A Processor Architecture Defense against Buffer Overflow Attacks

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

***Abstract*—Buffer overflow vulnerabilities in the memory stack continue to pose serious threats to network and computer security. By exploiting these vulnerabilities, a malicious party can strategically overwrite the return address of a procedure call, obtain control of a system, and subsequently launch more virulent attacks. Software countermeasures for such intrusions entail modifications to applications, compilers, and operating systems. Despite the availability of these defenses, many systems remain vulnerable to buffer overflow attacks.**

In this paper, we present a hardware-based solution that prevents buffer overflow attacks involving procedure return address corruption. We add a secure return address stack to the processor that provides built-in, dynamic protection against return address tampering without requiring any effort by users or application programmers. Also, the performance impact is negligible for most applications. Changes are not required of application source code, so both legacy and future software can enjoy the security benefits of this solution.

***Index Terms*—buffer overflow, processor architecture, computer security, network security, return address corruption**

I. Introduction

Buffer overflows have caused security problems since the early days of computing. In 1988, the Morris Worm caused Internet havoc using a buffer overflow vulnerability as one of its means of intrusion. The Code Red worm and its variants, which stung companies over the summer of 2001, exemplifies the severity of problems that buffer overflow vulnerabilities still cause today. Code Red spread by taking advantage of a buffer overflow problem in Microsoft IIS. The total economic cost of these worms is estimated at $2.6 billion by Computer Economics [16]. In addition, various intrusion tools that establish distributed denial of service (DDoS) networks often exploit buffer overflow vulnerabilities to compromise oblivious hosts [10].

The vast majority of buffer overflow exploits involve an attacker "smashing the stack" and changing the return address of a targeted function to point to exploit code. Thus, protecting return addresses from corruption prevents most attacks. Past work addresses the problem through 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, and the continuing emergence of such problems in the face existing software solutions.

We seek to implement a hardware-based, built-in, non-optional layer of protection against common buffer overflow vulnerabilities in all future systems. Some processors contain return address stacks to reduce performance penalties due to delayed branch resolution. We detail how a modified hardware return address stack (RAS) can be used to protect return addresses. Our method preserves a correct copy of the return address and also provides a means of detecting buffer overflows with high probability. Since the RAS is of finite size, various safe spill and fill methods are described and evaluated. Our proposal is not meant to be the only defense against such attacks, however. We recommend that our proposal be used in conjunction with existing software techniques to ultimately provide more pervasive protection.

In Section II, we detail the buffer overflow problem, and we compare past work in Section III. In Section IV, we describe our secure return address stack. We discuss hardware return address stacks in existing processors, and we describe the architectural and OS changes that are required to achieve our security goals. In Section V, we discuss methods for handling non-LIFO procedure control flow in our proposal. In Section VI, we analyze the performance impact of our proposal, and we conclude in Section VII.

II. Buffer Overflows and Stack Smashing

Despite existing countermeasures, buffer overflow vulnerabilities continue to plague computer systems and networks. Table I shows the percentages of CERT advisories from 1996 to 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 [18].

The majority of 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 nested fashion. Thus, a procedure call stack, which is a LIFO data structure, is used to save the state between procedure calls and returns. We describe stack behavior and

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.

buffer overflow attacks for the IA-32 architecture [11], but the general procedures apply to all conventional ISAs.

Figure 1 illustrates the operation of the memory stack for the example program shown in Figure 2. The memory stack consists of a set of stack frames; a single frame is allocated for each procedure (e.g., `g`) that has yet to return control to an ancestor procedure. The stack pointer (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 the currently executing procedure.

When function `f()` calls `g()`, a new stack frame is pushed onto the stack. The stack on the left of Figure 1 shows the state of the memory stack immediately following the call to `g()`. The new frame includes the input pointers `x` and `y`, the procedure return address, the frame pointer, and the local variables `a` and `b`. Upon completing `g()`, the program should return to the return 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 should also be restored to their former values, and the stack frame belonging to `g()` should be effectively popped from the stack.

A security vulnerability exists in the code shown in Figure 2 because `strcpy()` does not perform bounds checking. In the function `g()`, if the string to which `x` points exceeds the size of `b`, `strcpy()` will overwrite data located adjacent to `b` 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 `x` points to such a string, `strcpy()` will copy malicious code into the stack, and the return address in `g()`'s stack frame will be set to the initial instruction of the malicious exploit code. This is illustrated in Figure 1. Consequently, once `g()` completes, the program will jump to and execute the exploit code instead of returning control to `f()`. There are many variations of this form of attack [13], but the majority relies on the ability to modify the return address. 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.

III. Past Work

Researchers have proposed many software-based countermeasures for buffer overflow exploits. 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 stack in non-executable pages. This can prevent an attacker from executing code injected into the memory stack. However, the return address may instead be redirected to preexisting 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. For instance, Linux depends on executable stacks for signal handling.

StackGuard is a compiler-based solution involving a patch to `gcc` that defends against buffer overflow attacks that

TABLE I. 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 %**

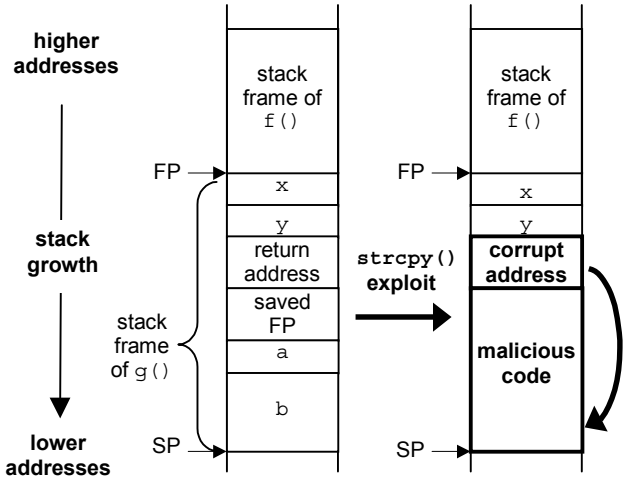

FIGURE 1. Buffer Overflow Attack

```
int f()
{
    …
    g(x, y);
    …
}

int g(char * x,
      char * y)
{
    int a;
    char b[64];
    …
    strcpy(b, x);
    …
    return;
}
```

FIGURE 2. Vulnerable Code Example

corrupt procedure return addresses [7]. 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 [8].

TABLE II. PRIOR WORK COMPARISON

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

[1]By "complete protection," we mean complete protection against buffer overflow addresses that directly corrupt procedure return addresses.
[2]Compiler changes may be required for certain programs depending on how non-LIFO procedure control flow is handled (see Section V).
[3]Depending on how non-LIFO procedure control flow is handled, some programs may experience a very small increase in code size (see Section V).

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 [20].

Researchers have also proposed using more secure (or safe) dialects of C and C++, for 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 the prevention of stack smashing attacks [9]. 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 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.

Transparent run-time defenses have also been proposed. The dynamically loaded libraries `libsafe` and `libverify` provide a run-time defense 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 in the copied function is replaced with a jump to code that verifies the return address before 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 to our solution in Table II. Our hardware-based solution enables built-in, transparent protection against common buffer overflow vulnerabilities without requiring user or application programmer effort. We observe that our proposal is the only solution that combines the features of support for legacy code, wide applicability to various platforms, low performance impact, a negligible increase in code size, and strong protection against procedure return address corruption.

IV. A SECURE RETURN ADDRESS STACK

We now describe low-cost enhancements to the core hardware and software of future programmable machines that enable the detection and prevention of return address corruption. More specifically, we modify the implementation of procedure call and return instructions, employ a special hardware return address stack, and present a secure method for swapping the contents of the hardware stack to and from memory. Since we do not require changes to programming languages or application source code, both legacy and future software applications can benefit from the security provided by these enhancements.

A. Hardware Return Address Stacks

The branch target of a procedure return instruction is often calculated using the contents of one or more registers and/or memory words. Therefore, the target address cannot be resolved until the return instruction has passed through several stages of the processor pipeline. Due to the LIFO nature of procedure calls, a simple stack structure that stores return addresses can facilitate highly accurate prediction of the return

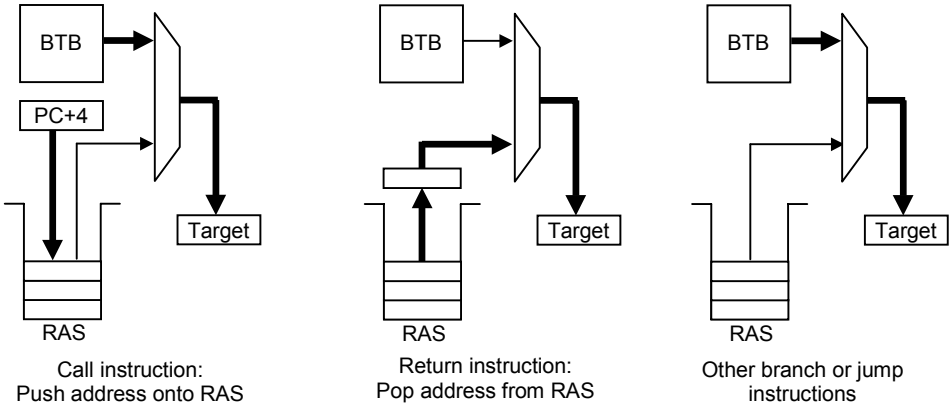

FIGURE 3. BRANCH TARGET PREDICTION WITH THE BTB AND THE RAS

instruction targets [12, 25]. Many processors, such as the Alpha 21164 [5] and the Alpha 21264 [6], employ such return address stacks (RAS) to improve performance by predicting procedure return addresses early in the pipeline.

Processor branch predictors can employ a hardware return address stack in conjunction with a branch target buffer (BTB) as illustrated in Figure 3. Upon executing a procedure call instruction, an entry from the BTB indexed by the program counter (PC) is used as the predicted target of the call instruction. The address of the following instruction (i.e., the return address) is pushed onto the RAS. During the execution of a return instruction, the topmost entry of the RAS is popped and used as the predicted target (instead of using an entry from the BTB). The RAS is unaffected during the target prediction of other branch and jump instructions. RAS structures are often implemented as circular buffers. When overflow of the RAS occurs, the least recently pushed address is overwritten with the value of the most recent return address. We henceforth refer to the hardware return address stack as the RAS, and we refer to the call stack for storing local variables and return addresses in memory as the memory stack.

Unfortunately, the RAS provides no protection against corruption of the return addresses in the memory stack. When a call instruction is executed, a valid return address is pushed on to the hardware RAS, and depending on the ISA, the return address may also be stored in the memory stack. Suppose the return address is subsequently corrupted by a buffer overflow in memory. At the moment when a valid return instruction is fetched, the corrupted address is located in a register or a memory location specified by the return instruction. Upon full execution of this instruction, the processor learns that the value popped from the hardware RAS does not equal the return address associated with the instruction. However, rather than jump to the correct return address popped from the hardware RAS, the processor starts executing instructions beginning at the corrupted return address. The instructions issued and executed speculatively based upon the correct return address from the RAS are nullified. The processor does not employ the uncorrupted address from the RAS because the RAS contents are treated as branch prediction "hints" that are not expected to always be correct.

B. *Architectural Modifications*

By using special protected hardware and memory structures, we can defend against return address corruption in the memory stack. We require a special hardware RAS, the Secure Return Address Stack (SRAS), which always provides correct, uncorrupted return addresses [15]. To properly employ the SRAS, we first modify the operation of procedure call and return instructions. We require that these call and return instructions are clearly recognizable. For instance, in a RISC ISA, 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 [14].

We maintain the ISA definitions and visible behavior of call and return instructions, but we alter the manner in which the processor executes these instructions. Upon executing a procedure call instruction, the processor always pushes the return address onto the top of SRAS. The program counter is set to the target of the call instruction. When a processor executes a return instruction, the return address popped from the top of the hardware SRAS – not the target specified by a register or memory value – is assigned to the processor's program counter. The processor then determines whether the return address from the memory stack equals the return address popped from the SRAS. If these addresses differ, corruption has occurred, and the processor can terminate the current process, inform the OS of the corruption by issuing a new invalid return address trap, or continue execution of the program based upon the correct address popped from the secure RAS.

C. *Swapping Contents of the SRAS*

The SRAS core consists of an *n*-address stack implemented as a circular buffer. We now rely on the SRAS to store an arbitrary number of return addresses, but the number of nested return addresses in a program may exceed the number of entries in the SRAS. Therefore, to avoid overwriting valid addresses, we must define an efficient method for the processor to securely swap the contents of the SRAS to memory when the SRAS becomes completely full. 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. We discuss

the typical stack depth sizes observed for common programs in Section VI.

Upon overflow, the processor will store the $n/2$ least recently pushed addresses to memory underflow, and the processor will retrieve up to $n/2$ most recently pushed addresses from memory in the event of underflow. The processor stores and retrieves $n/2$ addresses to and from memory rather than all n addresses to prevent an SRAS thrashing scenario. In some programs, a policy of transferring the entire contents of the SRAS could lead to frequent storage of all SRAS addresses to memory immediately followed by the retrieval of n SRAS addresses from memory. We investigate two different approaches to handling SRAS swapping: operating system-managed and processor-managed swapping.

OS-managed SRAS Swapping. In the first approach, the operating system assumes complete responsibility for swapping SRAS entries. The processor issues an OS interrupt in the events of SRAS overflow or underflow. The OS then executes code that transfers contents of the SRAS to or from memory; the application does not observe or 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 simply 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.

Processor-managed SRAS Swapping. In this scheme, we implement hardware enhancements to reduce the number of OS invocations associated with SRAS overflow and underflow. We store information in the processor concerning the physical memory locations of the OS-managed data structures that contain spilled SRAS addresses. Upon SRAS underflow or overflow, the processor can employ this information to directly transfer SRAS contents to and from physical memory rather than invoking the OS. To support this functionality, the processor maintains two pointers to two physical pages that store spilled SRAS contents for the active process. Also, the processor maintains a counter that indicates how much space is available in the two physical pages. Although the two pages may be virtually or physically separated in memory, the two pages are treated as being adjacent to form a single "superpage." When the superpage underflows or overflows, the OS is invoked to allocate a new physical page or deallocate one of the two physical pages.

The processor maintains two pointers to two physical pages rather than one pointer for one physical page in order to avoid a thrashing scenario in which a page is repeatedly allocated and immediately deallocated. In such a trashing situation, the OS could be invoked for every SRAS spill and fill, and performance would decay. By providing the processor with access to two physical frames at once, we can avoid calls to the OS caused by jumping back and forth over a page boundary. The processor logic required to manage the two pointers and the counter is based upon a simple 4-state machine. The two bits that represent the machine state are stored by the processor in the high order bits of the counter.

The OS is invoked much less often in this scheme than in the OS-managed swapping scheme. For example, if the SRAS consists of 64 8-byte return address entries and the page size is 8 KB, the OS is invoked only once after $8192/((64/2)\times 8) = 32$ consecutive SRAS overflows. In the OS-managed scheme, the OS would be invoked for each of those 32 SRAS overflows. In Section VI, we compare the performance impact of these two schemes on a set of benchmark programs.

We also note that since the values popped from the SRAS must always be valid to preserve correct execution, all of the SRAS contents and any associated configuration state bits must be transferred to and from memory on context switches.

D. Security Analysis

Our primary design goal is to prevent attacks in which hostile code is injected and executed on innocent hosts by exploiting buffer overflow vulnerabilities that corrupt procedure return addresses. The modifications described above accomplish this goal: we defined architecture based on a secure return address stack that detects and prevents any corruption of return addresses. In our system, only call and return instructions can modify the contents of the SRAS. Hence, the correct return addresses will be preserved in the event of a standard buffer overflow attack that corrupts the values of return addresses in the memory stack. If such corruption does occur in memory, the processor detects this and can respond appropriately.

Since the SRAS is finite in size, its contents must be securely swapped to and from memory upon overflow and underflow, respectively, to guarantee security. In both the OS-managed and processor-managed SRAS schemes, we protect spilled SRAS contents by storing the addresses in physical pages that are not accessible by the virtual memory spaces of user-level applications. Because non-kernel processes cannot access the contents spilled from their respective SRASes, no software bug or buffer overflow vulnerability in such processes can affect the spilled return addresses.

To provide truly pervasive and comprehensive protection against buffer overflow and related attacks, our solution should be implemented in conjunction with existing software defenses. Software-based security solutions encourage safe programming practices and identify a wide variety of security vulnerabilities in new code. Our proposal complements these solutions by offering specialized dynamic protection for legacy code and preventing potential attacks in new code that may be unrecognized by the software defenses. In addition, since we require changes to hardware, our proposal is meant to be a long-term solution. Software defenses, however, can and should be applied as they become available.

V. 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. However, the SRAS mechanism described so far does not accommodate non-LIFO procedure control flow. We can address this issue in at least four ways. These four options trade varying degrees of security and non-LIFO support for implementation cost and complexity.

The first two options enable zero or complete support for non-LIFO behavior while facilitating high or low security against procedure return address corruption, respectively. 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. Legacy executables that exhibit non-LIFO procedure calling behavior will terminate with an error (if not recompiled). The second option is to allow users to disable the SRAS with a new `sras_off` instruction. This enables the execution of potentially insecure code that exhibits any non-LIFO behavior as permitted in systems without an SRAS.

The third 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 to and from the SRAS without actually 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 fourth 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.

Regardless of the method(s) 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.

VI. Performance Impact

We now examine the performance impact of spilling and retrieving the contents of the SRAS to and from memory during program execution. The architectural enhancements that we propose enable security features for all programs, rather than just for network-based applications. Because of this, a performance study that examines the impact of our proposed architectural changes on all programs is more useful than one limited to network applications. Thus, we use SPEC2000 benchmarks to represent a "general-purpose" workload [21].

A. Simulation Methodology

To obtain performance data for the benchmarks, we use SimpleScalar, a cycle-accurate out-of-order superscalar processor simulator [3]. We consider the scenario in which the operating system is invoked each time a SRAS swap is required and the scenario where the processor primarily handles SRAS swapping. The OS-managed swapping scheme is easier to implement, but the processor-managed scheme can provide better performance. We simulate the execution of 500 million instructions of 12 SPEC2000 integer benchmarks after skipping at least 1 billion instructions in order to capture steady state behavior [17].

Our base processor model closely represents an Alpha 21264 processor [6]. We summarize the processor simulation parameters in Table III. The base processor includes a hardware return address stack that is implemented as a circular buffer. In some situations, speculative execution can pollute the RAS with invalid addresses. Hence, to maintain the integrity of the SRAS, the processor must include a perfect

repair mechanism to recover from SRAS corruption due to branch mispredictions [19]. Such mechanisms involve saving the top-of-stack (TOS) pointer and the return address to which the TOS points following the prediction of a branch instruction. If the processor discovers that a particular branch was mispredicted and the wrong program control path was followed, the processor can use information such as the saved TOS to restore the SRAS to its former, correct state (preceding the mispredicted branch).

We gather performance results for all 12 benchmarks, SRAS sizes of 16, 32, 64, 128, and infinite entries, and page sizes of 8 KB, 16 KB, and 32 KB. To model the SRAS swapping code in the OS-managed swapping scheme, we wrote a swapping and memory management routine in C. We also wrote C code that models the allocation and deallocation of physical pages during OS invocations in the processor-managed swapping scheme. We compile the SPEC2000 benchmarks and our OS swapping code on an Alpha machine with full optimizations to produce Alpha executables.

We simulate the execution of our SRAS swapping and page allocation routines assuming the caches are initially cold. We obtain cycle counts for all four OS routines: spilling the SRAS to memory in the OS-managed scheme, filling the SRAS from memory in the OS-managed scheme, allocating a new physical page in the processor-managed scheme, and deallocating a physical page in the processor-managed scheme. We find that each of these four routines requires between 23,000 and 25,000 cycles to complete. These cycle counts vary with the size of the SRAS.

In addition, we simulate the spilling and filling of SRAS contents directly to and from physical memory in the processor-managed SRAS swapping scheme by stalling the processor for a number of cycles. This stall time represents the number of cycles required to transfer $n/2$ SRAS entries directly to or from main memory. The bus from the processor to main memory in our model can transfer one 64-bit value every two cycles after an initial latency. In the event of underflow, the first load experiences the main memory latency of 100 cycles, and then the $(n/2 - 1)$ subsequent addresses arrive at the processor every other cycle for $(n - 2)$ cycles. Hence, the total stall time is $(98 + n)$ cycles. Similarly, in the event of overflow in this scheme, the processor stalls for $(98 + n)$ cycles to store $n/2$ addresses to main memory.

B. Performance Results

Figure 4 illustrates maximum and mean depths of the return address stack associated with the benchmarks. We observe that the return address stacks for the SPEC2000 benchmarks never exceed depths of 108 addresses. The mean depths of the return address stacks for the SPEC2000 benchmarks range from 4.72 to 29.94 addresses. Although Figure 4 provides an appropriate guide for choosing the SRAS size, the figure does not present precise information concerning the performance impact of the SRAS on the benchmarks. The SRAS only negatively impacts performance when its contents are swapped to or from memory. Hence, instead of hinging on maximum and mean stack depths, performance primarily

TABLE III. SIMPLESCALAR SIMULATION PARAMETERS

Parameter	Characteristics
Instruction window	64-entry RUU
Fetch/decode/issue width	4 instructions
Commit Width	8 instructions
Functional Units	4 integer ALUs, 1 integer mult. 4 FP ALUs, 1 FP multiplier
BTB	4K-entry, 2-way set associative
Branch Predictor	Hybrid: 4K 2-bit selector 4K 2-bit bimodal predictor 1K 2-bit local w/ 10-bit history
L1 data cache	64 KB 2-way set-associative 64 byte blocks, 2 cycle latency
L1 instruction cache	64 KB 2-way set-associative 64 byte blocks, 1 cycle latency
L2 unified cache	2 MB 4-way set associative 64 byte blocks, 12 cycle latency
Main memory	100 cycle latency
Load/store queue	64 entries
Data memory ports	2 ports
I-TLB and D-TLB	128-entry fully-associative

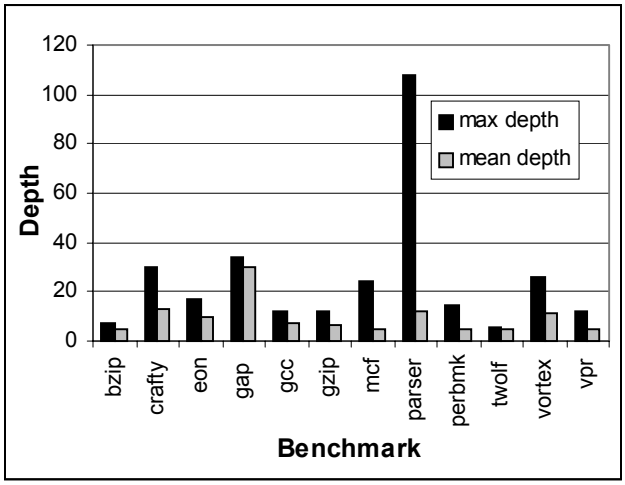

FIGURE 4. RETURN ADDRESS STACK DEPTHS

TABLE IV. PERFORMANCE IMPACT OF SRAS SWAPPING

Bench-mark	Percent Performance Degradation for Different SRAS Sizes							
	OS-managed				Processor-managed			
	16	32	64	128	16	32	64	128
bzip	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
crafty	**5.8**	0.1	0.0	0.0	0.0	0.0	0.0	0.0
eon	**67.9**	0.0	0.0	0.0	1.0	0.0	0.0	0.0
gap	**4.7**	**1.5**	0.0	0.0	0.0	0.0	0.0	0.0
gcc	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
gzip	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
mcf	**7.3**	0.0	0.0	0.0	0.0	0.0	0.0	0.0
parser	**64.2**	**29.3**	**9.2**	0.1	0.8	0.2	0.1	0.0
perlbmk	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
twolf	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
vortex	**36.7**	0.0	0.0	0.0	0.3	0.0	0.0	0.0
vpr	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

depends on the rates at which the memory stack (and thus the SRAS) grows and shrinks.

The performance penalties caused by SRAS swapping in the SPEC2000 benchmarks are presented in Table IV. These statistics represent percent performance degradation caused by an *n*-entry SRAS relative to the base machine model that includes an *n*-entry return address stack. The entries listed in bold indicate the situations in which the performance degradation exceeds 1%. For an SRAS size of 16 entries in the OS-managed scheme, 6 of the 12 SPEC2000 integer benchmarks experience performance reductions ranging from 4.7% to 67.9%. If the SRAS contains 64 entries, the performance degradation caused by OS-managed swapping is negligible (i.e., 1% or less) for all benchmarks except for `parser`. The `parser` benchmark is a syntactic parser of English in which the memory stack grows and shrinks quickly; thus, SRAS swapping penalties can be significant. When the SRAS contains 128 or more entries, the performance impact is negligible for all of the benchmarks.

In the processor-managed scheme, however, the performance degradation is less than or equal to 1% for all of the benchmarks when using a SRAS of size 16 entries or greater. We therefore conclude that the processor-managed SRAS swapping scheme is superior to the OS-managed SRAS swapping scheme. The processor-managed scheme achieves much higher performance than the OS-managed scheme at the cost of a small, incremental implementation effort.

VII. Conclusion

Malicious parties often exploit buffer overflow vulnerabilities to enable the insertion or execution of hostile code on 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, addressing such buffer overflow vulnerabilities has become a security priority. Although software-based countermeasures are available, a processor architecture defense is justified because of the fact that major security problems stemming from buffer overflows continue to plague networks and computer systems.

We described a secure hardware return address stack (SRAS) that detects and prevents corruption of procedure return addresses. The SRAS only requires minor changes to the compiler, the operating system, and the branch prediction structures found in many microprocessors, so legacy and future software can enjoy the security benefits without modifying application source code. We presented new approaches to managing and swapping the SRAS, and we presented new results that demonstrate the SRAS causes a negligible performance impact in most applications.

Our hardware-based solution should be applied in tandem with existing software countermeasures to provide truly robust protection against buffer overflow attacks. In future work, we will explore SRAS enhancements, and we will investigate alternative techniques for preventing buffer overflow attacks.

References

[1] A. Baratloo, N. Singh, and T. Tsai, "Transparent Run-time Defense Against Stack Smashing Attacks," *Proceedings of 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: Hardware Reference Manual*, December 1998.

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

[7] 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.

[8] M. Frantzen and M. Shuey, "StackGhost: Hardware Facilitated Stack Protection," *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[9] 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.

[10] K. J. Houle, G. M. Weaver, N. Long, and R. Thomas, "Trends in Denial of Service Attack Technology," CERT Coordination Center, October 2001.

[11] Intel Corporation, *The IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, Intel Corporation, 2001.

[12] D. R. Kaeli and P. G. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns," *Proc. of the International Symposium on Computer Architecture*, pp. 34-41, May 1991.

[13] klog, "The Frame Pointer Overwrite," *Phrack Magazine*, vol. 9, no. 55, Sept. 1999.

[14] R. B. Lee, "Precision Architecture," *IEEE Computer*, vol. 22, no. 1, pp. 78-91, January 1989.

[15] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi, "Enlisting Hardware Architecture to Thwart Malicious Code Injection," *Proceedings of the International Conference on Security in Pervasive Computing*, 2003.

[16] J. McCarthy, "Take Two Aspirin, and Patch That System – Now," *SecurityWatch*, August 31, 2001.

[17] S. Sair and M. Charney, "Memory Behavior of the SPEC2000 Benchmark Suite," IBM T. J. Watson Technical Report RC-21852, Oct. 2000.

[18] The SANS Institute, "The SANS/FBI Twenty Most Critical Internet Security Vulnerabilities," Available at http://www.sans.org/top20/, October 2002.

[19] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark, "Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms," *Proceedings of the 31st International Symposium on Microarchitecture*, Nov. 1998.

[20] SPARC International, Inc., *The SPARC Architecture Manual*, 1992.

[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," *Proc. of the 2000 Annual Computer Security Applications Conference (ACSAC 2000)*, Dec. 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.