Enlisting Hardware Architecture to Thwart Malicious Code Injection

Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi
Princeton Architecture Laboratory for Multimedia and Security (PALMS)
Department of Electrical Engineering
Princeton University
{rblee, dkarig, mcgregor, zshi}@ee.princeton.edu

Abstract. Software vulnerabilities that enable the injection and execution of malicious code in pervasive Internet-connected computing devices pose serious threats to cyber security. In a common type of attack, a hostile party induces a software buffer overflow in a susceptible computing device in order to corrupt a procedure return address and transfer control to malicious code. These buffer overflow attacks are often employed to recruit oblivious hosts into distributed denial of service (DDoS) attack networks, which ultimately launch devastating DDoS attacks against victim networks or machines. In spite of existing software countermeasures that seek to prevent buffer overflow exploits, many systems remain vulnerable.

In this paper, we describe a hardware-based secure return address stack (SRAS), which prevents malicious code injection involving procedure return address corruption. Implementing this special hardware stack only requires low cost modifications to the processor and operating system. This enables the hardware protection to be applied to both legacy executable code and new programs. Also, this hardware defense has a negligible impact on performance in the applications examined. The security offered by this hardware solution complements rather than replaces that provided by existing static software techniques. Thus, we detail how the combination of the proposed secure return address stack and software defenses enables comprehensive multi-layer protection against buffer overflow attacks and malicious code injection.

1 Introduction

As the number and networking capabilities of pervasive computing devices increase, built-in security for these devices becomes more critical. Hostile parties can exploit any of several security vulnerabilities in Internet-enabled computing devices to inject malicious code that is later employed to launch large-scale attacks. Furthermore, attacks involving billions of compromised pervasive computing devices can be much more devastating than attacks that employ thousands or millions of traditional desktop machines. Malicious code is often inserted into victim computers by taking advantage of software vulnerabilities such as buffer overflows, which can alter the

This work was supported in part by the NSF under grants CCR-0208946 and CCR-0105677 and in part by a research gift from Hewlett-Packard.

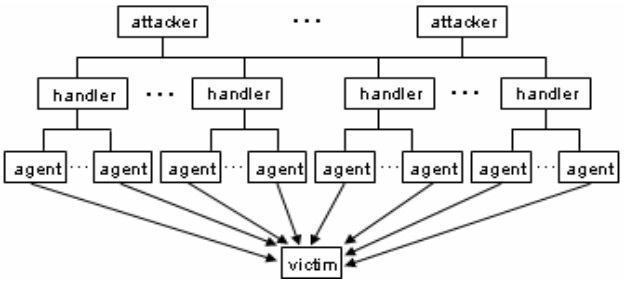

Fig. 1. Distributed denial of service attack network

Table 1. CERT buffer overflow advisories

Year	Advisories	Advisories involving buffer overflow	Percent buffer overflow
1996	27	5	18.52 %
1997	28	15	53.57 %
1998	13	7	53.85 %
1999	17	8	47.06 %
2000	22	2	9.09 %
2001	**37**	**19**	**51.35 %**

control flow of the program. In this paper, we propose a built-in hardware defense for processors to prevent malicious code injection due to buffer overflow attacks.

Buffer overflows have caused security problems since the early days of computing. In 1988, the Morris Worm, which resulted in large-scale denial of service, spread throughout the Internet using a buffer overflow vulnerability as one of its means of intrusion. The Code Red worm further exemplifies the severity of problems that buffer overflow vulnerabilities still cause today. Code Red and its variants, which stung companies over the summer of 2001, took advantage of a buffer overflow problem in Microsoft IIS. The total economic cost of these worms was estimated at $2.6 billion by Computer Economics [19].

Buffer overflow vulnerabilities also play a significant role in distributed denial of service (DDoS) attacks. In such attacks, an adversary compromises a large number of machines to set up a DDoS network that is later used to launch a massive, coordinated attack against a victim machine or network. A typical DDoS network is shown in Figure 1. An adversary controls one or more handler machines, which in turn command the agent machines (also called “zombies”) that actually carry out the attack. This network structure allows an attacker to easily control a large number of machines and makes the attacker difficult to trace. Furthermore, as the number of pervasive computing devices grows rapidly, the potential destructiveness of DDoS attacks greatly increases.

Various tools are available that provide for the large-scale compromise of machines and the installation of DDoS attack software. These tools scan thousands of hosts for the presence of known weaknesses such as buffer overflow vulnerabilities. Susceptible hosts are then compromised, and attack tools are

installed on the oblivious handler or agent machines. The compromised hosts can then be used to scan other systems, and this cycle of intrusion may be repeated indefinitely [11]. The tools differ in the types of attacks they execute and in the communication between nodes in the attack network, but all allow the attacker to orchestrate large-scale, distributed attacks [15, 16]. Popular attack tools include Trinity, trinoo, Tribal Flood Network (TFN) and TFN2K, and Stacheldraht [4].

Defending against DDoS attacks in progress is extremely difficult. Hence, one of the best countermeasures is to hinder attack networks from being established in the first place, and defending against buffer overflow vulnerabilities is an important step in this direction. Table 1 shows the percentages of CERT advisories between 1996 and 2001 relating to buffer overflow weaknesses. In 2001, more than 50 percent of CERT advisories involved buffer overflow. Furthermore, buffer overflow weaknesses play a very significant role in the 20 most critical Internet security vulnerabilities identified by the SANS Institute and the FBI [20].

The majority of buffer overflow exploits involve an attacker "smashing the stack" and changing the return address of a targeted function to point to injected code. Thus, protecting return addresses from corruption prevents many attacks. Past work addresses the problem through static and dynamic software methods, such as safe programming languages, operating system patches, compiler changes, and even run-time defense. However, the examination of potential solutions at the hardware architecture level is justified by the frequency of this type of attack, the number of years it has been causing problems, the continuing emergence of such problems despite existing software solutions, and the explosive increase of vulnerable devices.

We propose a hardware-based, built-in, non-optional layer of protection to defend against common buffer overflow vulnerabilities in all future systems. We detail how a hardware secure return address stack (SRAS) mechanism can be used to achieve this goal. The mechanism preserves a correct copy of every procedure return address for correct program control flow, and it provides a means of detecting buffer overflow attacks with high probability. Our proposal is a "hardware safety net" that should be applied in conjunction with safe programming techniques and compiler-inserted checking mechanisms to provide a multi-layered defense.

In Section 2, we describe the problem of return address corruption caused by buffer overflows. We summarize and compare past work in Section 3. In Section 4, we present a multi-layer software and hardware protection mechanism for buffer overflow attacks in pervasive computing devices. We describe the hardware architectural support for our proposal in Section 5. In Section 6, we discuss performance and implementation costs, and we conclude in Section 7.

2 Stack Smashing via Buffer Overflow

Most buffer overflow attacks involve corruption of procedure return addresses in the memory stack. During the execution of a procedure call instruction, the processor transfers control to code that implements the target procedure. Upon completing the procedure, control is returned to the instruction following the call instruction. This transfer of control occurs in a LIFO (i.e., Last In First Out) fashion, or properly

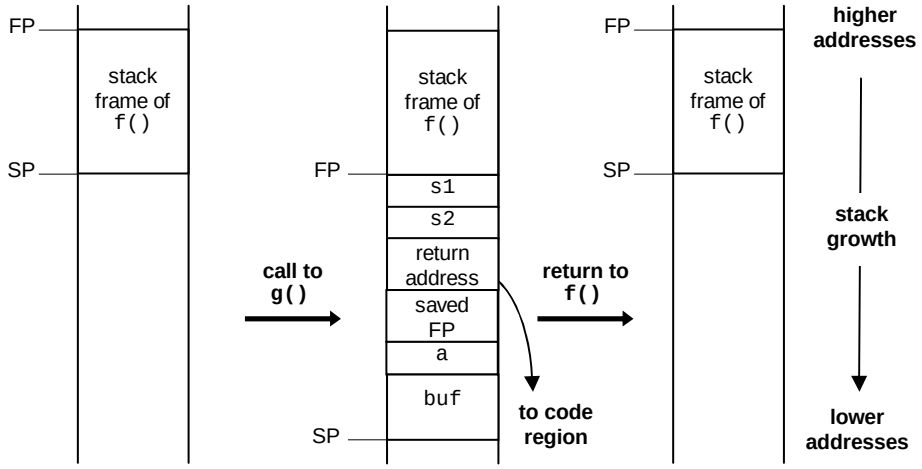

Fig. 2. Example of stack operation

```
int f()
{
    …
    g(s1, s2);
    …
}

int g(char *s1,
      char *s2)
{
    int a;
    char buf[100];
    …
    strcpy(buf, s1);
    …
}
```

Fig. 3. Code example

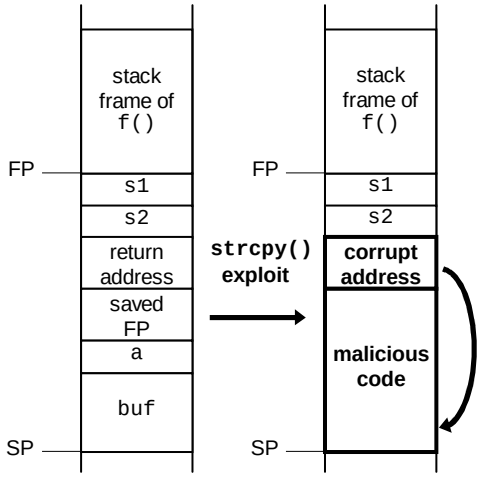

Fig. 4. Buffer overflow attack

nested fashion. Thus, a procedure call stack, which is a LIFO data structure, is used to save the state between procedure calls and returns. Compilers for different languages use the same stack format, and therefore a function written in one language can call functions written in other languages. We describe memory stack behavior for the IA-32 architecture [12], but the general procedures apply to all conventional ISAs.

The memory stack is typically implemented as a contiguous block of memory that grows from higher addresses toward lower addresses (as shown in Figure 2). The stack pointer (SP) is used to keep track of the top of the stack. When an item is pushed onto or popped off the stack, the SP is adjusted accordingly. Anything beyond the SP is considered to be garbage. We can reference data on the stack by

adding an offset to the SP, and modifying the SP directly can either remove a batch of data or reserve space for a data such as local variables. The stack consists of a set of stack frames; a single frame is allocated for each procedure that has yet to return control to an ancestor procedure. The SP points to the top of the stack frame of the procedure that is currently executing, and the frame pointer (FP) points to the base of the stack frame for that procedure. To avoid destroying the value of the current FP upon calling a new procedure, the FP must be saved on entry to the new procedure and restored on exit.

Figure 2 illustrates the operation of the memory stack for the example program in Figure 3. The leftmost stack shows the state of the stack immediately preceding the call to `g()`. When function `f()` calls `g()`, a new stack frame will be pushed onto the stack. This frame includes the input pointers `s1` and `s2`, the procedure return address, the frame pointer, and the local variables `a` and `buf`. Upon completing `g()`, the program will return to the address stored in `g`'s stack frame; this address should equal the location of the instruction immediately following the call to `g()` in the function `f()`. The SP and the FP are also restored to their former values, and the stack frame belonging to `g()` is effectively popped from the stack.

Figure 4 illustrates a buffer overflow attack on the code listed in Figure 3. A security vulnerability exists because `strcpy()` does not perform bounds checking. In the function `g()`, if the string to which `s1` points exceeds the size of `buf`, `strcpy()` will overwrite data located adjacent to `buf` in the memory stack. A malicious party can exploit this situation by strategically constructing a string that contains malicious code and a corrupted return address. If `s1` points to such a string, `strcpy()` will copy malicious code into the stack and overwrite the return address in `g()`'s stack frame with the address of the initial instruction of the malicious code. Consequently, once `g()` completes, the program will jump to and execute the malicious code instead of returning control to `f()`. There are many variations of this form of attack, but most rely on the ability to modify the return address [17]. For example, rather than the attacker injecting his own exploit code, the return address may be modified to point to legitimate, preexisting code that can be used for malicious purposes. In another variant, the malicious code inconspicuously installs agent software for a future DDoS attack and returns execution to the calling function `f()`. Thus, the program appears to execute normally, and the user is unaware that his machine may become a DDoS zombie in a future attack.

3 Past Work

Researchers have proposed many software-based countermeasures for thwarting buffer overflow attacks. These methods differ in the strength of protection provided, the effects on performance, and the ease with which they can be effectively employed.

One solution is to store the memory stack in non-executable pages. This can prevent an attacker from executing code injected into the memory stack. For

Table 2. Required system changes

Technique for defending against procedure return address corruption	Required system changes			
	Source code	Compiler	OS	Processor
Safe programming languages	Yes	Yes	No	No
Static analysis techniques	Yes	No	No	No
StackGuard	No	Yes	No	No
StackGhost	No	No	Yes	No
`libsafe`	No	No	Yes	No
`libverify`	No	No	Yes	No
Our SRAS proposal	**No**	**No[1]**	**Yes**	**Yes**

[1]Compiler changes may be required for certain programs to operate properly depending on the method used to handle non-LIFO procedure control flow (see Section 5).

Table 3. Benefit and cost comparison

Technique for defending against procedure return address corruption	Provides complete protection[1]	Applies to many platforms	Application code size increase	Adverse performance impact
Safe programming languages	Yes[3]	Yes	Can be high	Can be high
Static analysis techniques	No	Yes	Varies	Varies
StackGuard	No	Yes	Low	Moderate
StackGhost	Yes	No	None	Low
`libsafe`	No	Yes	Low	Low
`libverify`	Yes	Yes	High	Moderate
Our SRAS proposal	**Yes**	**Yes**	**None[2]**	**Low**

[1]By "complete protection," we mean complete protection against buffer overflow attacks that directly corrupt procedure return addresses.
[2]Depending on how non-LIFO procedure control flow is handled, some programs may experience a very small increase in code size (see Section 5).
[3]Provided that programmers comply and write correct code.

example, Multics was one of the first operating systems to provide support for non-executable data memory, i.e., memory pages with execute privilege bits [14]. However, the return address may instead be redirected to preexisting, legitimate code in memory that the attacker wishes to run for malevolent reasons. In addition, it is difficult to preserve compatibility with existing applications, compilers, and operating systems that employ executable stacks. Linux, for instance, depends on executable stacks for signal handling.

Researchers have proposed using more secure (or safe) dialects of C and C++, since a high percentage of buffer overflow vulnerabilities can be attributed to features of the C programming language. Cyclone is a dialect of C that focuses on general program safety, including prevention of stack smashing attacks [10]. Safe programming languages have proven to be very effective in practice. While programs written in Cyclone may require less scrupulous checking for certain types of vulnerabilities, the downside is that programmers have to learn the numerous distinctions from C, and legacy application source code must be rewritten and

recompiled. In addition, safe programming dialects can cause significant performance degradation and executable code bloat.

Methods for the static, automated detection of buffer overflow vulnerabilities in code have also been developed [22, 23, 24]. Using such static analysis techniques, complex application source code can be scanned prior to compilation in order to discover potential buffer overflow weaknesses. The detection mechanisms are not perfect: many false positives and false negatives can occur. Also, as true with Cyclone, these techniques ultimately require the programmer to inspect and often rewrite sections of application source code. Re-coding may also increase the total application code size.

StackGuard is a compiler-based solution involving a patch to `gcc` that defends against buffer overflow attacks that corrupt procedure return addresses [8]. In the procedure prologue of a called function, a "canary" value is placed on the stack next to the return address, and a copy of the canary is stored in a general-purpose register. In the epilogue, the canary value in memory is compared to the canary register to determine whether a buffer overflow has occurred. The application randomly generates the 32-bit or 64-bit canary values, so the application can detect improper modification of a canary value resulting from a buffer overflow with high probability. However, there exist attacks that can circumvent StackGuard's canaries to successfully corrupt return addresses and defeat the security of the system [2].

StackGhost employs the SPARC architecture's register windows to defend against buffer overflow exploits [9]. Return addresses that have stack space allocated in register windows are partially protected from corruption. The OS has the responsibility of spilling and filling register windows to and from memory, and once a return address is stored back in memory, it is potentially vulnerable. Various methods of protecting such spilled stacks are defined. Buffer overflow protection without requiring re-compilation of application source code is a benefit of StackGhost, but the technique is only applicable to SPARC systems.

Transparent run-time software defenses have also been proposed. The dynamically loaded libraries `libsafe` and `libverify` provide run-time defenses against stack smashing attacks and do not require programs to be re-compiled [1]. `libsafe` intercepts unsafe C library functions and performs bounds-checking to protect frame pointers and return addresses. `libverify` protects programs by saving a copy of every function and every return address in the heap. The first instruction of the original function is overwritten to execute code that stores the return address and jumps to the copied function code. The return instruction for the copied function is replaced with a jump to code that verifies the return address before actually returning.

The downside to `libsafe` is that it only defends against buffer overflow intrusions resulting from certain C library functions. In addition, static linking of these C library functions in a particular executable precludes `libsafe` from protecting the program. Implementations of `libverify` can double the code space required for each process, which is taxing for embedded devices with limited memory. Also, `libverify` can degrade performance by as much as 15% for some applications.

We compare past work in Tables 2 and 3. We observe that no existing solution combines the features of support for legacy applications (indicated by no changes to source code or the compiler), wide applicability to various platforms, low performance overhead, and complete protection against procedure return address corruption. Therefore, we propose a low-cost, hardware-based solution that enables built-in, transparent protection against common buffer overflow vulnerabilities without depending on user or application programmer effort in complying with software safeguards and countermeasures.

4 A Multi-layer Defense

We advocate a multi-layer approach to solving buffer overflow problems that lead to procedure return address corruption. By "multi-layer", we mean a combination of static software defenses and dynamic software or hardware defenses. Static software techniques include safe programming languages, static security analysis of source code, and security code inserted into executables at compile-time. Dynamic software security solutions include run-time defenses such as StackGhost, `libsafe`, and `libverify`. We present a dynamic hardware defense in the next section.

We categorize programs as new software and legacy software. With new software, the source code is available, so the programmer can apply static software techniques for defending against buffer overflows. In addition, the platform can provide dynamic software or hardware defenses to supplement these static techniques. Legacy software consists of compiled binary executables – the corresponding source code is no longer available. Hence, the only applicable protection for legacy software is dynamic (i.e., run-time) software or hardware defense. The dynamic software countermeasures described above may provide incomplete coverage (`libsafe`), only apply to a certain platform (StackGhost), or cause performance degradation and code bloat (`libverify`). Therefore, we recommend using a dynamic hardware countermeasure, which is designed to transparently provide protection for both new and legacy software.

We propose low-cost enhancements to the core hardware and software of future programmable machines that enable the detection and prevention of return address corruption. Such a processor-based mechanism would complement static software techniques in a multi-layered defense by overcoming some deficiencies of existing software solutions. Our proposed hardware defense provides robust protection, can be used in all platforms, causes negligible performance degradation, and does not increase code size. Since we require changes to processor hardware, our proposal is meant to be a longer-term solution. In the interim, software patches and defenses against buffer overflow vulnerabilities should continue to be applied when available.

5 The Processor-based Defense

In instruction set architectures, procedure call and return instructions are clearly recognizable from other branch instructions. For instance, in many RISC ISAs, a

branch and link instruction is identified as a procedure call, and a branch to the link register (such as R31) is identified as a procedure return instruction [18]. Furthermore, as explained in Section 2, procedure calls and returns occur in a properly nested, or LIFO, fashion. Since the processor can clearly identify call and return instructions, it can maintain its own LIFO hardware stack to store the correct nested procedure return addresses. The processor does not need to depend on the memory stack in which return addresses can be corrupted by external sources (that exploit software vulnerabilities such as buffer overflows).

We propose that security-aware processors implement a *secure return address stack* (SRAS) that preserves correct values of dynamic procedure return addresses during program execution. Only call and return instructions can modify the contents of the SRAS, and the processor can rely on the SRAS to provide the correct return address when executing a procedure return instruction. If the return address given by the SRAS hardware differs from that stored in the memory stack, then it is highly likely that the return address in the memory stack has been corrupted. In this event, the processor can terminate execution, continue execution using the correct address from the top of the SRAS, or issue a new *invalid return address* trap. With the SRAS, we can achieve our goal of thwarting buffer overflow attacks in which hostile code is injected into innocent hosts.

Our SRAS solution differs significantly in the security function it provides compared to the performance function provided by hardware return address stacks [13, 25] found in some high-performance processors like the Alpha 21164 [5] and the Alpha 21264 [6]. In these processors, the hardware return address stack provides a mechanism for branch prediction; the target address of a procedure return instruction is highly predictable, and thus it can be made available earlier in the pipeline. The processor uses a return address stack in conjunction with other mechanisms such as branch target buffers to perform branch prediction. Since branch prediction mechanisms are not expected to be 100% accurate, if the address predicted by the hardware return address stack differs from the return address saved in the memory stack, the processor assumes that the branch prediction is incorrect. It will "squash" instructions based upon the address popped from the hardware return address stack and start fetching instructions beginning at the return address stored in the memory stack. Hence, in the event of return address corruption due to buffer overflow exploitation, existing processors will jump to the malicious code pointed to by the corrupted return address. In contrast, our SRAS solution places trust in the processor's hardware stack rather than in the memory stack, which can be modified by external sources.

5.1 SRAS Architectural Requirements

Supporting a Secure Return Address Stack mechanism in a processor requires a hardware return address stack (the SRAS itself), modification of the implementation of procedure call and return instructions to use the SRAS, and a method for securely spilling and filling of the contents of the SRAS to and from memory upon SRAS overflow or underflow. Since we do not require re-compilation or changes to

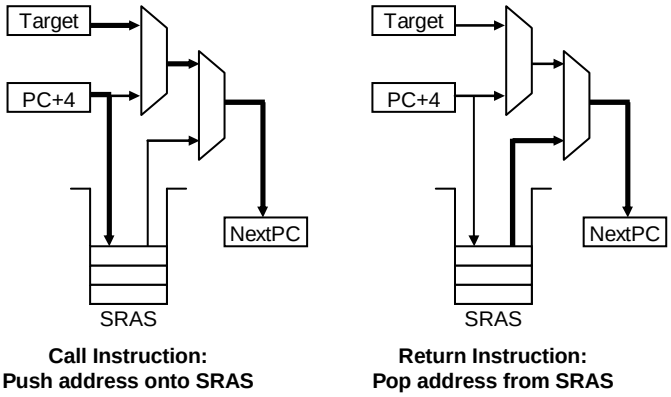

Fig. 5. SRAS operation for call and return instructions

programming languages and application source code, both legacy and new software can benefit from the security provided by these enhancements.

The hardware SRAS is simply an *n*-entry LIFO stack. We transparently modify the execution of procedure call and return instructions to place trust in the SRAS rather than the memory stack as follows. We maintain the ISA definitions and visible behavior of call and return instructions, but we alter the manner in which the processor executes call and return instructions to use the SRAS (see Figure 5). This enables protection for legacy programs as well as new programs. During the execution of a call instruction, the target of the procedure call is assigned to the next PC. Also, the return address (i.e., PC + 4 assuming the call instruction size is 4 bytes) is pushed onto the top of the SRAS. When a processor fetches a return instruction, the return address popped from the top of the hardware SRAS is *always* assigned to the next PC. The processor then determines whether the return address from the memory stack is the same as the return address popped from the SRAS. If these addresses differ, return address corruption (or some other error) has occurred, and the processor should take appropriate action.

A hardware SRAS structure contains a finite number of entries, which may be exceeded by the number of dynamically nested return addresses in the program. When this happens, the processor must securely spill SRAS contents to memory. We define the event in which the SRAS becomes full following a call instruction as overflow; the event where the SRAS becomes empty following a return is defined as underflow. The processor issues an OS interrupt to write or read SRAS contents to or from protected memory pages when SRAS overflow or underflow occurs. To prevent thrashing in some programs due to SRAS spilling and filling, we only transfer half (instead of all) of the SRAS entries to or from memory on an SRAS overflow or underflow.

This SRAS overflow space in memory is protected from corruption by external sources by only allowing the OS kernel to access spilled SRAS contents. The OS executes code that transfers contents of the SRAS to or from these protected memory pages; the application does not, and cannot, participate in SRAS content transfers. The kernel is responsible for managing the memory structures required to store the

spilled SRAS entries for all threads running on the system. This is achieved by maintaining one stack of spilled SRAS return addresses for each process. In addition, the virtual memory regions that store the SRAS contents are mapped to physical pages that can only be accessed by the kernel. Hence, user-level application threads cannot corrupt the contents of their respective spilled stacks. Also, since the values popped from the SRAS must always be valid to preserve correct execution, the OS must transfer the SRAS contents to and from memory during context switches.

5.2 Non-LIFO Procedure Control Flow

If software always exhibited LIFO procedure control flow behavior, the SRAS would transparently provide hardware-based protection of return addresses for all programs. No compiler changes or recompilation of existing source code would be necessary: the system would provide protection for all legacy and future binary executables. Unfortunately, however, some existing executables use non-LIFO procedure control flow. For example, some compilers seek to improve performance by allowing certain procedures to return to an address located deep within the stack. The memory stack pointer is then set to an address of a frame buried within the stack; the frames located in between the former top of the stack and the reassigned stack pointer are effectively popped and discarded. Exception handling in C++ is one technique that can lead to such non-LIFO behavior.

Other common causes of non-LIFO control flow are the C `setjmp` and `longjmp` library functions. These functions are employed to support software signal handling. The `longjmp` function may cause a program to return to an address that is located deep within the memory stack or to an address that is no longer located in the memory stack. More specifically, a particular return address may be explicitly pushed onto the stack only once, but procedures may return to that address more than once. Note that tail call optimizations, which seem to involve non-LIFO procedure control flow, do not cause problems for the SRAS. Compilers typically maintain proper pairing of procedure call and return instructions when implementing tail call optimizations.

Our security proposal depends on the correctness of the address popped from the top of the hardware SRAS. Hence, the SRAS mechanism described so far does not accommodate non-LIFO procedure control flow. We can address this issue in at least four ways. The first option prohibits non-LIFO behavior in programs, providing the greatest security at the lowest cost but also the least flexibility. The fourth and last option disables the SRAS, providing the least security but the greatest flexibility for programs that exhibit non-LIFO behavior. There exist several possible alternatives between these two options that trade varying degrees of non-LIFO support for implementation cost and complexity. We present two of these possibilities: the second option described below relies on re-compilation, while the third option described below uses only dynamic code insertions. Both options only support certain non-LIFO behavior for cost and complexity reasons.

The first option is to implement the SRAS as described above and completely prohibit code and compiler practices that employ non-LIFO procedure control flow.

This provides the highest degree of security against return address corruption. To support this option, we may need to rewrite or re-compile source code for certain legacy applications. Legacy executables that exhibit non-LIFO procedure calling behavior will terminate with an error (if not recompiled).

The second option is to permit certain types of non-LIFO procedure control flow such as returning to addresses located deep within the stack. This option requires re-compilation of some legacy programs. During re-compilation, the compiler must take precautions to ensure that the top of the SRAS will always contain the correct target address for an executed return instruction in programs that use non-LIFO techniques. We define new instructions, `sras_push` and `sras_pop`, which explicitly push and pop entries from and to the SRAS without necessarily calling or returning from a procedure. Compilers can employ these new instructions to return to an address deep within the SRAS (and to the associated frame in the memory stack) when using `longjmp`, C++ exception handling, or other non-LIFO routines.

The third option is to provide dynamic support for common non-LIFO behavior. This approach does not support all instances of non-LIFO behavior that the second option can handle via re-compilation, but it does allow execution of some legacy executables (where the source code is no longer available) that exhibit non-LIFO procedure control flow. First, we implement the `sras_push` and `sras_pop` instructions described above. We also need an installation-time or run-time software filter that strategically injects `sras_push` and `sras_pop` instructions (as well as other small blocks of code) into binaries prior to or during execution. The software filter inserts these instructions in recognized routines that cause non-LIFO procedure control flow. For instance, standardized functions like `setjmp` and `longjmp` can be identified at run-time via inspection of linked libraries such as `libc`. This option only handles executables that employ known non-LIFO techniques, however. For new manifestations of non-LIFO procedure control flow, the software filter may not identify some locations where the new instructions should be inserted.

The fourth option is to allow the users to disable the SRAS with a new `sras_off` instruction. This enables the execution of code that exhibits non-LIFO procedure control behavior as permitted in systems without an SRAS. In some situations (e.g., program debugging), a user may wish to turn off the SRAS and run insecure code. In other cases, the user may disable the SRAS to execute legacy code with unusual non-LIFO behavior.

Regardless of the method used to handle non-LIFO procedure control flow, we require that the SRAS be “turned on” by default in order to provide built-in protection. Our architecture definition stipulates that the SRAS is always enabled unless explicitly turned off by the user, at his own risk.

6 Performance Impact

We now analyze the implementation costs of our proposal. First, we investigate the performance degradation caused by the SRAS mechanism on typical programs. The SRAS does not impact the performance of procedure call and return instructions. Any performance degradation is due to spilling and retrieving the contents of the

SRAS to and from memory during program execution. Although network-processing software is most vulnerable to buffer overflow attacks, the SRAS provides transparent protection for all applications, and therefore any SRAS-induced performance degradations apply to all software. Hence, we examine the performance impact of our SRAS solution on the SPEC2000 benchmarks [21], which are typically used to model a representative workload in processor performance studies.

We gather performance data using SimpleScalar version 3.0, a cycle-accurate processor simulator [3]. Our base machine model closely represents an ARM11 processor core, which is used in many network-enabled, embedded computing devices [7]. The ARM11 is a single-issue processor with 8 KB L1 instruction and data caches. Also, the ARM11 core supports limited out-of-order execution to compensate for the potentially high latencies of load and store instructions.

We simulate the execution of the first 1.5 billion instructions of 12 SPEC2000 integer benchmarks [21]. Our performance data is based upon the last 500 million instructions of each 1.5 billion instruction simulation in order to capture steady-state behavior. We obtain performance results for all 12 benchmarks and 6 SRAS sizes of 8, 16, 32, 64, 128, and infinite entries. Hence, we performed 12×6 = 72 simulations. To model the code executed by the OS upon SRAS overflow and underflow, we wrote a swapping and memory management routine in C. All of the benchmarks and swapping code were compiled using `cc` with `-O2` optimizations.

We find that the performance degradation caused by SRAS swapping is negligible (i.e., less than 1%) for all the benchmarks when using SRAS sizes of 128 or more entries. When using a 64-entry SRAS, the only benchmark that suffers a non-negligible performance penalty is `parser`, which experiences a small performance degradation of 2.11%.

Next, we compare the implementation costs of our proposed processor-based solution to `libverify`, a dynamic software-based solution that provides robust security against procedure return address corruption. We do not consider StackGhost and `libsafe`, for these solutions only function on SPARC platforms and only provide protection against buffer overflows in certain C functions, respectively. `libverify` does not require any changes to processors or hardware, which is an advantage over our proposal. Although our solution does require hardware enhancements, the necessary modifications are minor. In addition, many processors already contain a return address stack that would serve as the core of the SRAS.

Our SRAS solution causes a negligible performance penalty in the set of benchmarks examined, whereas `libverify` causes performance degradation as high as 15% in some common applications. Furthermore, our solution requires little or no expansion of executable code size. Since `libverify` copies functions to the heap at run-time, `libverify` can increase code size by a factor of two. Such code bloat can be very taxing for constrained devices in pervasive computing environments. Hence, our dynamic hardware-based solution is superior to dynamic software defenses from a performance perspective. As future processors are designed to include SRAS mechanisms, our dynamic hardware defense may be used to replace dynamic software defenses against procedure return address corruption.

7 Conclusion

Malicious parties utilize buffer overflow vulnerabilities to inject and execute hostile code in an innocent user's machine by corrupting procedure return addresses in the memory stack. Due to the growing threat of attacks such as distributed denial of service that exploit the rapidly increasing number of pervasive computing devices, addressing such buffer overflow vulnerabilities is a high priority for network and computer security. Although software-based countermeasures are available, a processor architecture defense is justified because major security problems stemming from buffer overflow vulnerabilities continue to plague computer systems.

We propose a built-in, non-optional, secure hardware return address stack (SRAS) that detects corruption of procedure return addresses. The SRAS mechanism only requires minor changes to the operating system and the processor, so legacy and new software can enjoy the security benefits without the need to modify application source code or re-compile the source, which may no longer be available. Also, the SRAS mechanism causes a negligible performance penalty in the applications examined. For greatest security, we suggest that new software disallow non-LIFO procedure control flow techniques. However, we describe compiler, OS, and hardware methods for supporting non-LIFO behavior when it is necessary. We also discuss the tradeoffs between security, implementation complexity, and software flexibility associated with supporting non-LIFO procedure control flow.

We describe a multi-layer software and hardware defense against buffer overflow attacks. Our hardware-based solution should be applied in tandem with existing static software countermeasures to provide robust protection in pervasive computing devices. In future work, we will explore SRAS enhancements and alternative techniques for preventing buffer overflow and distributed denial of service attacks.

References

[1] A. Baratloo, N. Singh, and T. Tsai, "Transparent Run-time Defense against Stack Smashing Attacks," *Proc. of the 9th USENIX Security Symposium*, June 2000.

[2] Bulba and Kil3r, "Bypassing StackGuard and StackShield," *Phrack Magazine*, vol. 10, issue 56, May 2000.

[3] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *University of Wisconsin-Madison Computer Sciences Department Technical Report*, no. 1342, June 1997.

[4] CERT Coordination Center, http://www.cert.org/, Nov. 2001.

[5] Compaq Computer Corporation, *Alpha 21164 Microprocessor (.28μm): Hardware Reference Manual*, December 1998.

[6] Compaq Computer Corporation, *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.

[7] D. Cormie, "The ARM11 Microarchitecture," available at http://www.arm.com/support/White_Papers/, April 2002.

[8] C. Cowan, C. Pu, D. Maier, 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," *Proceedings of the 7th USENIX Security Symposium*, Jan. 1998.
[9] M. Frantzen and M. Shuey, "StackGhost: Hardware Facilitated Stack Protection," *Proceedings of the 10th USENIX Security Symposium*, August 2001.
[10] L. Hornof and T. Jim, "Certifying Compilation and Run-time Code Generation," *Proceedings of the ACM Conference on Partial Evaluation and Semantics-Based Program Manipulation*, January 1999.
[11] K. J. Houle, G. M. Weaver, N. Long, and R. Thomas, "Trends in Denial of Service Attack Technology," CERT Coordination Center, October 2001.
[12] Intel Corporation, *The IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, Intel Corporation, 2001.
[13] D. R. Kaeli and P. G. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns," *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 34-41, May 1991.
[14] P. A. Karger and R. R. Schell, "Thirty Years Later: Lessons from the Multics Security Evaluation," *Proceedings of the 2002 Annual Computer Security Applications Conference*, pp. 119-126, December 2002.
[15] F. Kargl, J. Maier, and M. Weber, "Protecting Web Servers from Distributed Denial of Service Attacks," *Proceedings of the Tenth International Conference on World Wide Web*, pp. 514-525, April 2001.
[16] D. Karig and R. B. Lee, "Remote Denial of Service Attacks and Countermeasures," Princeton University Department of Electrical Engineering Technical Report CE-L2001-002, October 2001.
[17] klog, "The Frame Pointer Overwrite," *Phrack Magazine*, 9(55), Sept. 1999.
[18] R. B. Lee, "Precision Architecture," *IEEE Computer*, 22(1), pp. 78-91, Jan. 1989.
[19] J. McCarthy, "Take Two Aspirin, and Patch That System – Now," *SecurityWatch*, August 31, 2001.
[20] The SANS Institute, "The SANS/FBI Twenty Most Critical Internet Security Vulnerabilities," http://www.sans.org/top20/, October 2002.
[21] The Standard Performance Evaluation Corporation, http://www.spec.org/, Nov. 2001.
[22] J. Viega, J. T. Bloch, T. Kohno, and G. McGraw, "ITS4: A Static Vulnerability Scanner for C and C++ Code," *Proceedings of the 2000 Annual Computer Security Applications Conference*, December 2000.
[23] D. Wagner and D. Dean, "Intrusion Detection via Static Analysis," *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pp. 156-169, 2001.
[24] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A First Step towards Automated Detection of Buffer Overrun Vulnerabilities," *Network and Distributed System Security Symposium*, Feb. 2000.
[25] C. F. Webb, "Subroutine Call/Return Stack," *IBM Technical Disclosure Bulletin*, 30(11), April 1988.