SmashGuard: A Hardware Solution to Prevent Attacks on the Function Return Address

H. Özdoğanoğlu, C. E. Brodley, T. N. Vijaykumar and B. A. Kuperman
{cyprian,brodley,vijay,kuperman}@ecn.purdue.edu

School of Electrical and Computer Engineering
1285 Electrical Engineering Building
Purdue University
West Lafayette, IN 47907-1285

December 5, 2002

Contents

List of Tables

List of Figures

Abstract

We present a hardware-based solution to protecting the function return addresses stored on the program stack. Our solution protects against all known forms of attack on the return address pointer while incurring negligible performance overhead. Our solution requires that we add a small hardware stack to the processor that is inaccessible to a user process. With each function call instruction a new return address is pushed onto the hardware stack at the same time it is saved in the process stack. A return instruction pops and compares the return addresses from the top of both the hardware and process stacks. If a mismatch is detected, then an exception is raised. Because the stores and checks are done in hardware, our approach has virtually no overhead. This approach is inexpensive and non-intrusive because our hardware modification does not require that we modify the instruction set of the architecture. We report performance statistics with SPEC-2K benchmarks and we discuss the changes that need to be made to the kernel to securely process context switches and deeply nested function calls.

Chapter 1

Introduction

The most well-known example of attacks on the function return address pointer are buffer overflow attacks [15]. Although it is fairly simple to fix an individual instance of a buffer overflow vulnerability, they continue to remain one of the most popular methods by which attackers compromise a host. Buffer overflows are found in 10 of the 31 advisories published by CERT for 2002 [7] and in 5 of the top 20 twenty vulnerabilities compiled by the SANS Institute [14].

Buffer overflow attacks overwrite data on the stack and can be used to redirect execution by changing the value stored on the process stack for the return address of a function call. Our hardware-based solution protects against all known forms of attack on the return address pointer while incurring virtually no overhead in terms of performance.

Our hardware solution, which we call SmashGuard, has the following strengths:

Performance: Because previous approaches are software-based schemes, a high level of security implies performance degradation. In contrast, our hardware-based approach provides security with negligible (less than .01% for our benchmarks) performance degradation. This performance superiority while retaining the ability to protect against *all* forms of attack on the return address pointer is the key advantage of our approach.

Unmodified binaries: Another strength of our approach is that no recompilation of the source code is necessary.

Implementation cost: The hardware cost of our solution is the addition of 1 KB of data storage into the CPU, which is negligible when compared to the size of the data storage used in L1 and L2 caching (which are usually in the order of megabytes). Further, our modification does not require that we modify the instruction set of the architecture and therefore can be quickly incorporated into today's microprocessors.

In Section 2 we describe the vulnerability of the function return address and different ways an attacker can exploit this vulnerability. In Section 3 we summarize related work to point out their strengths and weaknesses, both in terms of performance and functionality. Then in Section 4 we describe the proposed hardware solution in detail. In Section 5 we present performance results of our modification on SimpleScalar, an Alpha simulator. Finally in Section 6 we provide our conclusions and we outline future extensions to our work.

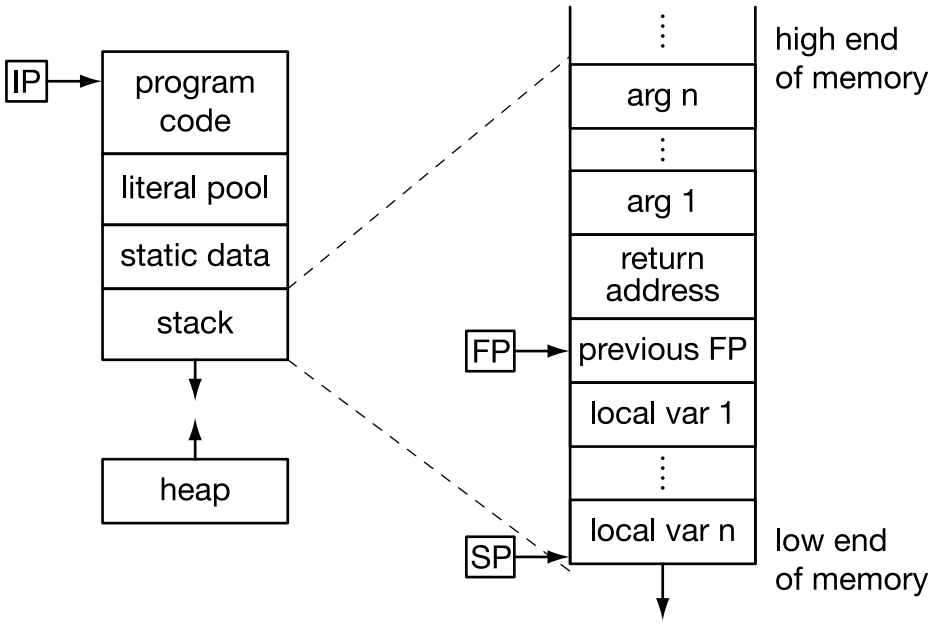

Figure 1.1: Memory Organization

Chapter 2

Anatomy of an Attack

This section provides an overview of the vulnerability of return address pointers stored on the stack and describes how "stack smashing" attacks exploit this vulnerability to execute the attacker's code.

2.1 The Stack

Before describing the vulnerabilities and the attacks affecting the function return address, we first briefly review the memory organization of a process. On the left hand side of Figure 1.1, we show the three logical areas of memory used by a process. The text-only portion contains the program instructions, literal pool and static data. The stack is used to implement functions and procedures, and the heap is used for memory that is dynamically allocated by the process during run time. We focus here on the stack, because attacks on the return address are possible because the return address is stored on the stack when a function is called and is popped off the stack on function exit. During the *function prologue*, the function arguments are pushed on to the stack in reverse order and then the return address is pushed onto the stack.[1] The return address holds the address of the instruction immediately following the function call and is an address in the program code section of the process' memory space. The prologue finishes by pushing on the previous frame pointer followed by the local variables of the function. The function arguments, return address, previous frame pointer and local variables comprise a *stack frame*. Because functions can be nested, the previous frame pointer provides a handy mechanism for quickly deallocating space on the stack when the function exits. During the *function epilogue*, the return address is read off of the stack and the stack frame is deallocated dynamically by moving the stack pointer to the top of the previous stack frame.

2.2 Vulnerability of the Function Return Address

The return address in the stack frame at the top of the stack points to the next instruction to execute after the current function returns (finishes). This introduces a vulnerability that allows an attacker to cause a program to execute arbitrary code. An attacker can inject code (shellcode) somewhere in the memory (in the stack or the heap) and modify – either by a buffer overflow or a format string attack – the return address to point to the start of the shellcode. When the function exits, the program execution will continue

[1] Our discussion is based on the x86 architecture because it is perhaps the most widely known; for other architectures, details will vary slightly.

from the location pointed to by the stored return address. On successful modification of the return address, the attacker can execute commands *with the same level of privilege* as that of the attacked program. If the compromised program is running as root, then the attacker can use the injected code to spawn a root or superuser shell, and take control of the machine. Note that the attacker can choose any code to execute, but typically will spawn a shell in order to install a backdoor on the compromised host.

2.3 Exploiting the Vulnerability

Currently there are several methods for overwriting the function return address and two types of approaches to where to redirect execution. In this section we first describe three types of buffer overflow attacks that redirect execution to either a shellcode or to the `system()` call in the libc. We then present an overview of format string attacks, a relatively recent method to alter the return address in order to redirect execution.

Buffer overflow attacks are the undesirable side-effects of unbounded string copy functions. The most common example from the C programming language is `strcpy()` which copies each character from a source buffer to a destination buffer until a `null` character is reached. The vulnerability arises because `strcpy()` does not check whether the destination buffer is large enough to fit the source buffer's contents. If the destination buffer is a local variable (and therefore stored on the stack in the frame), then an attacker can exploit this vulnerability to overflow the buffer and overwrite a pointer on the stack, a function pointer, or even the return address. Note that for most architectures (e.g., x86, SPARC, MIPS) the stack grows down from high to low addresses, whereas a string copy on the stack moves up from low to high addresses. It is trivial to overflow a buffer to overwrite the return address which is higher in the stack than the local variables in that particular stack frame.
There are three types of buffer overflow attacks that can be used to redirect program execution. In all three cases a local buffer is overflowed to:

Type 1: directly overwrite the return address on the stack;

Type 2: overwrite a pointer variable adjacent to the overflowed local buffer to make it point to the return address, and then overwrite the return address by an assignment to the pointer; or

Type 3: overwrite a function pointer adjacent to the overflowed local buffer, so that when the function is called the control transfers to the location pointed by the overwritten function pointer. Strictly speaking this is not an attack on the return address, but it can be used to redirect execution. Note that our method protects the return address stack and does not protect against this type of attack.

The second characteristic of an attack is based on where execution is redirected. The most well-known approach is to write shellcode (a hexadecimal representation of assembly code to spawn a shell) 1) into the local buffer being overflowed, 2) above the return address on the stack, [15] or 3) in the heap [9]. If the attacked program has root privilege, then when control is redirected to the injected shellcode, the attacker gets a *root shell*. The second choice for redirecting execution is called the *return-to-libc* attack. It was invented to bypass protection methods that mark the stack as non-executable [12] (a non-executable stack prevents execution from being rerouted to shellcode injected on the stack). The return-to-libc attack eliminates the need for shell code, by redirecting execution to the `system()` library function in `libc` with the argument "/bin/sh". Note that this attack can be done either locally or across the network.

Format string attacks are relatively new and are thought to have first appeared in mid 2000 [5]. We provide a brief overview here, but for details the reader is referred to [5, 16]. Similar to a buffer overflow attack, format string attacks modify the return address in order to redirect the flow of control to execute the attacker's code. In the C programming language, format strings allow the programmer to format inputs

and outputs to a program using conversion specifications. For example, in the statement `printf("%s is %d years old.",name,age)`, the string in quotes is the *format string*, `%s and %d` are conversion specifications, and `name` and `age` are the specification arguments. When `printf()` is called, a stack frame is created and the specification arguments are pushed on the stack along with a pointer to the format string. When the function executes, the conversion specifiers will be replaced by the arguments on the stack. The vulnerability arises because, programmers write statements like `printf(string)` instead of the proper form: `printf("%s",string)`. The statements appear identical unless `string` contains conversion specifiers. In this case, for each conversion specifier, `printf()` will pop an argument from the stack. For example, consider the following:

```
int foo1(char *str) {
  printf(str);
}
```

If the user calls `foo1` with an argument string `"%08x.%08x"`, the function will pop two words from the stack and display them in hex format with a dot (.) in between. Using this technique, the attacker can dump the contents of the entire stack. The key to this attack is the `"%n"` conversion specifier, which pops four bytes off the stack and writes the number of characters in the format string before `"%n"` to the address pointed to by the popped four bytes. Basically, if we move up the stack popping values using `"%08x"`, and then just when we reach the return address we insert a `"%n"` in the format string, we would write the number of characters we had in the format string so far onto the return address on the stack. Note that length specifiers allow the creation of arbitrarily long format string "lengths" without needing the string itself to be an arbitrary length.

Like buffer overflow attacks, format string attacks can be used to redirect execution to shellcode in the stack (or heap) or to the `system()` call in libc. Format string attacks are similar to Type 2 buffer overflow attacks, in the sense that the return address can be modified without touching anything else on the stack, so methods that can prevent Type 2 buffer overflow attacks can also prevent format string attacks.

Chapter 3

Related Work

Various tools and methods have been devised to stop these attacks with varying levels of security advantage and performance overhead. Solutions that trade off a high level of security for better performance are eventually bypassed by the attackers and prove incomplete. On the other hand, high security solutions seriously degrade the system performance due to the high frequency of integrity checks and high cost of software based memory protection. Another issue that diminishes the feasibility of these tools and methods is their lack of transparency to the user or to the operating system. We have split the existing work into two groups: those that modify the compiler and therefore require that the source code be recompiled, and those that require a modification to the system software.

3.1 Modifications to the Compiler

StackGuard [10, 11] is perhaps the most well-known compiler-based solution. StackGuard places an integer of known value (called a canary) between the return address and the local variables on the stack during the function prologue. If a local buffer on the stack is overflowed, the attacker must overwrite the canary to reach the return address. StackGuard supports two types of canaries. The *random canary* method inserts a 32-bit random canary after the return address in the function prologue, and checks the integrity of its value before using the return address at epilogue. The *terminating canary* consists of four string termination characters: `null, CR, -1`, and `LF`. Note that each one of these characters is a terminating value for at least one unbounded data copying function. If the attacker tries to overwrite the canary with the same terminating values, the overflow will never reach the return address because the string copy will be terminated at the canary.

As pointed out by Bulba and Kil3r [6], StackGuard does not protect against buffer overflow attacks that overwrite pointers or function pointers (attack Types 2 and 3). It does not protect against format string attacks either. In addition, it requires recompilation of the source code, and because it modifies the stack contents, programs dependent on the stack structure (e.g., debuggers) will no longer work. Finally, the random canary needs to be protected.

StackGuard Performance: For every function call and return instruction executed StackGuard must write the random canary to the stack and compare it on return. A varying performance overhead of 6-80% is reported in [11] which is a function of the ratio of the instructions required for the modified prologue and epilogue to the number of original function instructions.

StackShield [18] is a modification to the compiler that provides three different protection mechanisms: two for protecting the return address and one for protecting function pointers. *Global ret stack* implements a separate stack for the return addresses in a global array of 256 32-bit entries. For each function call, the return address is pushed onto both the real stack and this additional stack. On function return, the address stored on the separate stack is used. Because there is no comparison to the address stored on the stack, in contrast to the other solutions, attacks are prevented but *not detected*, which may be dangerous. Another weakness of this approach is that the separate stack needs to be protected, otherwise the attacker can use a pointer to overwrite the return address on the separate stack. *Ret range check*, a faster alternative to global return stack, saves the return address of the currently executing function in a global long integer, and then compares it to the return address on the stack when the function returns. This method does not protect against attacks on the return address that come through pointers or function pointers. Moreover, unless the global long integer variable is protected, it can also be found and overwritten. The final method, *Function pointer protection* is implemented by adding check code in the functions where there are function pointers, right before the function calls. This check code ensures that the function pointer points to somewhere in the text segment. This method does not prevent the "return to libc" attacks, because the `system()` call is already in the text segment. In addition, this protection mechanism is not compatible with code that requires an executable stack.

StackShield Performance: All three of these methods have little performance overhead because the array is unprotected.

Finally, **Return Address Defender** [8] creates a global integer array called the *Return Address Repository (RAR)* that holds the copies of the return addresses pushed on the stack. There are two versions of RAD that differ in the amount/cost of protection to the RAR. The first and less expensive method, *MineZone RAD* inserts two minezones above and below the RAR and marks them as read-only with the `mprotect()` system call. Any attempt by the attacker to overflow a buffer and overwrite the RAR would cause a trap and be denied by the OS. This method protects against type 1 buffer overflows but can be defeated by Types 2 and 3. The second version of RAD, *Read-Only RAD* marks the entire RAR as read-only with `mprotect()` to achieve high security. This incurs a large overhead because during the function prologue the RAR is marked as writable, the return address is saved into the RAR and then the RAR is marked as read-only again. Similar to MineZone RAD, this method cannot prevent the return-to-libc attack.

RAD Performance: Chiueh, et al., report a modest performance degradation of 5-40% for Minezone RAD, and up to 1000% for the more secure Read-Only RAD [8].

3.2 Modifications to System Software

Currently there are three system-based methods for protecting against buffer overflow attacks. One that modifies the libraries and two that modify the kernel.

The first, **Libsafe** [3, 4], intercepts all calls to library functions known to be vulnerable, and executes their safe versions. For statically linked programs this requires recompilation of the source code. The safe versions of the functions estimate a safe upper bound on the size of buffers automatically, so as to prevent the buffer overflows. Since no legitimate C program should ever overwrite the frame pointer, this is a safe upper bound to detect attacks. Libsafe can protect against all three types of buffer overflow attacks because it disables unbounded string copy. However, it cannot protect against the format string attacks since the format string attacks are irrelevant to the vulnerable functions for buffer overflows. **Libverify** is a run-time implementation of StackGuard, which does not require re-compilation of the source code. Like StackGuard, Libverify can only prevent buffer overflows of type 1.

Libsafe/Libverify Performance: Baratloo, et al., report an average overhead of 15% for for applications protected by Libsafe, Libverify, and StackGuard [4]. While Libverify and StackGuard behave very similarly, Libsafe is slightly faster since it need only intercept vulnerable functions, whereas StackGuard modifies the prologue and epilogue of all called functions.

The first kernel-based solution **StackGhost** [13] is a patch to the OpenBSD 2.8 kernel under the Sun SPARC architecture. Frantzen and Shuey performed experiments on three methods for protecting the return address. The first two XOR the return address on the stack with a cookie before writing it on the stack and then XOR it again with the same cookie before the return address it is popped off the stack. This method distorts any attack to the return address but does not detect it. Therefore, another method is used to detect the attacks. In SPARC architecture, the memory is four byte aligned and the least significant (LS) two bits are always 0s. So, the two bits are set at the function prologue and verified to be set at the epilogue. If the attacker is not aware of this, they will inject a four byte aligned address in the return address and therefore the attack will fail. But, once the attacker figures this out, they can set the two LS bits of the address that they want to jump to, and then overwrite the return address with the modified address. The XOR cookie method comes with two flavors. XOR-cookie per kernel and XOR cookie per-process. Both of these methods (especially per kernel XOR cookie) are easily bypassable since the cookie can be figured out if the contents of the stack frame can be observed (e.g., using the method in the format string attacks as described in Section 2.3) and the return addresses are extracted from the program binaries.

StackGhost Performance: Frantzen and Shuey report 17.44% overhead for the per-kernel XOR cookie and 37.09% overhead for the per-process XOR cookie.

Finally, to prevent execution of the shellcode on the stack, Solar Designer proposed the **Non-Executable Stack**. This solution is a Linux kernel patch from the Openwall Project [12]. This method can be bypassed with return-to-libc attacks or running the shellcode in the heap. To prevent the return-to-libc attacks, this patch also changes the default addresses of the shared libraries in libc to contain a zero byte. It is not possible to overwrite the return address with a value that contains a zero byte (null), since zero byte is a string terminator, and terminates copying a string. Torvalds did not approve including this in the Linux kernel because a non-executable stack causes debuggers to fail and some programs require an executable stack [17].

Chapter 4

SmashGuard: A Hardware Solution

We propose a hardware solution that is secure and inherently faster than the aforementioned software methods. In this section we present our hardware solution and the architecture that we chose to modify to illustrate our approach. We elaborate on the complications we face with process context switches and deeply nested function calls, and how we plan to solve them.

4.1 Overview

Our approach, which we call SmashGuard, protects attacks on the return address by saving the return address in a hardware stack added to the CPU. With each function call instruction, a new return address is pushed onto the hardware stack, at the same time it is saved in the stack frame of the process in the run-time stack. A return instruction pops the most recent return address from the top of the hardware stack and compares it to the return address stored on the process stack in memory. If a mismatch is detected between the two return addresses, then a hardware exception is raised. In the exception handler the process may be killed and a report may be sent to syslog.

In the common case (single process and nesting of functions less that the size of the hardware stack), all read and writes to the hardware stack are done in hardware via the function call and the return instructions, so there is no instruction that is permitted to read/write directly from/to the hardware stack. Specifically, no *user-level* load or store instruction can access the hardware stack. To handle the more complicated cases of multiple processes requiring context switching, and deeply nested function calls, the hardware stack needs to be accessible by the Operating System (OS). As we explain in Sections 4.5 and 4.6 we solve this problem by memory mapping the hardware stack. The user cannot access the hardware stack via the OS either since it is protected at the kernel privilege level.

Our hardware-based approach will protect against format string attacks and attacks that modify the return address of a function (Types 1 and 2 in Section 2.2). Our approach does not protect against Type 3 buffer overflow attacks, which use the overflow to overwrite a function pointer adjacent to the overflowed buffer.

4.2 Implementation

We first introduce the Alpha CPU architecture and the operation of the relevant instructions. We chose the Alpha architecture because it has a RISC instruction set which is simple to explain and simulate. However, our technique is directly applicable to a more complex instruction set such as the x86. The Alpha is a superscalar pipelined RISC architecture processor with fixed size and simple instruction set. In the Alpha architecture, register 26 is used implicitly for storing the return address of the current function. It is one of the 32 general purpose registers. When a function is called, Jump-to-Subroutine (`jsr`) or Branch-to-Subroutine (`bsr`) instructions write the address of the next instruction after the function to register 26 and the execution continues from the address of the specified function. When a nested function is to be called, register 26 is copied to the program stack (done in software via code generated by the compiler) and loaded with the return address of the new function. When the function exits, the return (`ret`) instruction copies the contents of register 26, to the instruction pointer.

Our hardware solution modifies the `jsr` and `bsr` instructions of the CPU to copy the contents of register 26 to the top of the hardware stack.[1] The feasibility of this modification from the perspective of cost is addressed in Section 4.6. The `ret` instruction is modified to retrieve the last return address on top of the hardware stack and compare it with the current value of register 26. If there is a mismatch, then the return address must have been altered during the execution of the function, and an exception may terminate execution.

Because modern processors execute instructions out of program order and speculatively under branch prediction, return instructions may be executed under misspeculation and out of program order. Consequently, comparing the return address of the ret instruction with the top of the hardware stack *at the time of execution* is not reliable. Instead, we perform the comparison at the time the `ret` instruction *commits,* which occurs in program order and after all outstanding speculations are confirmed.

However, there is one difficulty: The ret instruction does not carry the register 26 value with it at commit. The value would have been written to the register file at execution, which occurs well before commit. As such, the only way at commit to obtain the register 26 value is to read the register file. To read from register 26 and compare to the top of the hardware stack, we use one of the register read ports. In special cases this can cause a small overhead, which we explain in detail next.

A register file has sufficient data read and write ports to enable it to handle the maximum possible number of references by all instructions issued simultaneously. The maximum number of read ports implemented is twice the issue width of the processor.[2] This is because the maximum number of ports used by a single instruction is two (e.g., reading two source operands). If we have a machine with an issue width of k then, our modified machine encounter a *stall* whenever k instructions are issued simultaneously *and* all k each need to read two source operands. Because we are working with a pipelined architecture, while an instruction is issuing (reading source operands, getting ready to execute), another instruction can be at the commit stage trying to complete a return instruction. If all data ports are used, the return instruction cannot read the value of register 26 to make the comparison with the return address on the process stack. Therefore, the issuing of instructions is stalled to allow a port to be used for the return instruction. It is an engineering trade-off whether to add an extra read port to eliminate the stalls or just to stall one of the issuing instructions. In our case, the stalls are very infrequent (see Section 5) therefore they do not impose a real threat to performance. Whereas adding an extra read port is a substantial modification leading to an increase in cost.

[1] Our modification is to the implementation of these instructions in hardware.

[2] Issue width is the number of instructions that can be issued simultaneously subject to the number of available functional units.

4.3 Handling `setjmp()` and `longjmp()`

One of the more complicated aspects of any attempt to protect the call stack is considering how to handle `setjmp` and `longjmp` functions. Briefly, `setjmp` stores the context information for the current stack frame and execution point into a buffer, and `longjmp` causes that environment to be restored. This allows a program to quickly return to a previous location, effectively short-circuiting any intervening return instructions. One place this might be used is in a complex search algorithm: the program uses `setjmp` to mark where to return once the item is found, begins calling search functions, and once the target is found it will `longjmp` back to the entry point.

However, since this avoids going through the usual function return sequence, our hardware stack becomes inconsistent with respect to the software stack. (Actually, the `longjmp` moves the stack pointer back to the previous location; the inconsistency is not with the data, but the location that is pointed to as top-of-stack.) We are currently working on finding an elegant solution to this problem.

Only mechanisms that store a copy of the return address stack need to handle `setjmp` and `longjmp`. Their solution has been to continue to pop return addresses off of their stored table of return addresses until a match is found (e.g., RAD). The problem with this approach is that it is possible that the modified return address value exists somewhere further down on the hardware stack, causing execution to continue without detecting the problem. As has been pointed out before, failing to stop execution is no worse than the current situation where no check is being made, but this answer is unsatisfactory.

If we were to store the current height of the hardware stack with our `setjmp` call, then at the time that `longjmp` is invoked, we would know how many return addresses need to be removed from the stack to be at the proper location. Unfortunately, `setjmp` and `longjmp` are both handled in user space, and there is a considerable security risk to allowing a user process to modify our hardware stack directly. If we instead let the OS or CPU continue to pop off addresses, we run the risk of entering an inconsistent state.

As the risk of inconsistency is less than the problems encountered by simply ignoring the situation, we have elected to allow the OS to continue popping off addresses until an underflow occurs. We are still looking for an alternate solution, and currently expect that one can be found but it will require OS support.

4.4 Handling Process Context Switches

A *context switch* is the process of switching the currently running process with one of the other processes ready to execute to implement a concurrent multi-process operating system. The context switch function is called by an exception handler – that is raised by a timer interrupt – either when the allowed time quota for execution of the running process expires, or when the running process is blocked (for I/O etc.). This function checks to see whether there is a higher priority process ready to execute. If not, the interrupted process continues to execute until the next call to the context switch function. A context switch requires saving the process and processor state information, in a structure called the Process Control Block (PCB), and loading the state information of the scheduled process. For our modification, we have to page out the hardware stack of the running process (less than 1K) either to the PCB or a memory location pointed by a special pointer in the PCB and page in the hardware stack of the scheduled process. We describe how the copy to/from the hardware stack is done seamlessly using memory mapped I/O in Section 4.6.

We do not expect swapping the hardware arrays at every context switch to cause a substantial overhead for two reasons. First, context switches happen infrequently (tens of milliseconds). Second, the overhead incurred by copying two 1K arrays (one for the process being swapped out and the other for the process being swapped in) is negligible with respect to the overhead of the rest of the context switch. The storage

and retrieval of the hardware stack to kernel memory will be as safe as all other kernel operations (i.e., the OS protects the kernel's memory space from other processes).

4.5 Handling Deeply-Nested Function Calls

Our hardware stack has a hard-limit on its size because it is inside the CPU and there is no dynamic memory allocation in hardware. This means that the buffer may fill up for programs with deeply nested function calls. A 1 KB stack of registers holds 256 32-bit addresses (e.g., x86) or 128 64-bit addresses (e.g., Alpha). To handle nested function calls deeper than 128 (256) we will raise a *hardware-stack-overflow exception*, which will copy the contents of the hardware stack to the kernel memory where it is saved at context switch. This kernel memory location will be a stack of stacks and every time a stack is full, then it will be appended to the previous full stack. Another exception, *hardware-stack-underflow*, will be raised when the hardware stack is empty to page in the last saved full-stack from the kernel memory. Just as with context switches, saving and retrieving the hardware stack will be handled by the kernel so it will be secure.

4.6 Feasibility of Hardware Modification

SmashGuard's implementation cost is minimal. The main component of the cost is the hardware stack in the processor to hold function return addresses. Considering that modern microprocessors employ on-chip level one (L1) caches of sizes 32-64 KB, and on-chip L2 caches exceeding 1 MB, the 1 KB stack adds minimal overhead (less than one-tenth of one percent) to the chip.

Adding the stack to the next implementation of an instruction set (e.g., Pentium III and Pentium IV are both implementations of the x86 instruction set) does not present any difficulties. It is common practice for newer implementations to incorporate optimizations for better performance. Indeed, such optimizations often involve employing tables which are similar to SmashGuard's hardware stack.

When introducing new hardware, a key cost factor to avoid is the introduction of new instructions to the instruction set. New instructions imply an implicit cost in future implementations which must support the new instructions (in their original form) for compatibility reasons. Because SmashGuard introduces a hardware stack, we have to ensure that the stack does not imply new instructions. If the hardware stack were completely invisible to software (e.g., the hardware caches are usually invisible to the user-level code, unless the code optimizes for cache performance), then the stack will not require new instructions.

In our approach the hardware stack is invisible to software except for context switches and when the call depth exceeds the stack size. In the later case, an exception is raised and the exception handler copies the stack to locations in memory owned by the OS. Upon a context switch, the previous process's stack contents have to be saved and the new process's stack contents have to be restored. To handle both these scenarios without adding any special instructions to the instruction set, we employ *memory mapping* (similar to memory-mapped I/O), so that regular load or store instructions may be used to read and write the stack in these scenarios. We map a part of the address space to the hardware stack, much like other parts of the address space are memory-mapped to I/O devices. A regular load or store access to this part translates to a read or write access to the hardware stack, much as memory-mapped I/O devices are read and written. Recall that I/O devices are protected from direct access by user-level code via virtual memory protection.

Similarly, direct access to the hardware stack is forbidden by virtual memory protection of that part of the address space. Thus, only the OS can read or write the memory-mapped stack, and the OS does so to handle both scenarios.

Chapter 5

Experiments and Results

Our preliminary experiments were run under the assumptions of a single process machine and that no process contains function call nesting deeper than 127 levels. In this section we first describe the simulation tool we used to determine the performance overhead of our proposed solution. We then present performance statistics on the SPEC-2K benchmarks for the Alpha architecture [1]. We illustrate how our method can detect Types 1 and 2 buffer overflow attacks on the return address pointer while incurring virtually no overhead.

5.1 The Simulation Tool: SimpleScalar

SimpleScalar[2] simulates a (fictitious) MIPS-like architecture and the Alpha architecture. The simulator is freely available for academic research and it is by far the most extensively used simulation tool in computer architecture research and instruction. SimpleScalar simulates the modern multiple-stage pipeline, multiple-issue Reduced Instruction Set Computer (RISC) architectures. The SPEC-2K binaries we use in our experiments were compiled on a Dec Alpha running Digital UNIX (Tru64) v.4 operating system.

We chose SimpleScalar for our research because it is a standard in processor performance estimation and because there was no mature x86 simulator with available benchmark suites. SimpleScalar provides low-level hardware performance metrics in terms of the number of cycles a program takes to execute. One drawback of SimpleScalar is that it is a CPU simulator, not a system simulator. Therefore, it does not have operating system support for exception handling, virtual memory, file I/O or multi-processing. In future work we will evaluate the cost of the proposed changes to the kernel to support saving of the hardware stack during process context switches and deeply nested function calls.

The simulation parameters used with SimpleScalar are presented in Table 5.1. In a superscalar (multiple-issue) processor, the issue width indicates the number of instructions issued for execution simultaneously. The ReOrder Buffer (ROB) holds the results of instructions that execute out of order between the time the instructions complete execution and the time they commit. Level 1 (L1) and level 2 (L2) caches are two levels of cache hierarchy, to bridge the huge gap between the CPU and memory speeds. The low level data (d-cache) and instruction caches are separated to increase the cache bandwidth to feed the agressive CPU. The branch predictor helps reduce stalls in a pipeline due to control hazards (e.g., conditional branches).

Issue Width	4 instrucitons
ROB Size	16 entries
Load/Store Queue	8 entries
L1 i-Cache	16K 1-way, 1 cycle hit time
L1 d-Cache	16K 4-way, 1 cycle hit time
L2 Cache	256K 4-way, 6 cycle hit time
Branch Predictor	2-level hybird, 8K entries

Table 5.1: Simulation parameters

Name	Cycles without SmashGuard	Cycles With SmashGuard	Percent Overhead	Max Stack Size
gcc	508833442	508834551	0.0002	63
gzip	280107128	280128539	0.0076	13
bzip	280631599	280631599	0.0000	11
mesa	281512571	281512571	0.0000	16
ammp	664087874	664087874	0.0000	20

Table 5.2: Performance results for modifications to the SimpleScalar simulator run on the SPEC-2K benchmarks for the Alpha architecture.

5.2 Performance Results

To verify that our hardware modifications can actually detect changes in the stored return address, we created a binary for the Alpha that overwrites one of it's own local buffers and executed it in the simulator. We were limited to self-attacking code because SimpleScalar only supports single process execution. Our hardware modification was able to detect that the return address value being pulled from the stack did not match the value stored in the hardware stack and terminated the process.

Our second set of experiments were designed to assess the overhead of the proposed hardware solution. Our experiments were conducted to compare the cycles required with and without our modification on the SPEC-2K hardware performance benchmarks. We chose three SPEC INT and two SPEC FP benchmarks. To compute our timing results, we followed the convention and skipped the first two billion instructions and then executed the next 500 million. Note that SimpleScalar *skips* instructions by doing functional simulation (so that the state of the processor is simulated) as opposed to a timing simulation. Skipping is done because the first set of instructions is typically initialization code and is not representative of the benchmark. Skipping two billion instructions and running 500 million instructions is typical in architecture research.

The results are shown in Table 5.2. The first column reports the number of cycles for the unmodified version of SimpleScalar. In computer architecture, it is customary to show processor cycles rather than wall clock time (which includes I/O times) for techniques that incur instruction execution overhead. Processor cycles isolates the incurred overhead without being diluted by I/O times. The second column reports the number of cycles when SmashGuard is used. Recall that the overhead occurs because by using one of the read ports to read the return address from register 26 during `ret` instruction we cause instructions to be stalled. Simulations show no more than one `jsr/bsr` or `ret` instruction in the commit stage when all ports are full. The last column reports the maximum stack size for each of the benchmark programs. The deepest nesting in the benchmarks was less than half the size of the hardware stack.

Chapter 6

Conclusions and Future Work

In this paper we presented an inherently fast and secure hardware-based solution to protecting the function return addresses that are stored on the program stack. The strengths of our approach are that

1. no recompilation of the source code in necessary;
2. in contrast to software-based approaches, our method requires minimal CPU overhead while retaining the ability to protect against *all* known forms of attack on the return address pointer; and
3. the hardware modification is inexpensive and non-intrusive because it does not require that we modify the instruction set.

Our experiments with the SPEC-2K benchmarks show less than 0.01% performance overhead, and a maximum nesting depth of 63.

Our experiments were limited to a single process machine that handled programs whose nesting depth did not exceed 127. Our next goal is to assess the performance overhead of our proposed modifications to handle process context switching and deeply nested functions. We are currently investigating systems simulators that can be modified in the hardware and OS level to implement our hardware method on a multi-process environment. The key points in our simulator search are concurrent multi-processing operating system support and low-level performance timing measurements. Finally, we would like to find an elegant solution to handle `setjmp` and `longjmp`.

Bibliography

[1] Spec2k alpha binaries, 2001.

[2] T. Austin. Simplescalar llc, 2001.

[3] A. Baratloo, N. Singh, and T. Tsai. Libsafe: Protecting critical elements of stacks. White Paper at `http://www.research.avayalabs.com/project/libsafe/`, December 1999.

[4] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Technical Conference*, San Diego, California, USA, June 2000.

[5] J. Bowman. Format string attacks: 101, 2000.

[6] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack Magazine*, 10(56), May 2000.

[7] CERT Coordination Center. CERT/CC advisories 2002.

[8] T. Chiueh and F-H. Hsu. Rad: A compile time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, Arizona, USA, April 2001.

[9] M. Conover and w00w00 Security Team. w00w00 on heap overflows, 1999.

[10] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J Lokier. Formatguard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[11] C. Cowan, C. Pu, Maier D., H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, January 1998.

[12] Solar Designer. Linux kernel patch from the openwall project: Non-executable user stack, January 2001.

[13] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[14] The SANS Institute. The twenty most critical internet security vulnerabilities (updated) – the experts' consensus, Version 3.21 October 17 2002.

[15] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[16] Team Teso Scut, September 2001.

[17] L. Torvalds. Reply to non-executable stack patch, August 1998.

[18] Vendicator. Stack shield: A "stack smashing" technique protection tool for linux, January 2001.