack pointer manipulation. Invoking a `longjmp` violates the last-in/first-out order of stack frames by bypassing multiple stack frames. Since the `longjmp` software implementation updates the stack register to bypass stack frames while the RAS entries remain unchanged, RAS mispredictions will be triggered by subsequent `return` instructions. Generally speaking, any

manipulation that crafts the stack register or a memory word to be referenced for a `return` instruction would trigger misprediction – even if it is legitimate.

However, we suspect that a validation procedure to distinguish RAS misses would be invoked infrequently. First, the RAS miss rate is low in general because of the last-in/first-out order of the return addresses. Second, stack pointer manipulations like `longjmp` are not widely utilized in practice. These two experimental observations substantiate our claim that the RAS miss handler could be utilized to counter payload injection attacks with a low performance impact.

3.2.5 Taint status data for protection

As discussed in the previous section, every cache miss and every branch misprediction are not necessarily caused by malicious attacks; therefore, it is important to determine whether or not a control flow redirection handled by these handlers is legitimate. This infers that we need auxiliary data to assess the legitimacy of control flow redirection.

In general, such auxiliary data could be generated either ahead of time or at runtime. Examples of the former are CFI (21) and Shi's work (40); these build branch information during compilation or by training and incorporate the data collected into software modules or an architectural component. On the other hand, IFT approaches manage the statuses of memory blocks that could contain suspicious foreign objects. This thesis takes the latter approach in managing taint status data for control flow redirection validation to support security at the processor architecture level.

Most IFT-based approaches utilize and manage the taint status for unreliable objects to counter malicious attacks. "An object being tainted" in this thesis indicates that a target object

originated from an I/O device during runtime. As most runtime environments with files systems fetch binary executables from I/O devices like hard disk drives during binary loading, to regard every object in an address space that was loaded from I/O as being tainted, regardless of its loading status, appears to be problematic. Therefore, this thesis considers only objects loaded during runtime (or execution time) as tainted and assumes that binary executables prepared by the OS kernel during a binary load are reliable. It is highly likely that when a tainted object is fetched by or for control flow redirection, the operation is because of a malicious attack and the validation unit based on the proposed scheme will trigger an exception for the post-detection procedure.

This section discusses three technical issues for utilizing taint status data as a protection measure. All of these issues will be addressed and resolved in Section 5.

3.2.5.1 Fine granularity

Taint-based protection schemes like IFT have their own taint status management approaches. One critical issue in designing taint status management is the supporting granularity.

In most current memory systems, the basic management unit is the page frame. Many microprocessors supporting virtual memory systems adopt a 4KB granularity for their page frame management and provide several data structures like paging translation entries and their table. At first glance, extending a virtual memory system with taint status information, like assigning a bit field in translation entries, seems to be viable approach. However, the 4KB granularity is not small enough to distinguish the reliability of memory blocks in one page frame.

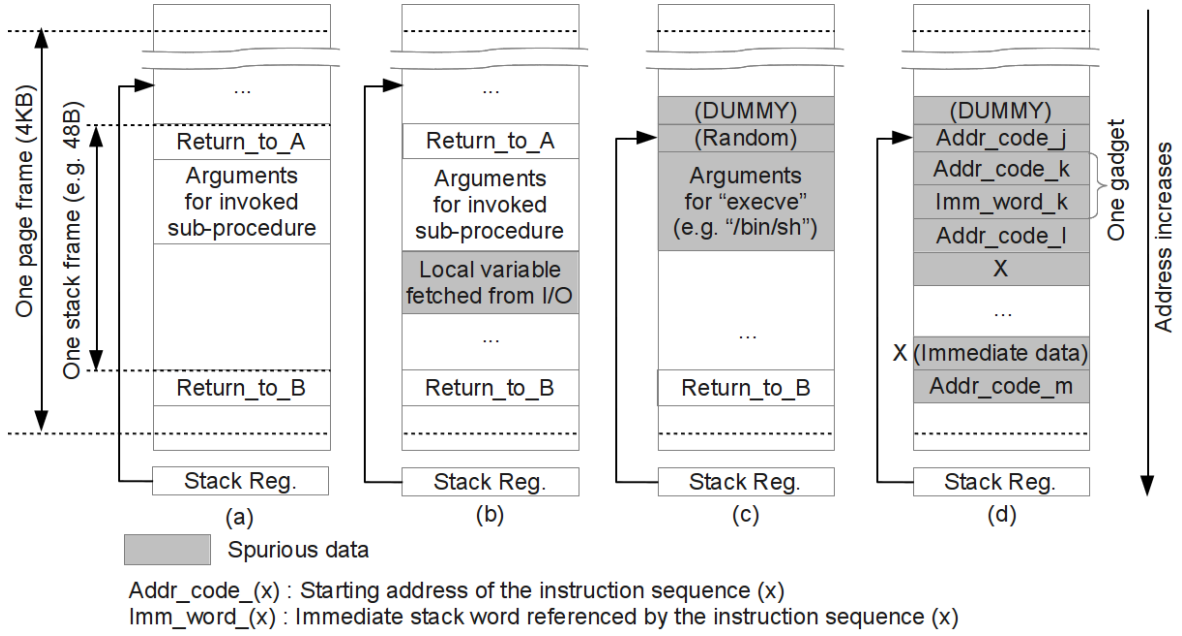


Figure 8: Active stack region and spurious data
(a) normal active stack frame without spurious data; (b) safe stack frame containing spurious data; (c) stack frame overflow by brute-force return-to-libc attack; (d) stack frame overflow by return-oriented-programming attack.

Figure 8 illustrates the coarse granularity problem. While a page frame size is usually 4KB and access control attributes are applied per page frame, stack frame sizes are usually much smaller than the page frame size and vary depending on the invoked sub-procedures. In our simulations, most stack frames were smaller than 1KB and varied from 4B to 8KB. These results mean that an attribute value representing the reliability of a page frame would not be able to differentiate (b) from (c) as well as (d).

IFT approaches support fine granularity like 1B or multiple granularities for taint status data to address problems caused by coarse granularity. Obviously, those mechanisms require

significant modifications in target architecture platforms because existing memory systems are designed for fixed-size blocks like cache lines and page frames.

3.2.5.2 Supporting virtual memory systems

Many microprocessors extensively utilize a virtual address space to enable secure multi-tasking environments. One critical issue for taint status data to be incorporated into multi-tasking environments is to ensure isolation between address spaces. In order to support an arbitrary number of address spaces, taint status data should be contained in main memory and assigned per address space. This means that taint status data is vulnerable to cross-contamination unless the platform – both the processor and the OS kernel – enforces strict isolation of taint status data between address spaces.

In supporting a virtual address space, many microprocessors provide several types of data structure and special purpose hardware registers. The OS kernel manages numerous instances of the data structure and updates those registers for each address space, and the hardware paging unit translates a virtual address into a physical one by traversing those instances starting with the base pointer register for translation information. Those instances contain address translation information for virtual address spaces and can be seen as auxiliary data for address space management because applications being executed will never access them.

One noteworthy observation on address space setup in multi-tasking environments is that the OS kernel carefully initializes the translation information for virtual address spaces so that they do not share (or minimally share) the same physical address ranges. Figure 9 illustrates how this approach works for four virtual address spaces, each of which is organized with translation

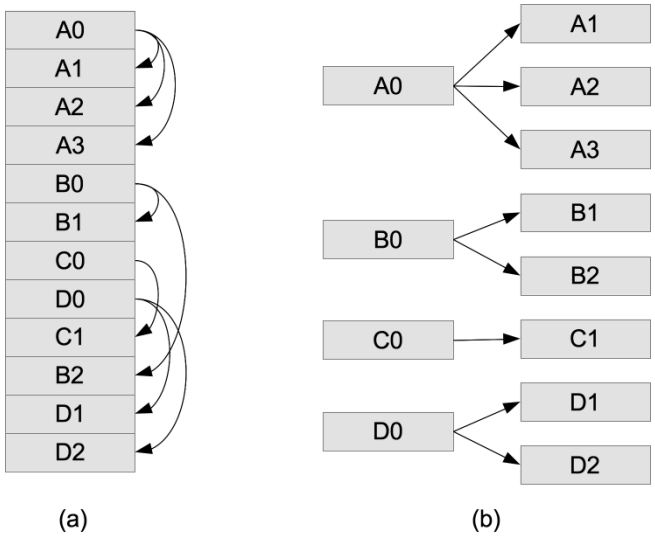


Figure 9: Isolation of translation information
Each block represents one translation table, and arrows indicate the traversal path set by the OS kernel and visited by the paging unit: (a) physical layout of translation tables; (b) logical organization of translation tables.

tables. The left-hand side shows the physical layout of the translation tables, the right-hand side shows their logical organization, and the arrows indicate traversal paths configured by the OS kernel. Although the translation tables are located next to each other in the physical layout, the paging unit is able to traverse translation entries without going to another space's entries. Through this strict isolation mechanism, the OS kernel is able to enforce address space isolation between virtual address spaces and each application running on a platform does not have to consider cross-access for instruction fetches or data accesses from other address spaces. A similar approach as the virtual address space setup could be applied to taint status data organization

and initialization – assigning taint status data per address space and incorporating a hardware mechanism to prevent cross-access of taint status data across virtual address spaces.

Knowing how to locate taint status data for one address space is as important as knowing how to ensure isolation. In most microprocessors supporting virtual address spaces, there is a special hardware register to point to the base of the translation information – for example, the `CR3` register in the x86 architecture. In Figure 9, the `CR3` register must point to the base of the A0, B0, C0, or D0 translation table to select an active virtual address space.

As taint status data in main memory should be accessed by a hardware unit in the same way that translation tables are visited by the paging unit, a designated special purpose hardware register is necessary and the OS kernel must update the register at critical events like context switches.

3.2.5.3 Taint propagation and block copying

In this thesis, we assume that malicious payloads are introduced into an address space through I/O interfaces. In a Linux kernel, the system call interface provides a programming interface for I/O operations – such as the `read` system call. When a foreign object is loaded into the address space of an application, the application processes it with respect to its purpose. For example, a JPEG file transferred over a network interface card is decoded with an image viewer for screen display, or a database query string is parsed by the lexical analyzer to locate the data elements in the target database.

This means that a foreign object or a part of such an object could be copied or moved to other locations in the given address space. It is obvious that the taint statuses of such relocated

foreign objects must be duplicated or transferred along with the corresponding target data. This mechanism is frequently referred to as "taint propagation" and most IFT approaches support such mechanisms. Generally speaking, there are five dependencies for taint propagation: (a) copy dependency; (b) computation dependency; (c) load dependency; (d) store dependency; and (e) control dependency (31). Confirming all of these dependencies appears to be necessary in tracking spurious foreign objects. However, with such propagation tracking it is easy to trigger unexpected taint propagation.

For the load and store dependencies, if a pointer variable referenced for memory access is tainted, the taint propagation mechanism considers the value contained in the target register or memory block accessed through the pointer variable tainted (or spurious). Unfortunately, in such a blind propagation tracking mechanism it is easy to incur taint pollution (46). Taint pollution is the explosive taint propagation across the kernel and user processes and is triggered by the OS kernel and blind taint propagation. Such out of control propagation would quickly render the whole system unreliable. Not only the load and store dependencies, but also other dependencies can easily trigger out of control taint propagation. There are several mitigation approaches for taint pollution: excluding stack registers in taint propagation (46), incorporating a security policy (47)(31), or scrubbing taint statuses at designated points of the OS kernel (48). However, these approaches control only limited propagation paths, and none of them is able to actively control taint propagation per memory region.

This thesis tries to confirm one dependency – the copy dependency. Our taint propagation policy is based on the observations that malicious payloads are usually relocated verbatim and

that such relocations could be monitored at the memory system level on the fly by analyzing data cache accesses.

CHAPTER 4

TLB MON

In Section 3.1, we discuss how the virtual-physical address translation mechanism could be utilized in detecting shell-code injection attacks. Based on that observation, this chapter presents a protection approach against shell-code injection attacks, called "TLB Mon" – TLB Monitor. The proposed protection is designed as an extension of the virtual-physical address translation for a virtual memory system.

4.1 Working mechanism

We choose a two-TLB structure – an instruction TLB (ITLB) and a data TLB (DTLB) – as our target architectural component because these TLBs are transparent to the software level and because we found that hit-miss statuses in both TLBs represent instruction fetch attempts for modified memory page frames as follows.

Once a malicious shell code is injected into the victimized address space, the only step left to accomplish the code injection attack is to redirect control flow into the payload in the main memory to execute it. For this redirection, an adversary exploits one of the vulnerabilities discussed in Section 2.1.1. Regardless of how the control flow is redirected to another instruction, the address translation request for the instruction fetch is routed to the ITLB and so is the request for the malicious payload. For the newly loaded machine code, even if the code is loaded by the OS kernel, its execution is always subject to triggering an ITLB miss, because

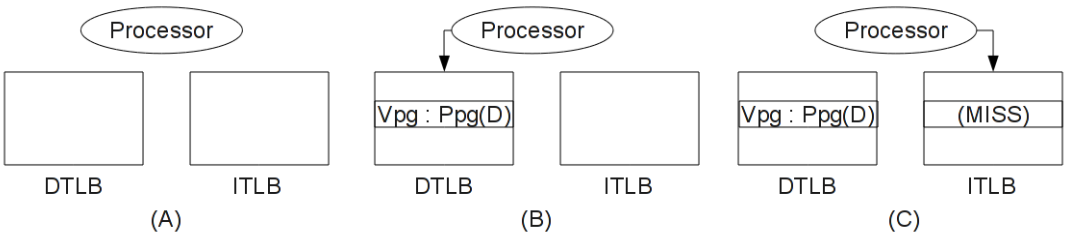


Figure 10: TLB status changing during a code injection attack
Vpg: virtual page number, Ppg: physical page number, D: dirty

the memory page containing the machine code has never been accessed for an instruction fetch before.

In summary, the activation of a code injection attack can be identified during an instruction fetch by following the hit/miss statuses in the two TLBs; Figure 10 shows how this detection scheme works.

- ITLB miss
- DTLB hit with "dirty" bit set in the hit entry

However, as discussed in Section 3.1.3, referring only to address translation statuses will trigger false alarms for legitimate runtime codes. The next section discusses how to deal with such runtime codes from an architectural standpoint.

4.2 Runtime codes and I/O

In Section 3.1.3, we present two types of runtime codes: device drivers and code snippets generated from runtime environments like JVM. This section discusses how to distinguish these machine codes from an architectural standpoint.

In general, device drivers are programmed to run with the root privilege. This means that the processor's current privilege level (CPL in the x86 architecture) must be elevated from the user level at which most applications work to the root level that the OS kernel is running under. This privilege elevation provides reliable and efficient centralized management of shared platform resources – CPU clock quantum, memory pages, I/O devices.

On the other hand, vulnerable applications generally run at the user privilege level. Malicious attacks are designed to redirect program flow (running with the user privilege level) into injected payloads by exploiting vulnerabilities. Because the program execution flow is redirected into injected codes without privilege elevation, such codes are executed at the user privilege level. Figure 11 is an example of privilege switching between the user and root levels.

In summary, the execution of device drivers can be distinguished from that of malicious attacks by referring to the current privilege level – if a software module runs at the root privilege level, the program execution can be trusted regardless of the hit/miss statuses in the ITLB and DTLB.

The other type of runtime code that we try to distinguish from a malicious shell code comprises code snippets generated from runtime environments like JVM. Unfortunately, such runtime codes are generated and executed at the user privilege level. This means that referring

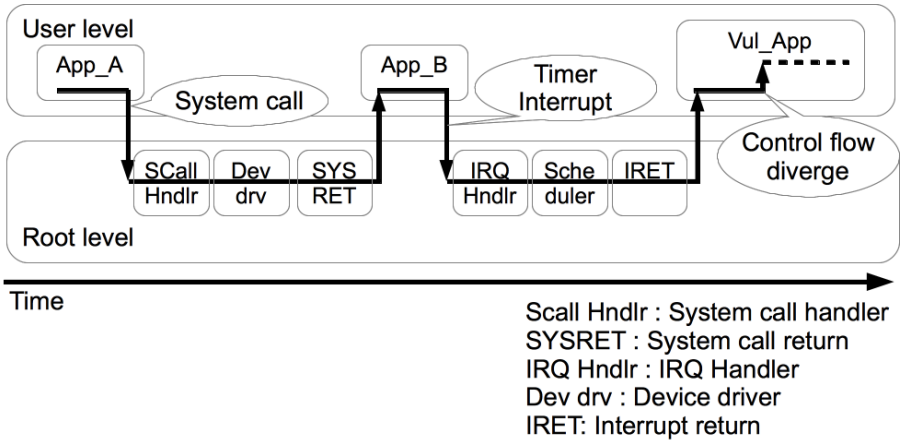


Figure 11: Privilege switchings between the root and user levels
While device drivers are executed at the root level, the control flow is diverged into the injected payload (illustrated with the dotted line) without privilege switching.

to the current privilege level of the processor cannot be used to distinguish between legitimate runtime code execution and shell-code injection attacks.

This work heuristically addresses this problem under an assumption regarding the *attack channel* – a malicious payload is transferred to a victim application's memory space via the I/O channel. This assumption is based on the observation that malicious payloads are not generated internally but inserted by the intruders, and an infiltration typically occurs via an I/O operation.

In other words, if a suspicious machine code – detected by referring to the hit/miss statuses in the ITLB and DTLB – is found to originate from an I/O device, it is highly likely the code is an arbitrarily injected machine code. This work proposes to utilize a manifest that

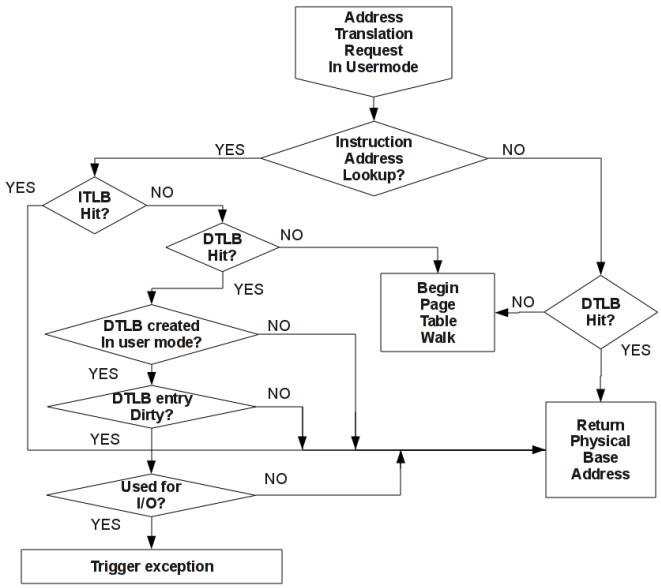


Figure 12: Flow chart for TLB Mon

contains page frame numbers used for I/O activity, for example, the `read` system call in Linux environments.

Figure 12 shows the overall detection flow of the proposed detection approach against shell-code injection attacks.

4.3 Experiments

4.3.1 Bochs

We used the Bochs-x86 emulator for our simulation. With Bochs, programmers can change the internal behavior or configuration of an emulated x86 processor, and instrumentation can be added so that users can monitor under-surface activities such as exception triggering. Bochs

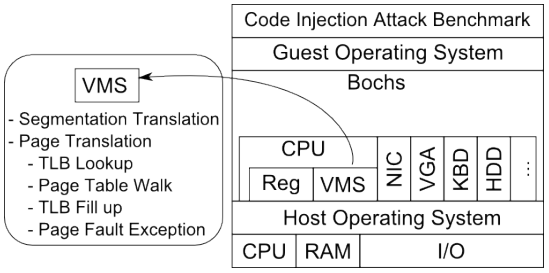


Figure 13: Hierarchy of Bochs, MMU and paging unit – Virtual Memory System

also emulates the x86 architecture's segmentation and paging structure, in which users can modify memory access behavior or the TLB configuration (Figure 13).

4.3.2 Modifications to the TLB

According to the Intel x86 manual, the TLB entry also has the same access control information as the PTE and PDE. The Bochs' implementation does also. However, the simulator has a unified TLB that is different from modern processor architecture: every address translation is cached in one TLB array with 1024 entries. To simulate our feature in a unified structure, we assigned two bits in the access control attribute to indicate which translation the entry has been used for – code, data, or both.

4.3.3 I/O monitoring

In order to implement the buffer-address logging feature described previously, instructions used for the system call request (`INT 0x80` and `SYSENTER`) are instrumented and the pointer – i.e. buffer address – for the `read` operation is logged inside the simulator on the fly. In the x86 architecture, this pointer is saved in one of the registers. Because it is nearly impossible

to know what application or server process is being exploited at runtime, every `read` operation needs to be monitored. In order to manage this log, we tested with two structure types – a binary tree and a hash table.

4.3.4 Attack simulation

In Wilander et al. (49), buffer overflow attacks are implemented in various configurations with the shell code fixed in the data segment.

In this benchmark, insecure instructions such as `strcpy` are exploited to directly overwrite control flow data into vulnerable buffer layouts. Victimized control flow data are: the return address, the old base pointer, the function pointer assigned as a local variable, the `longjmp` buffer assigned as a local variable, the function pointer assigned as a function parameter, and the `longjmp buffer` assigned as a function parameter. In addition to single-stage attacks, multi-stage attacks (50) are provided. Multi-stage attacks exploit a pointer variable next to the input buffer as a "bridge" to compromise the control flow data while circumventing protection measures like StackGuard (11).

Riley et al. (51) modified this benchmark to allow users to inject a shell code into various regions of the address space – data, BSS, heap, or stack region. In addition to this code placement option, we added one more feature into this benchmark – a virtual device driver delivering a payload to the benchmark running at the user privilege level. In order to simulate a shell-code injection attack via an I/O device, this device driver provides a small shell code to the benchmark running via the `read` system call of a device driver interface.

	Guest System 1	Guest System 2
Linux distribution	Redhat 6.2	Gentoo 2007.0
Linux kernel version	2.2.14	2.6.22
Kernel release date	April 3, 2000	July 8, 2007
GCC version	2.96	4.1.1

TABLE II: VULNERABLE GUEST SYSTEMS

4.4 Results

Table II shows the two vulnerable operating systems we used as guest systems in our simulation. The code injection attacks were copied, compiled, and executed in each system.

4.4.1 Modified Wilander/Riley benchmark

As described in Section 4.3.4, a modified Wilander benchmark provides various types of code injection attack and we ran the benchmark 10 times per attack configuration – varying victimized control flow data and payload location – to see if our method can detect them. In Guest System 1, 14 attacks out of 18 were able to execute injected code, and the effective attacks were detected by our protection measure. Some of the attacks could not execute the injected shell code, since this depends on the target control flow data such as the return address or function pointer. In Guest System 2, 5 code injection attacks out of 18 were effective, and the other 13 attacks were thwarted at the OS level. For the remaining 5 attacks, our protection was able to detect them as it did in Guest System 1.

One noteworthy difference between the two guest systems is whether or not the ASLR was employed. In the Linux system, the basic memory layout determined during compilation is fixed

	Detected/Effective/Exploits				
	Return address (3)	Base pointer (3)	Function pointer (6)	Longjmp buffer (6)	Total
Guest System 1	1/1/3	1/1/3	6/6/6	6/6/6	14/14/18
Guest System 2	1/1/3	0/0/3	4/4/6	0/0/6	5/5/18
StackGuard	3/3	2/3	0/6	0/6	5/18
StackShield	3/3	3/3	0/6	0/6	6/18
ProPolice	2/3	2/3	3/6	3/6	10/18
Libsafe/libVerify	1/3	1/3	1/6	1/6	4/18

TABLE III: EVALUATION OF EFFECTIVENESS AND COMPARISON
Example: For Guest System 2, of the six attacks that exploit the function pointer variable, four of them were able to execute the injected shell code and our protection detected all of those four effective attacks. There were 10 iterations for each attack configuration in our experiments. Lower four results are from Qin et al. (34).

– the code section is filled with instructions, the data section is filled with initialized values, and the BSS section is used without initialization. Other sections (or segments), like the heap section, are located in general at runtime. However, legacy OS kernels like Guest System 1 use the fixed starting address for the stack segment, for example, `0xBFFFFFFF`. This fixed location for the stack segment allows adversaries to locate vulnerable control flow data easily. Therefore, newer OS kernels randomize the starting addresses of stack segments – as does Guest System 2. This is one of the reasons why Guest System 1 was more vulnerable to the attack benchmark.

In order to see if our protection would raise a false alarm for legitimate runtime generated/loaded code, we ran a couple of simple applications and algorithms[1] programmed in Java. The results showed that our protection regards instructions generated by the JVM as legitimate.

[1]Fibonacci, Hanoi tower, wget, File I/O

4.4.2 Storage overhead

As described in Section 4.3.3, the addresses of buffers used in the `read` operation were logged to distinguish legitimate runtime generated/loaded code from injected code. In order to assess the storage overhead of this logging operation, we ran a secure shell server process in the simulator and tried to connect and exchange files with a remote client PC while JVM was executing simple algorithms or the code injection attack simulator was running. We found that every attack was detected while the runtime generated/loaded code was not detected as a code injection attack. However, we found that it was hard to assess the storage overhead for this log.

First we logged the input buffer addresses in a tree structure to find out how big the manifest would be, and the result showed that its size varied from 1000 to 2000 entries regardless of the client or server behavior. There are several explanations for this: (1) some buffers can be recycled depending on the control flow within one context; and (2) the OS kernel randomly assigns the virtual address space for buffer areas.

This result and analysis suggest that a hash table organization is a better choice over a tree structure. Even though a hash table has the inherent weakness of collisions, results with 256 and 512 entry hash table implementations showed that it was effective in managing the buffer address log and in detecting code injection attacks under the same conditions as the tree structure.

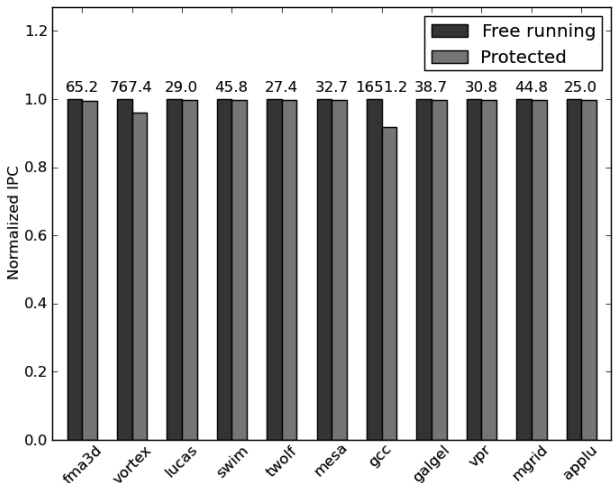


Figure 14: SPEC2K benchmark results from modified SimpleScalar
Numbers on top of each cluster show the number of ITLB misses per 1 million translation requests. Average degradation was 0.68%. Only benchmarks results showing more than a 0.1% performance degradation are presented.

4.4.3 Performance overhead

Figure 14 shows the performance degradation from the proposed protection. The performance degradation in the IPC from the TLB lookup delay is negligible except for the GCC (8.31%) and the vortex (3.91%). This is because the overall ITLB hit ratio is so high that the delay caused by the address translation going through the DTLB under an ITLB miss does not affect the overall performance, and the number of ITLB misses on top of each cluster in the figure supports this reasoning. The performance overheads from the other types of software-based protection shown in Table III are negligible except for libVerify, with a 25% slowdown in program execution time.

The performance impact from other changes – manifest operation and the modified page table walk – was not assessed in this work due to the limitations of our simulation environment. However, several observations on existing implementations of related features imply that the performance impact would be insignificant. Firstly, considering that the cost of a system call ranges from 350 clocks to more than 1000 clocks (52), and the manifest update could be performed in parallel, we expect that the overhead from a manifest update would be insignificant. Moreover, the modified page table walk also would not affect overall performance very much because our modification only requires a few conditional branches, while there are usually already conditional branches in the finite state machine implementation for a page table walk.

4.5 Discussion

4.5.1 Revision of paging unit

Future work for TLB Mon is to make this protection approach detect instruction fetch attempts to modified memory pages in the swap region. When memory utilization of a system exceeds a certain threshold value, the OS kernel selectively moves modified memory pages of some address spaces to the memory swap, which is usually a storage peripheral like a hard disk drive, and brings them back later when applications in those address spaces try to access the swapped-out pages.

If a memory page containing a malicious payload happens to be swapped out and later the processor core tries to access the page for an instruction fetch, the detection flow shown in Figure 12 cannot detect the instruction fetch attempt because neither of the TLBs has

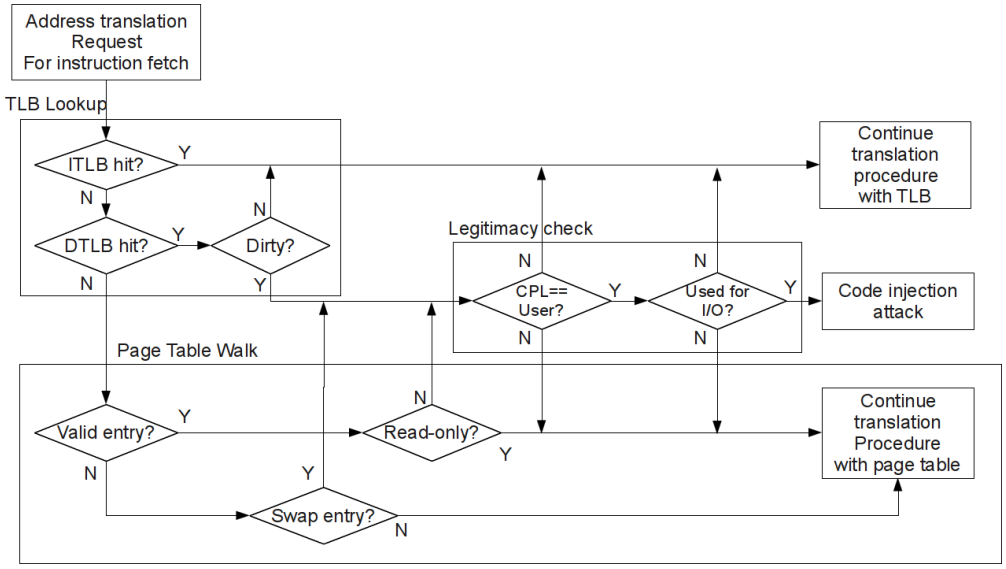


Figure 15: Revised version of TLB Mon
The protection flow detects instruction fetch attempts for swapped-out memory pages

corresponding entries. This means that we need to extend the proposed detection scheme to the paging unit, which fills the TLBs by traversing paging tables.

Figure 15 shows the revised version of the proposed scheme. Basically, this detection flow is the same as Figure 12 except for conditional branch orders and the paging table walk – the DTLB hit entry with a "dirty" bit under the ITLB miss. The lower half is the extended part – conditional branches checking the "valid" attribute and the swap-entry format are added for the detection flow. Most modern OS kernels that can handle swapped-out pages exploit invalid page translation entries to invoke their page fault exception handlers. By checking whether or not the page translation entry is in the swap entry format, this detection flow is able to detect instruction fetch attempts to swapped-out modified memory pages.

To verify the effectiveness of this revised detection flow, we modified the same simulator that we used for the TLB Mon experiments and ran the same attack benchmark as before. One difference in the experiments was that we ran heavy-load applications in background so as to increase the possibility that the OS kernel would swap out memory pages containing injected payloads. The experiments showed the same results as Table III regardless of the location of an infected page frame – in main memory or swap memory.

4.5.2 Limitations

The detection approach proposed in this section has two limitations: coarse granularity and vulnerability to circumvention techniques exploiting existing machine codes. This section discusses these limitations.

4.5.2.1 Coarse granularity

Because this scheme is based on translation information, its supporting granularity is fixed by the hardware configuration, usually 4KB. This granularity limitation mandates the page separation between code and data, because the proposed protection depends on the hit/miss status in the two TLBs. In legacy platforms, only OS kernels utilize shared memory pages. This memory utilization was to minimize the memory footprint of the OS kernels and their extensions (such as device drivers). In addition, in order to ensure the integrity of user level applications, memory pages containing machine codes were protected with read-only attributes.

However, since the computing power of modern microprocessors has improved drastically and the portability of applications across various platforms is now necessary – especially for runtime environments like web browsers – from the late 1990s software vendors began to use

runtime-generated codes. Since runtime environments running at the user privilege level compose machine codes in memory pages allocated from OS kernels and execute them, OS kernels are less able to control memory pages containing binary executables. The heap-spraying attacks discussed in Section 2.1.5 exploit this security loophole.

One limitation of the proposed approach regarding such runtime environments is that this approach could generate false alarms for memory pages containing both legitimately written machine codes and I/O data. In general, runtime environments request memory pages in chunks using `malloc` or `mmap` and use their own memory allocator to manage memory pages containing raw data and runtime-generated codes. Because neither the OS kernel nor the processor knows which page contains runtime-generated codes, our detection flows shown in Figure 12 and Figure 15 are highly likely to trigger false alarms if a memory page happens to contain legitimate machine codes and I/O data.

4.5.2.2 Vulnerability against attacks exploiting existing machine codes

The other limitation is that the protection approach is unable to counter sophisticated attacks exploiting existing machine codes. In Section 2.1.6, we discuss attack vectors designed to circumvent NX-based protection and Section 3.2 briefly mentioned the vulnerability of protection approaches utilizing address translation mechanisms.

In our detection flows shown in Figure 12 and Figure 15, a DTLB with a dirty bit set is essential in detecting an arbitrarily injected code. However, circumvention techniques, like return-to-libc or return-oriented programming attacks, exploit existing machine codes without shell-code injection (Section 2.1.6). Because exploited codes are legitimately prepared by the

OS kernel and binary loaders, corresponding entries in the active paging table do not have their dirty bits set.

This means that even if an ITLB miss is triggered for a machine code, into which a malicious injected payload is trying to hijack the program flow, the DTLB will not have a corresponding entry, therefore the detection flows presented in this section are unable to detect such attacks

CHAPTER 5

MEMORY-ACCESS VALIDATION AGAINST PAYLOAD INJECTION ATTACKS IN MULTI-TASKING ENVIRONMENTS

In Section 3.2, we discussed payload injection attacks from an architectural standpoint and found that two hardware miss handlers – the L1 I-cache miss handler and the RAS miss handler could be utilized to counter payload injection attacks. We also discussed that we need auxiliary data to determine whether or not a miss event in those handlers is caused by a malicious attack and presented three requirements for such data to be incorporated into existing platforms. Based on the discussion, we propose a memory-access validation scheme against payload injection attacks for multi-tasking environments.

Figure 16 gives an overview of our validation scheme. Basically, the proposed scheme consists of two components – the validation unit and taint status data. The validation unit handles queries from other processor components by managing and referring to the taint status data associated with the active address space. We have designed these components to address the issues and requirements discussed in the previous chapter.

5.1 Taint storage format

For the proposed validation scheme, we devise storage formats for taint status data. A good data structure for taint status data is critical because its instances must be stored in main

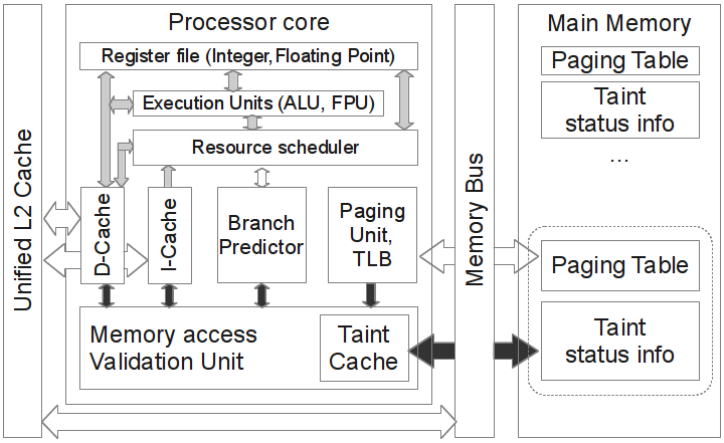


Figure 16: Overview of the proposed validation scheme

memory and those instances must have a fixed size for the hardware validation unit to traverse in constant time. This section develops our data structure based on the requirements discussed in Section 3.2.5.

5.1.1 Fine granularity and memory page frame

As is discussed in Section 3.2.5.1, it is essential to support a granularity smaller than a page frame size in a memory-access validation scheme. Moreover, the OS kernel allocates/de-allocates memory page frames on the fly, and access attempts to unallocated address ranges trigger exceptions. Based on these circumstances, we propose a two-level approach for our validation scheme – the first level dealing with page frames and the second level dealing with the memory blocks in the page frames listed in the first level. Only if our validation unit finds that the page frame that a queried memory block belongs to contains spurious data, would it proceed to the next level for dealing with memory blocks in the page frame. Otherwise,

the queried memory block is considered "authentic" and instruction execution continues. In this work, we have chosen a virtual address space as our reference address space to support multi-tasking OS kernels.

5.1.2 Matrix format for page frame numbers

For the first level of taint status data, the proposed scheme manages page frame numbers (PFNs) of page frames containing spurious data in matrix format – as an array of arrays. Each PFN is placed in its assigned sub-array – in matrix term a "row", and specified bit fields of a PFN are used as the row index. There are several benefits of this format: (a) the validation unit can manage PFNs like cache lines in set-associative caches with replacement policies; (b) whether or not a page frame contains spurious data can be quickly determined using a linear search over its assigned row; and (c) constant access time is guaranteed because the target row of a PFN is fixed.

We design our validation scheme to manage a fixed number of PFNs in each matrix – 32, 64, 128, or 256. This estimation is based on our experimental result with I/O-intensive applications (GCC toolchain, Apache web server, and sshd server) that more than 99.8 percent of I/O-active address spaces utilize fewer than 256 pages for I/O. The I/O-active address space in this work refers to an address space that directly communicates with I/O devices through system calls like `read` during execution time.

5.1.3 Validation granularity and taint status information in bitmap format

In this work, we chose the cache line size for validation granularity. Managing the taint status at the granularity of the cache line size has several advantages: (a) cache hit/miss

statuses could be utilized for taint status management; (b) we can put only validated memory blocks into a cache so that cache lines in the cache no longer needed to be validated. The limitation of this coarse granularity compared to IFT approaches will be discussed in Section 5.6.1.

In this work, we chose the bitmap format for our second level storage format. Each bit of a bitmap indicates the taint status of the corresponding memory block (with cache line size) in a page frame that is listed in the associated PFN matrix. Because there are 32, 64, 128, or 256 PFNs in a PFN matrix, the same number of bitmaps for those PFNs will be stored in an array. For simplicity, we designed a bitmap array to be placed right next to the PFN matrix.

5.1.4 Matrix/bitmap location

Section 3.2.5.2 discusses several issues regarding the taint status data – isolation between address spaces and its location. In this work, we propose to concatenate our taint data (one PFN matrix and one bitmap) to one of the paging tables of a virtual address space to be protected to address those issues (Figure 17). Such an augmented data structure facilitates isolation of taint status data between address spaces and synchronization between address space changes and taint status data switching. In other words, our validation unit would be able to locate the active PFN matrix and its bitmap array without an additional special purpose register for taint status data.

The location of the queried memory block's taint status is determined as follows. As explained previously, each PFN is placed in its assigned row and specified bit fields of a PFN are used as the row index in the matrix. Because each bitmap in the bitmap array corresponds to

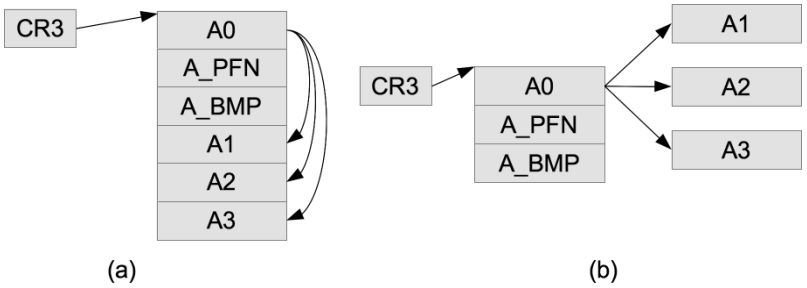


Figure 17: Translation table organization augmented with taint status data set
The validation unit is able to locate the taint status data set – one PFN matrix and one bitmap array – by adding a translation table size to CR3 value. The paging unit still traverses translation data from the CR3 register without being influenced by taint status data.

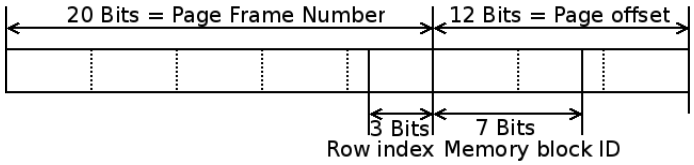


Figure 18: Row index and memory block ID assignment example
32-bit architecture with 4KB page frame and 32B cache line, $8 \times N$ page frame number matrix

a PFN in the matrix, we use the coordination of a PFN in the matrix to calculate the index for the associated bitmap stored in the bitmap array. For the memory block level, we use the MSBs of the page offset of the memory block's address as the identifier within a page frame. Figure 18 shows an example of row assignment and memory block ID extraction.

Table IV and Figure 19 illustrate a decomposition example of a virtual address `0x0871c230`.

Virtual address	`0x0871c230`
Page frame number	`0x0871c`
Page offset	`0x230`
Row index	`0x0871c & 0x7` = 4
Memory block ID	`(0x230 >> 5) & 0x7F = 0x11` = 17

TABLE IV: VIRTUAL ADDRESS AND MATRIX/BITMAP COORDINATION
0x871c230 using 8×4 page frame number matrix and row/memory block ID assignment in Figure 18.

5.1.5 Memory space requirement

The memory requirement for two sets of taint information for one address space – one PFN matrix and one bitmap array – can be calculated with the following equation (CLS : Cache line size / PGN : Number of page frame numbers / BMP : Bitmap).

$$\underbrace{4 \times PGN}_{PFNMatrix} + \underbrace{\frac{4096}{CLS} \div 8 \times PGN}_{BMParray} \tag{5.1}$$

For the matrix structure, we assign one 32-bit word for the PFN matrix element and utilize higher bit fields (20 bits) for the PFN and the remaining lower bit fields for other purposes – like checksums for PFNs and bitmaps. Depending on the matrix configuration, memory consumption of these storage formats could go up to 5KB (32B cache line/256 PFNs) or 3KB (64B cache line/256 PFNs) per address space. In case of the 32-bit x86 architecture, at least 8KB are required to initialize one virtual address space, and this size increases as more memory page frames are allocated. This means that the memory overhead in kernel address space for

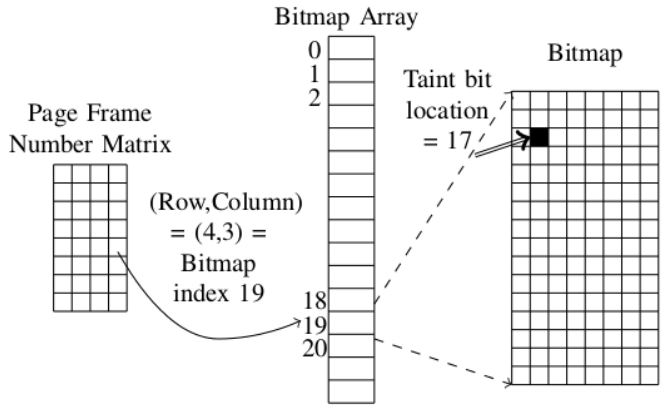


Figure 19: Locating the taint status of the memory block
The virtual address 0x0871c230 is decomposed to locate corresponding taint status data based on virtual address decomposition in Table IV

address space initialization would account for 62.5%/37.5% and would decrease as more page frames are allocated.

We believe that our approach for taint status data augmentation has three advantages: (a) the memory-access interface does not require non-trivial modifications like Minos because taint status data are contained in main memory; (b) hardware designs required for implementation – like the LRU, bitmap – are already available; and (c) taint status data need to be allocated and initialized only once during the initialization of a protected address space.

5.1.6 Supporting 64-bit architecture processors

The 64-bit architecture is widely adopted in modern processors – especially in desktop and server processors. Basically, the address width of virtual addresses affects only our PFN matrices and not the bitmaps because the page frame size remains unchanged. At first glance,

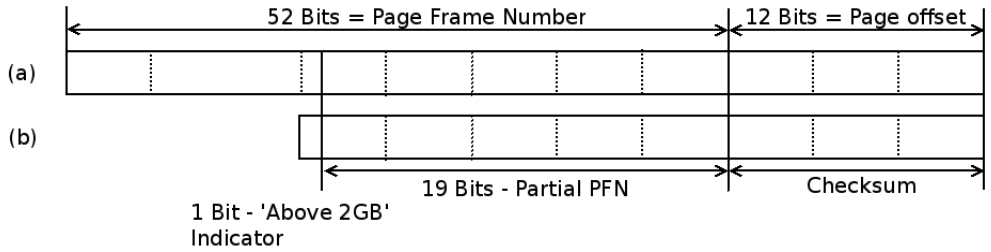


Figure 20: 64-bit virtual address (a) and its 32-bit PFN matrix element (b)

doubling the matrix size seems to be a simple and efficient way for our scheme to support 64-bit architecture processors. However, this extended format is highly likely to incur an access overhead for PFN matrices, because each PFN matrix element is doubled in bit-width and because our validation unit does a linear search on the row pointed to by the row index field of a PFN. Furthermore, such an access delay is critical in supporting the caching structure for our validation scheme discussed in Section 5.2.4. In order to address this problem, this work proposes a shortened format for PFN matrix elements for 64-bit architecture. This format stores only the lower 19-bits of a PFN in its corresponding PFN matrix element and uses the remaining 1 bit to indicate that the actual PFN points to the memory region over 2GB (Figure 20). Row assignment and block ID extraction are the same as explained in the previous section.

This PFN abbreviation may have aliasing due to its shortened format. In order to see whether or not this approach incurs aliasing in PFN matrices, we ran extensive aliasing tests with 64-bit Linux OS kernel, the gcc compiler (4.5.3), and Linux source codes. In all PFN matrix configurations, the results did not show any aliasing. This is because of the virtual

address layout of the OS kernel. For the Linux OS kernel, only lower-half of the 64-bit canonical virtual address space (53) is utilized by user-level applications. In addition, the kernel rarely assigns virtual address ranges that incur aliases in our address abbreviation approach. Based on this simulation result, we propose to abbreviate PFNs for 64-bit addressing to a 32-bit shortened format.

5.2 Memory-access validation unit

The validation unit has two roles – one is to handle queries from other processor components and the other is to update taint status data. Other processor components, such as the cache miss handlers, invoke taint operations for memory blocks in the validation unit. The validation unit processes these queries and returns the results by updating and referring to the active taint status data. This section discusses the overall working flow of the validation unit.

5.2.1 Taint operation

Three operations are performed on the taint information: Paint, Erase, and Lookup. Other hardware components issue operation queries to the validation unit and this unit returns results.

If a requested PFN is found in the PFN matrix, the Paint operation sets the bit field representing the taint status of a memory block that is written with spurious data transferred via I/O. Because user applications request I/O operations from the OS kernel through the system call interface, the straightforward approach to activating the Paint operations is to make the hardware unit handling system calls invoke the Paint operation with arguments like the starting address of an I/O buffer and its size. The Erase operation clears the bit field that corresponds to a memory block being overwritten with data from a register. This operation is necessary

because a memory block that once contained spurious data may be updated with reliable data later and corresponding changes must be applied to the taint status data. The Lookup operation answers validation queries issued from other hardware components by referring to the bit field representing the taint status of a requested memory block.

The procedure followed after failing to locate a requested PFN in the matrix varies depending on the requested operation. Only the Paint operation puts the PFN into its assigned row in the matrix (either by finding an available spot or replacing an existing one) and initializes the linked bitmap in the bitmap array. If the Erase or Lookup operation fails to locate the PFN in the matrix, the validation unit returns a miss. The hardware component that issued the query ignores the response and resumes execution. Note that frequent misses do not necessarily mean that this matrix structure is unsuitable for managing PFNs. This is because queries that result in misses include those targeting memory pages that have never been involved in an I/O operation.

5.2.2 Integration into memory hierarchy and the return address stack

In Section 3.2.2 and 3.2.3, we discuss how two architectural miss handlers – the L1 I-cache miss handler and the RAS miss handler – could be utilized in countering payload injection attacks. This section presents our approach to integrating our taint-based validation mechanism into those handlers and how it works.

As discussed in Section 3.2.2, validating an instruction fetch attempt at the L1 instruction cache miss handler could be utilized for countering shell code injection attacks. In terms of our taint operations, having the miss handler issue the Lookup operation for the missed cache line

can counter those attacks. Payloads for shell code injection attacks contain machine code to be executed after control flow redirection. As an injected shell code has never been executed before at control flow redirection, an I-cache miss should be triggered. The miss handler then invokes a Lookup operation for the missed memory block address and our validation unit returns that the queried address is marked as spurious or not by referring to the taint status data.

We discussed that utilizing the RAS miss handler could detect stack-compromising attacks (Section 3.2.3). As is widely known, the processor core pushes return addresses for `call` instructions into the RAS and the branch predictor pops contained addresses out of the stack for speculative execution. This stack contains a limited number of return addresses in a circular buffer. In the following explanation, we use Figure 8 as an example and assume that the processor core has already pushed "Return_to_B", followed by "Return_to_A", into the RAS just before the attack.

In return-to-libc attacks, a stack frame is overwritten with a crafted payload through a buffer overflow. This payload contains the starting address of a library procedure to be exploited and arguments arranged for the procedure as illustrated in (c) in Figure 8. During a stack overflow, the starting address usually replaces a return address previously pushed by the processor – "Random", which is pointed to by the stack register for a brute-force attack. As the corrupted return address in the stack frame would differ from the corresponding entry in the RAS – "Random" in the stack frame against "Return_to_A" in the RAS – an RAS miss should be triggered to discard operands updated during the speculative execution based on branch prediction by the RAS. During this discarding operation, the RAS miss handler issues

a Lookup query, for the memory block address of the memory word containing the problematic return address, to our validation unit. The unit traverses the active taint status data set and returns that the queried memory block address is marked as spurious.

The recent prevalence of 64-bit architectures makes brute-force return-to-libc attacks less threatening because of the increased entropy compared to 32-bit architectures (22). However, the stack region is still attractive to adversaries as demonstrated in return-oriented-programming attacks (10). Our claim is that our validation scheme is effective against those attacks as well.

The same reasoning for the interaction between the RAS and return-to-libc attacks can be applied to return-oriented-programming attacks. As noted in (10), the easiest way to place a payload filled with *gadgets* into a victimized process address space is through a stack overflow. Each gadget consists of a pointer to instruction sequences concluding with `return` and data words referenced by those instruction sequences for a basic operation like load, store, or addition.

For example using (d) of Figure 8, assume that "Addr_code_k" points to a code snippet of (`pop %edx; ret;`). When the control flow is redirected to this snippet, the `pop` loads the value "Imm_word_k" into the `edx` register. The `ret` instruction in the snippet reads "Addr_code_l" for its next instruction address. In the given example, at least two mismatches would incur RAS miss predictions during execution of the payload – "Return_to_A" in the RAS against "Addr_code_j" in the stack frame and "Return_to_B" against "Addr_code_k". Therefore, our validation approach with the RAS miss handler is effective against ROP attacks.

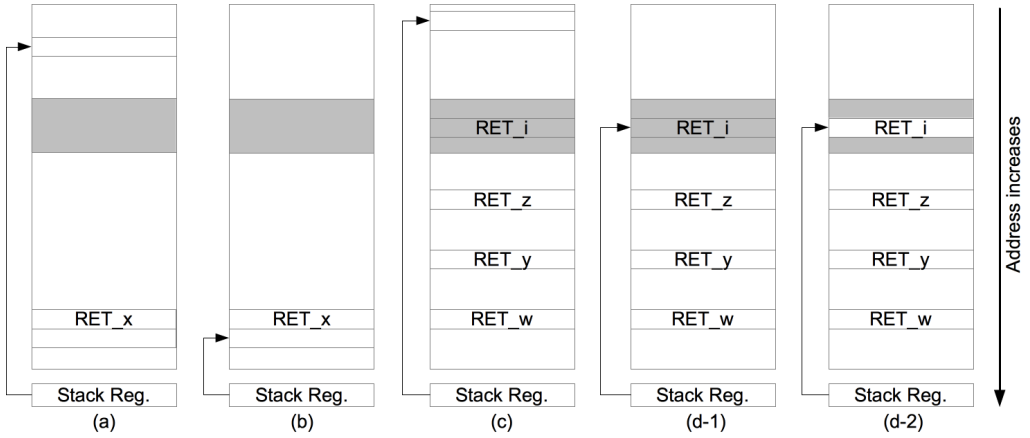


Figure 21: Call chain and taint status data
Gray block represent memory blocks marked as "tainted": (a) only a small portion of a large stack frame is used for bufer I/O; (b) control flow returns to its caller; (c) multiple functions are called; (d-1) legitimate return address (RET_i) is recognized as tainted after an RAS miss; (d-2) False positive could be prevented by clearing the taint bit associated with the stack word containing (RET_i) at function call.

The RAS must issue not only validation queries but also Erase queries to prevent false alarms from legitimate miss events. In Section 3.2.4, we present two legitimate miss events in the RAS mechanism – an RAS buffer underflow and `longjmp` utilization. Under these events, the processor core is forced to fetch the missed return address from the active stack frame. This means that the RAS must ensure the legitimacy of return addresses pushed by `call` instructions. We will assume the following scenario:

1. A memory block in a stack frame is used as an I/O buffer. ((a) in Figure 21)
2. The control flow of the procedure that used the I/O buffer returns to its caller (b).
3. One or more nested procedures are invoked by the caller (c).

4. One of the nested procedures happens to have its return address in the memory block previously used for the I/O buffer (c).
5. A context switch occurs and the RAS is flushed.
6. The nested procedure exits and the RAS miss is triggered.
7. The RAS miss handler invokes a Lookup operation for the memory block containing the missed return address (d-1).

After Step 1, the active taint status data should indicate that the memory block used for I/O contains spurious data. Without the correct status update along with the `call` instructions in Step 3, our validation unit will trigger a false alarm at the last step even if the queried memory block contains a legitimate return address. The straightforward resolution to this problem is to make the RAS invoke the Erase query for every `call` instruction – in the given example, at Step 3.

Legitimately invoking `longjmp` will not trigger a false alarm in our validation scheme, because the `longjmp` implementation does not manipulate stack frames in main memory but restores execution contexts of the processor – i.e. hardware register values including the stack register – from the `jmp_buf` buffer. As the stack pointer register will point to another uncompromised word in the stack frames, our validation scheme will not trigger a false alarm for legitimate `longjmp`s.

While the Lookup operation is invoked by the L1 instruction cache miss handler and the RAS miss handler for control flow validation, the Erase operation should be invoked to clear the taint status of memory blocks updated with reliable data - in the same way that `call`

instructions must invoke Erase operations for the RAS buffer. One problem in invoking Erase operations is that if there are too many queries for memory blocks in the L1 data cache, then it is highly likely that a performance overhead will be incurred. Our solution to this problem is to utilize the L1 data cache miss handler. By having the L1 data cache miss handler invoke Erase operations, the validation unit based on the proposed scheme would be able to reset the taint status of memory blocks being updated using fewer queries.

5.2.3 Block copy and taint transfer

Conforming copy dependency (Section 3.2.5.3) necessitates a sophisticated mechanism other than merely exploiting the cache miss handlers and the RAS miss handler. This is because Paint and Erase operations at the system call interface and the miss handlers will not transfer the taint status of a memory block being copied. This section presents our approach to addressing this problem by utilizing a processor component.

In general, string copies use two pointer variables to point to two memory locations – the source character and the destination character and increase those variables to advance to the next character locations until the program flow encounters a loop exit condition – an end-of-string character in the source string or the number of characters to be copied.

Basically, it is very hard to distinguish between the variety of pointer and arithmetic operations constituting block copies from other pointer-based memory accesses and arithmetic operations. This work proposes a taint status transferring mechanism that monitors memory accesses in loops and transfers taint statuses of a source string (or memory block) to the destination taint status location for string copies. This approach is based on the observation that

string (memory block) copies are performed in loops and their basic blocks are simple and small. For the GNU C library version 2.14 for x86-32, the loop size is 12B for `strcpy` (without loop-unrolling) and 61B for `strncpy` (with loop-unrolling). If the processor core is able to detect small-looped operations during instruction execution, a string (or memory block) copy detector would be implementable, with minor modifications, to monitor data accesses at the data cache controller side.

Our integration approach is to exploit the Loop-Stream Detector (LSD) found in recent Intel Core processors. The LSD is designed to optimize loop operation like calculation-intensive loops, searches and string moves by replaying decoded instructions in the processor pipeline based on branch predictions (42). There are several requirements that have to be met before decoded instructions can be locked down in the processor pipeline – such as the number of micro-ops (up to 28 instructions) and no function calls or returns inside the loop.

When the LSD locks down looped instructions, it lets our validation unit know that a small-looped operation has been detected. Our validation unit starts to monitor data cache accesses to analyze the queried access size and location to determine whether or not the loop being executed is a string (or memory block) copy. This analysis is simple and straightforward – two one-byte accesses through sequential addresses while alternating between read and write. If our validation unit detects a string copy operation, it reads the taint statuses of the source string and updates the taint statuses of the destination memory block accordingly.

One important issue for our validation scheme in utilizing the LSD mechanism is that we have to leave enough time between reading the taint status of the source string and updating

the taint status of the destination memory region. As mentioned earlier, our taint status data are contained in main memory, therefore our validation unit accesses them through shared data buses. Although this work proposes a caching structure for taint status data in the next section, the memory-access overhead caused by our validation unit must be alleviated.

Our mitigation approach for the memory-access overhead is to delay taint reading until the loop iteration count reaches a threshold value, and then read the taint status of the starting character of the source string, and write the taint status data at the loop exit. This procedure relies on two assumptions: (a) the taint status of the source string is uniform; and (b) any payloads span more than one cache line because of our taint granularity. We believe that these assumptions are plausible for our validation scheme because we are interested in carbon-copy relocation of payloads and it is highly likely that such payloads are longer than one cache line (64B).

In summary, with a minor extension of the LSD and the data cache controller, our validation scheme is able to transfer the taint status of relocated memory blocks.

5.2.4 Caching structure

We propose a caching structure for our taint status data. Our validation unit processes three taint operations on this cache structure and replaces PFNs in this cache structure, *not* in the matrices and the bitmap arrays in main memory. This structure allows our validation scheme to operate while not affecting existing (memory) cache operations, unlike CFI (Section 2.2.3).

The proposed caching structure consists of a row cache and a bitmap cache. The row cache stores the most recently accessed row and is filled with all of the PFNs in the row that a requested frame number belongs to. This row-loading enables our validation scheme to promptly determine whether or not the PFN of a requested virtual address is contained in the matrix (Section 5.1.2). If the row index of a requested PFN does not match that of PFNs in the row cache, modified PFNs in the row cache are written back to the PFN matrix in main memory. After write-back, all PFNs in the row that the requested PFN belongs to are loaded into the row cache. If a requested PFN is found in the row cache (either after a row cache hit or row cache loading), our validation unit proceeds to the bitmap cache. Otherwise, the unit returns a "miss" and aborts the processing of the query. The bitmap cache has the same number of slots as the number of elements in the row cache, and each slot is allocated for the linked bitmap of each PFN in the row cache. Unlike the row cache, each slot of the bitmap cache is filled with the original bitmap in main memory only when its corresponding PFN is requested. We use this request-based bitmap loading approach because the bitmap slot which will be referenced is unknown when loading the row cache.

All of the modified elements in row and bitmap caches must be written back to the original data in main memory at the following events: (a) row cache miss; (b) context switching; (c) interrupt/exception; and (d) cache control instructions (e.g. a cache flush). Three events other than row cache misses must invoke row/bitmap cache write-backs because they affect cache line statuses.

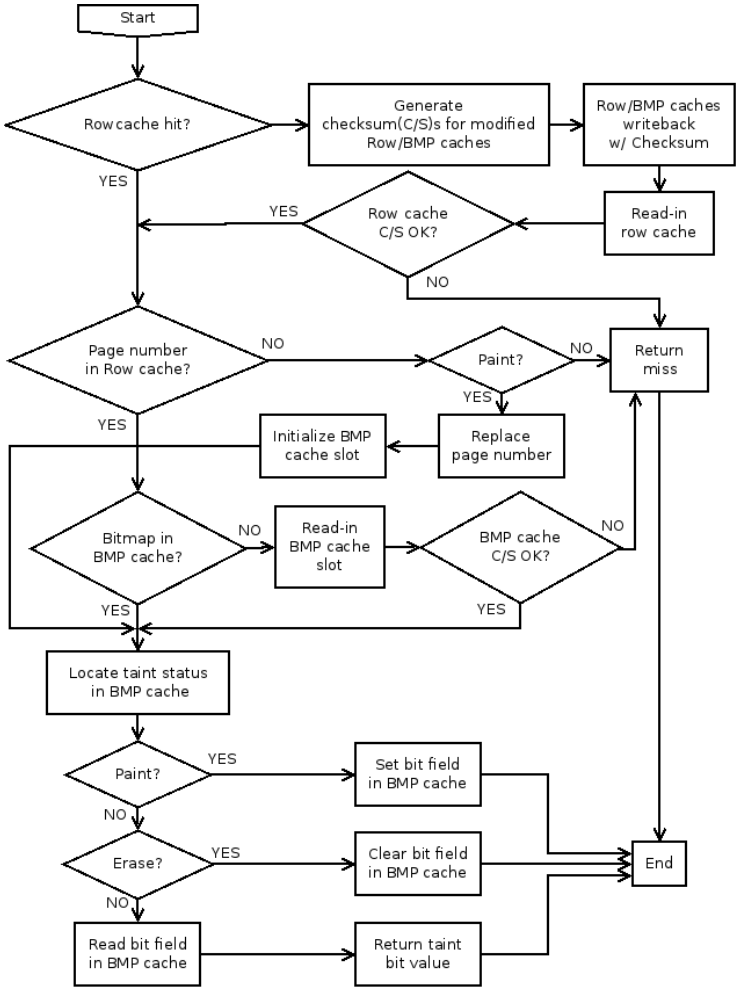


Figure 22: Taint operation flow of the memory-access validation scheme

Figure 22 shows the overall detection flow of the proposed validation scheme including our two-level caching structure.

5.3 System software support

As the OS kernel manages system resources for multi-tasking features, the proposed validation scheme needs system software support.

The first is memory allocation for the PFN matrix and the bitmaps per address space. If a process is to be protected with the proposed validation scheme, the OS kernel has to allocate memory space for its taint information – one PFN matrix and one bitmap array. Because taint status data for one address space are concatenated to one of the paging tables as proposed in Section 5.1.4, the OS kernel can easily manage memory regions allocated for taint status information. The OS kernel must initialize one PFN matrix and its associated bitmap array with zero values, I/O activities must be logged and referenced only by our validation unit.

The second required support is to handle shared memory pages. Multi-tasking OS kernels extensively utilize shared memory pages for various purposes, like shared library procedures and the Inter-Process Communication feature. Because we assign one taint information set per address space, taint status changes occurring in memory pages of one address space are unknown to other address spaces sharing those pages. This problem could be addressed by making the OS kernel duplicate taint statuses from one address space to another at critical events like context switching. The OS kernel is responsible for this synchronization because the kernel manages the page sharing information.

5.4 False positives

The proposed scheme has two false positives cases. This section briefly discusses how such false positives could occur and how to eliminate them.

The first false positive is from the matrix structure. Because of our row assignment (Figure 18), some PFNs could be removed from a PFN matrix and a "miss" will be returned if a removed PFN is requested for an Erase or Lookup operation. This means that Lookup misses will result in false positives because program execution must continue. Due to the limited number of elements in our matrix structure, we cannot fundamentally eliminate such false positives. We have two ways to reduce the false positive rate – one is to use larger matrices and the other is to adopt a row assignment that would evenly distribute PFNs over rows.

The second case is caused by cache line size granularity. If a part of a string buffer allocated for spurious data happens to share the same memory block (with a cache line size) with a memory word containing a return address and an RAS miss prediction is observed due to a circular buffer underflow, a false alarm will be triggered. During our experiments with the Bochs simulator, we observed false alarms caused by this problem. A remedy for this problem is to organize memory layouts in a way such that taint-candidate data and control flow data are placed far enough apart not to share a cache line. This work-around could increase memory consumption for an extended memory layout as a side effect.

5.5 Comparison with IFT

This thesis borrowed the concept of "taint-status data" from the IFT approaches (Section 2.2.10). In those works taint-status data is referred to as "taint tags" or "taint bits". This section briefly compares the proposed scheme with other architectural approaches in two aspects.

5.5.1 Word-size granularity and cache-line-size granularity

In contrast to our target granularity of the cache line size, general IFT approaches support finer granularity like 4B. Supporting such a sub-cache-line granularity requires non-trivial changes to the processor core – especially the cache structures. This is because the cache line size in modern microprocessors is 32B or 64B and the existing cache controllers are designed to support these cache line sizes. On this structure, monitoring and controlling memory accesses at word-size 4B/8B granularity (for 32/64-bit architecture) or 1B requires signal/control mechanisms to resolve taint bit fragmentation at the sub-cache-line level.

For example, assume that a dirty cache line in the L1 data cache is to be written back to the L2 cache and only one 4B word in the cache line got tainted while stored in the L1 cache. In this case, the taint tag of the modified 4B word and the buffered tags of remaining 4B words must be put together into one block and stored in the tag storage – like a taint cache inside the processor or an external storage like a separated DRAM proposed in Minos. The sub-cache-line level modifications to support these mechanisms would be significant and are highly likely to incur performance degradation. Some IFT works try to address this performance issue (34)(47)(36) using optimization techniques.

Since our scheme chooses the cache line size for its target granularity, such fragmentation would not occur, therefore asynchronous events in the cache structure – like cache misses – could be directly used to issue taint queries for the associated cache lines. Our claim is that this simplified structure would be more adaptable to the existing memory hierarchy than

IFT approaches. One problem for this granularity configuration is that our scheme may have granularity issues as discussed in Section 5.4.

5.5.2 Augmentation of the memory system

Minos (32) employs the 33rd-bit augmentation approach, which requires two significant modifications to the memory-access interface: (a) a separate DRAM to contain taint tags; and (b) a widened memory bus for taint tags. Unlike Minos, our scheme stores taint status data in main memory and the validation unit fetches and stores the matrix rows and associated bitmaps using the two-level cache structure, which accesses main memory through the existing memory-access interface.

Our two-level approach is an efficient format compared to multi-granularity tags proposed in (31). In the multi-granularity tags approach, the OS kernel must allocate memory space for security tags when a memory block in a newly allocated frame becomes tainted. The authors claim that granularity switching from one to another (for example, all-tainted-page to one-byte granularity) occurs infrequently; therefore, the overall runtime overhead is insignificant. However, in terms of its hardware implementation, processing more than two granularities would require dedicated procedures in hardware design to handle each of the granularity configurations.

Unlike the multi-granularity tags approach, our scheme allocates memory pages for taint status data during binary loading only and revokes those pages at address space deletion. In addition, with two widely known storage formats in each level of validation, our approach would not require complicated implementations, unlike the multi-granularity tags approach.

Our approach has two downsides though – the false positive from PFN matrices (Section 5.4) and a vulnerability to rootkit attacks (Section 5.6).

5.6 Vulnerability and limitations

5.6.1 Vulnerability of taint information

This validation scheme is vulnerable to rootkits compromising a target system with the same access level as its OS kernel. A plausible attack scenario is that a rootkit attack duplicates a sparsely marked bitmap to other bitmap locations. If a rootkit identifies or crafts a sparsely marked bitmap and populates the bitmap array of a vulnerable process with this bitmap using the duplication procedure described in Section 5.3, our validation scheme would not be able to detect attacks accessing memory blocks whose taint status information has been compromised.

5.6.2 Limitations

The proposed validation scheme has two limitations. One limitation is that our protection scheme is unable to detect payload relocation using hardware-optimized instructions. In Section 5.2.3, we described our taint transfer mechanism leveraging the LSD and the data cache controller to transfer taint statuses of copied memory blocks. This mechanism is effective only if memory blocks are copied through looped procedures. While we ran our simulators with a 64-bit OS kernel and applications, we found that block copies were frequently done with a few loop iterations or sometimes even without loops. This is because some software modules generated for x86-64 architecture processors utilize XMM/AVX (previously SSE) instructions for string copies. These instructions are able to move one 16/32B memory block with only two instructions (one for read and one for write), therefore optimized string or memory block copy

procedures can move memory blocks with fewer data move and branch instructions compared to looped instructions. The LSD would not be activated for these XMM/AVX instructions unless they are heavily looped, and our validation scheme would not transfer the taint statuses of memory blocks moved by hardware-optimized instructions.

Another limitation is that our validation scheme cannot detect attacks exploiting indirect branches other than `return`s – like `(call *)` or `(jmp *)`. During an indirect branch execution, the next instruction address contained in the data cache is loaded into the program counter either via a general purpose register (a register indirect jump) or directly from the cache (a memory indirect jump). Because these branches do not trigger RAS misses and our scheme does not validate data cache accesses, our validation unit is unable to detect exploitation of indirect branches for return-to-libc-style or ROP-without-returns attack (9).

5.7 Experiment

This section discusses how we have evaluated the effectiveness and performance impact of the proposed validation scheme.

5.7.1 Experimental environment

We used two simulators in our experiment – the Bochs x86 simulator and SimpleScalar. The Bochs simulator was for large-scale statistics from I/O intensive applications running on a multi-tasking environment and for effectiveness assessment. The SimpleScalar simulator with architectural parameters was to assess the performance impact from our validation scheme.

Architectural configuration	Value
Matrix size	32 / 64 / 128 / 256 PFNs
Number of columns in PFN matrices	4 / 8 columns
Replacement policy in PFN matrices	LRU / FIFO / Random
Page frame size	4K Bytes
Cache line size	64B
L1 Instruction cache	32K Bytes, 4-way set assoc. LRU
L1 Data cache	32KB, 8-way set assoc. LRU
Return Address Stack	Circular buffer with 32 entries

TABLE V: BASELINE ARCHITECTURE CONFIGURATION

Table V shows the baseline configurations applied to both simulators. These architectural parameters, like the cache line size, were adopted from the Intel Core i5 processor (2.67GHz, 45nm).

Bochs x86 simulator

The Bochs x86 simulator is an emulator that fully simulates an x86 ISA as well as peripherals like network interface cards. Because this simulator provides instrumentation features that enable programmers to monitor instruction executions and the processor's internal behavior, it is widely used in computer architecture research projects.

We have modified the instrumentation procedures in our Bochs simulator to have architectural features as shown in Table V. To those features, we have added our validation mechanism and the caching structure discussed in Chapter 5.

The taint transfer mechanism discussed in Section 5.2.3 was also implemented as discussed. For this feature, we have emulated the LSD mechanism as described in (42) with the branch

instruction instrumentation and made the emulated LSD invoke taint operations accordingly – read the taint status at string copy detection and issue either Paint or Erase for relocated memory blocks at loop exit.

In order to create and delete PFN matrices and associated bitmap arrays, our simulator was instrumented to monitor address space switches and deletions. For Paint queries, `read` system calls from user applications were trapped to query Paint operations with I/O buffer addresses passed as arguments in the hardware registers. We installed and executed two target OS kernels out-of-the-box – one for the 32-bit mode and the other for the 64-bit mode.

Using the Bochs x86 simulator and two multi-tasking OS kernels, we ran two I/O intensive jobs for evaluation – the Linux kernel compilation with the GCC compiler and the dacapo Java benchmark. The GCC toolchain takes advantage of the multi-tasking environment for parallel compilation and high throughput, and handles various sizes of I/O operation. The Java benchmark executed runtime-generated codes from `.class` files. These resource utilization characteristics were helpful in evaluating various aspects of this validation scheme like the miss rates of PFN matrices. Other simulation configurations are shown in Table VI.

SimpleScalar

The SimpleScalar simulator allows programmers to add new features and to apply architectural parameters to an emulated environment so that they can evaluate the performance impact of their architectural modifications. Unlike the Bochs simulator, SimpleScalar emulates various architectural features like cache structures, branch predictors, and an out-of-order ex-

Simulator configuration	Value
Emulated CPU	Pentium 4
DRAM	256MB
Virtual HDD	6GB
Linux Kernel	2.6.31 from Gentoo Linux
GCC compiler	4.3.4
Apache version	2.2.17

TABLE VI: ARCHITECTURAL PARAMETERS FOR SIMULATION

ecution engine. After a simulation execution, the simulator provides statistics of the emulated environments such as IPC (Instructions-Per-Clock) and branch prediction hit rates.

We have modified the simulator as we did the Bochs simulator – the cache miss handlers and the RAS miss handler were modified to invoke Lookup and Erase queries. One limitation of SimpleScalar is that this simulator executes applications in single-threaded mode and does not utilize any multi-tasking features. This means that the simulator does not assess the performance impact from multi-tasking features and I/O operations and it cannot evaluate the performance impact from the system call interface like `read` system calls.

Instead of implementing our own system call interface in SimpleScalar, we modified the simulator to flush both the row cache and the bitmap cache at every system call request from the simulator to the host system. This is to create a conservative simulation environment.

Table VII shows the architectural parameters we applied in our SimpleScalar simulations. The architectural parameters for memory hierarchy like cache hit latencies were adopted from the same processor referenced for our Bochs simulator and a DDR3 memory interface running

Architectural parameters	Value
Fetch/Decode Issue/Commit width	4 / 4 / 4 / 4 per cycle
L2 Unified cache	256 KB, 8-way set assoc., LRU
L1 latency	4 cycles
L2 latency	42 cycles
DDR3 memory latency (First-chunk / inter-chunk)	72 / 2 cycles
Row cache / Bitmap cache hit latency	42 / 42 cycles
Number of instructions executed	10,000,000

TABLE VII: ARCHITECTURAL PARAMETERS FOR SIMPLESCALAR
Cache and RAS configurations are same as Table VI).

at 666MHz (1333 MT/s) with an 8-burst-deep pre-fetch buffer. For latencies of row/bitmap caches, we assumed that those caches were implemented with the same circuitry as L2 caches; therefore, the same latencies were used. We ran the SPEC2K benchmark suite for performance impact assessment.

5.7.2 Effectiveness evaluation

We verified the effectiveness of the proposed scheme with two synthesized attacks on the modified Bochs simulator. For this evaluation, we used two benchmark suites – one for shell code injection attacks explained in Section 4.3.4 and the other for an ROP attack. The shell code injection attack benchmark was executed 10 times per attack configuration varying victimized control flow data and payload location like in the previous experiment.

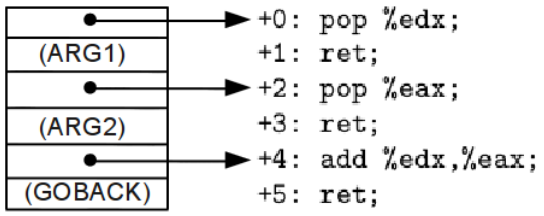


Figure 23: Payload layout and code snippets
The payload contains three gadgets (left-hand), and the code snippets are implanted in the heap region. "GOBACK" is a pointer value that passes control flow to another location of the exploited process after addition.

The ROP attack test suite was our own and executed (harmless) code snippets only by overflowing a stack frame with a payload containing three gadgets from (10). The code snippets shown in Figure 23 were implanted in the heap region before overflowing the buffer so as to emulate existing machine codes, each of which ends with a `return`. With four gadgets crafted for these codes, we could add values contained in the payload (`ARG1` and `ARG2`) instead of executing basic blocks written in the code region and could make the control flow return to the exploited procedure.

The modified Bochs simulator was able to detect all of the synthesized attacks. For the code injection attack benchmark, our simulator detected executions of injected payloads through the I-cache miss handler regardless of the type of the exploited control flow data and the location of the injected shell code (Table VIII). The modified simulator also successfully detected all of the de-references of the pointer values contained in the payload for the simulated ROP attacks as RAS mispredictions.

	Detected (/Effective) / Exploits				
	Return address (3)	Base pointer (3)	Function pointer (6)	Longjmp buffer (6)	Total
Mem-validation	1/1/3	1/1/3	6/6/6	5/5/6	6/6/18
Low-overhead IFT	3/3	2/3	0/6	0/6	5/18
StackGuard	3/3	2/3	0/6	0/6	5/18
StackShield	3/3	3/3	0/6	0/6	6/18
ProPolice	2/3	2/3	3/6	3/6	10/18
Libsafe/Libverify	1/3	1/3	1/6	1/6	4/18

TABLE VIII: EFFECTIVENESS EVALUATION AND COMPARISON
Example: In case of the proposed scheme under 6 attacks that compromise longjmp buffer, 5 out of 6 attacks were able to execute injected shell code and our protection detected all of those 5 effective attacks. 10 iterations for each attack configuration in our experiments. Lower 5 results are from Qin et al(34).

5.7.3 Matrix structure evaluation

The most important metric regarding the matrix format is the miss rate of the Lookup operations, because PFN matrices have to reliably manage PFNs used for I/O. Experimental results were collected only from our modified Bochs simulator. This is because the input files for SPEC2K benchmarks running on SimpleScalar were not large enough to populate PFN matrices and do not utilize any multi-tasking features. For reliable statistics from our modified Bochs simulator, more than 400 I/O-active address spaces were profiled for each matrix configuration in our experiments. The percentages for each operation were: 51.34 percent from Lookup, 48.65 percent from Erase, and less than 0.001 percent from Paint.

First, we checked what row assignment approach would be efficient in distributing PFNs in rows in PFN matrices. This issue is briefly discussed in Section 5.4. The row assignment is

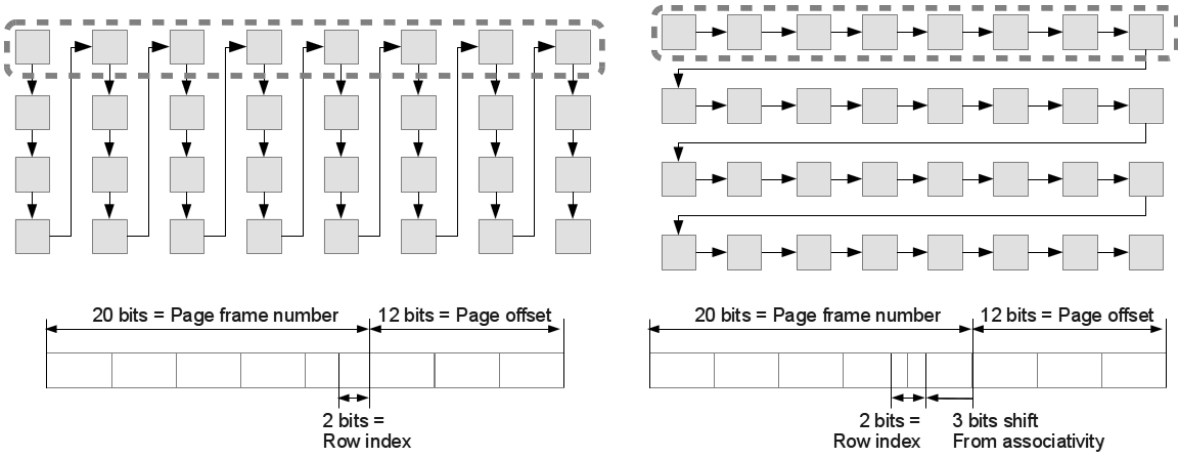


Figure 24: Row distribution for contiguous PFNs in a 4×8 PFN matrix
Vertical assignment versus horizontal assignment. Each block stands for a PFN element.

important in managing PFNs reliably because distributing PFNs unevenly is highly likely to incur frequent thrashing of frequently assigned rows. As proposed in Section 5.1.2, specified bit fields of a PFN are referenced as its row index. The row assignment exemplified in Figure 18 puts contiguous PFNs in different rows. Another approach is to refer higher bit fields than Figure 18 so as to put adjacent PFNs in the same row. In terms of the row assignment direction for contiguous PFNs, we refer to the former approach as "vertical assignment" and the latter approach as "horizontal assignment" (Figure 25).

Each approach has advantages and disadvantages. The vertical assignment distributes adjacent PFNs more evenly than the horizontal assignment, therefore it is less prone to thrashing of PFNs. One disadvantage of this assignment is that we cannot exploit the spatial locality of PFNs in the row cache structure. When the PFN next to the current PFN is requested, both

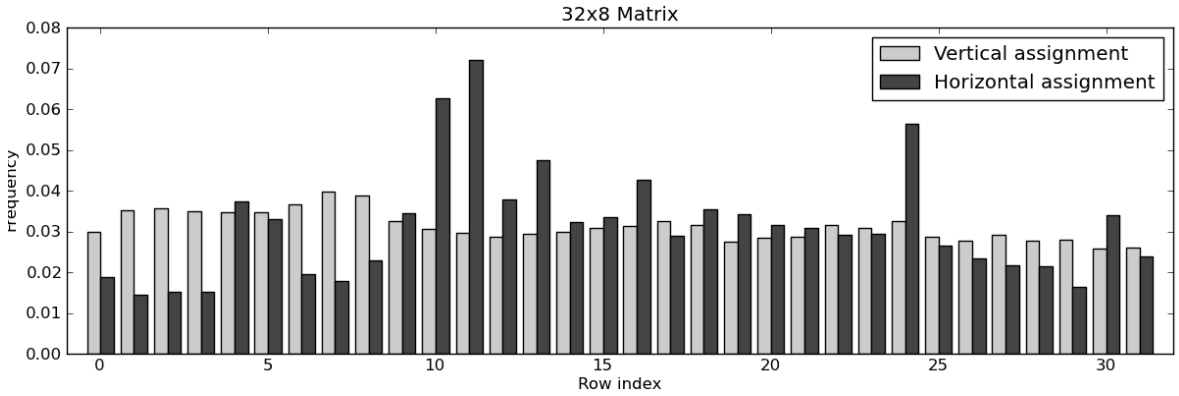


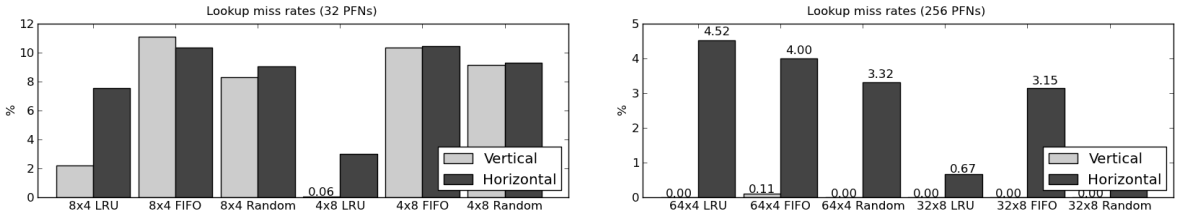
Figure 25: PFN assignment distribution in each row from the GCC toolchain
X axis – row index, Y axis – normalized frequency.

the row cache and the bitmap cache must be written back because the next PFN will be in the next row of the active PFN matrix.

On the other hand, horizontal assignment can take advantage of the spatial locality between adjacent PFNs. One problem of this approach is that there could be frequent thrashing if PFNs are distributed randomly rather than in order.

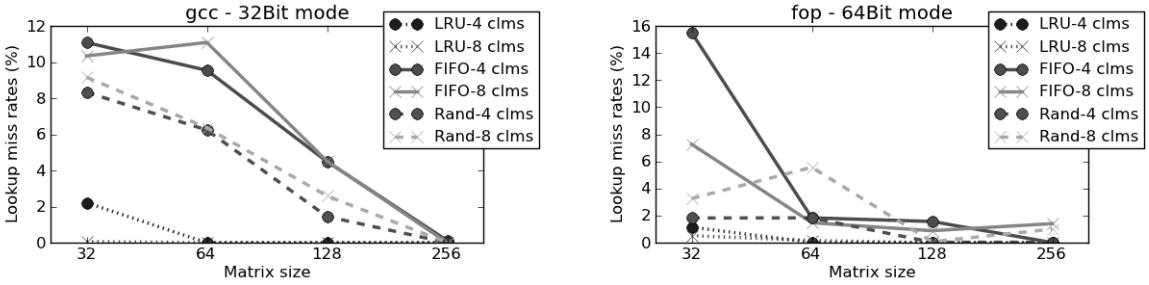
Figure 25 illustrates the distribution of PFNs in each row of the GCC toolchain. The 64×4 matrix configuration had a similar distribution as this figure. As shown in this figure, horizontal assignment distributes PFNs unevenly compared to vertical assignment. If a more frequently assigned row happens to be requested for a Lookup operation, it is highly likely that the row will have Lookup misses, which result in false positives.

Figure 26 compares Lookup miss rates between vertical assignment and horizontal assignment over two matrix configurations. The results from 32-PFN matrices showed quite high miss



(a) 32-PFN matrix (b) 256-PFN matrix

Figure 26: Comparison of two row assignments for Lookup misses
Vertical versus horizontal. Smaller is better.



(a) The GCC toolchain (b) The dacapo Java benchmark (fop)

Figure 27: Lookup miss rates in the 32-bit mode environment. Smaller is better.

rates regardless of replacement policies and row assignments. However, when we increased the matrix size to 256 elements, the vertical assignments showed nearly zero miss rates while the horizontal assignments still showed considerable miss rates. Our understanding is that virtual addresses randomized by ASLR incurred frequent thrashing in PFN matrices and such rows are prone to Lookup misses. Subsequent experimental results will be shown only for vertical assignments unless otherwise noted.

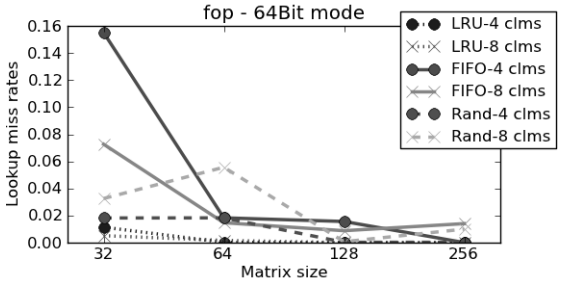
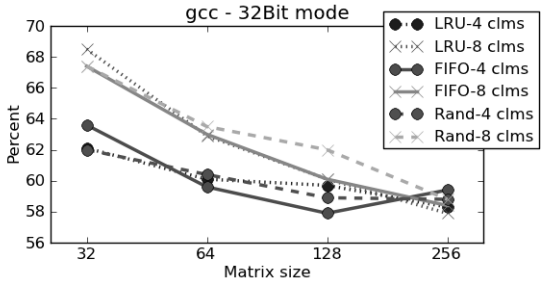


Figure 28: Lookup miss rates for the dacapo Java benchmark (fop)
In 64-bit mode. Smaller is better.

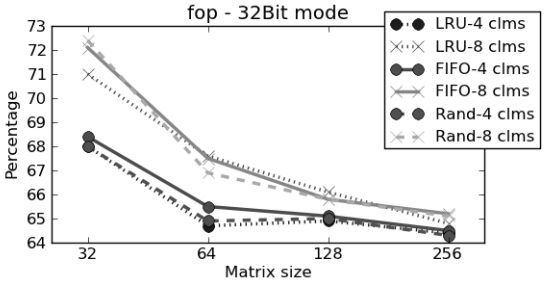
Figure 27a through Figure 28 illustrate Lookup miss rates of the proposed validation scheme for two benchmarks – the GCC toolchain and the dacapo Java benchmark. We found that: (a) smaller matrices were susceptible to Lookup misses regardless of replacement policies; (b) the LRU replacement policy outperformed the other two policies by several orders of magnitude; and (c) a PFN matrix for one address space had to contain at least 64 entries to ensure low miss rates with LRU. One noteworthy result from the Java benchmark is that runtime-generated codes are prone to Lookup misses with replacement policies other than LRU. Our understanding of this result is that those replacement policies replace PFNs regardless of whether a PFN is frequently requested for taint operations therefore it is highly likely that popular PFNs are removed frequently.

5.7.4 Row cache and bitmap cache evaluation

We evaluated the cache structure proposed in Section 5.2.4 using two statistics – the hit rates of the row and bitmap caches and the distribution of the write-back size under a row

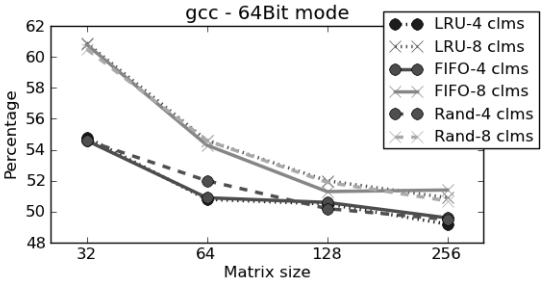


(a) The GCC toolchain

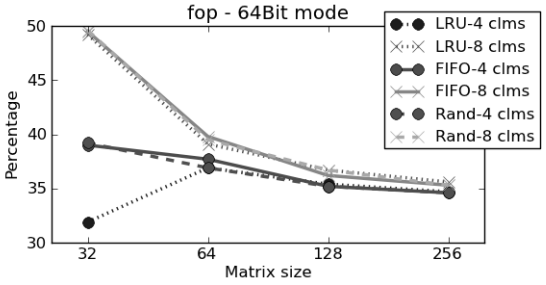


(b) The dacapo Java benchmark (fop)

Figure 29: Row cache hit rates in the 32-bit mode environment. Higher is better.



(a) The GCC toolchain

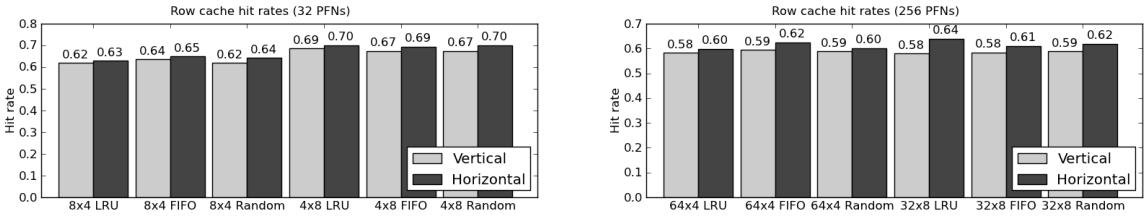


(b) The dacapo Java benchmark (fop)

Figure 30: Row cache hit rates in 64-bit mode environment. Higher is better.

cache miss. Only the results from the Bochs simulator are presented because as with matrix evaluation, the input sizes for the SPEC2K benchmarks for SimpleScalar were too small.

Figure 29 and Figure 30 show the row cache hit rates from the same benchmark used for the matrix evaluation. Overall, the row cache hit rates were not high – between 0.31 and 0.73, and 8-column matrices showed slightly higher hit rates than 4-column matrices. There was no difference in row cache hit rates for the replacement policies. This is because the choice of a



(a) 32-PFN matrix (b) 256-PFN matrix

Figure 31: Comparison of two row assignments in Row cache hit rates
Vertical versus horizontal. Higher is better.

replacement policy only affects which PFN is removed from a row – not which row in the active PFN matrix a PFN is located. The lowest hit rate in 30b (32-PFN matrix, LRU) was found to be statistical noise. Another experiment batch with the same configuration showed the same hit rates as other placement policies.

In the previous section, we mention that the horizontal assignment for PFN could be helpful in exploiting spatial locality among contiguous page frames. Figure 31 compares two row assignment approaches for the same matrix configurations - the 32-PFN matrix and the 256-PFN matrix. Contrary to our expectation, the actual advantage from horizontal assignment was insignificant – about 2 to 6 percent in the row cache hit rates. Considering the Lookup miss rates from the horizontal PFN assignment, these figures substantiate that the vertical PFN assignment is a better choice than the horizontal one.

The bitmap cache rates were high – 0.923 to 0.999 for all matrix configuration and replacement policies. This result is straightforward because taint operations are likely to occur in adjacent memory regions within a page frame.

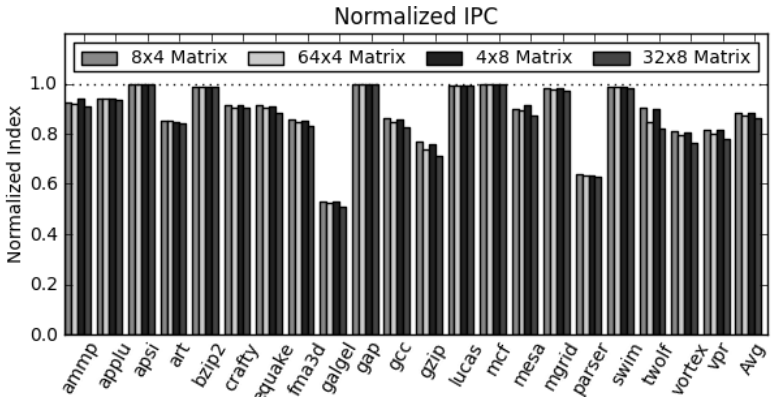


Figure 32: IPC from SPEC2K benchmark
64B cache line, 32/256 PFN matrix, and 4/8 columns

The write-back sizes for row cache misses vary depending on how many row/bitmap cache elements were modified. Regardless of replacement policies or matrix configurations, all of the write-back size distributions were similar to other configurations; one bitmap write-back (8B) accounted for more than 99.5 percent of the total write-backs. Other write-backs like one new PFN with one new bitmap were less than 1 percent.

5.7.5 Performance impact assessment

Figure 32 shows normalized IPC values. The average IPC degradation was 12.3 percent, and the largest degradation was 48 percent. Experiments with the 32B cache line size exhibited results similar to Figure 32 except for less degradation in several benchmarks.

Differences in performance degradation were mostly caused by queries and row cache misses. In case of *swim* with less than 1 percent degradation, only about 30,000 queries were issued while 10 million instructions were executed and the row cache hit rate was low – from 49.7 to

52 percent. On the other hand, *apsi* issued about 1.2 million queries, and the row cache hit rate was as high as 99.2 percent. This benchmark had no performance degradation. *Galgel* suffered the largest performance degradation; about 200,000 queries were issued, and the row cache hit rate ranged between 32.0 (64 pages) and 47.8 (128 pages) percent.

With the proposed caching structure, the read-in size for the row cache is 16B (4-column matrix) or 32B (8-column matrix) while one bitmap is 8 or 16B (Section 5.2.4). Although the row cache size is small compared to the cache line size (32/64B), row cache read-ins influenced the overall performance. There are several reasons for this performance degradation: (a) our row/bitmap cache structure does not have high-speed secondary storage; (b) the first-chunk latency of the DDR3 memory interface is large; and (c) row cache misses are triggered more frequently than system calls and context switches.

In summary, our experimental results with the modified SimpleScalar and SPEC2K benchmarks show that our validation scheme has a modest performance impact, mostly influenced by queries and row cache misses. Unfortunately, according to results presented in Section 5.7.4, our row cache hit rates were found to be low in the kernel compilation experiment. This implies that the performance degradation in I/O-intensive applications like kernel compilation or internet browsing could be significant.

CHAPTER 6

CONCLUSION

6.1 Summary of Thesis Work

Modern computing systems are vulnerable to payload injection attacks because of naive assumptions adopted in programming environments and because of several features of memory systems.

First, we propose a protection scheme against shell code injection attacks. We show that a viable protection measure can detect shell code injection attacks by monitoring address translation queries to two TLBs – instruction TLB and data TLB. The key observation on shell ocde injection attacks from an architectural standpoint is that Certain hit/miss statuses in those TLBs show that an arbitrarily written data is about to be accessed for instruction fetch. We develop a protection scheme based on this observation and verfify it effectiveness and performance impact.

Our second work is more fine-grained approach to address limitations in the first work. We have found that two acceleration techniques – the cache structure and the return-address-stack branch predictor – widely adopted in modern microprocessors could be utilized in countering payload injection attacks. As miss handlers of those components are invoked by unprecedented or unexpected circumstances, validating control flow redirections at these handlers could be effective in countering some types of payload injection attack. However, as not every instruction

cache miss and branch miss prediction would account for payload injection attacks, we need auxiliary data to assess the legitimacy of memory blocks referenced for control flow redirection. We have clarified the requirements for such auxiliary data, such as fine granularity and supporting multi-tasking environments.

Based on these discussions and observations, we have proposed a memory-access validation scheme countering two types of payload injection attack at the memory system level: shell code injection attacks and stack-compromising attacks. The proposed scheme consists of the validation unit and taint-status data set. The cache miss handlers and the RAS branch predictor issue queries to the validation unit at memory word updates and control flow redirections. The validation unit validates memory accesses for control flow redirection at lookup queries by referring to the taint-status data set associated with the active address space. A taint-status data set consists of one page frame number matrix and one bitmap array, and multiple instances of taint-status data are contained in main memory to support an arbitrary number of address spaces. By concatenating one taint-status data set to the top-level paging table in a virtual-physical address translation structure, our validation unit is able to locate the active taint-status data set without a dedicated hardware register. In order to alleviate performance degradation caused by frequent accesses to the active taint-status data contained in main memory, we have proposed a two-level caching structure.

Our experimental results have shown that our data structure for taint-status data is suitable for multi-tasking environments and shows very low miss rates with a fixed number of elements for a PFN matrix. To evaluate the effectiveness, we simulated payload injection attacks and found

that our system was able to detect all of the attacks. It was also shown that the performance impact from the proposed validation scheme is negligible to moderate depending on the row cache hit rates.

6.2 Open issues

While the proposed memory-access validation scheme against payload injection attacks shows effectiveness as well as moderate performance impact, there are several open issues for improvement.

1. Instances of the taint-status data in the kernel address space are vulnerable to rootkit attacks. We expect that protection measures against rootkit attacks implemented by anti-virus software could help mitigate threats from those attacks.
2. With an explicit policy to be enforced at the memory system level, validating data cache accesses could be helpful in countering payload injection attacks that compromise function pointer values referenced by indirect branch instructions like (`call *`) or (`jmp *`).
3. Currently the validation unit is designed to search the row cache first to see if a requested PFN is contained in the active PFN matrix. Whether or not a PFN is contained in the row cache could be promptly determined with an abbreviated representation of the contained PFNs such as a Bloom filter.
4. A high row cache hit rate is critical in alleviating the performance impact from the proposed scheme. A row assignment approach other than the ones given in this work could be helpful in achieving both high row cache hit rates and low lookup miss rates.

CITED LITERATURE

1. Microsoft Inc.: Protect yourself from conficker. http://www.microsoft.com/security/worms/conficker.aspx, 2008.

2. Microsoft Inc.: Microsoft security bulletin ms04-011. http://technet.microsoft.com/en-us/security/bulletin/ms04-011, 2004.

3. Moore, D., Shannon, C., and k claffy: Code-red: a case study on the spread and victims of an internet worm. pages 273–284, 2002.

4. McAfee: The new reality of stealth crimeware. www.mcafee.com/stealthcrimeware, 2006.

5. Christey, S.: Format string vulnerabilities in perl programs. http://seclists.org/bugtraq/2005/Dec/30, 2005.

6. Skypher: Heap spray generator. http://skypher.com/SkyLined/heap_spray/small_heap_spray_generator.html, 2005.

7. Jiang, X., Wangz, H. J., Xu, D., and Wang, Y.-M.: Randsys: Thwarting code injection attacks with system service interface randomization. In SRDS '07: Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems, pages 209–218. IEEE Computer Society, 2007.

8. MSISAC: Multiple vulnerabilities in adobe reader and acrobat could allow for remote code execution (apsb12-08). http://msisac.cisecurity.org/advisories/2012/2012-023.cfm, 2012.

9. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M.: Return-oriented programming without returns. In Proceedings of the 17th ACM conference on Computer and communications security, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

10. Roemer, R., Buchanan, E., Shacham, H., and Savage, S.: Return-oriented programming: Systems, languages, and applications. ACM Trans. Inf. Syst. Secur., 15(1):2:1–2:34, March 2012.

CITED LITERATURE (Continued)

11. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In Proceedings of the 7th USENIX Security Symposium, pages 63–78, 1998.

12. Lee, G. and Tyagi, A.: Encoded program counter: Self-protection from buffer overflow attacks. In International Conference on Internet Computing, pages 387–394, 2000.

13. Frantzen, M. and Shuey, M.: Stackghost: Hardware facilitated stack protection. In Proceedings of the 10th USENIX Security Symposium, pages 55–66, 2001.

14. Park, Y.-J., Zhang, Z., and Lee, G.: Microarchitectural protection against stack-based buffer overflow attacks. IEEE Micro, 26:62–71, July 2006.

15. Stack shield : A stack smashing technique protection tool for linux. `http://www.angelfire.com/sk/stackshield`.

16. Necula, G. C., Condit, J., Harren, M., McPeak, S., and Weimer, W.: Ccured: type-safe retrofitting of legacy software. ACM Trans. Program. Lang. Syst., 27(3):477–526, 2005.

17. Grossman, D., Hicks, M., Jim, T., and Morrisett, G.: Cyclone: A type-safe dialect of C. C/C++ User's Journal, 23(1), January 2005.

18. Etoh, H.: GCC extension for protecting applications from stack-smashing attacks (ProPolice), 2003. `http://www.trl.ibm.com/projects/security/ssp/`.

19. Tsai, T. and Singh, N.: Libsafe 2.0: Detection of format string vulnerability exploits. Technical report, 2001.

20. Baratloo, A., Singh, N., and Tsai, T. K.: Transparent run-time defense against stack-smashing attacks. In USENIX Annual Technical Conference, General Track, pages 251–262, 2000.

21. Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur., 13(1):4:1–4:40, November 2009.

22. Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D.: On the effectiveness of address-space randomization. In Proceedings of the 11th ACM

CITED LITERATURE (Continued)

conference on Computer and communications security, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

23. Riley, R., Jiang, X., and Xu, D.: An architectural approach to preventing code injection attacks. IEEE Transactions on Dependable and Secure Computing, 7:351–365, 2010.

24. Ratanaworabhan, P., Livshits, B., and Zorn, B.: Nozzle: A defense against heap-spraying code injection attacks. In Proceedings of the Usenix Security Symposium, August 2009.

25. Akritidis, P., Markatos, E. P., Polychronakis, M., and Anagnostakis, K.: Stride: Polymorphic sled detection through instruction sequence analysis. In 20th IFIP International Information Security Conference, 2005.

26. Polychronakis, M., Anagnostakis, K. G., and Markatos, E. P.: Network level polymorphic shellcode detection using emulation. In Proceedings of the Third international conference on Detection of Intrusions and Malware & Vulnerability Assessment, DIMVA'06, pages 54–73, Berlin, Heidelberg, 2006. Springer-Verlag.

27. Toth, T. and Kruegel, C.: Accurate buffer overflow detection via abstract payload execution. In RAID, pages 274–291, 2002.

28. Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N.: Native Client: A sandbox for portable, untrusted x86 native code. Communications of the ACM, 53(1):91–99, 2010.

29. Kc, G. S.: Countering code-injection attacks with instruction-set randomization. In Proceedings of the ACM Computer and Communications Security (CCS) Conference, pages 272–280. ACM Press, 2003.

30. Barrantes, E. G., Ackley, D. H., Forrest, S., and Stefanovic, D.: Randomized instruction set emulation. ACM Trans. Inf. Syst. Secur., 8(1):3–40, 2005.

31. Suh, G. E., Lee, J. W., Zhang, D., and Devadas, S.: Secure program execution via dynamic information flow tracking. SIGARCH Comput. Archit. News, 32:85–96, October 2004.

32. Crandall, J. R. and Chong, F. T.: Minos: Control data attack prevention orthogonal to memory model. In Proceedings of the 37th annual IEEE/ACM International

CITED LITERATURE (Continued)

Symposium on Microarchitecture, MICRO 37, pages 221–232, Washington, DC, USA, 2004. IEEE Computer Society.

33. Newsome, J. and Song, D. X.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In NDSS, 2005.

34. Qin, F., Wang, C., Li, Z., Kim, H.-s., Zhou, Y., and Wu, Y.: Lift: A low-overhead practical information flow tracking system for detecting security attacks. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 39, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society.

35. Nethercote, N. and Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not., 42:89–100, June 2007.

36. Kannan, H., Dalton, M., and Kozyrakis, C.: Decoupling dynamic information flow tracking with a dedicated coprocessor. In DSN, pages 105–114, 2009.

37. Shiv, K., Iyer, R., Newburn, C., Dahlstedt, J., Lagergren, M., and Lindholm, O.: Impact of jit/jvm optimizations on java application performance. Interaction between Compilers and Computer Architecture, Annual Workshop on, 0:5, 2003.

38. Gal, A., Eich, B., Shaver, M., Anderson, D., M, D., Haghighat, M. R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., and Corporation, M.: Trace-based just-in-time type specialization for dynamic languages.

39. Google Inc: Google chrome web browser. `http://www.google.com/chrome/intl/en/more/index.html`.

40. Shi, Y. and of Illinois at Chicago, U.: Architectural Support to Secure Program Execution. University of Illinois at Chicago, 2007.

41. Stiliadis, D. and Varma, A.: Selective victim caching: A method to improve the performance of direct-mapped caches. IEEE Transactions on Computers, 46:603–610, 1997.

42. Intel Corp.: Intel 64 and ia-32 architectures optimization manual. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`, 2011.

CITED LITERATURE (Continued)

43. Castro, M.: Securing software by enforcing data-flow integrity. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation, pages 147–160, 2006.

44. Lee, M.-S., Kang, Y.-J., Lee, J., and Maeng, S. R.: Opts: increasing branch prediction accuracy under context switch. Microprocessors and Microsystems, 26(6):291–300, 2002.

45. Kim, H., Joao, J. A., Mutlu, O., Lee, C. J., Patt, Y. N., and Cohn, R.: Virtual program counter (vpc) prediction: Very low cost indirect branch prediction using conditional branch prediction hardware. IEEE Trans. Computers, 58(9):1153–1170, 2009.

46. Slowinska, A. and Bos, H.: Pointless tainting?: evaluating the practicality of pointer tainting. In Proceedings of the 4th ACM European conference on Computer systems, EuroSys '09, pages 61–74, New York, NY, USA, 2009. ACM.

47. Dalton, M., Kannan, H., and Kozyrakis, C.: Raksha: a flexible information flow architecture for software security. SIGARCH Comput. Archit. News, 35(2):482–493, June 2007.

48. Ermolinskiy, A. and Shenker, S.: Design and Implementation of a Hypervisor-Based Platform for Dynamic Information Flow Tracking in a Distributed Environment. Doctoral dissertation, EECS Department, University of California, Berkeley, May 2011.

49. Wilander, J. and Kamkar, M.: A comparison of publicly available tools for dynamic buffer overflow prevention. In Proc. of the 10th Network and Distributed System Security Symposium, Feb 2003.

50. Piromsopa, K.: Defeating buffer-overflow prevention hardware, 2006.

51. Riley, R., Jiang, X., and Xu, D.: An architectural approach to preventing code injection attacks. In DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pages 30–40, Washington, DC, USA, 2007. IEEE Computer Society.

52. Vasudevan, A., Yerraballi, R., and Chawla, A.: A high performance kernel-less operating system architecture. In ACSC '05: Proceedings of the Twenty-eighth Australasian conference on Computer Science, pages 287–296, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

CITED LITERATURE (Continued)

53. Intel Corp.: Intel 64 and ia-32 architectures software developer's manuals. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, 2012.

54. Kang, M. G., McCamant, S., Poosankam, P., and Song, D.: DTA++: Dynamic taint analysis with targeted control-flow propagation. In Proceedings of the 18th Annual Network and Distributed System Security Symposium, San Diego, CA, February 2011.

55. Anonymous: Bypassing PaX ASLR protection. Phrack, 11(59), Jul 2002.

56. CPUID: Cache latency computation. http://www.cpuid.com/download/latency.zip.

VITA

Name

– Dongkyun Ahn

Education

– B.S. Electronic Engineering, Yonsei University, Seoul, South Korea, 1998

– M.S. Electrical and Computer Engineering, Yonsei University, Seoul, South Korea, 2000

– Ph.D. Electrical and Computer Engineering, University of Illinois at Chicago, Chicago, Illinois 2013

Experience

– Software Engineer, LG Electronics, Seoul, South Korea, 2000 – 2006

– Research Assistant, Department of Electrical and Computer Engineering, University of Illinois at Chicago, Chicago, Illinois. 2007 – 2008, 2011

Publications

– Dongkyun Ahn and Gyungho Lee : A Memory Access Validation Scheme against Payload Injection Attacks, 15th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID) 2012, September 2012.

– Dongkyun Ahn and Gyungho Lee : Countering code injection attacks with TLB and I/O monitoring 28th International Conference on Computer Design (ICCD) 2010, October 2010.

– Dongkyun Ahn and Gyungho Lee : Prospect of Fine Grain Dynamic Memory Access Control with Profiling Emerging Security Information Systems and Technologies (SECURWARE), July 2010.

– Dongkyun Ahn and Gyungho Lee : StackLock with simple FSM, IEEE International Conference on Electro/Information Technology, EIT 2009, June 2009.

– Gyungho Lee, Dongkyun Ahn, Youngjeong Ahn, and Yong-seok Lee: Detecting Code Injection Attack at TLB Miss, Submitted to Transaction on Computer, July 2012

– Dongkyun Ahn and Gyungho Lee : A Memory-Access Validation Scheme against Payload Injection Attacks, Submitted to IEEE Transactions on Dependable and Secure Computing, November 2012