Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow

Nathan Tuck Brad Calder George Varghese

Department of Computer Science and Engineering
University of California, San Diego
{ntuck,calder,varghese}@cs.ucsd.edu

Abstract

Buffer overflow vulnerabilities are currently the most prevalent security vulnerability; they are responsible for over half of the CERT advisories issued in the last three years. Since many attacks exploit buffer overflow vulnerabilities, techniques that prevent buffer overflow attacks would greatly increase the difficulty of writing a new worm.

This paper examines both software and hardware solutions for protecting code pointers from buffer overflow attacks. We first evaluate the performance overhead of the existing PointGuard software solution for protecting code pointers, and show that it can be applied using binary modification to protect return pointers on the stack. These software techniques guard against write attacks, but not read attacks, where an attacker is attempting to gain information about the pointer protection mechanism in order to later mount a write buffer attack. To address this, we examine encryption hardware to provide security for code pointers from read and write attacks. In addition, we show that pure software solutions can degrade program performance, and the light-weight encryption hardware techniques we examine can be used to provide protection with little performance overhead.

1. INTRODUCTION

This paper focuses on protecting code pointers against buffer overflow attacks. A buffer overflow occurs when an intruder passes data to the victim that is longer than the buffer the victim has allocated for the data. If the victim does not do bounds checking, the attacker can read or write data which he or she is not intended to have access to, and can cause the victim program to execute in an unintended manner. In the worst case, the attacker can cause the victim machine to run code sent by the attacker, and can potentially even gain administrative control over the machine.

Even the simple ability to run arbitrary user-level code allows the attack to unleash a *worm* wherein the victim is now enlisted in the task of attacking other randomly chosen victims. If the attacker can go further and gain administrative control, arbitrary damage can be done to the victim machine(s).

Three fairly recent worms provide good examples of remote buffer overflow attacks. First, Code Red exploited a buffer overflow in Microsoft's Internet Information Services (IIS). Next, the Slammer worm exploited a buffer overflow in SQL network messages. Recently, the Blaster worm exploited a buffer overflow in DCOM. Each of these attacks only required a machine to be hooked up to a network with the available port open. While these worms only ran at normal user privileges, the resulting denial-of-service due to the computation and bandwidth used by the worm had a very high cost.

Year	Advisories	Advisories of Buffer Overflows	Percent Overflow
1996	27	5	19%
1997	28	15	54%
1998	13	7	54%
1999	17	8	47%
2000	22	3	14%
2001	37	19	51%
2002	37	21	57%
2003	28	18	64%
2004	9	7	78%
Total	218	103	47%

Table 1: Summary of recent CERT advisories

Given the cost of these worms and the fact that they exploit buffer overflows, buffer overflow vulnerabilities are recognized [27] as one of the most prevalent security vulnerabilities. In Table 1 (derived from the recent data on the CERT website and [13]) we summarize the number of buffer overflows over the last eight years and the fraction which have been buffer overflow related. The table shows that for the last four years, buffer overflow accounts for over 50% of the CERT advisories per year. The one exceptionally low year is 2000, for which a disproportionate number of the advisories came from browser-related vulnerabilities.

Unfortunately, buffer overflow vulnerabilities are exceedingly easy to generate in C code, either through the use of unsafe library functions or through sloppy programming practices. As such, C is particularly prone to a range of static and dynamic buffer overflows, even in well audited code.

An attacker can take control of a machine with a buffer overflow attack by overflowing the buffer and overwriting a function pointer. Then when the function pointer is invoked via a return instruction or a jump, the overwritten value will jump to the attacker's code (which was written into the buffer), or to a valid routine (e.g., system/shell routine) in the existing binary with parameters supplied by the attacker.

Because new buffer overflow vulnerabilities are constantly being found and because buffer overflows can cause serious damage, companies that provide Internet services are in a costly arms race with criminals and delinquents from around the world who seek to subvert their systems.

The standard approach, exemplified by Norton Antivirus or Snort [19], is to detect and block a signature of the bad data used in the attack. Unfortunately, this approach is akin to posting a picture of a burglar after he has robbed a thousand banks. Much damage has already been done, and

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 **IEEE**

IEEE COMPUTER SOCIETY

it requires considerable human effort to extract signatures for each new vulnerability and each new attack. Further, a small change in the attack code will most likely not be detected by current antivirus tools.

Signature detection can be regarded as adding checks for the specific tool the burglar uses (e.g., a crowbar). Why not secure the valuables in the first place? In other words, why not add mechanisms that will prevent buffer overflows — the root cause of many attacks — in the first place? The research community has proposed many innovative solutions to this problem without as much overhead as doing run-time bounds checking. One set of software techniques requires the binary to be modified using compiler techniques to guard function addresses. Other hardware approaches require the use of a separate return address stack, which is not generally applicable to all types of branching constructs used in binaries.

In this paper we examine both software and hardware solutions for protecting code pointers from buffer overflow attacks. We first evaluate the performance overhead of the existing PointGuard [6] software solution for protecting code pointers, and show that it can easily be applied to protect return pointers on the stack using binary modification. PointGuard stores the code pointers in an obfuscated form in memory. It protects a code pointer by XORing a key with it to obfuscate the value, and performing an XOR to unobfuscate the value. These software techniques guard against write attacks, but not read buffer attacks, where an attacker is attempting to learn the key in order to mount a write buffer attack. To address this, we propose using encryption hardware to provide efficient security for code pointers from read and write attacks.

The contributions of this paper are:

- We examine applying PointGuard using automatic binary modification, and show that this works by booting the linux kernel with these changes in place. This requires handling call and return instructions that do not have matching return and call instructions. In doing this, we find that previously proposed pure hardware return address protection techniques would not work without performing this binary modification on the program first.
- We examine reducing the overhead of software solutions like PointGuard by providing hardware to protect code pointers by adding special purpose encryption (e-store) and decryption (d-load) instructions. We examine using these as an XOR-Key mechanism to provide obfuscation of code pointers with little performance overhead.
- We examine using these instructions to protect against read attacks using cryptographic encryption. We propose using a Random Hash Table and a hardware Luby-Rackoff Feistel Network for function pointer cryptographic encryption/decryption for processes that need to have read buffer attack protection. To achieve low overheads for strong read attack security, we propose a caching mechanism for the encrypted code pointers that still provides the same protection semantics.

In Section 2 we discuss the types of buffer overflow attacks that are of concern, and Section 3 provides an overview of related work. Section 4 presents the methodology we use for evaluating our design. Section 5 describes using automated binary modification to apply PointGuard buffer overflow protection. In Section 6, we discuss using special purpose instructions to aid pointer encryption and decryption as well as hardware techniques to provide read buffer attack protection. Finally in Section 7 we draw our conclusions.

2. TYPES OF ATTACKS

In this section, we classify buffer attacks into a few major categories which an ideal system would guard against. At the highest level, buffer overflow attacks can be either write or read attacks: write buffer overflow attacks can change the behavior of programs and thereby subvert systems, while read buffer overflow attacks can be used to obtain privileged information from a program. There are several subcategories of write and read attacks that we now describe.

2.1 Write Attacks to Gain Process Control

A write attack attempts to gain control of a process/thread by overwriting a function pointer. This modified function pointer will allow execution of either code inserted by the attacker's buffer overflow or will call an existing routine in the process' address space with attacker supplied arguments.

In order to launch this attack the attacker must be able to find or place into the process address space code that can be used to perform desired actions, and then cause the program to start running this code. Although many processors are now attempting to make it more difficult to execute code created in a data area, it is still very easy to find code already in the program to exploit with attacker supplied arguments. Standard library system and shell invocation routines are just a few examples. Once the attack code location is determined, the attacker can force control flow into that piece of code by altering return addresses, function pointers, or `setjmp`/`longjmp` structures. We will now discuss each of these attacks on program control flow.

2.1.1 Return Address Attack

The first and most common attack is a *return address* attack. In most ISAs and languages today, when a procedure call is invoked, the return address is (either lazily or immediately) saved on the stack. If a buffer overflow can be exploited to overwrite values on the stack, then the return address can be overwritten with the address of a function chosen by the attacker. When the program attempts to return from the function call, the program will begin executing the code chosen by the attacker.

2.1.2 Function Pointer Attack

The second type of write buffer overflow attack is a function pointer attack. This attack uses a buffer overflow to overwrite a function pointer stored in the stack or heap to point to the code of the attacker's choice. In this attack, the attacker is not necessarily able to choose the arguments passed to the function; hence this attack has been less used than the return address attack. The most common uses of this attack are overwriting the Global Offset Table (GOT) or virtual function pointers [4].

2.1.3 Setjmp/Longjmp

A third kind of write buffer overflow attack is the `setjmp` / `longjmp` attack. `setjmp` and `longjmp` are used in C to implement non-local gotos and exception handling. `setjmp` stores the program's current register values, processor state, and

IEEE COMPUTER SOCIETY

PC; `longjmp` uses `setjmp`'s data structure to return to that point in the code. Because of the amount of state restoration incurred by `longjmp`, if `setjmp`'s data-structure can be overwritten by a buffer overflow, the attacker can choose to return to any place in the code with almost any set of register state he or she chooses. This makes `setjmp`/`longjmp` exploits particularly dangerous, but they are somewhat less common because `setjmp`/`longjmp` and similar constructs have been generally deprecated in favor of structured exception handling (which in turn can be defeated with a function pointer attack described above).

2.2 Data Corruption/Manipulation Attacks

Data corruption attacks involve an attacker changing the behavior of the program by overwriting data rather than code pointers. These attacks can have just as devastating effects as attacks on code pointers, but are generally more difficult to mount. Our techniques could be used against some of these attacks, but we do not address these attacks in this paper.

2.3 Read Attacks

Read attacks occur either when a user is able to trick a program into returning a longer response than it has allocated buffer space for (in which case it is called a read overflow attack) or when the program accidentally returns data which has not been properly initialized. By exploiting this, an attacker can take snapshots of memory that may include stack contents, function pointers, and internal state that is meant to be kept secret.

Read attacks are potentially a threat in code which attempts to provide informational services to other pieces of code, particularly when both sides must negotiate expected sizes of information passed in a shared buffer. When one side forgets to check a size returned by the other side of the protocol, it can end up reading past the buffer and into its own data structures when preparing a response. In another common (but less useful) incarnation, the attacker can request information "early" in a protocol causing the return of seemingly uninitialized data. Because this memory is likely to be used for other purposes in a long running process, it is possible that valuable information will leak; this can give a determined attacker enough insight to launch a write attack to gain control of the process.

CERT Vulnerability Note VU#738331 is an example of a read overflow attack. The library used for DNS name resolution passes a buffer of insufficient length into a function which fills it in and returns an integer that indicates the size of the buffer that would be required to fill in a complete response. However, this integer is treated as the length of the actual response; the library then returns extra data from past the end of the buffer to the attacker.

These read attacks, though currently uncommon, devastate most prior proposed methods of buffer overflow protection. Because most of these methods are based on XOR rather than true cryptographic security, they can be broken with a single successful read attack of a single known protected value.

The attack against an XOR encrypted key can be mounted as follows. First, one assumes that the attacker has access to the same binary that is used by the victim (a typical case given that most systems do not run fully custom software). By XORing the plain-text version of the pointer (known from the binary) and the encrypted value of the pointer (known from the read overflow), the attacker now has the XOR-key used by the process. The attacker can then encrypt any desired address and use it in a write buffer overflow attack against the same process.

2.4 Our focus

We propose hardware techniques to provide encryption of function pointers to guard against read and write attacks targeted at taking control of a process. We do not address any data corruption/manipulation attacks. Some of our techniques can be used against some non-code pointer attacks, but our current experimental framework did not allow us to examine this case.

Wilander and Kamkar [28] present a taxonomy of 20 different categories of buffer overflow attacks and test the effectiveness of then-current buffer overflow techniques. A full implementation of PointGuard protects against all overflowing pointer attacks. In Section 5 we present detailed emulation results for automatically applying PointGuard using binary modification to provide protection against the 3 attacks related to overwriting the return address. Then in Section 6 we provide performance simulation results evaluating the overhead of protecting against 14 of the attacks specified in [28].

3. RELATED WORK

3.1 Prior Work in Software Protection

Considerable effort has been expended on devising software solutions to this problem [28]. Replacements for system libraries, static analysis, and altered compilers have all been proposed.

One of the most well known software solutions to buffer overflow is StackGuard [5]. StackGuard protects the stack from buffer overflow attacks by placing an extra value called a *canary* on the stack in front of the return address. Before a function returns, it loads both the return address and the canary and checks that the canary is the expected value. StackGuard supports multiple kinds of *canaries*. A canary can be a *terminator canary*, which prevents the attacker from using a string library function such as `strcpy` to replace it. Alternately, a canary can be a special (possibly random) number XORed with the return address. In the first case, the attacker cannot attack by misusing a standard library function; in the second case, the attacker must be able to guess a weakly hidden value. In particular, any code which has both a read buffer overflow and a write buffer overflow can be exploited to circumvent the XOR cipher.

StackGhost [10] implements (in software) an XOR encryption of the return address value. StackGhost modifies the SPARC kernel register window overflow handler to save modified return addresses by either XORing return addresses with a per kernel or per process value. Protecting the return address occurs whenever a register window overflow (encryption) or underflow (decryption) occurs. They show a maximum of 40% overhead using their technique with a carefully constructed microbenchmark and a 0.1-0.4% overhead running the SPEC95 benchmarks. The StackGhost approach proposed can only be implemented on SPARC due to its unique architectural feature of register windowing. The StackGhost approach is susceptible to a single read buffer overflow attack which motivates us to consider stronger hard-

IEEE COMPUTER SOCIETY

ware assisted encryption functions.

Compiler based software techniques other than StackGuard include safe dialects of programming languages [12] and static detection of buffer overflow vulnerabilities [27, 26]. The disadvantages to these approaches are that they require recompilation of a program and even rewriting the code. They also typically incur high overhead in the resultant binary due to array bounds checking.

Library based approaches such as libsafe [1] and libverify [2] avoid the problem of recompilation, but generally do it at the expense of missing attacks. libsafe contains bounds-checked versions of unsafe C library functions. libverify copies all functions and return addresses into the heap and patches up the copied code to check the return address on exit. libverify is less efficient than StackGuard in memory space and can degrade program performance by up to 15% [2].

3.2 Prior Work in Hardware Protection

Hardware protection against buffer overflows has also recently become a popular topic for research. With the increase in industry interest in digital rights management and secure systems such as TCPA and Palladium [20], many approaches have been proposed which could also be used to stop buffer overflows.

Lee et. al. [13, 16] and Ozodganoglu et. al. [18] independently present a method of protecting the return address stack by copying the return address and putting it on another hardware managed stack with backing store in a protected area of memory. When the code returns from a call, the processor checks to make sure that the return value matches the value stored on the duplicate stack. They explore both hardware and operating system initiated save-on-overflow. This method has been shown to have less than a 1% effect on performance for a cached hardware stack size of 128.

This approach requires extra backing store for every process to save the duplicate stack to and introduces some complexity to the processor in handling stack overflows. An interesting challenge that must be solved for these approaches is as follows: they must keep the protected return address stack in sync when dealing with constructs that arbitrarily pop off several stack frames like setjmp/longjmp (and other APIs which do not use stack entries in the order they were pushed on the stack). Instead of this approach, we examine using hardware encryption to leave the encrypted return value on the stack. In doing this, we do not have to do anything special to correctly handle constructs that pop off several stack frames at a time like setjmp/longjmp. Also, this approach does not require a backing store for the hardware stack like the return stack approaches. In addition, in Section 5 we show existing code sequences in the linux kernel that would break the return address stack semantics causing the linux kernel to not run. To address this, we examine using an automated binary modification approach for fixing these code sequences in Section 5.

HSAP [22] employs a stack smashing protection which does not allow variables passed into other functions to be allocated on the stack. More similarly to our work, it also provides an extra register and XOR instruction for protecting function pointers, but does not concentrate on maintaining existing code compatibility when providing protection. More recently, HSDefender [21] was proposed concurrently with our work examining the use of an XOR encrypted call/return mechanism. Both HSAP and HSDefender are proposed with weak XOR encryption, and do not provide much performance analysis of their design space. In comparison, we focus on encryption to protect against read attacks, provide code compatibility with binary modification, and provide detailed analysis and comparison between this and other protection techniques.

Other research in secure computing such as XOMOS [14] could be used to solve the buffer overflow problem through the use of signed memory. In order to implement this, every piece of code which wishes to protect itself from the execution of other pieces of code must have a key. The key is used to cryptographically sign each piece of memory (for instance a cache-line) which is written back to storage. Untrusted pieces of code are given a different key (or no key at all) and a separate piece of memory to execute with. If the untrusted code attempts to overwrite memory which it does not have privileges to write to, it will be unable to create a valid corresponding signature. When the memory is used, the incorrect signature will be detected and the code will trap before any damage is done. This approach could be used to provide protection against buffer overflows, but would require users to completely rewrite their application partitioning out all potential code that could have a buffer overflow from the main parts of the application. For some code this might not be possible. In addition, asking programmers to deal with explicit issues of trust and privilege has been problematic, as the number of buffer overflow vulnerabilities in the CERT databases will attest to.

More recently Crandall and Chong [7] proposed using an additional integrity bit for each memory word to identify trusted data vs non-trusted data. Initially all data in the system is set to be high integrity. A low-water-mark scheme is then used to propagate the integrity bit. On control transfers such as jump, call and return, the integrity bit is checked and an exception is triggered if it is not set.

4. METHODOLOGY

To verify our claims of compatibility for the techniques we examine, we first implemented the PointGuard protection and our techniques in the IA-32 architecture emulator Bochs [3]. In doing this, the return address pointers on the stack are actually encrypted and decrypted to test that the approach works. We then successfully used Bochs to boot and run a real linux kernel and some linux programs, showing that return addresses can be protected using binary modification as described in the next section.

We also measure the performance overhead of PointGuard and our proposed encryption techniques using cycle level simulation [25]. We simulate a *single threaded* wide-issue out-of-order processor with baseline architectural parameters shown in Table 2. The parameters are meant to be similar to those of a slightly aggressive desktop CPU. Just as in Bochs, our simulations actually perform buffer overflow protection (encryption and decryption) for code pointers.

5. AUTOMATIC BINARY MODIFICATION FOR RETURN ADDRESS PROTECTION

In this section, we examine using binary modification to provide return stack address protection.

IEEE COMPUTER SOCIETY

Pipeline	20 stages
Fetch	Up to 8 instructions from 1 cacheline
Predictor	2bcgskew, 64K entry Meta and gshare, 16K entry bimodal table
Prefetcher	PC based stride, 256 entry with 8 stream buffers
ROB	128 entry
HW Registers	126
Queues	64 entries each IQ, FQ, and MQ
Issue	6 instructions per cycle, up to 4 Integer, 2 FP, 2 load/store
ICache L1	32 KB 2-way set associative, 2 cycles latency
Data L1	16 KB 2-way set associative, 2 cycles latency
L2	512 KB 8-way set associative, 20 cycles latency
L3	2 MB 16-way set associative, 40 cycles latency
Memory	450 cycles latency

Table 2: Simulator Architectural Parameters

5.1 PointGuard

PointGuard [6] is a recent approach to buffer overflow vulnerabilities using a compiler solution to the buffer overflow problem for write attacks. The PointGuard compiler generates a random key which is generated at process creation time and stored in a protected area of memory. The same key is then XORed with every pointer that is loaded from or stored to memory for a process. By effectively obfuscating pointers stored in memory, PointGuard prevents pure write buffer overflow attacks from succeeding in changing control flow to known locations. However, it is trivial to exploit if there is a read attack in the program which leaks the value of any code pointer in the program. Although the PointGuard approach is quite comprehensive, the overhead of doing this in software can result in up to 20% slow-down for the benchmarks examined in [6]. Our hardware extension in Section 6 provides efficient hardware encrypt/decrypt instructions that PointGuard could use to provide this level of protection with low overhead and security against read attacks.

5.1.1 Calls and Returns

In order to properly handle calls and returns in PointGuard, there is a minimal set of operations that must be implemented for every call and return in the code. For calls, PointGuard encrypts the return address at the destination call site, since the return value was just pushed onto the stack by the call. This following code sequence would be added at the start of each procedure to encrypt the return address:

```
pop r0  (pop off the return address)
load r1 key_location
xor r0 r0 r1
push r0  (push obfuscated return address on stack)
```

As we can see, this adds overhead of three memory references and one ALU operation per call. In order to properly handle the corresponding return, PointGuard must replace each return with the following sequence to decrypt the return address:

```
pop r0
load r1 key_location
xor r0 r0 r1
push r0
return
```

This also adds 3 memory references, and an ALU operation in overhead to our return. Note, the push and return could be replaced by an indirect jump to r0, but this would interfere with the return address stack predictor used in modern processors.

When functions are small, the above overhead can be quite large. Performing binary translation of these sequences is not difficult, but does require relocating instructions due to the extra instructions introduced. In Section 6 we present providing hardware support for special purpose instructions, which eliminate the performance and binary modification overhead for call/return pairs. By allowing the call and return to automatically perform the encryption and decryption in hardware, the call and return instructions do not need to be changed in the binary. Then the binary modifier only needs to patch and transform the instruction sequences in Sections 5.2.1 and 5.2.2. This will significantly simplify and increase the speed of the binary patching process for providing pointer protection, potentially allowing it to be performed at process load-time without much overhead.

5.2 Protecting Calls and Returns using Binary Modification

A goal of this study is to examine applying PointGuard to commercial code using binary modification. To do this we implemented our binary modification inside the IA-32 architecture emulator Bochs [3]. In Bochs we changed the behavior of the call and return instructions to automatically obfuscate every return pointer pushed by a call and to de-obfuscate the pointer used in every return. We then successfully used Bochs to boot a linux kernel showing that our binary modification approach works on a complicated piece of code.

For our study, we are mainly concerned with the x86 ISA, which is currently the most widely deployed (and attacked) ISA in the world. Programs compiled for this ISA use direct and indirect `call` instructions to perform procedure calls, which *implicitly* store the PC on the stack; programs also use `return` instructions that pop the return address off the stack and assign it to the PC. While this is the majority of the calls and returns seen, there are cases where these semantics are not used. These cases will cause pure hardware techniques [18, 13] to break, unless those code sequences are removed. This is because these techniques assume an ISA that makes strict usage of call and return pair semantics (every call instruction is matched by a return instruction).

In implementing our binary modification technique, the two interesting code sequences we found that needed special handling are unmatched call and return instructions. We found that using binary modification either statically or dynamically at load-time can be used to accurately transform these cases and provide return address protection.

5.2.1 Calls without Corresponding Returns

Code generated by a compiler will generally use standard call/return semantics for the ISA. But for some legacy x86 code or assembly code it is always possible that the return will be implemented by some manual means, which does not match the normal semantics. To address this, these code sequences must be corrected, and binary patching provides an efficient mechanism to accomplish this.

For the x86 ISA, the PC register cannot be directly read; thus the only practical way to get its value onto the stack is

IEEE COMPUTER SOCIETY

to perform a call instruction. We found when running the linux kernel with return address protection that there are many locations where a local PC is obtained in a register by performing a jump to the *very next instruction* and the resulting PC is then popped off the stack and used as input to an indirect jump (e.g., switch table). A short call appears as follows:

```
call next_instruction;
pop  r0;
```

We can easily identify these calls by looking for any call that has a destination equal to the next PC. More generally, it is any call that has a destination where the return is popped off the stack into a register. When these "short calls" are identified, they are not obfuscated. After implementing this in bochs, we found no instances where procedure calls were improperly obfuscated. Note, encryption can be provided for the above code sequence if a d-load (described in the next section) along with a stack pointer adjustment was used instead of the pop instruction.

5.2.2 Returns without Corresponding Calls

In running the linux kernel we found locations in the code that had return sequences without corresponding calls. These code blocks appear as follows:

```
push r0 (save an indirect jump address
         on top of the stack)
...
arbitrary code sequence
...
return (pop the indirect jump address off
        the top of the stack and jump to it)
```

This code sequence occurred because a register contained the value for an indirect jump. Then due to register pressure or a procedure call, the register was saved on the stack to be restored later. The binary would normally execute a pop and indirect jump, but this is the exact semantics of a return instruction, but with 1 less instruction. Therefore, to save an instruction the compiler used a return instead of a pop and indirect jump code sequence.

At the binary level, this is harder to detect than our earlier code sequence. We found that the push and return (pop+jump) occurred in the same function, and it was therefore possible to detect that they are accessing the same stack location. During binary modification for the Bochs emulation, we replace the return with a pop and indirect jump, so this code pointer on the stack would not be protected. Note, an alternative would be to use the e-store (described in the next section) along with a stack pointer adjustment instead of the push to provide encryption protection for the code pointer.

In performing the above simple transformations we were able to boot and run a real linux kernel and some linux programs, showing that return addresses can be protected using binary modification.

6. HARDWARE SUPPORT FOR CODE POINTER PROTECTION

In this section we examine the benefit of adding hardware support to more efficiently provide encryption of the code pointers to guard against read and write buffer overflow attacks.

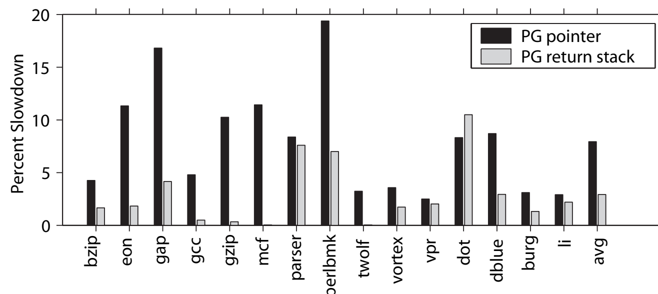

Figure 1: Percent slowdowns on SPEC2000 and other benchmarks using PointGuard techniques applied to either all code pointers or only to return addresses.

6.1 Motivation

Because of the weaknesses in PointGuard, we explore in this section (1) reducing the overhead of encrypting and decrypting the code pointers using special purpose instructions, and (2) adding protection for read attacks by using cryptographic security. In addition, this approach reduces the overhead and complexity of the binary patching for protecting return addresses as discussed in Section 5.1.1.

The PointGuard approach does not have security against read attacks, whereas the approach we describe next offers cryptographic security. A fundamental concept of security is that one must assume that the attacker has access to everything on a system except the key. This would include compiled binaries (which are widely distributed). This makes it trivial for the attacker to recover the key from a single pointer read if XOR is used.

Figure 1 shows that PointGuard induces a 7% slowdown across the SPECint 2000 benchmark suite and a handful of C++ programs when encrypting and decrypting all pointers in memory and approximately a 2% slowdown when encrypting only return stack addresses. However, some benchmarks do far worse. For example, gzip, parser and perl slow down quite a bit with perl slowing down almost 19% with PointGuard applied. Large commercial applications tend to have lots of function calls and heavy use of object-oriented code, both of which put heavy demands on software encryption/decryption. We suspect that the slowdowns for PointGuard on these applications would be even greater than what we were able to measure.

6.2 Encrypt-Stores and Decrypt-Loads

6.2.1 Targeted Encrypt/Decrypt Instructions

The fundamental idea behind our approach is that the instruction set architecture (or internal micro-op ISA) provides the following two extra instructions:

- E-Store - The store encrypts the value to be stored and the resulting value is stored at the effective address.
- D-Load - The value loaded is decrypted and the decrypted value is stored into the register destination.

When a function pointer or return address is stored, the value is encrypted using one of the hardware techniques described in Section 6.4. When the address is loaded back and used, it is decrypted with the inverse function. Our mechanism for preventing buffer overrun attacks rely upon the following properties:

IEEE COMPUTER SOCIETY

- Code pointers being protected (either only return address or all code addresses) are written using e-store instructions and only loaded using d-load instructions for all trusted store and load instructions.
- Untrusted store instructions that do not need to have access to code pointer data use a normal non-encrypted store instruction.
- Untrusted load instructions that do not need to have access to code pointer data use a normal non-decrypted load instruction.

An untrusted store or load instruction, and e-store and d-load instructions can write and access all parts of the processor's normal address space. Whether the store/load is trusted or not determines whether it does the encryption/decryption.

With the above enforced at the instruction execution level, an attacker filling a buffer will be using a non-encrypted store of the data for a code pointer. But when the code pointer is used, a d-load will decrypt the data; if it was not correctly encrypted the code pointer will jump to an undetermined location, which may or may not be in the valid address space. Therefore, the attacker has to know the cryptographic function in order to be able to write the correct value to assume control of the process.

6.2.2 A Failed Attack, Detection, and the Aftermath

Assuming the encryption of the value is cryptographically secure, an attacker who overwrites the return address will have no chance of creating a value which decrypts to anything other than a random value. When an attack fails the decryption (d-load) will decrypt the value to an apparently random address which may even be illegal or unaligned. This will cause the program to crash either due to an illegal memory access, or incorrect program execution. We address the issues and probability of the attacker succeeding with each of the hardware encryption techniques described in Section 6.4.

An important aspect of any protection scheme to detect attacks and to prevent the attacker from retrying or replaying the attacks. We discuss these two issues next.

6.2.2.1 Detecting the Attack.

The above scenario will cause the process or daemon to fail and restart with a new key. This is the primary form of detection for our approach. Existing intrusion detection methods and operating systems already check for this behavior and flag these processes [6]. If this occurs, then the system will flag the user or system administrator and not restart the process. We feel that this approach is sufficient for detecting attacks.

6.2.2.2 Preventing Retrying and Replaying the Attack.

The scenario described above will occur during an attack, which will ultimately result in the restarting of a process or daemon. To prevent retrying and replaying from succeeding, at process creation time, a random number is created and used as a key for a cryptographic engine. This means that all machines and all processes use different keys. A similar idea is used in PointGuard [6], where they XOR a random per-process random key with code pointers.

In addition, to provide additional protection for the stack, we assume a randomized starting stack location is chosen at program startup. This is not needed for our approach, but we believe helps add yet another level of randomness across different machines and processes. This technique is currently being applied to several operating systems such as several flavors of BSD and RedHat Linux.

6.2.3 Targeted Code Pointer Protection

Our approach uses encryption to protect function pointers. We examine two approaches. The first providing protection for only return addresses on the stack, and the second provides protection for all code pointers.

6.2.3.1 Automated Return Address Protection.

A goal of this research is providing protection of return addresses on the stack requiring minimal binary modification. An architecture that performs an e-store to the stack for a call and a d-load whenever a return instructions is executed will allow binaries to automatically benefit from return address protection. Then the only modification to the binary that needs to be performed is to transform the few unstructured call or return sequences in the binary as discussed in Sections 5.2.1 and 5.2.2. To handle these two cases, calls without returns would be replaced an "unencrypted call" instruction, and returns without calls would be replaced with "non-decrypting return" instructions.

6.2.3.2 All Function Pointers and Setjmp/Longjmp.

PointGuard can be used to provide protection for general function pointers and `setjmp` / `longjmp` (as discussed in Section 2). In Section 6.4, we examine using hardware support to provide efficient protection in the presence of read attacks. Having special purpose e-store and d-load instructions significantly reduces the overhead for the encrypt/decrypt; it also provides a level of protection in the presence of read and write attacks. In Section 6.5, we examine the overhead of using our different hardware techniques for protecting all code pointers in memory using the e-store and d-load instructions.

6.2.4 Issues

Our code pointer encryption approach efficiently handles several important issues for stack and function pointer protection, which we now discuss.

6.2.4.1 Copying Memory.

It is important that our encryption approach still allows encrypted data to be moved around memory and to be correctly used. The hardware encryption techniques we examine in the next section are based only on the value of the data being stored. This allows the encrypted data to be moved around memory using normal load and store instructions. A memory copy would use a normal untrusted store and untrusted load instruction. Therefore, the encrypted data would be read with a normal load into a register. At this point the register contains the encrypted value. Then a normal store instruction would store that encrypted value back into memory. The value can later be read with a d-load instruction to decrypt and use the correct code pointer value.

6.2.4.2 Jumping Around The Stack.

As in PointGuard, our approach stores the encrypted value on the stack in the location of the return address, so we do not have to do anything special to correctly handle constructs that pop off several stack frames at a time like `setjmp` / `longjmp`. In comparison, other hardware techniques [18,

IEEE COMPUTER SOCIETY

16] have to have special mechanisms to restore their return address stack to the correct location.

6.2.4.3 *Continuations.*

There are programming constructs, such as Continuations and Futures [17], which allow a stack to be shared across threads. For these programs, a set of continuations (stack pointers) is kept; when a thread stalls due to a lock, it stores its stack pointer into the continuation list. It then sees if there is a ready continuation to execute, and starts running using that stack pointer (continuation). Therefore, the stacks being used for this parallel program are shared across multiple threads. For this approach, threads would be assigned the same key, since they are in the same process. Since we store the encrypted data in the return location on the stack and the threads share the same key our approach works for these types of programs and systems.

6.2.4.4 *Replay Attacks.*

One category of attacks which we do not handle is what we will call "replay attacks" where a known good function pointer is read and copied verbatim to a new location where it is hazardous. An example of where this style of attack can be used is in the GOT, where functions from libc can be swapped with potentially devastating effect. Preventing these kinds of attacks requires making the GOT read only at load-time, or preventing the copying of function pointers except through use of d-load/e-store for the GOT. Thus our scheme provides only strong read attack security without replay protection as defined in the next section.

6.3 Levels of Security

In this section, we describe our taxonomy of security of pointer-protection relative to protection against an attacker who is capable of both read and write attacks. To put our work into a framework more familiar to the security community, we assume a program with an arbitrary number of read and write buffer holes in it and define an adversary who would like to inject a pointer into our program that will cause it to execute malicious code. This adversary is allowed a certain number of read attacks where he can view memory (e.g., the value of an encrypted pointer) in our program. After these reads, he is allowed to issue write attacks that attempt to direct control flow to a place of his choosing. If the attacker can be expected to direct control flow to the area of his choosing with less total effort than randomly issuing write attacks, then the attacker has broken the pointer protection scheme.

We classify the security of an underlying pointer protection mechanism by the ability or inability to provide protection against read attacks and the additional ability to provide protection against replay attacks.

6.3.1 *Weak Read Attack Security*

We define a pointer protection scheme with weak read attack security to be – if an adversary cannot perform a read attack then they cannot launch a successful write attack. Techniques with this level of security must rely upon no leakage from the program of values encrypted by their key. PointGuard [6] fits this model well. PointGuard relies on an attacker not being able to use a read attack. If the attacker can "see" a single (known) pointer value encrypted by PointGuard's XOR function, then the attacker can immediately launch an attack against the still running program.

6.3.2 *Strong Read Attack Security*

The next level of security is what we call strong read attack security. We define this level of protection to be – if adversary can perform a read attack they cannot launch a successful write attack. At this level, it is assumed that the attacker can read values from memory, and it still requires great effort to correctly inject a pointer to take control of the program's execution. Given that read buffer attacks are rarer than write attacks and that the bar to actually implementing a replay attack (described in Section 6.2.4.4) is significant, this form of security may be adequate for systems.

6.3.3 *Strong Read Attack Security with Replay Protection*

We define strong read attack security with replay protection to be a pointer protection system in which read attacks and replay attacks give the attacker no significant advantage. This is the eventual goal of the secure processor community, but poses substantial challenges for naive implementation with existing software.

6.4 Three Levels of Hardware Encryption

In this section we present a description of three levels of hardware support to provide encrypted protection against hardware buffer overruns.

6.4.1 *XOR-Key*

The XOR-key algorithm provides hardware support for a very efficient encryption scheme used by the PointGuard [6] compiler. The hardware has a key stored in a special purpose hardware register that is assigned to each process. For the e-store instruction, the key is XORed with the data value and stored. For the d-load instruction, the key is XORed with the data value read, before using the data value. We model this architecture taking one additional cycle for each encrypted load and store code pointer instruction.

As mentioned previously in Section 2.3, this approach is secure from a remote attack only if a process can be guaranteed free of read attacks. Therefore, it provides weak read attack security. If read attacks are possible on the process, we propose using one of the following two hardware techniques for providing strong read attack security.

6.4.2 *Random Permutation Table*

To provide strong read attack security, we propose adding a Random Permutation Table (RPT) to the processor [9]. The random permutation table is loaded by the operating system at system startup time, and is of size 2^N, where N is the number of bits we are encrypting. The table is randomly filled with all of the possible values from 0 to 2^N. Table permutations are actually ideal cryptographic functions because if an attacker has not seen a given input/output pair before, random guessing (with probability 1 in $2^N - q$ of being correct) is the best possible strategy for finding such a pair, where q is the number of different possible read attacks.

For this approach, we use two keys, K_B to XOR with the data (D) being encrypted *before* the RPT lookup, and a K_A to XOR with the RPT table entry. The following function is used to encrypt the code pointer (Data):

$\text{Index} = D \oplus K_B$
$D_E = \text{RPT[Index]} \oplus K_A$

IEEE COMPUTER SOCIETY

The encrypted value is then stored into the data location. Note, two RPT tables are needed. One for the encryption (RPT) and one performing the inverse function (Inverse RPT) to decrypt the data. To decrypt the data, the following function is used:

Index = $D_E \oplus K_A$
D = Inverse-RPT[Index] $\oplus K_B$

The keys are assigned per process. It is important to XOR the key with the original data before indexing into the RPT to thwart attacks. Note that the RPT is *shared among all processes on the machine* because it can be quite large. It is only reset at machine bootup.

The individual process keys are used to randomize across processes so that different entries in the RPT will be used for different processes. This is important to prevent retry attacks.

For this approach, we propose permuting the lower N-bits of code pointer addresses. Then the attacker is left with randomly guessing a table entry to perform an attack, since the RPT is a random permutation of all the possible values from 0 to 2^N. If the attack fails, the process/daemon will be restarted using new keys. The attacker has only a 1 in $2^{16} - q$ chance of success, where q is the number of different possible read attacks. For detection, enough failed attacks will trigger a detection as described in Section 6.2.2.1. To provide this level of protection two tables of 64 KBytes are used. In addition, just two N-bit (e.g., 16 bits in our example) hardware keys are required for the current process. When context switching, the 2 keys for the process are stored along with process context state in OS-protected memory.

Based on the amount of on-chip storage available, the table sizes can be increased or decreased by controlling the value of N. This provides a simple inverse trade-off between the amount of storage and the probability an attacker has of succeeding in a random guess. However, this is only a realistic solution for small values of N due to the storage size of the table. This leads us to our next approach.

6.4.3 Feistel Network

Our previous approach of table lookup can scale to be used on 32-bit processors, but cannot create the higher levels of security possible on 64-bit processors because of the inherently small table sizes that must be used to fit on chip. On a 64-bit processor, we would like to increase the number of read attacks required for an attacker to mount a successful write attack. To achieve this, we propose the use of an on-chip pipelined version of a 4 round Luby-Rackoff Feistel Network with the standard 10 rounds of AES to perform encryption and decryption [15]. A single round of Luby-Rackoff (F) breaks a 64-bit address into two 32-bit chunks, L and R and then performs the following:

$$F(L, R) = (R, L \oplus f(R))$$

Where f is a function which is difficult to distinguish from a pseudo-random function. For a 64-bit implementation, we use the output of the lower 32 bits of AES as f. A custom AES [11] datapath is used to perform the encryption. This approach takes longer to encrypt and decrypt than the RPT, since it is no longer just a simple table lookup, but we expect the attacker to require 2^{16} chosen read attack queries to construct a birthday attack. We implement a standard 4 round Luby-Rackoff Feistel Network with each round being AES, and ensure that each round uses an independent random function. In addition, the keys for AES are initialized at process creation time. We point the interested reader to [15, 11] for the full details of the algorithms. For the Feistel Network and a 32-bit ISA, only the bottom 16-bits of AES is used, and for a 64-bit ISA only the bottom 32-bits of AES are used.

For an attacker to achieve full key recovery, the attacker would have to do 2^X operations, where the key length is X bits; we assume a 128-bit key length for our design. Given this approach, the attacker is left to try to exploit some property of the Feistel Network. For an ISA where the data addresses and code pointers are 32-bit, using the Feistel Network, the attacker will have only a probability of being correct of approximately q^2 out of 2^{16}, where q is the possible number of unique read attacks. For a 64-bit architecture (where data addresses are 64-bits), this would be a probability of q^2 in 2^{32} of exploiting the attack. This leads us to expect that our adversary can construct an attack if they can read 2^{16} encrypted code pointers from memory.

6.4.3.1 How the Feistel Network Fits into the Pipeline.

We examine two implementations of the Feistel Network, one where it is put onto the e-store and d-load critical path to use for protection for all code pointers, and the other where the encryption and decryption can be overlapped with execution for call/return instructions. This second approach is targeted specifically at protecting return address pointers on the stack.

When performing a procedure call, the return PC is available at the fetch stage in the processor pipeline. We can therefore start the encryption using the Feistel Network of the return PC at fetch, and the PC is not needed to store onto the stack until the call commits. For return instructions, there are two paths it can take. The common case fast path uses the predicted return address from the return address stack on a processor. For this approach, we first *encrypt* the return address when it is predicted; we then match the encrypted predicted return address with the encrypted address read from the stack. If they are the same, then we have a match. If they are different, we then decrypt the data read from the stack and jump to that value because we either had a return address misprediction or an attack.

6.4.3.2 Caching Decrypted Code Pointers.

We calculated the latency of implementing the full Feistel Network we are using to be between 40 to 80 cycles for the processor we are simulating. This latency can be tolerated when using this for only protecting return addresses on the stack using the approach in the previous subsection. However, it is too inefficient to be used for protecting general purpose code pointers through e-store and d-loads.

To address this inefficiency, we examine caching the decrypted values in the L1 cache, and marking the decrypted code pointers stored in the cache as protected with a single bit for every data word stored in the cache. For this design, only d-loads and e-stores can set the bit. On a d-load that misses in the cache, we decrypt the data, and store it in the cache; we also set a bit indicating that this is a protected code pointer. If a d-load occurs and the bit is already set, the value is ready to use without any extra delay. This makes

the common case fast. This removes the decryption of the Feistel Network for code pointers that hit in the cache. For this study, we examine applying this caching just to the L1 virtually indexed data cache.

Our caching policy allows d-loads to directly access unencrypted code pointers from the L1 cache, and e-stores to store unencrypted data into the L1 cache. All other untrusted loads and stores only see encrypted data when accessing a protected code pointer. These are the semantics enforced by the following policies.

For e-stores, if there is a cache hit or the cache is write-allocate, then the unencrypted data would be stored in the cache with the code pointer protection bit set. When the dirty block is written to the next level of the memory hierarchy (or if the cache is write through) the code pointer would at that time be encrypted using Feistel Network.

If a d-load occurs and there is a hit in the cache and the encrypted bit is not set, then the data item needs to be decrypted and recached. When this occurs the d-load goes through the decryption process, recaches this unencrypted value, and sets the bit in the cache. This situation can occur when a cache line is prefetched, or when a spatial hit brings in the cache block.

If an untrusted store instruction writes to a data item that is already in the cache and the encryption bit is set for that data item, then the register value is stored and the encrypted bit is *cleared* because it was not stored with an e-store.

The last remaining issue occurs if an untrusted load loads a data item, and the protection bit is set. This can only occur if a d-load or e-store last accessed that code pointer. The reason why an untrusted load might be accessing it is to do a memcopy or something similar. Our policy is that untrusted loads should only see encrypted data. Therefore, the data item is encrypted before storing the value into the register for that load. The encrypted data item is then stored into the cache and the bit is cleared. This should occur significantly less often than the d-load and e-store accessing that cached code pointer.

6.5 Hardware Support Results

6.5.1 Simulating D-Loads and E-Stores

To gather performance overhead results, we perform cycle level simulation as described in Section 4. To simulate return address encryption, we detect loads and stores to and from the return address register and introduce extra latency to them. To simulate targeted encryption of function pointers we add extra latency to every load or store in a program that accesses a value which could be interpreted as a pointer to the text segment and we add extra latency to them. We then run our simulator for 100 million instructions using the early SimPoints [23] on the SPECint benchmarks and a handful of C++ programs and measure the effects of our modification. In doing the cycle level simulations we modeled in detail all pipeline modifications and latencies, including details like the return address check.

Our latencies for the various techniques are conservative. The RPT method requires two XORs and a table lookup. In all likelihood this is a single cycle operation in any conceivable processor, but we give 5 cycle numbers to show that the performance impact for additional latency is not substantial.

For the 4 round Luby-Rackoff Feistel Network, we assume the AES [8] implementation. We provide results for assuming each Feistel Network round to take 10 cycles or 20 cycles for a total of 40 or 80 cycles for the two configurations examined. In terms of area, from published AES implementations, we can estimate that the cost of a pipelined AES implementation would be hundreds of thousands of gates - a very small fraction of a future microprocessor. Note, each round of the Luby-Rackoff Feistel Network could be represented as a direct table lookup by expanding AES, but examining the performance of this approach is beyond the scope of this paper.

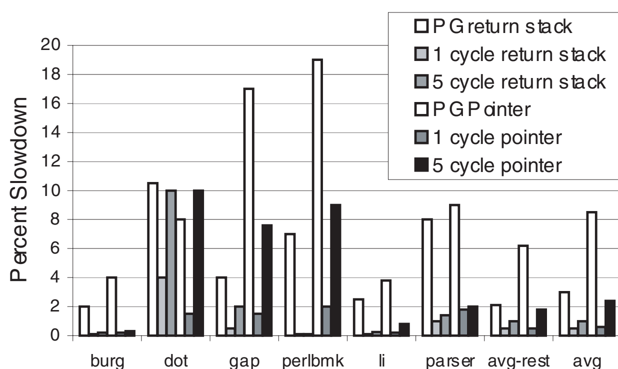

Figure 2: SPECint and other benchmark results for XOR-Key and RPT. The first 3 bars are for protecting only return addresses, and the last 3 bars are for protecting all code pointers.

6.5.2 XOR-Key and Random Permutation Table Results

We first assume simple XOR encryption on the load/store path and perform two different experiments. The first experiment assumes that we only encrypt call/return sequences. The second approach assumes compiler support which has replaced *all* pointer stores and loads with e-store and d-load.

Figure 2 shows the performance overhead of using the pure software approach of PointGuard (PG), the XOR-Key architecture, and the RPT architecture. The results are shown for two latencies (1 and 5 cycle) for encrypting all code pointers (pointer) and only return addresses (return stack). For the XOR-Key results we assume a 1-cycle latency to perform the encryption and decryption. The effects on the SPECint benchmarks is fairly benign. The XOR-Key results show that we can provide PointGuard level of protection for less than 0.5% slowdown on average.

For the RPT approach we model that the RPT function is performed on the critical path of the e-store and d-load for all code pointers and assume that this takes 5 cycles. Only with 5 cycle encryption of all code pointers do *perlbmk*, *gap* and *dot* show slowdowns of up to 10%. This implies that the RPT approach may benefit from the caching approach discussed in Section 6.4.3.2. Even so, if we use the RPT approach for just return stack addresses, effectively no overhead is seen except for `dot`. The reason for not having any performance degradation for 5 cycle return stack is that the check is performed off of the predicted return address as described in Section 6.4.3. This effectively masks the latency of the return address check. In all cases, except for `dot`, the RPT table creates less than half the slowdown of PointGuard.

IEEE COMPUTER SOCIETY

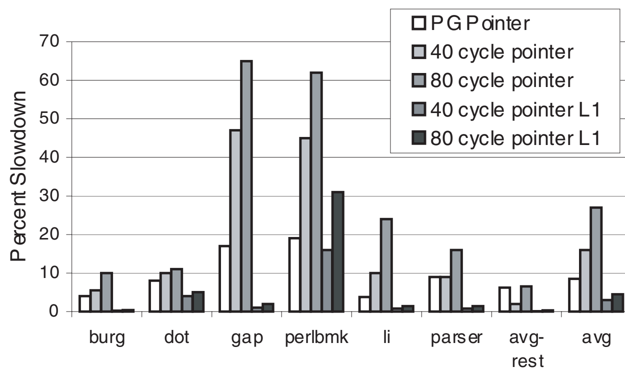

Figure 3: SPECint benchmark results for Feistel Network with and without caching. The first bar shows results for the software only solution of PointGuard, and the last 4 bars are using hardware support for protecting all code addresses with strong read attack security.

6.5.3 Performance of Feistel Network encryption

As previously mentioned, we estimated the implementation for the 4 round AES Feistel Network to be 40-80 cycles for our processor assumptions based on published implementations of AES [11]. Figure 3 shows the results for these two latencies using the Feistel Network to protect *all* code pointers (pointer). The performance decrease caused by naively placing the encryption on the critical path of the d-load/e-store leads to debilitating performance drops.

Our final experiment is to assume that we add extra bits to the L1 cache tags to indicate that encryption must be performed when the data words are accessed as described in Section 6.4.3.2. These are the L1 bars in Figure 3. As we see, the penalty from this is greatly reduced relative to the non-cached approach when protecting all code pointers.

6.5.4 Results Summary

From the results in this section it is clear that the return addresses can have strong read attack security with very low overhead. Applying the Luby-Rackoff Feistel network to all code pointers is more expensive than PointGuard and therefore not likely to be a popular option in a processor. However, using the caching approach allows strong read attack security for all pointers with performance degradations only a fraction of the PointGuard approach. Even the 80 cycle implementation of the Feistel network with caching has a performance degradation of approximately 3% compared to 9% for software PointGuard. An alternative design left for future research is to use a direct table lookup by expanding AES used in each round of the Luby-Rackoff Feistel Network. This will significantly reduce the latency of the Feistel Network, but increase the area required to implement it.

7. SUMMARY

Buffer overflows are a key part of the hacker's arsenal because the efficient languages used for system development have chosen not to protect against them. Vulnerabilities are currently being found at a tremendous rate and machines frequently remain unpatched for long periods of time. Write and read buffer overflows can be used to allow an attacker to run arbitrary code on a victim's machine and are the basis for the many of the damaging worms seen recently. Thus, providing new mechanisms for preventing the root cause of such attacks will have considerable economic and societal benefits, compared to the more indirect approach used today of finding and blocking the signature of an attack in the network once it has caused damage.

In this paper we consider hardware approaches to eliminating buffer overflow attacks. When considering adding such a hardware change, a processor vendor would like a mechanism that is:

1. **General:** The mechanism should work with as many attacks as possible, and deal with nuances like continuations and setjmp/longjmp;

2. **Backwards Compatible:** The mechanism should ideally not require re-compilation (just minor binary modification), and deal with the vagaries of legacy code which may not always respect standard call/return semantics;

3. **Secure:** There should be a reasonable argument that the security provided cannot be finessed by attacks such as read buffer overflow attacks.

4. **Efficient:** The mechanism should fit in easily with existing ISAs, require a small amount of hardware to implement, take little extra memory, and cause very little slowdown.

Our paper shows that PointGuard can be applied via binary modification, as we show with Bochs, to execute legacy code and code that does not following the strict call/return semantics. In doing this, we show that previous return stack security architectures must have binaries (like the linux kernel) cleaned of non-strict call/return semantics in order to work.

We show that existing software techniques such as PointGuard have performance overhead, and we examine using special purpose hardware instructions to eliminate this overhead. These instructions significantly reduce the time and complexity of binary patching to protect return addresses, and would enable fast dynamic patching of code.

In addition, the existing software techniques like PointGuard do not protect against read attacks. To address this, we examine cryptographically secure mechanisms for targeted encryption of function pointers. Our approach also considers a wide spectrum of security needs ranging from the simplest and most efficient cypher (XOR) to a cryptographically strong Luby-Rackoff Feistel network. We address the inefficiency of the Luby-Rackoff Feistel network by introducing a scheme to avoid encryption and decryption of addresses that hit in the cache. Finally, we have shown that our approach requires little modification to existing ISAs (especially our target, the widely deployed x86 ISA), requires only a small amount of hardware changes, requires only a modest amount of extra memory even for cryptographically strong variants, and causes a very small slowdown of less than 3% on average for the SPECint benchmarks for even our most secure encryption algorithm when using caching.

Overall, this approach trades the generality and overhead of a more *revolutionary* approach (e.g., secure processor design) [14, 24, 29] for an efficient *evolutionary* approach that focuses only on protecting buffer overflow attacks that hijack processor control flow. In particular, mechanisms such as ours cannot prevent an attacker from crashing a process.

1072-4451/04 $20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

However, if generally implemented in the next generation of processors, such mechanisms could greatly limit the power of attackers and make writing a worm a much harder task than it is today.

Acknowledgments

We would like to thank Tadayoshi Kohno for detailed discussions and suggestions about the security algorithms used in this paper, and the anonymous reviewers and Jedidiah Crandall for providing useful comments on this paper. This work was funded in part by NSF grant No. CCR-0311712 and an equipment grant from Intel.

8. REFERENCES

[1] Arash Baratloo, Timothy Tsai, and Navjot Singh. Libsafe: Protecting critical elements of stacks. Technical report, Avaya Labs, 1999.

[2] Arash Baratloo, Timothy Tsai, and Navjot Singh. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[3] Bochs. Bochs ia-32 emulator project. http://bochs.sourceforge.org, 2001.

[4] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack Magazine*, 0xa(0x38), May 2000.

[5] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[6] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium, Washington DC*, August 2003.

[7] J.R. Crandall and F.T. Chong. A security assessment of the minos architecture. In *37th International Symposium on Microarchitecture*, 2004.

[8] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer-Verlag, 2002.

[9] S. Even and Y. Mansour. A construction of a cipher from a single pseudorandom permutation. *Journal of Cryptology*, 10(3), 1997.

[10] Mike Frantzen and Mike Shuey. StackGhost: Hardware facilitated stack protection. In *10th USENIX Security Symposium*, pages 55–66, 2001.

[11] K. Gaj and P. Chodowiec. Comparison of the hardware performance of the aes candidates using reconfigurable hardware. In *3rd Advanced Encryption Standard Candidate Conference*, 2000.

[12] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference*, pages 275–288, June 2002.

[13] Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi. Enlisting hardware architecture to thwart malicious code injection. In *Proceedings of the International Conference on Security in Pervasive Computing (SPC-2003)*, March 2003.

[14] David Lie, Chandramohan Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *Symposium on Operating Systems Principles*, October 2003.

[15] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Computation*, 17(2), April 1988.

[16] John P. McGregor, David K. Karig, Zhijie Shi, and Ruby B. Lee. A processor architecture defense against buffer overflow attacks. In *Proceedings of IEEE International Conference on Information Technology: Research and Education (ITRE 2003)*, pages 243–250, August 2003.

[17] E. Mohr, D. Kranz, and R. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. *ACM Transactions on Parallel and Distributed Systems*, 2(3):264–280, 1991.

[18] H. Ozdoganoglu, C.E. Brodley, T.N. Vijaykumar, and B.A. Kuperman. Smashguard: A hardware solution to prevent attacks on the function return address. Technical report, Electrical and Computer Engineering Department, Purdue University, Dec 2000.

[19] Martin Roesch. Snort – lightweight intrusion detection for networks. In *Proceedings of LISA'99: 13th Systems Administration Conference*, pages 229–238, November 1999.

[20] Ahmad-Reza Sadeghi and Christian Stüble. Bridging the gap between TCPA/palladium and personal security.

[21] Zili Shao, C. Xue, Q. Zhuge, E.H.-M Sha, and B. Xiao. Security protection and checking in embedded system integration against buffer overflow attacks. In *Proceedings of Information Assurance and Security special track in conjunction with the International Conference on Information Technology: Coding and Computing (ITCC 2004)*, volume I, pages 409–412, April 2004.

[22] Zili Shao, Qingfeng Zhuge, Yi He, and Edwin Sha. Defending embedded systems against buffer overflow via hardware/software. In *Proceedings of the 19th Annual Computer Security Applications Conference*, 2003.

[23] T. Sherwood, E. Perelman, G. Hammerley, and B. Calder. Automatically characterizing large-scale program behavior. In *Proceedings of the International Conference on 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[24] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of MICRO-36*, December 2003.

[25] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[26] John Viega, J. T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. *ACM Transactions on Information and System Security*, 5(2), 2002.

[27] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[28] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, pages 149–162, San Diego, California, February 2003.

[29] Jun Yang, Youtao Zhang, and Lan Gao. Fast secure processor for inhibiting software piracy and tampering. In *Proceedings of MICRO-36*, December 2003.

1072-4451/04 $20.00 © 2004 **IEEE**

IEEE COMPUTER SOCIETY