July 2002

UILU-ENG-02-2205
CRHC-02-05

ARCHITECTURE SUPPORT FOR DEFENDING AGAINST BUFFER OVERFLOW ATTACKS

Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel and Ravishankar K. Iyer

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801
University of Illinois at Urbana-Champaign

SF 298 MASTER COPY KEEP THIS COPY FOR REPRODUCTION PURPOSES

REPORT DOCUMENTATION PAGE

Form Approved OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 2002	

4. TITLE AND SUBTITLE
Architecture Support for Defending Against Buffer Overflow Attacks

5. FUNDING NUMBERS

6. AUTHOR(S)
Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel and Ravishankar K. Iyer

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main St.
Urbana, IL 61801

8. PERFORMING ORGANIZATION REPORT NUMBER
UILU-ENG-02-2205
(CRHC-02-05)

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Buffer overflow attacks are the predominant threat to the secure operation of network and Internet-based computing. Stack smashing is a common mode of buffer overflow attack for hijacking system control. It is performed by overflowing a stack-allocated buffer in order to overwrite a return address with the address of a malicious piece of code. This paper evaluates two architecture-based techniques to defend systems against such attacks: (1) the split control and data stack, and (2) secure return address stack (SRAS). The split stack approach separates control and data stack to prevent the function return address from being overwritten. It can be both implemented with compiler support or with architectural support. We implemented the compiler-based approach by modifying gcc compiler. Results show that it is effective in protecting real applications with slight performance overhead (2% for ftp server). The architecture-based split stack approach can eliminate the performance overhead and requires changes in the instruction set. The SRAS approach, instead detects an attack when a return instruction is being retired and requires no ISA changes or recompilation. It has been implemented in the SimpleScalar processor simulator. Simulation results show that the max overhead is 0.02% with a SRAS size of 64 entries for SPECINT 2000 benchmarks.

14. SUBJECT TERMS	15. NUMBER IF PAGES
buffer overflow, stack smashing, secure return address stack, split control and data stack, security	18
	16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Architecture Support for Defending Against Buffer Overflow Attacks

Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel and Ravishankar K. Iyer
Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana, IL 61801
Phone: 217-244-6104, Fax: 217-244-5686
{junxu,kalbar,sjp,iyer}@crhc.uiuc.edu

Abstract

Buffer overflow attacks are the predominant threat to the secure operation of network and Internet-based computing. Stack smashing is a common mode of buffer overflow attack for hijacking system control. It is performed by overflowing a stack-allocated buffer in order to overwrite a return address with the address of a malicious piece of code. This paper evaluates two architecture-based techniques to defend systems against such attacks: (1) the split control and data stack, and (2) secure return address stack (SRAS). The split stack approach separates control and data stack to prevent the function return address from being overwritten. It can be both implemented with compiler support or with architectural support. We implemented the compiler-based approach by modifying gcc compiler. Results show that it is effective in protecting real applications with slight performance overhead (2% for ftp server). The architecture-based split stack approach can eliminate the performance overhead and requires changes in the instruction set. The SRAS approach, instead detects an attack when a return instruction is being retired and requires no ISA changes or recompilation. It has been implemented in the SimpleScalar processor simulator. Simulation results show that the max overhead is 0.02% with a SRAS size of 64 entries for SPECINT 2000 benchmarks.

1. Introduction

The explosive growth of the Internet has brought with it an increase in the of Internet systems being compromised by malicious attacks. The extent of attacks ranges from exhaustion of system resources to seizing of root

privileges and ultimately unrecoverable damage, be it corruption of data or loss of privacy or even classified information. Among all attacks, a substantial and growing portion of attacks exploit *buffer overflow* vulnerabilities. In 1988, the Morris Internet worm [10] infected tens of thousands of Internet hosts and fragmented much of the Internet by exploiting a buffer overflow vulnerability in `fingerd` facility on Unix systems. In recent years, buffer overflow exploitation are becoming increasingly popular among attackers. In the Summer of 2001, *Code Red Worm* [7] spread over the Internet exploited a buffer overflow vulnerability in the Microsoft IIS (Internet Information Server) indexing service DLL. The *Code Red Worm* not only defaced web pages and launched new attacks to other system, but allowed arbitrary code to be executed on the compromised host. More recently, in December of 2001, several security related bugs were discovered in Microsoft Windows XP's UPnP (Universal Plug and Play) service [9], one of which is a stack buffer overflow vulnerability that allowed an attacker to gain remote SYSTEM level access to any default installation of Windows XP (SYSTEM is the highest level of access within Windows XP).

To exemplify the significance and seriousness of the problem, we studied security alerts and updates provided by CERT/CC [6] [1]. Figure 1 shows the number of security alerts and those that were based on buffer overflow vulnerabilities from 1988 to date (statistics prior to 1999 from [23]). Attacks that exploit buffer overflow vulnerabilities have accounted for approximately half of all the alerts after 1997 (with the exception of 2000 in which distributed denial of service dominated) and this trend is still growing. Among the six alerts during the first two months of 2002, *five* of them exploit buffer overflows.

Buffer overflow attack exploits vulnerabilities in programs (most often unchecked buffer on the process run-time stack) to overwrite control information (i.e., function return address). By overflowing a stack-allocated buffer, the attacker can seize the control of the process and force it to execute arbitrary, malicious code. Although, various software solutions have been proposed to tackle the problem, buffer overflow attacks still dominates as shown in Figure 1. This is mainly due to the following reasons: (1) thousands of legacy applications are still being used and many of them are vulnerable to buffer overflow attacks; (2) many proposed software solution incur undesirable performance overheads and/or cannot protect from all attacks. As a result, users are reluctant to patch their software; (3) new software products, due to its inherent complexity and lack of thorough testing due to time-to-market pressures, can leave serious vulnerabilities concealed. A general solution targeting fundamentals of this problem is clearly desired.

[1] The CERT Coordination Center was established after the Morris Internet worm incident in 1988 and funded by DARPA to coordinate communication among experts during security emergencies and to help prevent future incidents.

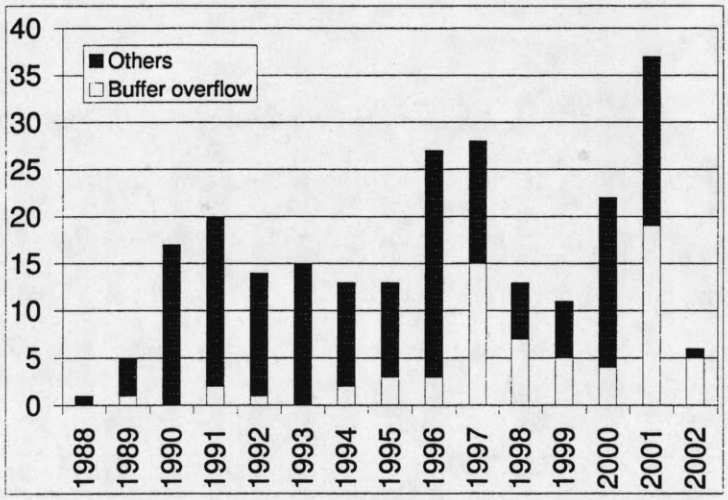


Figure 1. CERT Security Alerts by Years

This paper proposes two architectural solutions for run-time protection against buffer overflow attacks: (1) split control and data stack and (2) secure return address stack (SRAS). We brief describe the two approaches here.

Split Control and Data Stack: An important reason that an overrun buffer can result in control being seized is that current systems use a unified stack for both control information (i.e., return address) and data storage (buffers). We propose to split the unified stack into a control stack for return addresses and data stack for locally-allocated data items such as temporary buffers. This scheme can be implemented either in software or in hardware. We implemented the software approach by modifying the GNU C Compiler, *gcc* [12]. Results show it is effective in protecting real applications with slight performance overhead (2% for ftp server). By changing the semantics of function call/return instructions, hardware-based split stack implementation eliminates the overhead of the software-based implementation.

Secure Return Address Stack (SRAS) is hardware-based solution for detecting buffer overflow attacks. It uses the redundant copy of the return address maintained by the processor's fetch mechanism to validate return addresses and thereby detect malicious tampering. The operation of SRAS is similar to the Return Address Stack (RAS) implemented in most modern processors (e.g., Pentium and SPARC). Three different variants of SRAS, speculative SRAS, non-speculative SRAS, and non-speculative SRAS with overflow handling are evaluated using SimpleScalar simulator [5]. Performance evaluation shows that with SRAS of 64 entries, the performance degradation of the non-speculative SRAS with overflow handling is between 0% and 0.02%.

2. Buffer Overflow Exploit

A buffer overflow is the result of writing more data into a buffer than it can hold. This happens when a vulnerable program receives external input, and stores the input to a buffer without checking the boundary of it. In order for a buffer overflow attack to succeed, it needs to achieve the following two goals (1) Inject the attack code and (2) force the process to execute the injected code. If either goal fails, the attack fails. The most dominant form of buffer overflow exploitation is *stack smashing* attack. We explain how this kind of attack works using a synthetic example adapted from [1].

C Source Code

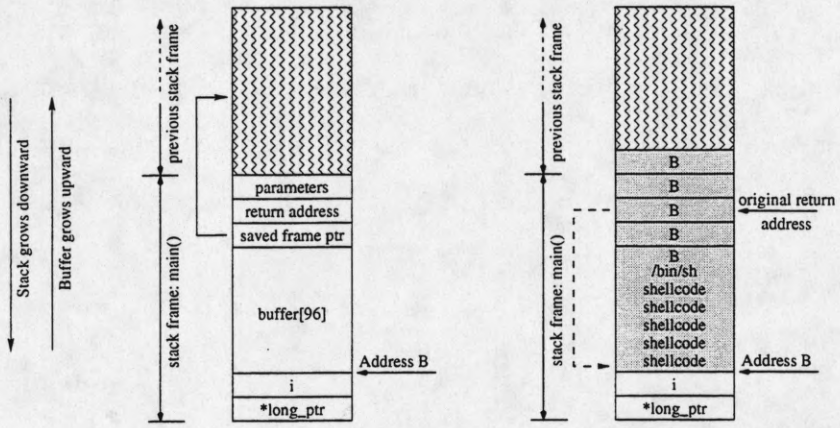
```
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
  "\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
  "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";
char large_string[128];
void main() {
  char buffer[96];
  int i;
  long *long_ptr;

  long_ptr=(long *)large_string;
  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;
  for (i=0; i<strlen(shellcode); i++)
    large_string[i] = shellcode[i];
  strcpy(buffer,large_string);
}
```

Disassembled Attack String `shellcode[]`

```
<shellcode>:      jmp  <shellcode+33>
<shellcode+2>:    pop  %esi
<shellcode+3>:    mov  %esi,0x8(%esi)
<shellcode+6>:    xor  %eax,%eax
<shellcode+8>:    mov  %al,0x7(%esi)
<shellcode+11>:   mov  %eax,0xc(%esi)
<shellcode+14>:   mov  $0xb,%al
<shellcode+16>:   mov  %esi,%ebx
<shellcode+18>:   lea  0x8(%esi),%ecx
<shellcode+21>:   lea  0xc(%esi),%edx
<shellcode+24>:   int  $0x80
<shellcode+26>:   xor  %ebx,%ebx
<shellcode+28>:   mov  %ebx,%eax
<shellcode+30>:   inc  %eax
<shellcode+31>:   int  $0x80
<shellcode+33>:   call <shellcode+2>
<shellcode+38>:   "/bin/sh"
```

Figure 2. Buffer Overflow Example Source Code (Adapted from [1])

Figure 2 shows the C source code of the simple synthetic example on the left and the disassembled malicious attack string (the content of the `shellcode` variable) on the right. The attack program first prepares the input in `large_string`, and then copies the content of it to the `buffer` variable on the stack. Note that `buffer` has only 96 bytes of space while `large_string` has 128 bytes. When copying `large_string` to `buffer`, function `strcpy()` blindly copies everything from the former to the latter withouth checking the latter's boundary, hence an overflow situation occurs. The content of `large_string` has been carefully crafted to both inject the malicious code and change the return address on the stack to the start of the malicious code. The stack layout before and after `strcpy()` is shown in Figure 3. The shaded area in Figure 3(b) shows the content of the overrun



(a) Stack Frames Before Buffer Overflow (b) Stack Frames After Buffer Overflow

Figure 3. Stack Layout for the Buffer Overflow Example

buffer after `strcpy()` which includes `buffer` and the location of orignal return address. The first part of the overrun buffer is filled using the code in `shellcode` and the second part is filled using the address of `buffer`, i.e., the starting address of the malicious shellcode on the stack. The function return address on the stack is overwritten by the address of `buffer` (*B* in Figure 3). When `main()` returns, it actually transfers control to address *B* and begin to execute the malicious code. The malicious code as shown on the right column of Figure 2 executes the `execv` system call to start a shell `/bin/sh`.

In a real security attack, the attack code normally comes from an environment variable, user input, or from a network connection. A successful attack on a privileged process such as a *set uid* program or a *daemon process* running as *root* would give the attacker an interactive *root shell.* Such an attack is often based on reverse-engineering the target program and is non-trivial to design. Using techniques from [1, 19, 20], however, an attack method can be relatively easily engineered. The danger is that, using the Internet, novice attackers can often download well-tested attack programs and launch attacks without deep knowledge of the inner workings of the target code.

3. The Split Control and Data Stack Approach

A fundamental reason that the stack buffer overflow is possible is that current systems use a unified stack for both control flow (function return addresses) and local data (temporary buffers, variables and function arguments) as shown Figure 4(a). When a function receives external data, and transfers that data to a stack-allocated buffer without checking its boundary, both the buffer and the return address (adjacent to it on the stack) can be possibly overwritten. We propose to split the unified stack into: (1) a *control stack* used to store function return addresses

only, and (2) a *data stack* stores temporary data and function arguments (Figure 4(b) and (c)). This section describes two ways of implementing the proposed solution, a software implementation by modifying an existing compiler and an architectural solution that changes the semantics of function call and return instructions. Since our implementation is in the context of Linux on Intel IA-32 architecture, we first briefly describe the function calling convention on IA-32.

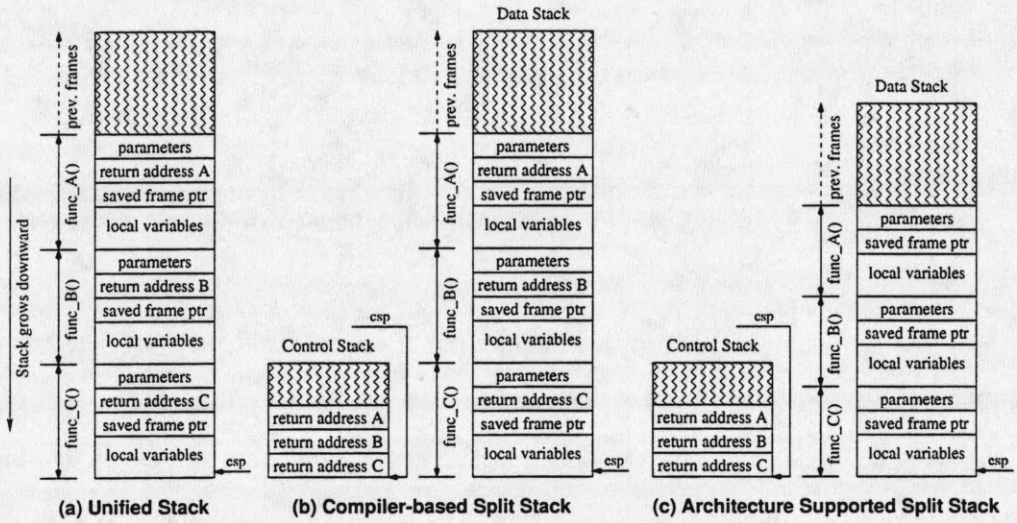


Figure 4. The Split Stack Scheme

3.1 Calling convention on Intel IA-32

On the Intel IA-32 architecture [15], function *call* and *return* are implemented using two instructions, `call` and `ret` respectively. When executing a `call` instruction, the processor pushes the function return address (the address of the instruction that immediately follows) onto the stack location pointed to by the *stack pointer register* `esp` and transfers control to the target function. When a `ret` instruction is executed, the processor pops off the current top of the stack pointed to by `esp` to the program counter and tranfers control to that address. When executing `ret`, the processor assumes that the top of the stack holds the correct return address. The compiler uses the same stack pointer register to grow and shrink the stack for buffer, variable, and function argument storage.

3.2 Compiler-based Split Stack Approach

The compiler-based split stack approach is illustrated in Figure 4(b). At the entry to each function, the return address saved on top of the *data stack* is also saved to on the top of the *control stack*. Before the function

returns, the top of the control stack is restored to the top of the data stack. As a result, the return instruction that immediately follows uses the saved return address from the control stack instead of the one that has been on the data stack through the lifetime of the function invocation. As Figure 4(b) shows, there are two copies of the return address, one on the data stack that can be tampered by buffer overflow, and another on the control stack that is immune from such tampering. Since a return instruction always uses the safe copy of the return address on the control stack, an overflow attack cannot seize the execution control of a process.

Our implementation of the compiler-based split stack required a changes the GNU C Compiler, *gcc* [12], on Linux for Intel IA-32. The compiler allocates space of the control stack and manages the control stack pointer variable `csp`. Saving of the return address to the control stack is implemented in the prologue for each function. Function prologue is responsible for setting up the stack frame, initializing the frame pointer register if used, saving registers, and allocating additional temporary storage. It is generated at the very beginning of each function. The added instruction copies the function return address from the top of the data stack pointed to by stack pointer register `esp` to the top of the control stack pointed to by the variable `csp` and increments `csp` by one word. Restoration of the return address from the control stack to the data stack is done in the epilogue of each function. Function epilogue code is responsible for restoring the saved registers and stack pointer to their original values, and returning control to the caller. The extra instructions restore the function return address at the top of the control stack (pointed to by `csp`) to current top of data stack (pointed to by stack pointer `esp`). The control stack pointer is then decremented by one word. When the `ret` instruction is executed, it uses the return address saved on the control stack.

Effectiveness of the compiler-based split stack approach. The effectiveness of this approach is evaluated using actual buffer overflow exploits provided by the LibSafe [4] team in their distribution package [3]. The distribution includes 5 synthetic exploits, which construct different malicious input, and one real attack, the *xlockmore* attack, which locks an X Window display. The *xlockmore* exploits a bug in the X window display lock program that usually run as *setuid* as *root*. Without any protection, these exploits all results in a interactive shell being forked due to the malicious code. After compiled using the modified version of the compiler, execution of these exploits resulted in the program terminating abnormally, preventing the intruder from compromising the system.

Performance of the compiler-based split stack approach. The performance overhead of the compiler-based split stack approach is evaluated by running the SPECINT 2000 benchmark programs [2] with the reference input set and

[2]The SPECINT 2000 benchmarks are a suite of integer applications used mainly to measure the performance of processor and memory architectures.

Benchmark	Base Time	Split Stack	Overhead(%)
164.gzip	355.7	384.6	8.14%
175.vpr	594.1	615.5	3.61%
176.gcc	335.9	347.8	3.52%
181.mcf	708.2	708.3	0.01%
186.crafty	407.6	466.2	14.38%
197.parser	487.8	515.6	5.69%
254.gap	241.9	278.7	15.20%
255.vortex	505.4	625.5	23.77%
256.bzip2	496.3	532.6	7.32%
300.twolf	1125.6	1154.8	2.59%
ftpd (1KB file)	0.117	0.123	5.43%
ftpd (1MB file)	0.127	0.130	2.38%

Table 1. Runtime Overhead for Compiler-based Split Control and Data Stack (All time in seconds)

the FTP server, wu-ftp-2.6.0 [26]. The results are shown in Table 1. The *Base Time* column provides execution time for the applications compiled using the original *gcc*. The figures in the *Split Stack* column are obtained while running the benchmark programs compiled by modified compiler. The overhead of the integer benchmarks ranges between 0.01% and 23%. Profiling runs of these programs showed that the variation in the overhead depends on the average number of instructions in each function called in the applications. Performance of the FTP serever is obtained by measuring the end to end delay from a FTP client. The client logs onto the server and transfers 1KB or 1MB file. The overhead for the FTP server with 1KB file transfer is about 5% and for 1MB file transfer is 2%. In the case of larger file transfer, more time is spent in I/O which hides the overhead due to the split stack implementation.

The measured overhead is mainly due to the extra memory accesses for saving and restoring of return addresses. Saving of an address includes two memory reads and two memory writes, i.e., read the address from data stack, write it to the control stack, read and write of control stack pointer `csp`. Similarly, restoring an address also requires four memory accesses. For each function invocation, eight extra memory accesses are required which might cause observable overhead. For server applications that are the primary targets for most attacks, the overhead shall be insignificant as in the case of FTP server.

Obviously, this overhead is due to the fact that the IA-32 has a very small register space. Creating an extra stack pointer creates additional register pressure. ISAs such as Alpha can possibility tolerate the reservation of an architectural register as the Control Stack Pointer. Architectures such as SPARC with register windows offer

additional protection because return addresses are kept in the register file and only propagate into memory on a window overflow.

In the next section, we show that with appropriate support from the processor, the overhead introduced by the compiler-based approach can be eliminated.

3.3 Split stack with IA-32 architectural support

The extra overhead introduced by the compiler-based approach can be eliminated by modifying the semantics of the function call and return instructions on IA-32. We propose to change the semantics of the `call` and `ret` instructions to eliminate the extra memory accesses due to saving and restoring of the return addresses. The new semantics of the two instructions are as follows: (1) for the `call` instruction, the processor pushes the function return address onto the control stack pointed to by the newly added control stack pointer register `csp` (instead of `esp`) and transfer control to the target function, and (2)for the `ret` instruction, the processor pops off the current top of the control stack (pointed to by the control stack pointer `csp`) to the program counter and tranfers control to that address. The extra register, `csp` is needed to manage the control stack and to eliminate extra memory accesses due to the read, write and adjust of the in memory varable `csp`. The semantics of instructions that manipulate the stack such as `push` and `pop` remain unchanged. In this new scheme, the function call and return instruction only manipulate the control stack and do not interfere with the operation of the normal data stack. In case of buffer overflow, the attacker can potentially overwrite the data stack, while those return addresses stored in the control stack are protected from being maliciously changed. The scheme is illustrated in Figure 4(c). The difference between the compiler-based approach shown in Figure 4(b) and the archecture-based approach is that the latter has only one copy of the return address is stored on the control stack and is hence more space efficient.

Employing the calling semantics, the run-time system (in particular, the program loader) needs to allocate control stack space for a process. The location of the control stack shall be far away from the data stack. Since buffers grow to higher address, the optimal control stack location is below the area for the data stack, so a buffer overflow can never reach the control stack.

3.4. Discussions

The advantage of the compiler-based implementation is that it can work on its own without any change in the processor. The disadvantage is that it requires recompilation in order to provide security benefit. There is possibly

performance overhead associated with any additional register pressure due to the allocation of one architectural register as the control stack pointer. Another source of potential overhead is due to the less efficient use of virtual memory space. The architectural approach does not incur performance overhead and is transparent to application programs, however, the biggest disadvantage is that it requires change in the instruction set.

Next, we discuss several system implemenation issues that are not addressed in our current implementation.

Control Stack Size. The amount of space required by the control stack depends on the number of nested function calls in an application. Table 2 shows the maximum function call depth for SPECINT 2000. Most applications have very limited call depth (less than 32). A control stack with one memory page (the granularity of kernel page allocation) shall suffice for almost all practical applications.

Benchmark	Maxium Call Depth
bzip2	11
crafty	28
eon	30
gap	362
gcc	32
gzip	14
mcf	41
parser	62
perlbmk	17
twolf	15
vortex	29

Table 2. Maxium Function Call Depth for SPEC-INT 2000

Multithreading and Longjmp. There are two system-level complications. First, for multi-threaded applications, each thread needs to have its own control stack. Thread creation APIs such as *pthread_create()* can be changed to automatically allocate the required space, transparent to application programs. Second, *setjmp* and *longjmp* are used in some applications for returning directly from multiple levels of nested functions calls. Adding an extra field in the *jmp_buf* structure [3] to save and later restore the control stack pointer solves the problem.

4. Secure Return Address Stack (SRAS)

In this section, we propose the Secure Return Address Stack (SRAS) In the split stack approach, return addresses are stored in the control stack in memory to prevent them from being changed. In contrast to the split stack

[3]The C structure *jmp_buf* is used by *setjmp* and *longjmp* to record and later to restore the current stack/frame pointer.

approach, SRAS does not try to prevent overwriting of return address stored on the stack, it instead detects an attack after the return address is maliciously changed but before the attack can have any negative impact.

The operation of the SRAS approach is similar to Return Address Stack (RAS) [17, 24, 22] implemented in modern processors. RAS is usually implemented at the instruction fetch stage of processor pipeline to maximize effective instruction fetch bandwidth. RAS can accurately predict the target address for a return instructions. When a function call instruction is fetched, its return address is pushed onto the RAS. When a return instruction is fetched, the top of the RAS is popped off and is used as the fetch address for the next instruction fetch cycle. RAS can usually achieve very high prediction accuracy (greater than 99%). Observe that the RAS contains a redundant hardware copy of the return address on the process's stack in memory and is immune from stack overflow. This redundant copy of the return address can be used to detect situations in whith the return address has been tampered with. In normal operations, RAS mispredictions are due to speculative update of the stack and overflows due to limited RAS size. A buffer overflow attack can also contribute to a RAS misprediction because it changes the correct return address on the stack.

Three alternative RAS extensions are proposed and evaluated for detecting buffer overflow attacks: (1) *Speculative SRAS* that extends the existing RAS and raises exceptions whenever mismatch/misprediction occurs; (2) *Non-speculative SRAS* that operates a new RAS at the commit stage of processor pipeline and eliminates RAS misprediction due to speculative RAS update; (3) *Non-speculative SRAS with overflow handling* that handles overflow in the *Non-speculative SRAS*. While hardware complexity and cost increase, the performance overhead is reduced going from alternative one to three. This section presents the three approaches and provides evaluation results from implementation in the SimpleScalar [5] processor simulator.

4.1 Speculative SRAS

The misprediction handling mechanism of the existing Return Address Stack (RAS) can be extended for buffer overflow attack detection. In the proposed speculative SRAS approach, any time a RAS misprediction occurs, an exception is raised so that the operating system can handle such a mismatch and determine whether the exception is due misprediction or buffer overflow. The exception handler can use the stack trace of the current process to make this decision or a table of all valid return points. If a buffer overflow has occured, the handler will not be able to trace back to previous stack frame. Such a handler will incur large overhead since it needs to go through a series memory indirection to trace the process stack. Currently, a penalty of 500 cycles is associated with each

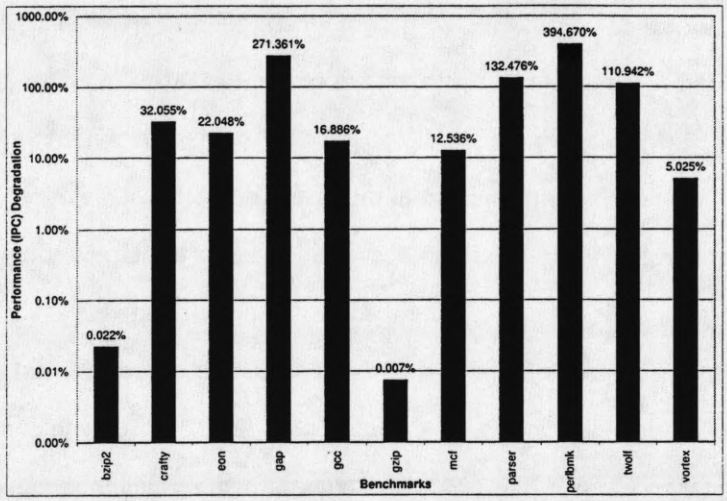


Figure 5. Performance Degradation of Speculative SRAS (SRAS Size=64)

misprediction exception. The performance overhead of this scheme is evaluted in the simulator. Figure 5 shows the simulation results for a RAS size of 64 entries (the Y-axis is in log scale). Except for *bzip2* and *gzip*, most of the applications experienced significant performance degradation, some of the them are even prohibitive (greater than 100%). The main reason for the high overhead is due to speculative update of the RAS at the fetch stage. The higher the RAS prediction rate, the lower the overhead. Currently, we used a RAS with *TOS-pointer only fixup* as defined in [22]. Although the observed overhead might be substantially lower if *pointer-and-data fixup* or *full RAS checkpointing* are used as shown in [22], the huge overhead in some of the benchmarks will remain high.

4.2 Non-speculative SRAS

To improve the performance of RAS-based detection, the speculative nature of the RAS at the fetch stage needs to be changed. A non-speculative SRAS is implemented in our simulator at the instruction commit stage, where we can more accurately establish the function call/return sequence without concern about the effect of speculative instruction. Incorrect mismatches (mismatches that are not due to an attack) in this scheme can occur due to RAS overflow—such mismatches trigger the exception code unnecessarily. Simulation results for this scheme are shown in Figure 6 (the Y-axis is on log scale). With a SRAS size of 64 entries, all but *gap* have no or less than 0.001% overhead. Benchmark *gap* still has around 4% overhead because its maximum call depth is larger than 64.

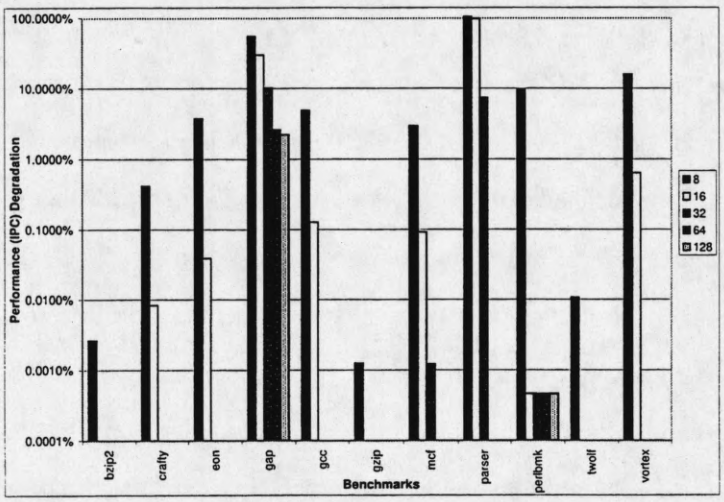


Figure 6. Performance Degradation of Non-speculative SRAS for Various SRAS Size

4.3 Non-speculative SRAS with Overflow Handling

We further improve the SRAS performance based on the *Non-speculative SRAS* from the previous subsection. Mismatches in the previous schemes are due to SRAS overflow. By handling the overflow situation more cleanly, a perfect SRAS can be achieved. Any mismatch with a perfect SRAS is a definitive signal of a buffer overflow attack.

A perfect SRAS is achieved by eliminating the two sources of false positives: (1) speculative update due to branch misprediction and (2) overflow due to too many levels of nested function calls. To eliminate polution of the SRAS due to speculative update, the SRAS is implemented in the commit stage of the processor pipeline instead of the fetch/decode stage — this is our non-speculative version of the SRAS. To eliminate SRAS overflows, we save part of its content to an corresponding data structure in memory allocated for each process. This is the same basic mechanism used to handle register window overflows on the SPARC architecture. The freed space in SRAS can be used for deeper levels of nested calls. Subsequently, when these nested function call return, SRAS might underflow. At that time, the content from the memory data structure are reloaded into the SRAS. Saving and reloading of the SRAS can be implemented either as an operating system exception handler or as an processor special unit much like the TLB (Translate Look-aside Buffer) miss handling by MMU (memory management unit). Finally, to preserve the content of SRAS across process context switch, its contents needs to be saved/restored at context switch time.

The SRAS technique has been implemented in the SimpleScalar processor simulator [5]. We present the perfor-

mance evaluation of the SRAS. Performance overhead of SRAS is mainly due to overflow and underflow handling. In the current simulation, any time SRAS overflows, half of its content is transferred to the in-memory data structure. A fixed number n, of penalty cycles is associated with each return address transfer. Let s be the size of SRAS, and p be the penalty associated with each SRAS overflow/underflow. then $p = n \cdot s/2$. In the simulation conducted, the fixed number n is set to be the memory acccess latency (the default value in the SimpleScalar simulator is 18 cycles). This formula of penalty calculation is perssmisitic in associating n cycles of with every tranfser of the $s/2$ return addresses from/to the SRAS, In practice, only part of them experience the n cycles of latency while the others can be tranferred in bulk mode. The formula is optimistic on the other hand in that the overflow/underflow handling requires some additional processing to locate the in-memory data structure and to adjust pointers. Taking these two factors, we believe that the formula above shall be a reasonable estimate of the overall penalty for an exception.

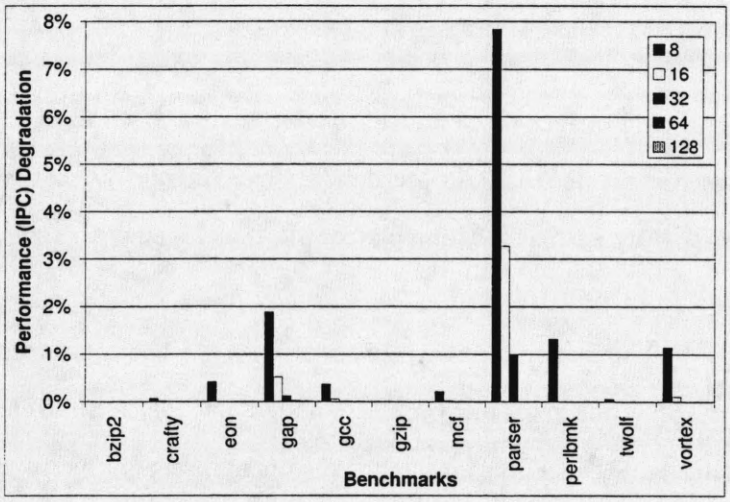


Figure 7. Performance Degradation of Non-speculative SRAS with Overflow Handling

Performance degradation of the SRAS is evaluated for SPECINT 2000 benchmarks. Figure 7 shows the performance (IPC, instructions per cycle) degradation due to the operation of the SRAS for various SRAS sizes. Most benchmark programs exhibit a certain amount of runtime overhead when SRAS size is much smaller than their maximum call depth. Except for *bzip2* and *gzip*, the benchmarks have measurable overhead for SRAS size of 8. Although, maximum call depth much larger than 8, *bzip2* and *gzip* only have 0.0003% and 0.0001% of overhead. This is because call depth fluctuaion for these two are very small. After extending SRAS size to 16 entries, only *gap*, *gcc*, *parser* and *vortex* exhibit measurable overhead, among which *gcc* and *vortex* have relatively small overhead (0.055% and 0.1% respectively). Only *gap* and *parser* exhibit overhead when SRAS is 32 since their

maximum call depth is 362 and 62. Although its call depth is much smaller, *parser* has much larger overhead than *gap* due to it larger call depth fluctuation. After SRAS reaches 64, only *gap* shows slight overhead (0.02%, 0.005% and 0.0006% for SRAS size of 64, 128 and 256 respectively). Its overhead disappears when SRAS has 512 entries. Overhead due to SRAS operation is a function of both maximum function call depth and the level of call depth fluctuation. A larger SRAS is needed for larger maximum call depth to eliminate the overhead. For a fixed SRAS size, the larger the call depth fluctuation, the larger its overhead. We believe that a SRAS size of 32 or 64 entries shall incur minimal performance overhead for most applications.

4.4 Discussions

An interesting problem was encountered while simulating the non-speculative SRAS with overflow handling. We were expecting return addresses to match perfectly with `ret` instructions. However, several benchmarks had non-zero mismatches. Careful tracing showed that the problem was caused by a *return folding* optimization used in the standard C library for the Alpha architecture. *Return folding* is done by the compiler when static analysis can determine that a `ret` instruction transfers control to a second `ret` instruction, and the return address of the second `ret` instruction can also be statically determined. In this case, the second `ret` instruction can be eliminated by directly transfering control from the first `ret` to the target of the second `ret` instruction. We implemented a quick fix for this problem in our mechanism. Any time a `ret` is committed and the top of the SRAS does not match the intended target, the second entry on the SRAS is tested, if it matches, both entries are popped off the SRAS.

As with the split stack approach, there are certain system implementation issues that affect the SRAS. Most noticably, for each context switch, the operating system kernel needs to save the content of the SRAS to the thread/process control block and to restore its content when execution is resumed. The overhead due to context switch is not measured in our current implementation. With some clever tricks, it is possible not to save or restore the entire stack. For example, with a SRAS size of 32, an application might only use 10 of the entries. It is clear that only the first 10 entries needs to be save and restored. We can keep improving this because the application might stays at the depth between 7 and 10 most of the time, in this case, only these four entries needs to be restored when resuming execution. A second issue is also with *setjmp* and *longjmp*. This can be solved by a special instruction to rewind the SRAS to an specified level.

5. Related Work

Since the Morris Internet Worm of 1988 [10], a significant amount of research effort has been dedicated to preventing and detecting buffer overflow attacks. Proposed solutions can be divided into two broad categories: (1) static program analysis for buffer overflow avoidance and (2) runtime buffer overflow detection. Our work falls into the runtime detection category. It is different from previous solutions in that it is based on architectural support for detection instead of pure software techniques. It does bear similarities to previous work. We discuss the related research in this section and compare our work when appropriate.

5.1 Static Program Analysis

Several commonly used tools, such as Lint [16], and those proposed in [11] use compile-time analysis to detect common programming errors. Existing compilers such as *gcc* have also been augmented to perform bounds-checking. Wagner et al. [23] used compile-time range analysis that formulate buffers as a pair of integer, the *allocated size* and *number of bytes currently in use*. The project specifically focuses on the set of unsafe library functions and check for each string buffer whether its inferred allocated size is at least as large as its inferred number of bytes currently in use. Larochelle et al. [18] proposed to use special comments (*annotations*) in program source code as a heuristic to infer and detect vulnerabilities in C programs. Both of the methods produce false positives false negatives.

5.2 Dynamic Runtime Detection

Compiler Extensions. This class of solution insert code for runtime detection at compile time. The instrumenation is done by changing existing compilers. StackGuard [8] instrument code at function entry and exit point to detect overflow. A *canary* is placed near the return address on the stack at function entry. The validity of this canary is checked when the function returns and the program is terminated if mismatch is detected. It bases on the assumption that tampering of the canary implies tampering of return address. Researchers from IBM [14] did some optimization based on the StackGuard scheme. With the exception of a few programs, the approaches has shown to be effective. Our compiler-based split stack implementation is similar to these two approaches in that it also changes the function epilogue and prologue code. It is different in that while StackGuard relies a canary around the return address for detection, our implemenation saves an extra safe copy of the return address and later restore. it. All three approaches incurs performance overhead and require access to source code. Our proposed

architecture-based solutions, however, do not have either of the two disadvantages.

Operating System Kernel Patches. By changing part of the O.S. kernel, it is possible to prevent or detect certain types of buffer overflow attacks. Most operating systems mark the stack region as executable for signal handling, trampoline and functional languages. The Non-excutable Stack Linux Patch [21] prevents execution of malicious code on the stack by making the stack non-executable. The attacker can still overflow the stack, but an O.S. exception is raised immediately before the malicious code is executed. This approach is transparent to application programs and offers zero performance overhead. The patch deals with trampolines and signal handler by detecting such usage and permanently or dynamiclly enable an executable stack. Both of these offer potential opportunities for vulnerabilities. Furthermore, the famous *return-into-libc* [25] attack completely defeats this scheme by overwriting the return address and transfers execution control to the heap.

C Library Patches. Since many buffer overflow vulnerabilities are caused by unsafe C library functions such as *gets* and *strcpy*, this class of solutions patch the standard C library for detections. Snarskii [2] implemented customized C library for FreeBSD. It targets unsafe functions, and inspects the process stack to detect overflows by checking if it write across frame pointers. Libsafe [4] from Bell Labs intercepts function calls to shared C Library and conducts the similar frame-pointer based boundary checking. Both implementation are transparent to user program. These approaches fail in two cases: (1) if a program chooses not to enable frame pointer at compile time (many programs do so for performance optimization) and (2) if the buffer overflow vulnerability is caused by applicaton internal function calls.

Runtime Sandbox. Janus [13] is a run-time sand-boxing environment that confines each application to a set of predefined operations. It relies on the operating system's debugging features such as strace, to observe and to confine a process to a sand-box. It works with existing binary but does not work with applications that legitimately need high privileges which are the cases for most server applications. Plus, it incurs very high overhead due to the use of debugging APIs.

References

[1] Aleph One. Smashing The Stack For Fun And Profit. *Phrack Magazine*, 49(7), Nov. 1996.

[2] Alexander Snarskii. Increasing Overall Security ... *ftp://ftp.lucky.net/pub/unix/local/libc-letter*, Feb. 1997.

[3] Avaya Labs Research. Libsafe: Protecting Critical Elements of Stacks. *http://www.research.avayalabs.com/project/libsafe/*, Feb. 2002.

[4] A. Baratloo, T. Tsai, and N. Singh. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings. of the USENIX Annual Technical Conference*, June 2000.

[5] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessor: the SimpleScalar Tool Set. Technical Report TR-1308, Dept. of Computer Science, University of Wisconsin-Madison, July 1996.

[6] CERT/CC. CERT Advisories. *http://www.cert.org/advisories/*.

[7] CERT/CC. CERT Advisory CA-2001-19 Code Red Worm Exploiting Buffer Overflow In IIS Indexing Service DLL. *http://www.cert.org/advisories/CA-2001-19.html*, July 2001.

[8] C. Cowan, C. Pu, D. Maier, H. Hinton, 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*, 1998.

[9] eEye Digital Security. UPNP - Multiple Remote Windows XP/ME/98 Vulnerabilities. *http://www.eeye.com/html/Research/Advisories/AD20011220.html*, Dec. 2001.

[10] M. W. Eichin and J. A. Rochlis. With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988. In *Proceedings. of IEEE Computer Society Symposium on Security and Privacy (SSP '89)*, pages 326–343, 1989.

[11] D. Evans. Static Detection of Dynamic Memory Errors. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1996.

[12] Free Software Foundation. The GNU C Compiler. *http://gcc.gnu.org/*.

[13] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th USENIX Security Conference*, 1996.

[14] IBM Research. Gcc extension for protecting applications from stack-smashing attacks. *http://www.trl.ibm.com/projects/security/ssp/*, June 2001.

[15] Intel Corporation. *Intel Architecture Software Developer's Manual, volume 2, Instruction Set Reference*, 1999.

[16] S. C. Johnson. Lint, a C Program Checker. *Bell Laboratories Computer Science Technical Report 65*, Dec. 1977.

[17] D. R. Kaeli and P. G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proc. 18th International Conference on Computer Architecture*, page 34, 1991.

[18] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proc. 10th USENIX Security Symposium*, Aug. 2001.

[19] Mudge. How to Write Buffer Overflows. *http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html*, 1995.

[20] Nathan P. Smith. Stack Smashing Vulnerabilities in the UNIX Operating System. *http://destroy.net/machines/security/nate-buffer.ps*, 1997.

[21] OpenWall Project. Linux Kernel Patch from the Openwall Project. *http://www.openwall.com/linux/*.

[22] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *International Symposium on Microarchitecture*, pages 259–271, 1998.

[23] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of 7th Network and Distributed System Security Symposium*, Feb. 2000.

[24] C. F. Webb. Subroutine call/return stack. *IBM Tech. Disc. Bulletin*, 30(11), Apr. 1988.

[25] R. Wojtczuk. Defeating Solar Designer Non-executable Stack Patch. *http://www.insecure.org/sploits/non-executable.stack.problems.html*, Jan. 1998.

[26] WU-FTPD Development Group. WU-FTPD. *http://www.wu-ftpd.org/*.