An Architectural Approach to Preventing Code Injection Attacks

Ryan Riley, *Member*, *IEEE*, Xuxian Jiang, *Member*, *IEEE*, and Dongyan Xu, *Member*, *IEEE*

Abstract—Code injection attacks, despite being well researched, continue to be a problem today. Modern architectural solutions such as the execute-disable bit and PaX have been useful in limiting the attacks; however, they enforce program layout restrictions and can oftentimes still be circumvented by a determined attacker. We propose a change to the memory architecture of modern processors that addresses the code injection problem at its very root by virtually splitting memory into code memory and data memory such that a processor will never be able to fetch injected code for execution. This virtual split memory system can be implemented as a software-only patch to an operating system and can be used to supplement existing schemes for improved protection. Furthermore, our system is able to accommodate a number of response modes when a code injection attack occurs. Our experiments with both benchmarks and real-world attacks show the system is effective in preventing a wide range of code injection attacks while incurring reasonable overhead.

Index Terms—Code injection, secure memory architecture.

1 Introduction

Despite years of research, code injection attacks continue to be a problem today. Systems continue to be vulnerable to the traditional attacks, and attackers continue to find new ways around existing protection mechanisms in order to execute their injected code. Code injection attacks and their prevention has become an arms race with no obvious end in sight.

A code injection attack is a method whereby an attacker inserts malicious code into a running process and transfers execution to his malicious code. In this way, he can gain control of a running process, causing it to spawn other processes, modify system files, etc. If the program runs at a privilege level higher than that of the attacker, he has essentially escalated his access level. (Or, if he has no privileges on a system, then he has gained some.)

A number of solutions exist that handle the code injection problem on some level or another. Architectural approaches [1], [2], [3] attempt to prevent malicious code execution by making certain pages of memory non-executable. This protection methodology is effective for many of the traditional attacks; however, attackers still manage to circumvent them [4]. In addition, these schemes enforce specific rules for program layout with regard to separating code and data, and as such are unable to protect memory pages that contain *both*. Compiler-based protection mechanisms [5], [6], [7] are designed to protect crucial memory locations such as function pointers or return addresses and detect when they have been modified. These methods, while effective for a variety of attacks, do not provide broad-enough coverage to handle a great many modern vulnerabilities [8]. Both of these techniques, architectural and compiler-based, focus on preventing an attacker from executing his injected code, *but do nothing to prevent him from injecting and fetching it in the first place*.

The core of the code injection problem is that modern computers implement a von Neumann memory architecture [9]; that is, they use a memory architecture wherein code and data are both accessible within the same address space. This property of modern computers is what allows an attacker to inject his attack code into a program as data and then later execute it as code. Van Oorschot et al. [10] proposed a technique to defeat software self-checksumming by changing this property of modern computers (and hence producing a Harvard architecture [11], [12]), and inspired us to consider the implications such a change would have on code injection.

In this paper, we propose virtualizing a Harvard architecture on top of the existing memory architecture of modern computers, including those without non-executable memory page support, so as to prevent the injection of malicious code entirely. A Harvard architecture is simply one wherein code and data are stored separately. Data cannot be loaded as code and vice-versa. In essence, we create an environment wherein any code injected by an attacker into a process' address space cannot even be addressed by the processor for execution. In this way, we are attacking the code injection problem at its root by regarding the injected malicious code as data and making it unaddressable to the processor during an instruction fetch.

Based on the availability of hardware support for the execute-disable bit, our technique can be used as either a

- *R. Riley is with the Department of Computer Science and Engineering, Qatar University, PO Box 2713, Doha, Qatar. E-mail: ryan.riley@qu.edu.qa.*
- *X. Jiang is with the Department of Computer Science, North Carolina State University, 890 Oval Drive, Campus Box 8206, Raleigh, NC 27695. E-mail: jiang@cs.ncsu.edu.*
- *D. Xu is with the Department of Computer Science and CERIAS, Purdue University, 305 N. University Street, West Lafayette, IN 47907. E-mail: dxu@cs.purdue.edu.*

Manuscript received 20 Feb. 2008; revised 1 May 2009; accepted 30 Nov. 2009; published online 18 Jan. 2010.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-2008-02-0040. Digital Object Identifier no. 10.1109/TDSC.2009.1.

stand-alone code injection prevention and response mechanism or in conjunction with the hardware support for improved performance. Our current prototype is implemented as a software-only patch for the Linux operating system and incurs a reasonable performance penalty, on average, between 10 and 20 percent, when used in stand-alone mode. Such a software-only technique is possible through careful exploitation of the two translation lookaside buffers (TLBs) on the x86 architecture in order to split memory in such a way that it enforces a strict separation of code and data memory.

In addition, our technique detects a code injection attack at a unique moment when the injected attack code is ready to run but not executed yet. This unique timing provides the opportunity to accommodate a number of flexible *response modes*. For example, instead of just letting the system crash when a code injection attack occurs, we can choose to let it "proceed" properly in a monitored environment to gather information about the attacker's intention and tactics. In our current prototype, three response modes, i.e., *break*, *observe*, and *forensics*, have been implemented. The experiments with a buffer overflow benchmark suite as well as five attacks on real-world software vulnerabilities successfully demonstrate the effectiveness of our technique.

2 Related Work and Motivation

Research on code injection attacks has been ongoing for a number of years now, and a large number of protection methods have been researched and tested. There are two classes of techniques that have become widely supported in modern hardware and operating systems; one is concerned with preventing the execution of malicious code after control flow hijacking, while the other is concerned with preventing an attacker from hijacking control flow.

The first class of technique is concerned with preventing an attacker from executing injected code using non-executable memory pages, but does not prevent the attacker from impacting program control flow. This protection comes in the form of hardware support or a software-only patch. Hardware support has been put forth by both Intel and AMD that extends the page-level protections of the virtual memory subsystem to allow for non-executable pages. (Intel refers to this as the "execute-disable bit" [3].) The usage of this technique is fairly simple: Program information is separated into code pages and data pages. The data pages (stack, heap, bss, etc.) are all marked non-executable. At the same time, code pages are all marked read-only. In the event an attacker exploits a vulnerability to inject code, it is guaranteed to be injected on a page that is non-executable, and therefore, the injected code is never run. Microsoft makes use of this protection mechanism in its current operating systems, calling the feature Data Execution Protection (DEP) [1]. This method is effective for traditional code injection attacks in many scenarios; however, it requires hardware support in order to be of use. Legacy x86 hardware does not support this feature. It is also unable to protect memory pages containing both code and data. This technique is also available as a software-only patch to the operating system that allows it to simulate the execute-disable bit through careful mediation of certain memory accesses. PaX PAGEEXEC [2] is an open source implementation of this technique that is applied to the Linux kernel. It functions identically to the hardware-supported version; however, it also supports legacy x86 hardware due to being a software-only patch.

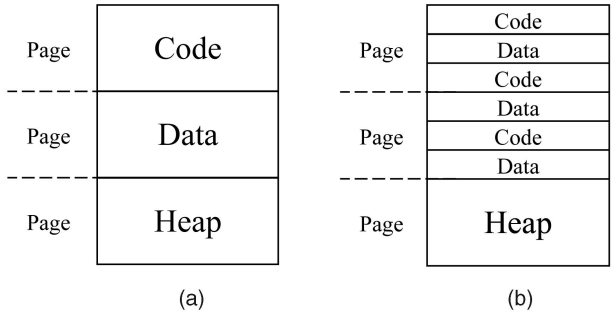


Fig. 1. (a) Separate code and data pages. (b) Mixed code and data pages in real-world software.

The second class of technique has a goal of preventing the attacker from hijacking program flow, but does not concern itself with the injected code. Works such as StackGuard [5] accomplish this goal by emitting a "canary" value onto the stack that can help detect a buffer overflow. ProPolice [6] (currently included in gcc) builds on this idea by also rearranging variables to prevent overflowed arrays from accessing critical items such as function pointers or the return address. Stack Shield [7] uses a separate stack for return addresses as well as adding sanity checking to `ret` and `call` targets. Due to the fact that these techniques only make it their goal to prevent control flow hijacking, they tend to only work against known hijacking techniques. That means that while they are effective in some cases, they may miss many of the more complicated attacks. Wilander and Kamkar [8], for example, found that these techniques missed a fairly large percentage (45 percent in the best case) of attacks that they implemented in their buffer overflow benchmark.

Due to the fact that the stack-based approaches above do not account for a variety of attacks, in this work, we are primarily concerned with addressing limitations in the architectural support of the execute-disable bit. While this technique is widely deployed and has proven to be effective, it has limitations. First, programs must adhere to the "code and data are always separated" model. See Fig. 1a for an example of this memory layout. In the event a program has pages containing both code and data (see Fig. 1b), the page-level protection scheme cannot be used. Unfortunately, these "mixed pages" do exist in real-world software systems. Sun's JavaVM loads some system library pages as both writable and executable. The Linux kernel uses mixed pages for both signal handling [13] as well as loadable kernel modules. Under Windows, applications protected by the SafeDisc DRM mechanism cannot make use of the execute-disable bit either [14]. Additionally, as operating systems begin to make use of larger page sizes [15], it becomes increasingly wasteful to strictly separate code and date onto different pages. Finally, mixed pages are easy to create. For example, the *mmap* system call in Linux allows programmers to set read, write, execute, or some *combination* of them for a memory object. The combination of write and execute accesses leads to mixed pages.

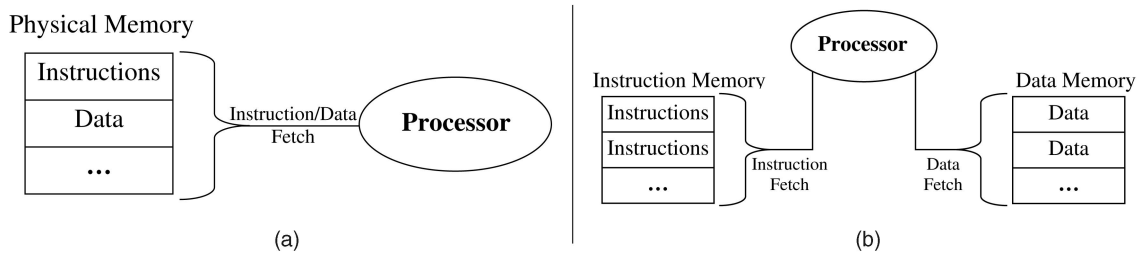


Fig. 2. (a) von Neumann architecture. (b) Harvard architecture.

A second problem with these schemes is that an advanced attacker can disable or bypass the protection bit using library code already in the process' address space and from there execute the injected code. Such an attack has been demonstrated for the Windows platform by injecting code into non-executable space and then using a well-crafted stack containing a series of system calls and library functions to cause the system to create a new, executable memory space, copy the injected code into it, and then transfer control to it. One such example has been shown in [4].

It is these two limitations in existing page-level protection schemes (the forced code and data separation and the above bypass methodology) that provide the main motivation for our work, which architecturally addresses the code injection problem at its core. Depending on whether certain hardware support is available, our system is designed to be run as either a stand-alone code injection prevention and response mechanism or in conjunction with a hardware-enforced execute-disable bit for improved performance.

Note that our architectural approach is orthogonal to research efforts on system randomization, such as Address Space Layout Randomization (ASLR) [16], [17], [18], [19] and Instruction Set Randomization (ISR) [20], [21], [22]. We are also distinct from other work that focuses specifically on preventing array overflow using a compiler or hardware, such as [23]. We point out that these alternate systems all work on a single memory architecture wherein code and data are accessible within the same address space. Our approach, to be described in the next section, instead creates a different memory architecture where code and data are separated.

3 An Architectural Approach

At its root, code injection is a problem because processors permit code and data to share the same memory address space. As a result, an attacker can inject his payload as data and later execute it as code. The underlying assumption relied on by attackers is that the line between code and data is blurred and not enforced. For this reason, we turn to an alternative memory architecture that does not permit code and data to be interchanged at runtime.

3.1 The Harvard and von Neumann Memory Architectures

Modern computers and operating systems tend to use what is known as a von Neumann memory architecture [9]. Under a von Neumann system, there is one physical memory which is shared by both code and data. As a consequence of this, code can be read and written like data and data can be executed like code. Many systems will use segmentation or paging to help separate code and data from each other or from other processes, but code and data end up sharing the same address space. Fig. 2a illustrates a von Neumann architecture.

An architecture not found in most modern computers (but found in some embedded devices or operating systems, such as VxWorks [24]) is known as a Harvard architecture [11], [12]. Under the Harvard architecture, code and data each have their own physical address space. One can think of a Harvard architecture as being a machine with two different physical memories, one for code and another for data. Fig. 2b shows a Harvard architecture.

3.2 Code Injection on Harvard Architecture

A code injection attack can be thought of as being carried out in four distinct, but related, stages:

1. The attacker injects code into a process' address space.
2. The attacker determines the address of the injected code.
3. The attacker somehow hijacks the program counter to point to the injected code.
4. The injected code is executed.

The mediation methods mentioned in Section 2 are designed to handle the problem by preventing either step 3 or 4. Non-executable pages are designed to prevent step 4, while compiler-based approaches are meant to prevent step 3. In both cases, however, the malicious code is injected, but execution is somehow prevented. Our solution, on the other hand, effectively stops the attack at step 1 by preventing the successful injection of the malicious code into a process' *code space*. (The purist will note that, in the implementation method described in Section 4, the attack is not technically stopped until step 4; however, the general approach described here handles it at step 1.)

The Harvard architecture's split memory model makes it suitable for the prevention of code injection attacks due to the fact that a strict separation between code and data is enforced at the hardware level. Any and all data, regardless of the source, are stored in a different physical memory from instructions. Instructions cannot be addressed as data, and data cannot be addressed as instructions. This means that in a Harvard-architecture-based computer, a traditional code injection attack is not possible because the architecture is not capable of supporting it after a process is initially setup. The attacker is simply unable to inject any information whatsoever into the instruction memory's address space, and at the same time, is unable to execute any code placed in the data memory. The architecture simply does

not have the "features" required for a successful code injection attack. However, we point out that this does not prevent an attacker from mounting non-control attacks (e.g., non-control-data attack [25]) on a Harvard architecture. We touch on these attacks in Section 7.

3.3 Challenges in Using a Harvard Architecture

While a Harvard architecture may be effective at mitigating code injection, the truth of the matter is that for any new code injection prevention technique to be practical, it must be usable on modern commodity hardware. As such, the challenge is to construct a Harvard architecture on top of a widely deployed processor such as the x86, without assuming hardware support for non-executable memory pages (Section 2).

In the following, we present a few possible methods for creating this Harvard architecture on top of the x86 architecture.

3.3.1 Modifying x86

One technique for creating such an architecture is to make changes to the existing architecture and use hardware emulation [26] to make them a reality. The changes required in the x86 architecture to produce a Harvard architecture are fairly straight forward modifications to the paging system.

Currently, x86 implements paging by having a separate pagetable for each process and having the operating system maintain a register (CR3) that points to the pagetable for the currently running process. One pagetable is used for the process' entire address space, both code and data. In order to construct a Harvard architecture, one would need to maintain two different pagetables, one for code and one for data. As such, our proposed change to the x86 architecture to allow it to create a Harvard architecture is to create an additional pagetable register in order that one can be used for code (CR3-C) and the other for data (CR3-D). Whenever an instruction fetch occurs, the processor uses CR3-C to translate the virtual address, while for data reads and writes, CR3-D is used. An operating system, therefore, would simply need to maintain two separate pagetables for each process. This capability would also offer backwards compatibility at the process level, as the operating system could simply maintain one pagetable and point both registers to it if a process requires a von Neumann architecture. We note that no changes would need to be made to the processor's translation lookaside buffer (TLB) as modern x86 processors already have a separate TLB for code and data.

While this approach to the problem may be effective, the requirement that the protected system be run on top of hardware emulation inhibits its immediate deployment. As such, a more practical approach is needed.

3.3.2 Exploiting x86

Another technique for creating this Harvard architecture is to make unconventional use of some of the architecture's features in order to create the appearance of a memory that is split between code and data. Through careful use of the pagetable and the TLBs on x86, it is possible to construct a Harvard memory architecture at the process level using only operating-system-level modifications. No modifications need to be made to the underlying x86 architecture, and the system can be run on conventional x86 hardware without the need for hardware emulation as in the previous method.

In following sections, we will further describe this technique as well as its unique advantages.

4 Split Memory: A Virtual Harvard Architecture

Now that we have established that it is our intention to exploit, not change, the x86 architecture in order to create a virtual Harvard architecture, we will now describe the technique in greater detail. The realization of the virtual Harvard architecture on other architectures (e.g., SPARC) will be discussed in Section 4.7.

4.1 Virtual Memory and the TLB

We first present a brief overview of paging on the x86 with an emphasis on the features we intend to leverage. The details are available in the Intel manual [3].

4.1.1 Pagetables and the TLB

Virtual memory on the x86 is implemented using operating-system-managed pagetables that are stored in memory. When the hardware needs to translate a virtual address to a physical address, it accesses the table (the address of which is stored in a register) to find the correct mapping. This procedure, while effective, can be very slow due to the fact that each memory access now requires three total accesses into memory, two into the pagetable and one into the data requested. To combat this slowdown, a small hardware cache called the translation lookaside buffer (TLB) is used to store recently accessed pagetable entries. As such, many pagetable lookups never actually need to go all the way to the pagetable, but instead can be served by the TLB. On the x86, the loading of the TLB is managed automatically by the hardware, but removing entries from it can be handled by either software or hardware. The hardware, for example, will automatically flush the TLB when the OS changes the address of the currently mapped pagetable (such as during a context switch) while software can use the `invlpg` instruction to invalidate specific TLB entries when making modifications to individual pagetable entries in order to ensure that the TLB and pagetables remain synchronized.

While the TLB is able to quite effectively speedup virtual memory on the x86, one problem is that due to the fact that it is limited in size, old entries are automatically removed when new ones come in. As a consequence of this, a program that has lots of random data access could end up removing the entries for its code, causing those code accesses to reference the pagetable once again. To help prevent this problem, the TLB is split into *two TLBs* on modern x86 processors, one for code and one for data. During normal operation one would want to ensure that the two TLBs do not contain conflicting entries (where one address could be mapped to different physical pages, depending on which TLB services the request).

4.2 Virtualizing Split Memory on x86

The key idea in our split memory virtualization technique is to exploit the dual-TLB feature of the x86 architecture. More specifically, via the two TLBs, the operating system is able to route data accesses for a given virtual address to one

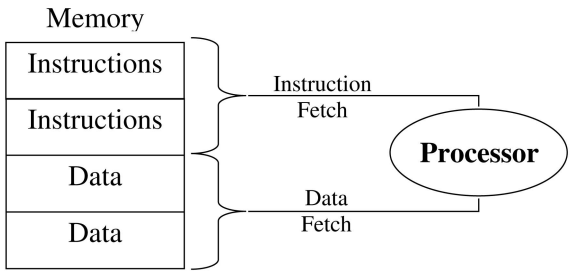


Fig. 3. Split memory architecture.

physical page while routing instruction fetches to another. By desynchronizing the TLBs and having each contain a different mapping for the same virtual page, every virtual page may have two corresponding physical pages: One for code fetch and one for data access. In essence, a system is produced where any given virtual memory address could be routed to two possible physical memory locations. This creates a split memory architecture, as illustrated in Fig. 3.

This virtual split memory architecture is an environment wherein an attacker can exploit a vulnerable program and inject code into its memory space, but never be able to actually fetch it for execution. This is because the physical page that contains the data the attacker managed to write into the program is not accessible during an instruction fetch, as instruction fetches will be routed to an uncompromised code page. This also creates the unique opportunity to support and protect pages that contain both code and data by keeping the two physically separated but logically combined.

4.2.1 What to Split

Before we discuss the technical details behind successfully splitting a given page, it is important to note that different pages in a process' address space may be chosen to split based on how our system will be used.

In systems where the execute-disable bit is available, our technique can be used to *complement* it by extending protection to mixed code and data pages. Under this usage scenario, the majority of pages (i.e., the "nonmixed" pages) in a process would be protected using the execute-disable bit, while the mixed pages would be protected using our technique. In this scenario, chances are high that only a few of the process' pages are mixed and need to be protected using our technique. This should result in a very low performance overhead. However, we point out that only protecting the mixed pages using our technique may limit the use of the various response modes described in Section 4.5.

When the execute-disable bit is not available, our technique is used to protect every page in a process' memory space. In this usage scenario, all relevant pages are split and protected.

4.2.2 How to Split

Once it is determined which pages will be split, the technique for splitting a given page is as follows:

1. On program startup, the page that needs to be split is duplicated. This produces two copies of the page in physical memory. We choose one page to be the target of instruction fetches, and the other to be the target of data accesses.
2. The pagetable entry (PTE) corresponding to the page we are splitting is set to ensure a page fault will occur on a TLB miss. In this case, the page is considered *restricted*, meaning it is only accessible when the processor is in supervisor mode. We accomplish it by setting or enabling the *supervisor* bit [3] in the PTE for that page. If *supervisor* is marked in a PTE and a user-level process attempts to access that page for any reason, a page fault will be generated and the corresponding page fault handler will be automatically invoked.
3. Depending on the reasons for the page fault, i.e., either this page fault is caused by a data TLB miss or it is caused by an instruction TLB miss, the page fault handler behaves differently. Note that for an instruction-TLB miss, the faulting address (saved in the *CR2* register [3]) is equal to the program counter (contained in the EIP register); while for a data-TLB miss, the page fault address is different from the program counter. In the following, we describe how different TLB misses are handled. The algorithm is outlined in Algorithm 1.

Algorithm 1. Split memory page fault handler
Input: Faulting Address (addr), CPU instruction pointer (EIP), Pagetable Entry for addr (pte)

```
1  if addr == EIP then                    /* Code Access */
2      pte = the_code_page;
3      unrestrict(pte);
4      enable_single_step();
5      return;
6  else                                   /* Data Access */
7      pte = the_date_page;
8      unrestrict(pte);
9      read_byte(addr);
10     restrict(pte);
11     return;
12 end;
```

4.2.3 Loading the Data-TLB

The data-TLB is loaded using a technique called a pagetable walk, which is a procedure for loading the TLB from within the page fault handler. The pagetable entry (PTE) in question is set to point to the data page for that address, the entry is unrestricted (we unset the *supervisor* bit in the PTE), and a read off of that page is performed. As soon as the read occurs, the memory management unit in the hardware reads the newly modified PTE, loads it into the data-TLB, and returns the content. At this point, the data-TLB contains the entry to the data page for that particular address while the instruction-TLB remains untouched. Finally, the PTE is restricted again to prevent a later instruction access from improperly filling the instruction-TLB. Note that even though the PTE is restricted, later data accesses to that page can occur unhindered because the data-TLB contains a valid mapping. This loading method is also used in the PaX [2] protection model and is known to bring the overhead for a

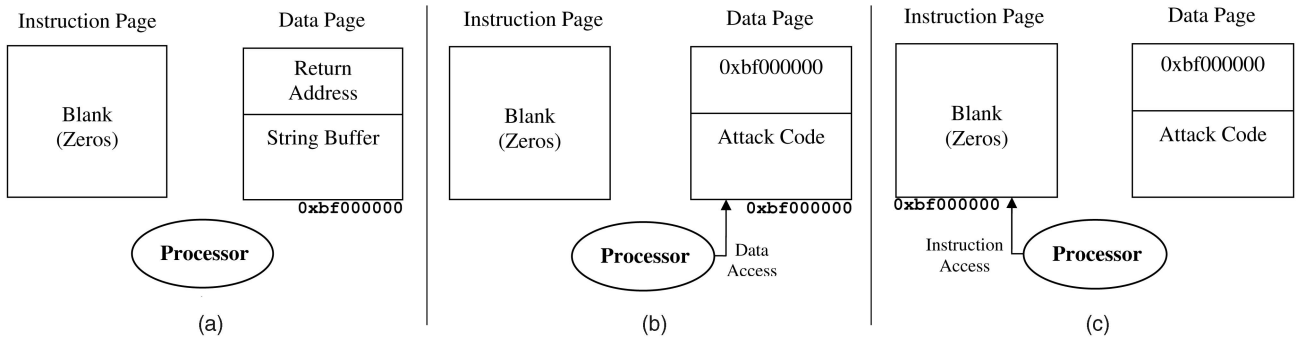


Fig. 4. (a) Before the attacker injects code. (b) The injection to the data page. (c) The execution attempt that gets routed to the instruction page.

data-TLB load down to reasonable levels (e.g., less than 2.7 percent in benchmarks on a Pentium III [27]).

The procedure above can be seen in lines 7-11 of Algorithm 1. First, the pagetable entry is set to point to the data page and unrestricted by setting the entry to be user-accessible instead of supervisor-accessible. Next, a byte on the page is touched, causing the hardware to load the data-TLB with a pagetable entry corresponding to the data page. Finally, the pagetable entry is reprotected by setting it into supervisor mode once again.

4.2.4 Loading the Instruction-TLB

The loading of the instruction-TLB has additional complications compared to that of the data-TLB, namely, because there does not appear to be a simple procedure such as a pagetable walk that can accomplish the same task. Despite these complications, however, a technique introduced in [10] can be used to load the instruction-TLB on the x86.

Once it is determined that the instruction-TLB needs to be loaded, the PTE is unrestricted, the processor is placed into single-step mode, and the faulting instruction is restarted. When the instruction runs this time, the PTE is read out of the pagetable and stored in the instruction-TLB. After the instruction finishes, then the single-step mode of the processor generates an interrupt, which is used as an opportunity to restrict the PTE.

This functionality can be seen in Algorithm 1 lines 2-5 as well as in Algorithm 2. First, the PTE is set to point to the corresponding code page and is unprotected. Next, the processor is placed into single-step mode and the page fault handler returns, resulting in the faulting instruction being restarted. Once the single-step interrupt occurs, Algorithm 2 is run, effectively restricting the PTE and disabling the single-step mode.

Algorithm 2. Debug interrupt handler
Input: Pagetable Entry for previously faulting address (pte)

```
1 if processor is in single step mode then
2       restrict(pte)
3       disable_single_step();
4 end
```

As an interesting side note, we created another instruction-TLB loading method that did not require the use of single-step mode through carefully adding a `ret` instruction to the page and then `call`ing it from the page fault handler, but surprisingly this actually decreased the system's efficiency. It is our understanding that the slowdown was caused by the x86 maintaining cache coherency. In essence, when the write to the code page occurs, the processor invalidates the memory caches corresponding to that page, and also invalidates any portions of the instruction pipeline currently containing instructions fetched from that page. Unfortunately, this causes undesirable performance degradation to the system.

4.3 Dynamic and Shared Libraries

Two important features of modern operating systems that need to be discussed are dynamic and shared libraries. For ease of presentation, we make the distinction between dynamic and shared libraries as follows: A dynamic library (sometimes called a plugin by applications) is a piece of code and data that is loaded into an application on demand at runtime while shared library is typically loaded into a process' memory space at load time. The split memory system detects the loading of these libraries at either load time or runtime and splits their pages appropriately.

It is important to note that in order for libraries to be handled in a secure way, they must be validated when being loaded. As a solution to this problem, we look to existing work [28], [29] that uses cryptographic primitives to verify binaries and libraries. Using one of these systems, memory splitting could simply validate the signature of the loaded library prior to loading and splitting it. This would prevent an attacker from loading a new or modified module into a running program's address space, while still permitting valid modules to be loaded and used unhindered. Given that this technique has already been implemented for both Linux [28] and NetBSD [29], we do not repeat this part in our implementation.

4.4 Effects on Code Injection

A split memory architecture produces an address space where data cannot be fetched by the processor for execution. For an attacker attempting a code injection, this will prevent him from fetching and executing any injected code. A sample code injection attack attempt on a split memory architecture can be seen in Fig. 4 and described as follows:

1. The attacker injects his code into a string buffer starting at address `0xbf000000`. The memory writes are routed to physical pages corresponding to data.
2. At the same time as the injection, the attacker overflows the buffer and changes the return address of the function to point to `0xbf000000`, the expected location of his malicious code.

3. The function returns and control is transferred to address `0xbf000000`. The processor's instruction fetch is routed to the physical pages corresponding to instructions.
4. The attacker's malicious code is not on the instruction page (the code was injected as data and therefore routed to a different physical page) and is not run. In all likelihood, the code page is empty (containing zeros) and the program simply crashes.

4.5 Attack Response Modes

As described, the virtual split memory system provides effective protection against the execution of injected code. Taking a step further, we can also take advantage of the provided protection as an effective means of detecting the injected code execution attempt and responding accordingly. More specifically, since the attack can be detected right before executing the first instruction injected by the attacker, we can develop a number of options to respond, including terminating the execution of the exploited process or permitting the attack to proceed while allowing its subsequent behavior to be closely monitored, similar to the way a honeypot is monitored. In the following, we describe three response modes.

4.5.1 Break Mode

This response mode will take no action and still route the instruction fetch to the uncompromised code page, which likely contains null content (a string of zeros). As a result, the operating system will typically terminate the offending application. Notice that this option, or the one that achieves the same results, is the defacto standard for most code injection prevention systems.

4.5.2 Observe Mode

This mode will log the code injection attempt and then somehow still permit the attack to continue. This can be applied to honeypot-style systems wherein notification of a previously unknown attack would be helpful while still allowing the attack to continue. The system could even be tightly integrated with honeypot monitoring tools (such as Sebek [30] and VMscope [31]) to allow features such as an incoming attack being seamlessly transferred to a sandbox system and allowed to continue.

The idea of accomplishing an observe mode is intuitive, but the devil is in the details. Particularly, to intervene prior to the execution of injected code, some sort of trap will need to be generated by the hardware. This challenge arises from the fact that the operating system does not normally intercede before every instruction fetch, and doing so would cause undue performance penalties. In our system, we take the following approach to cause a trap that will be handled by the operating system: Fill the previously empty code pages with invalid opcodes so that an invalid instruction fault will be generated when an execution attempt occurs.

Upon the detection of an invalid instruction fault, our response will be activated and Algorithm 3 will be executed. In essence, it works as follows: Once the trap is intercepted, log the attack attempt and record the timestamp when the injected attack code is executed. Next, the pagetable entry is updated to point to the data page (remember that the data page contains the actual attack code), memory splitting is turned off for the page, the TLB entry is invalidated, and the program is resumed. The net result is that the PTE has been updated to point to the page containing the attack code and the attack is able to continue unhindered by the intercession.

Algorithm 3. Observe Algorithm.
Input: CPU instruction pointer (EIP), Pagetable entry for EIP (pte)

```
1 if Invalid Instruction Fault then
2     log();
3     pte = the_data_page;
4     disable_splitting(pte);
5     invalidate_tlb(pte);
6     continue_execution;
7 end
```

4.5.3 Forensic Mode

In this mode, we aim at performing in-depth forensic analysis on the detected attack. Since the attack is detected right before the first injected attack code is executed, we consider it an opportune time to start forensic analysis. Given that the OS has access to the process' entire address space as well as the current instruction pointer before malicious code is executed, forensic investigation of the attack is quite feasible. Operations such as shellcode analysis (the instruction pointer points to shellcode in the data pages) or attack fingerprinting based on memory contents are fully realizable and can be initiated live during a previously unseen attack. A related project—Argos [32]—has offered the ability to replace injected code with its own, "forensic" shellcode. This same technique could easily be accommodated by this system by simply injecting the code into the process' address space, changing the EIP to point to it, and resuming program execution. In our current implementation, we dump the corresponding EIP content and the related injected attack code. An example will be presented in Section 5.

We point out that, in addition to the above modes, more customized response modes can be developed based on our system. For example, a unique potential is to develop a recovery mode in response to an attack. One common problem with interceding during an attack is that while the attacker has not successfully executed his malicious code, he has probably corrupted various parts of a program's memory space. Due to the fact that the operating system doesn't understand the intricacies of the running program, it would be infeasible for it to attempt any sort of recovery that would permit the application to continue running. It may be much more feasible, however, for the application itself to register a call-back function or a special signal handler that the operating system could transfer execution to in the event an attack is detected. The application writer would then be able to better attempt recovery by checking data integrity, restarting an earlier checkpoint, or terminating gracefully. This would, of course, require changes to the existing applications and would be interesting to investigate in future work [33].

4.6 Overhead

This technique of splitting memory does not come without a cost, there is some overhead associated with the methodologies described above.

One potential problem is the use of the processor's single-step mode for the instruction-TLB load. This loading process has a fairly significant overhead due to the fact that two interrupts (the page fault and the debug interrupt) are required in order to complete it. This overhead ends up being minimal overall for many applications due to the fact that instruction-TLB loads are fairly infrequent, as it only needs to be done *once* per page of instructions.

Another problem is that of context switches in the operating system. Whenever a context switch (meaning the OS changes running processes) occurs, the TLB is flushed. This means that every time a protected process is switched out and then back in, any memory accesses it makes will trigger a pagefault and subsequent TLB load. The overhead of these TLB loads is significantly higher than a traditional pagefault, and hence, causes the majority of our slowdown. The problem of context switches is, in fact, the greatest cause of overhead in the implemented system. The experimental details of the overhead can be seen in Section 6.2.

4.7 Portability to Other Architectures

We have reason to believe that x86 could be one of the most difficult architecture platforms to support virtual split memory. In some other architecture platforms, such as SPARC, the TLB is actually managed by software instead of by hardware. Given this, the split memory scheme should be much easier to build. More specifically, on an architecture with software-loaded TLBs, there would be no need for complex data or instruction TLB loading techniques. Instead, the processor's TLBs could be loaded directly. The basic procedure would be as follows: 1) Split and mark pages and pagetable entries just like the x86 implementation. 2) When a "memory splitting" pagefault occurs, use the architecture's TLB loading instructions to load the correct TLB with the proper physical page number.

Given that no complex loading procedures would be required, we believe that the code base required to construct virtual split memory on such an architecture would be smaller and that the performance overhead imposed on such a system would be noticeably lower.

5 Implementation

An x86 prototype of our design running in stand-alone mode has been created by modifying version 2.6.13 of the Linux kernel. In this section, we present a detailed description of the modifications to create the virtual split memory architecture.

5.1 Modifications to the ELF Loader

ELF is a format that defines the layout of an executable file stored on disk. The ELF loader is used to load those files into memory and begin executing them. This work includes setting up all of the code, data, bss, stack, and heap pages as well as bringing in the shared libraries used by a given program.

The modifications to the loader are as follows: After the ELF loader maps the code and data pages from the ELF file, for each one of those pages two new, side-by-side, physical pages are created and the original page is copied into both of them. This effectively creates two copies of the program's memory space in physical memory. The pagetable entries corresponding to the code and data pages are changed to map to one of those copies of the memory space, leaving the other copy unused for the moment. In addition, the pagetable entries for those pages get the supervisor bit cleared, placing that page in supervisor mode in order to be sure a page fault will occur when that entry is needed. A previously unused bit in the pagetable entry is used to signify that the page is being split. In total, about 90 lines of code are added to the ELF loader.

In this particular implementation of split memory, the memory usage of an application is effectively doubled; however, this limitation is *not* one of the technique itself, but instead of the prototype. A system can be envisioned based on demand paging (only allocating a code or data page when needed) instead of the current method of proactively duplicating every virtual page. This would result in a lower memory overhead because duplicate physical pages would only be needed when both code and data are accessed from the same virtual page. We would anticipate this optimization to not have any noticeable impact on performance.

5.2 Modifications to the Page Fault Handler

Under Linux, the page fault (PF) handler is called in response to a hardware-generated PF interrupt. The handler is responsible for determining what caused the fault, correcting the problem, and restarting the faulting instruction.

For our modifications to the PF handler, we simply modify it to handle a new reason for a PF: There was a permission problem caused by the supervisor bit in the PTE. We must be careful here to remember that not every PF on a split page is necessarily our fault, some PFs (such as ones involving copy-on-write), despite being on split memory pages, must be passed on to the rest of the PF handler instead of being serviced in a split memory way. If it is determined that the fault was caused by a split memory page and that it does need to be serviced, then the instruction pointer is compared to the faulting address to decide whether the instruction-TLB or data-TLB needs to be loaded. (Recall from Algorithm 1 that this is done by simply checking if the two are the same.)

If the data-TLB needs to be loaded, then the PTE is set to user mode, a byte on the page is touched, and the PTE is set back to supervisor mode. This pagetable walk loads the data-TLB.[1] In the event the instruction-TLB needs to be loaded, the PTE is set to user mode (to allow access to the page) and the trap flag (single-step mode) bit in the EFLAGS register is set. This will ensure that the debug interrupt handler gets called after the instruction is restarted. Before the PF handler returns and that interrupt occurs, however, a little bit of bookkeeping is done by saving the faulting address into the process' entry in the OS process table in order to pass it to the debug interrupt handler.

In total, there were about 110 lines of code added to the PF handler to facilitate splitting memory.

1. Occasionally, the pagetable walk does not successfully load the data-TLB. In this case, single stepping mode (like the instruction-TLB load) must be used.

5.3 Modifications to the Debug Interrupt Handler

The debug interrupt handler is used by the kernel to handle interrupts related to debugging. For example, using a debugger to step through a running program or watch a particular memory location makes use of this interrupt handler. For the purposes of split memory, the handler is modified to check the process table to see if a faulting address has been given, indicating that this interrupt was generated because the PF handler set the trap flag. If this is the case, then it is safe to assume that the instruction which originally caused the PF has been restarted and successfully executed (meaning the instruction-TLB has been filled), and as such, the PTE is set to supervisor mode once again and the trap flag is cleared. In total, about 40 lines of code were added to the debug interrupt handler to accommodate these changes.

5.4 Modifications to the Memory Management System

There are a number of features related to memory management that must be slightly modified to properly handle our system. First, on program termination, any split pages must be freed specially to ensure that both physical pages (the code page and data page) get put back into the kernel's pool of free memory pages. This is accomplished by simply looking for the split memory PTE bit that was set by the ELF loader above, and if it is found then freeing two pages instead of just one.

Another feature in the memory system that needs to be updated is the copy-on-write (COW) mechanism. COW is used by Linux to make `fork`ed processes run more efficiently. The basic idea is that when a process makes a copy of itself using `fork`, both processes get a copy of the original pagetable, but with every entry set read-only. Then, if either process writes to a given page, the kernel will give that process its own copy. (This reduces memory usage in the system because multiple processes can share the same physical page.)

An update similar to the COW update is also made to the demand paging system. Demand paging basically means that a page is not allocated until it is required by a process. In this way, a process can have a large amount of available memory space (such as in the bss or heap) but only have physical pages allocated for portions it actually uses. The demand paging system was modified to allocate two pages instead of just the one page it normally does. This required modifications to the code that allocates empty pages on demand as well as the code that allocates pages for memory-mapped files. Proper support of memory-mapped files also allows the system to protect dynamic and shared libraries as well.

Overall, about 75 lines of code were added to handle these various parts related to memory management.

5.5 Modifications to the Signal Handler

In order to accommodate the three response modes outlined in Section 4.5, we extend the Linux signal handler to better handle the `SIGILL` (illegal instruction) signal generated by the corresponding processor exception. In the event an attack is detected, the following three response modes have been implemented to respond to the attack: break mode, observe mode, and forensics mode.

The basic control flow in implementing the response modes is as follows: Once the attack has been detected, a log entry containing the EIP of the processor prior to malicious code execution is added to the system. After that, different modes lead to different responses:

- If the system is in observe mode, the corresponding pagetable entry is modified to point to the data page, split memory is disabled for that page, and the program is allowed to continue. In other words, the data page is locked in as the sole mapping and program execution is resumed. Note that this means that only the first unauthorized code execution on a given page will be logged, as future execution will occur unhindered from the data page.
- If the system is in forensics mode (a light version of what is described in our design), we first dump additional information about the attack. For example, we record the injected attack code or shellcode. The shellcode is considered the first payload executed after compromising the vulnerable program. Thanks to the unique timing of our system in detecting the attack, we can easily identify the location of the shellcode, namely, those bytes starting at the EIP in the data page. We record them in the log for later analysis. Moreover, we can also inject our own "forensic" shellcode into the address space, update the EIP to point to the new code, and resume normal program execution. Currently, the implementation copies the new code onto the empty code page being executed from and changes the EIP to point to the beginning of the page. The features of the forensic shellcode can range from a basic program exit to more advanced and customized code that collects runtime application semantic information.
- If the system is in break mode, the application will simply be terminated. This is what would occur if no modifications were made to the signal handler, and while it lacks elegance, it is effective at preventing the attacker from executing his malicious code.

Overall, about 70 lines of code were added to handle these various parts related to signal handling for response mode implementation.

6 Evaluation

Various experiments were run to test both the effectiveness of the system at preventing code injection attacks as well as the performance under a variety of workloads. Our testbed was a modest system, consisting of a Pentium III 600 MHz with 384 MB of RAM and a 100 MBit NIC.

6.1 Effectiveness

To evaluate the effectiveness, we used a buffer overflow benchmark as well as five representative, real-world attacks to see how our system performs.

6.1.1 Wilander Benchmark

The code injection benchmark used for evaluation was originally put forth by Wilander and Kamkar [8]. The benchmark was modified slightly in order to allow it to

TABLE 1
Benchmark Attacks Foiled when Code Is Injected onto the Data, Bss, Heap, and Stack Segments

Attack Type	Hijack Type	Injection Destination			
		Data	BSS	Heap	Stack
Buffer overflow on stack	Return address	✓	✓	✓	✓
	Old base pointer	✓	✓	✓	✓
	Function pointer as local variable	✓	✓	✓	✓
	Function pointer as parameter	✓	✓	✓	✓
	Longjmp buffer as local variable	✓	✓	✓	✓
	Longjmp buffer as function parameter	✓	✓	✓	✓
Buffer overflow on heap/bss	Function pointer	✓	✓	✓	✓
	Longjmp buffer	✓	✓	✓	✓
Buffer overflow of pointers on stack	Return address	N/A	N/A	✓	N/A
	Old base pointer	N/A	N/A	N/A	N/A
	Function pointer as local variable	✓	✓	✓	✓
	Function pointer as parameter	✓	✓	✓	✓
	Longjmp buffer as local variable	✓	✓	✓	✓
	Longjmp buffer as function parameter	✓	✓	✓	✓
Buffer overflow on heap/bss	Return address	N/A	N/A	✓	N/A
	Old base pointer	N/A	N/A	N/A	N/A
	Function pointer as variable	✓	✓	✓	✓
	Longjmp buffer as variable	✓	✓	✓	✓

handle having the code injected on the data, bss, heap, and stack portions of the program's address space. In addition, four of the test cases did not successfully execute an attack on our unprotected system, and so have been labeled "N/A." Table 1 shows the results of running the benchmark. The checkmarks indicate that the system successfully halted the attack. As can be seen, the system was effective in preventing all types of code injection attacks present in the benchmark. The effectiveness of the system is due to the fact that no matter what method of control flow hijacking the benchmark uses, the processor is simply unable to fetch the injected code.

6.1.2 Real-World Attacks

Five representative software packages containing real-world vulnerabilities that permit code injection and execution were run under our implementation. Vulnerabilities in five major Linux server packages from 2001 to 2003 were chosen and exploited using exploits publicly released at the time. Our software platform for the attacks was the RedHat 7.2 operating system (chosen due to its vulnerability to many attacks from that time period) that had been manually upgraded to use version 2.6.13 of the Linux kernel. Table 2 summarizes the results of the experiments, including the versions of software installed on our testing platform. Some software shipped with the default version of RedHat 7.2, other software was "forward" ported from previous releases. The results of the attacks when executed on an unpatched kernel is reflected by the "Attack Result" column.

1. Apache 1.3.20 with OpenSSL 0.9.6d. A bug in OpenSSL allows a buffer overflow to occur if an attacker sends a very large client master key to the server. The exploit we used, openssl-too-open by Solar Eclipse, overflows a heap buffer and makes use of an information leak in the SSL handshake to determine the proper address for its shellcode. If the attack successfully executes, a shell owned by `nobody` (the `uid` of the apache process) is spawned over the network to the attacker. When run under our system, the heap buffer is overflowed, but execution of the injected shell code is foiled because it is unavailable to the processor when it attempts to fetch instructions from the heap page.
2. Bind 8.2.2_P5. Bugs in the DNS server implementation allow either a stack or heap overflow to occur (depending on which bug is exploited) while handling transaction signatures. For our testing, we used a publicly released lsd-pl.net exploit. (A modified version of this same exploit code was used by the Lion worm.) Much like the apache attack, this exploit makes used of an information leak bug to determine the shellcode jump address. Once that occurs, a stack overflow is triggered and a shell is spawned over the network. When run under our system, the information leak bug still functions and the stack overflow still occurs, but the shellcode is unable to be fetched and the execution attempt fails.
3. ProFTPD 1.2.7. When transferring files in ASCII mode, ProFTPD contains a bug which causes newline

TABLE 2
Five Real-World Vulnerabilities

Software	CVE	Corruptible Memory Region	Attack Result	Stopped?
apache-1.3.20-16/mod_ssl-2.8.4-9	CVE-2002-0656	Heap	`nobody` shell	Yes
bind-8.2.2_P5-9	CVE-2001-0010	Stack or Heap	`named` shell	Yes
proftpd-1.2.7-1	CVE-2003-0831	Heap	`root` shell	Yes
samba-2.2.1a-4	CVE-2003-0201	Stack	`root` shell	Yes
wu-ftpd-2.6.1-18	CVE-2001-0550	Heap	`root` shell	Yes

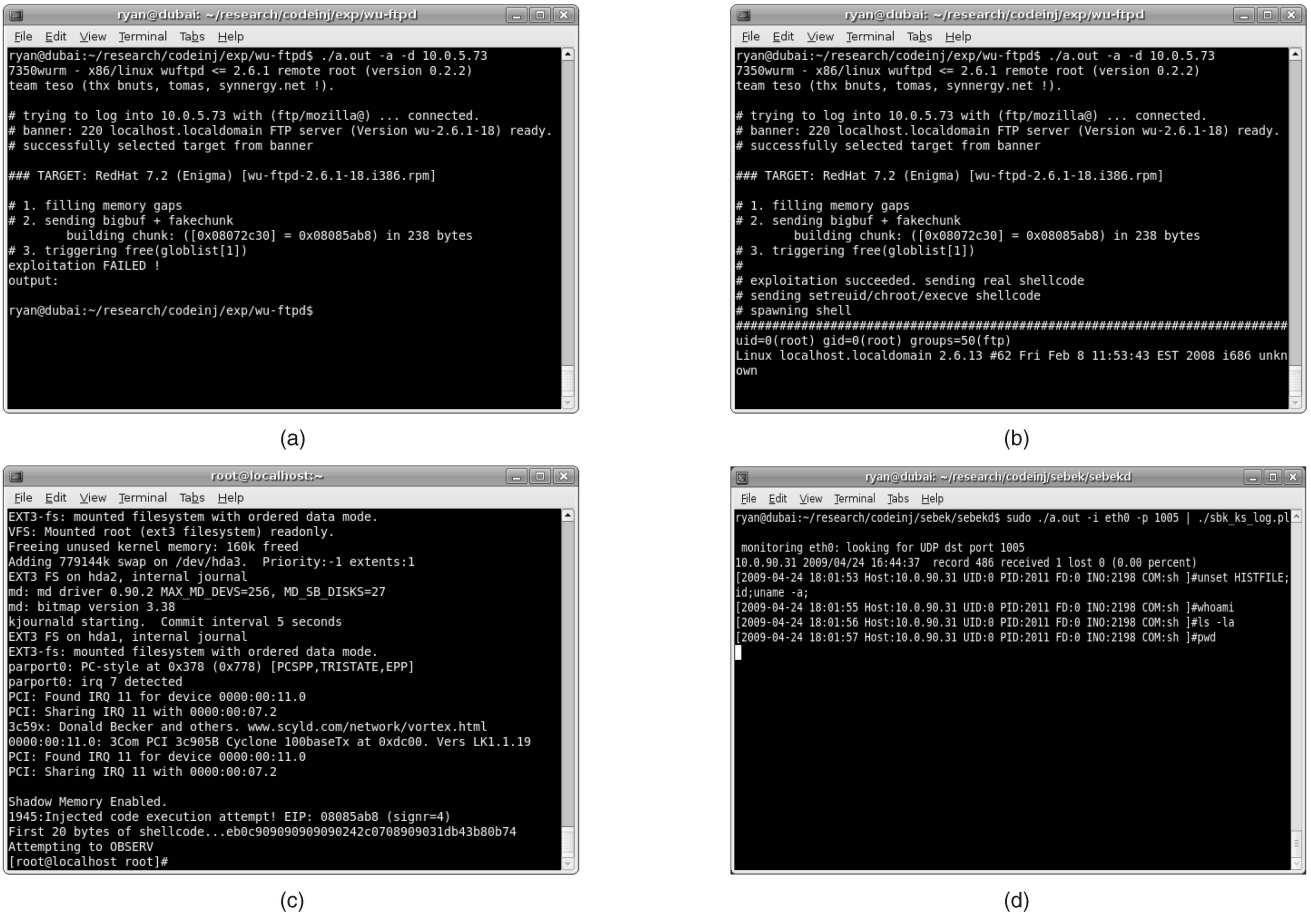


Fig. 5. Demonstration of response modes against the WU-FTPD exploit. (a) Attack failure during break mode. (b) Attack success during observe mode. (c) Output during forensics mode. (d) Sebek log during observe mode.

characters to be translated incorrectly and permits an attacker to execute arbitrary code. Our exploit of choice was proftpd-not-pro-enough by Solar Eclipse. In order to trigger the flaw, the exploit logs in to the server and uploads a file containing a malicious payload. Next, it puts the server in ASCII mode and downloads that file. During the ASCII translation process, the exploit code is executed off of the heap. The malicious code then breaks out of any `chroot` environments and spawns a root shell over the network. Executing the server under our system results in the instruction fetch from the heap failing, and hence, the attack is foiled.

4. Samba 2.2.1a. Samba contains a bug in the `call_trans2open` function that allows a stack buffer to be overflowed. For our testing, we used an exploit put out by eSDee. The exploit is a fairly simply stack-based overflow with a brute-force mode to guess the address of the shellcode on the stack based on a good "first guess" obtained by manual analysis of a similar vulnerable system. This bug was made a bit more difficult to exploit due to the fact that version 2.6 of the Linux kernel added slight randomization to the placement of an application's stack within memory. This means that it can take a fairly long time for the attack to properly guess the correct stack address. In order to better facilitate testing, the exploit was "helped" by providing a better first guess using insider information about the stack location. (We would like to note that an unmodified attack would still function given enough time.) When run under our system, the return address is still guessed properly, but the shellcode is unavailable to the processor when it attempts to transfer control to that location.
5. WU-FTPD 2.6.1. A bug in the WU-FTPD code handling filename globbing combined with the `free`'ing of attacker-controlled memory permits arbitrary code execution. This bug is different from, but related to, a traditional heap overflow. The exploit code we used was 7350wurm published by TESO Security. The exploit logs in to the server, adds its own malicious code to the heap, triggers the globbing flaw, and causes a root shell to be spawned. Under our system, the heap is still filled with malicious code and the globbing bug is still triggered, but the injected code is not fetched by the processor.

Overall, even with a variety of bugs and exploitation techniques, our system is able to defeat code injection in these real-world scenarios due to the fact that it prevents malicious code from ever being executed, even after successful injection into the process' data space.

6.1.3 Response Modes

In order to validate the attack response modes described in Section 4.5, the WU-FTPD vulnerability and exploit were executed under the various modes. Fig. 5 shows how the

exploit code reacts when the WU-FTPD daemon is run under break mode, observe mode, and forensics mode. First, the ftp server is run under break mode. As can be seen in Fig. 5a, the exploit fails to successfully launch a root shell. (This is due to the process crashing when attempting to execute the shellcode.) This is contrasted with our second test, executing the server under observe mode, where the exploit is allowed to continue unhindered and a rootshell is spawned (Fig. 5b). More information about this particular attack can be observed when running under forensics mode, which can be seen in Fig. 5c. A closer examination of the screenshot will find that the log entry contains the first 20 bytes of the injected shellcode. This is fairly easy to recognize because of the `nop` instructions (the `0x90` bytes).

A manual analysis of the exploit code reveals that these 20 bytes are indeed the first 20 bytes of injected code. The exploit actually functions using two stages of injected code. The initial stage (the first 20 bytes of which are in the figure) is used to write 4 bytes back to the attacker over the network in order to signal that the attack succeeded and then immediately reads a second stage of shellcode from the network and executes it. Currently, our system can successfully observe the execution of the initial stage of code, but does not intercede before the second stage because the memory page has been locked on to the data entry.

The last screenshot, Fig. 5d, demonstrates our system used in conjunction with Sebek, a kernel-level logging mechanism for honeypots. In our experiment, we integrate Sebek as a part of observe response mode. By default, Sebek's logging mechanism always runs. To reduce log volume, we modified Sebek to be activated by a buffer overflow event (caused by code injection) detected by our system. By doing so, log files can be significantly smaller, yet we can still ensure that an attacker's actions are captured thanks to our system's "right-before" detection of code injection attacks. The screenshot shows Sebek logging the commands the attacker types into his spawned shellcode.

We also tested the possibility of injecting custom shellcode into the program's address space. For demonstration purposes, we injected the code required to cause the program to call the `exit` system call and terminate gracefully. The injected shellcode (corresponding to `exit(0);`) is as follows:

```
"\xbb\x00\x00\x00\x00" /* mov $0x0, %ebx */
"\xb8\x01\x00\x00\x00" /* mov $0x1, %eax */
"\xcd\x80";            /* int $0x80 */
```

The code loads `percent ebx` with the program's return value (`0`), loads the system call number for `exit()` into `percent eax`, and finally generates the interrupt required for the system call. By replacing the attacker's injected code with this code, the program terminates *without* a segmentation fault. While this test shows the injection of fairly uninvolved code, it can easily be replaced with more sophisticated forensic shellcode to assist in attack investigation.

6.2 Performance

A number of benchmarks, both applications and microbenchmarks, were used to test the performance of the system. When applicable, benchmarks were run 10 times and the results averaged. Details of the configuration for the tests are available in Table 3. Each result has been normalized with respect to the speed of the unprotected system. Unless otherwise stated, the tests are run with the system running in stand-alone mode, which is an indicator of the worst-case performance.

TABLE 3
Configuration Information Used for Performance Evaluation

Item	Version	Configuration
Slackware	10.2.0	Using Linux 2.6.13
Apache	2.2.3	Worker mpm mode, set to spawn one process with threads
ApacheBench	2.0.41-dev	`-c3 -t 60 <url/file>`
Unixbench	4.1.0	N/A
Nbench	2.2.2	N/A
Gzip	1.3.3	Compress a 256 MB file.

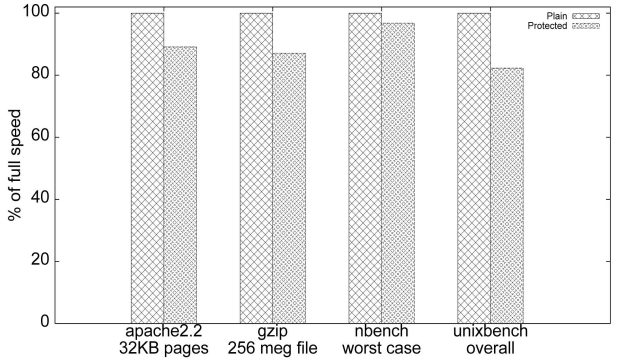


Fig. 6. Normalized performance for applications and benchmarks.

Four benchmarks that we consider to be a reasonable assessment of the system's performance can be found in Fig. 6. First, the Apache Webserver was run in a threading mode to serve a 32 KB page (roughly the size of Purdue University's main index.html). The ApacheBench program was then run on another machine connected via the NIC to determine the request throughput of the system as a whole. The protected system achieved a little over 89 percent of the unprotected system's throughput. Next, gzip was used to compress a 256 MB file, and the operation was timed. The protected system was found to run at 87 percent of full speed. Third, the nbench [34] suite was used to show the performance under a set of primarily computation-based tests. The slowest test in the nbench system came in at just under 97 percent. Finally, the Unixbench [35] Unix benchmarking suite was used as a microbenchmark to test various aspects of the system's performance at tasks such as process creation, pipe throughput, filesystem throughput, etc. Here, the split memory system ran at 82 percent of normal speed. This result is slightly disappointing; however, it can be easily explained by looking at the specific test that performed poorly, which we do below. As can be seen from these four benchmarks, the system has very reasonable performance under a variety of tasks.

If we simply left our description of the system's performance to these four tests, some readers may object that given the description of the system so far and the mention in Section 4.6 of the various sources of overhead, something must be missing from our benchmarks. As such, two benchmarks contrived to highlight the system's weakness can be found in Fig. 7. First, one of the Unixbench testcases called "pipe-based context switching" is shown.

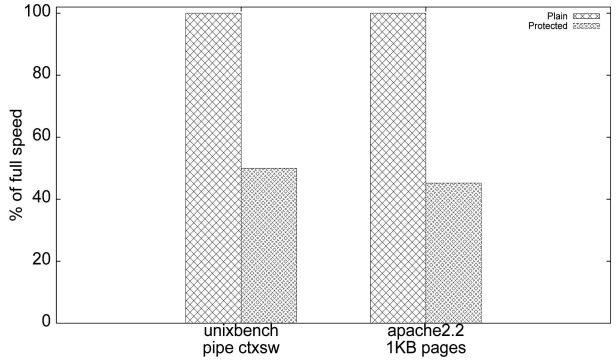


Fig. 7. Stress testing the performance penalties due to context switching.

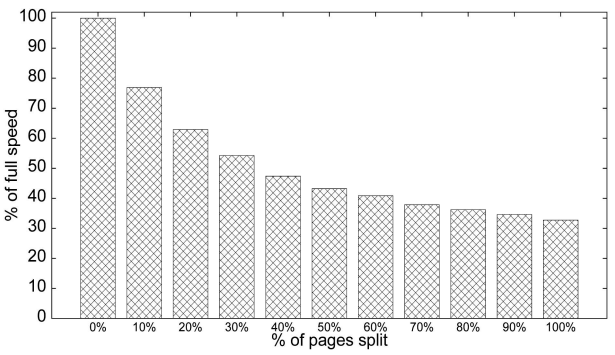


Fig. 9. Unixbench pipe ctxsw with varying percentages of pages being split.

This primarily tests how quickly a system can context switch between two processes that are passing data between each other. The next test is Apache used to serve a 1 KB page. In this configuration, Apache will context switch heavily while serving requests. In both of these tests, context switching is taken to an extreme, and therefore, our system's performance degrades substantially due to the constant flushing of the TLB. As can be seen in the graph, both are at or below 50 percent. In addition, in Fig. 8, we have a more thorough set of Apache benchmarks demonstrating this same phenomena, namely, that for low page sizes, the system context switches heavily and performance suffers, whereas for larger page sizes that cause Apache to spend more time on I/O as well as begin to saturate the system's network link, the results become significantly better. These tests show very poor performance; however, we would like to point out that they are shown here to be indicative of the system's *worst-case* performance under highly stressful (rather than normal) conditions.

As discussed previously, the system can be used in combination with the execute-disable bit. Under that scenario, only an application's mixed pages would be duplicated and split. In order to demonstrate the performance improvements that would come from not splitting every page, the poor-performing Unixbench "pipe-based context switching" benchmark was retested on recent hardware that supports the execute-disable bit (a 2.4 GHz Intel Quad-Core CPU) using a version of our system that splits only a certain percentage of an application's pages, while the rest of the pages are untouched. Since there are no real mixed pages in the benchmark tested, we chose the pages to be split at random for the sake of performance evaluation. Fig. 9 shows the results of this test. As can be seen, performance increases dramatically when a small percentage of an application's pages are being split. When only 10 percent of the pages are split, for example, even this "worst case" test is able to execute at about 80 percent of full speed. This means that memory splitting, when supplemented with hardware-based execute-disable bit, incurs a very low overhead.

Overall, the system's performance is reasonable; in most cases being between 80 and 90 percent of an unprotected system. Moreover, if split memory was supported at the hardware level, as described in Section 3.3, or used to supplement the protection of existing hardware-based systems, the overhead would be significantly reduced.

7 Limitations

There are a few limitations to our approach. First, as shown in other work [36], a split memory architecture does not lend itself well to handling self-modifying code. As such, self-modifying programs cannot be protected using our technique.

Next, this protection scheme offers no protection against attacks which do not rely on executing code injected by the attacker. For example, modifying a function's return address to point to a different part of the original code pages will not be stopped by this scheme. Fortunately, address space layout randomization [17] could be combined with our technique to help prevent this kind of attack. Along those same lines, non-control-data attacks [25], wherein an attacker modifies a program's data in order to alter program flow, are also not protected by this system.

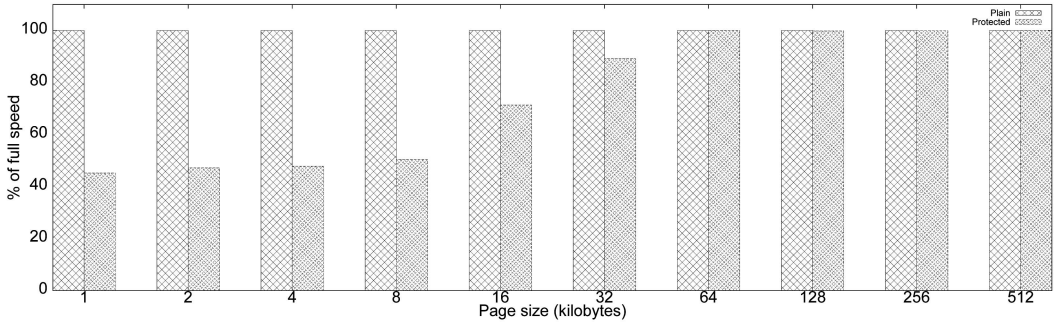


Fig. 8. Closer look into Apache performance.

8 Conclusions

In this paper, we present an architectural approach to preventing code injection attacks. Instead of maintaining the traditional single memory space containing both code and data, which is often exploited by code injection attacks, our approach creates a virtual split memory that separates code and data into different memory spaces. Consequently, in a system protected by our approach, code injection attacks may result in the injection of attack code into the data space. However, the attack code in the data space cannot be fetched for execution as instructions are only retrieved from the code space. We have implemented a Linux prototype on the x86 architecture, and experimental results show the system is effective in preventing and responding to a wide range of code injection attacks in both artificial and real-world scenarios while incurring acceptable overhead.

Acknowledgments

The authors would like to thank Glenn Wurster, the anonymous IEEE Transactions on Dependable and Secure Computing (TDSC) reviewers, and the anonymous reviewers of a preliminary conference version of this paper [37] for their insightful comments and suggestions. This work was supported in part by the US National Science Foundation (NSF) under Grants 0546173, 0716444, 0852131, and 0855297. One of the authors (X. Jiang) is also supported in part by the US Army Research Office (ARO) under grant W911NF-08-1-0105 managed by the NCSU Secure Open Systems Initiative (SOSI). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or the ARO.

References

[1] "A Detailed Description of the Data Execution Prevention (dep) Feature in Windows xp Service Pack 2, Windows xp Tablet pc ed. 2005, and Windows Server 2003," http://support.microsoft.com/kb/875352, Dec. 2006.

[2] "Pax Pageexec Documentation," http://pax.grsecurity.net/docs/pageexec.txt, Dec. 2006.

[3] Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual Volume 3A: System Programming Guide, Part 1.* Intel Corp., publication number 253668, 2006.

[4] "Buffer Overflow Attacks Bypassing dep (nx/xd bits)—Part 2: Code Injection," http://www.mastropaolo.com/?p=13, Dec. 2006.

[5] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *Proc. Seventh USENIX Security Conf.,* pp. 63-78, http://citeseer.ist.psu.edu/cowan98stackguard.html, 1998.

[6] H. Etoh, "Gcc Extension for Protecting Applications from Stack-Smashing Attacks," http://www.trl.ibm.com/projects/security/ssp/, Dec. 2006.

[7] Vendicator "Stack Shield: A 'Stack Smashing' Technique Protection Tool for Linux," http://www.angelfire.com/sk/stackshield/info.html, Dec. 2006.

[8] J. Wilander and M. Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," *Proc. 10th Network and Distributed System Security Symp.,* pp. 149-162, citeseer.ist.psu.edu/wilander03comparison.html, Feb. 2003.

[9] J. von Neumann, "First, Draft of a Report on the EDVAC," 1945, reprinted in, *The Origins of Digital Computers Selected Papers,* second ed., pp. 355-364, Springer, 1975.

[10] P.C. van Oorschot, A. Somayaji, and G. Wurster, "Hardware-Assisted Circumvention of Self-Hashing Software Tamper Resistance," *IEEE Trans. Dependable and Secure Computing,* vol. 2, no. 2, pp. 82-92, Apr. 2005.

[11] H.H. Aiken, "Proposed Automatic Calculating Machine," 1937, reprinted in, *The Origins of Digital Computers Selected Papers,* second ed., pp. 191-198, Springer, 1975.

[12] H.H. Aiken and G.M. Hopper, "The Automatic Sequence Controlled Calculator," 1946, reprinted in, *The Origins of Digital Computers Selected Papers,* second ed., pp. 199-218, Springer, 1975.

[13] "kernelthread.com: Securing Memory," http://www.kernelthread.com/publications/security/smemory.html, Dec. 2006.

[14] Skape and Skywing, "Bypassing Windows Hardware-Enforced dep," *Uninformed,* vol. 2, http://www.uninformed.org, Sept. 2005.

[15] R. Krishnakumar, "Hugetlb—Large Page Support in the Linux Kernel," *Linux Gazette,* vol. 155, http://linuxgazette.net/155/krishnakumar.html, Oct. 2008.

[16] "Pax aslr Documentation," http://pax.grsecurity.net/docs/aslr.txt, Dec. 2006.

[17] S. Bhatkar, D.C. DuVarney, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits," *Proc. 12th USENIX Security Symp.,* 2003.

[18] S. Bhatkar, R. Sekar, and D.C. DuVarney, "Efficient Techniques for Comprehensive Protection from Memory Error Exploits," *Proc. 14th USENIX Security Symp.,* 2005.

[19] J. Xu, Z. Kalbarczyk, and R.K. Iyer, "Transparent Runtime Randomization for Security," *Proc. 22nd Symp. Reliable and Distributed Systems (SRDS),* Oct. 2003.

[20] E.G. Barrantes, D.H. Ackley, S. Forrest, T.S. Palmer, D. Stefanovic, and D.D. Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," *Proc. 10th ACM Conf. Computer and Comm. Security (CCS),* 2003.

[21] G.S. Kc, A.D. Keromytis, and V. Prevelakis, "Countering Code Injection Attacks with Instruction-Set Randomization," *Proc. 10th ACM Conf. Computer and Comm. Security (CCS),* 2003.

[22] S. Sidiroglou, M.E. Locasto, S.W. Boyd, and A.D. Keromytis, "Building a Reactive Immune System for Software Services," *Proc. USENIX Ann. Technical Conf.,* 2005.

[23] L. Lam and T. Chiueh, "Checking Array Bound Violation Using Segmentation Hardware," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '05),* pp. 388-397, 2005.

[24] "Wind River: Vxworks," http://www.windriver.com/vxworks/, Mar. 2007.

[25] S. Chen, J. Xu, E.C. Sezer, P. Gauriar, and R. Iyer, "Non-Control-Data Attacks Are Realistic Threats," *Proc. USENIX Security Symp.,* Aug. 2005.

[26] "bochs: The Open Source ia-32 Emulation Project," http://bochs.sourceforge.net/, Dec. 2006.

[27] P. Venda, "PaX Performance Impact," http://www.pjvenda.org/linux/doc/pax-performance/, Oct. 2005.

[28] A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi, and V. Roy, "Digsig: Run-Time Authentication of Binaries at Kernel Level," *Proc. 18th USENIX Conf. System Administration (LISA '04),* pp. 59-66, 2004.

[29] B. Lymn, "Verified Exec—Extending the Security Perimeter," *Proc. Australian Unix Users Group Conf.* 2004.

[30] "Sebek," http://www.honeynet.org/tools/sebek/, 2010.

[31] X. Jiang and X. Wang, "'Out-of-the-Box' Monitoring of VM-Based High Interaction Honeypots," *Proc. 10th Recent Advances in Intrusion Detection (RAID '07),* Sept. 2007.

[32] G. Portokalidis, A. Slowinska, and H. Bos, "Argos: An Emulator for Fingerprinting Zero-Day Attacks for Advertised Honeypots with Automatic Signature Generation," *Proc. ACM SIGOPS/European Conf. Computer Systems (EuroSys '06),* pp. 15-27, 2006.

[33] A. Avizienis, J.C. Laprie, and B. Randell, "Fundamental Concepts of Dependability," *Proc. Int'l Workshop Information Security (ISW '00),* 2000.

[34] "Linux/Unix nbench," http://www.tux.org/mayer/linux/bmark.html, Dec. 2006.

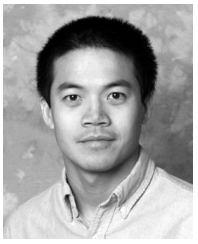
[35] "Unixbench," http://www.tux.org/pub/tux/benchmarks/System/unixbench/, Dec. 2006.

[36] J. Giffin, M. Christodorescu, and L. Kruger, "Strengthening Software Self-Checksumming via Self-Modifying Code," *Proc. 21st IEEE Ann. Computer Security Applications Conf. (ACSAC '05),* pp. 18-27, Dec. 2005.

[37] R. Riley, X. Jiang, and D. Xu, "An Architectural Approach to Preventing Code Injection Attacks," *Proc. 37th Ann. IEEE/IFIP Int'l Conf. Dependable Systems and Networks (DSN '07),* pp. 30-40, 2007.

Ryan Riley received the BS degree in computer engineering and the PhD degree in computer science in 2009 from Purdue University. He is an assistant professor of computer science at Qatar University in Doha. His current research interests include virtualization technologies, malware, and operating system security. He is a member of the IEEE.

Xuxian Jiang received the BS degree from Xi'an Jiaotong University in 1998 and the PhD degree in computer science from Purdue University in 2006. He is an assistant professor of computer science at North Carolina State University. His current research interests mainly focus on improving the security and reliability of commodity operating systems against various malware attacks. He is a member of the IEEE.



Dongyan Xu received the BS degree from Zhongshan (Sun Yat-Sen) University in 1994 and the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 2001. He is an associate professor of computer science and electrical and computer engineering (by courtesy) at Purdue University. His current research interests include virtualization technologies, computer malware defense, and cloud computing. He is a member of the IEEE and a recipient of the US National Science Foundation CAREER Award.