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
{junxu,kalbar,sjp,iyer}@crhc.uiuc.edu

Abstract

Buffer overflow attacks are the predominant threat to the secure operation of network and in particular, Internet-based applications. Stack smashing is a common mode of buffer overflow attack for hijacking system control. 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. This approach can be implemented with compiler support or with architectural support by modifying the semantics of call and return instructions. The compiler implementation shows slight performance overhead (e.g., 2% for ftp server), and the architectural support eliminates the overhead of the software solution. The SRAS is a hardware-based solution for detecting attacks. It uses the redundant copy of the return address maintained by the processor to validate return addresses and thereby detect malicious attacks. SRAS has been implemented in the SimpleScalar processor simulator. Simulation results show that the maximum 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 an increase in 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. Among all attacks, a substantial and growing portion of attacks exploit *buffer overflow* vulnerabilities. In 1988, the Morris Internet worm [10] exploited a buffer overflow vulnerability in `fingerd` 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. It 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 administrator access to any default installation of Windows XP.

Figure 1 shows the number of security alerts reported by CERT/CC [6][1] between 1988 and March, 2002 (statistics prior to 1999 from [18]). Attacks that exploit buffer overflow vulnerabilities have accounted for approximately half of all the alerts after 1997. Among the six alerts during the first two months of 2002, *five* of them exploit buffer overflows.

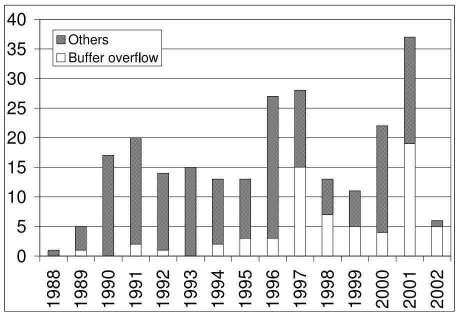


Figure 1. CERT Security Alerts by Years

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 soft-

[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.

ware solutions have been proposed to tackle the problem, buffer overflow attacks still dominate. 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 their inherent complexity and lack of thorough testing due to time-to-market pressure, can leave serious vulnerabilities concealed. 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).

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., function return addresses) 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 that the proposed approach is effective in protecting real applications with slight performance overhead (e.g., 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

Buffer overflow is the result of writing more data into a buffer than the buffer can hold. This happens when a vulnerable program receives external input, and stores the input to a buffer without checking the buffer's boundary. 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].

Figure 2 shows the C source code of the simple synthetic example. The attack program first prepares the input in `large_string`, and then copies the content of it to the `buffer` on the stack. Note that `buffer` has only 96 bytes of space while `large_string` has 128 bytes. Function `strcpy()` blindly copies everything from `large_string` to `buffer` without boundary checking, 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 Figure3(b) shows the content of the overrun buffer after `strcpy()` which includes `buffer` and the location of original 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 begins to execute the malicious code. The malicious code executes the `execv` system call to start a shell `/bin/sh`.

In a real security attack, the malicious 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. Using techniques from [1], an attack method can be relatively easily engineered.

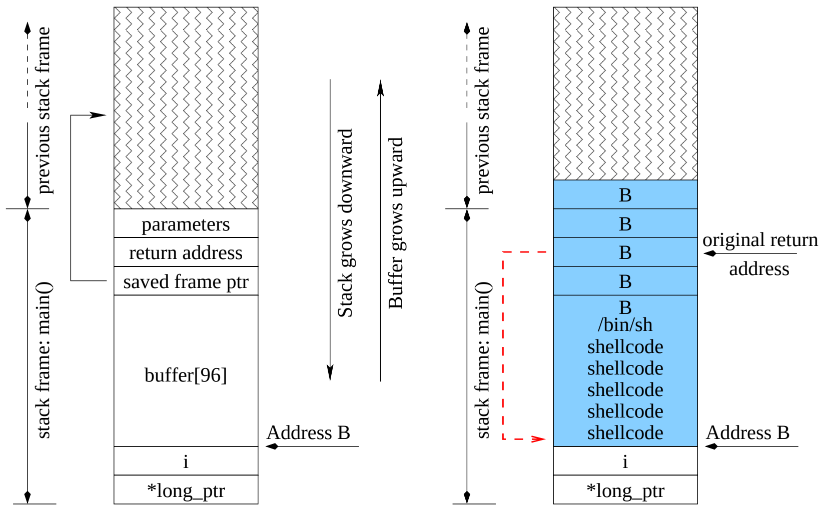
3. The Split Control and Data Stack Approach

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 the buffer's boundary, both the buffer and the return address (adjacent to it on the stack) can be 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* to store 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

```
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);
}
```

Figure 2. Example Source Code



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

Figure 3. Stack Layout for the Example

Linux on Intel IA-32 architecture, we first briefly describe the function calling convention on IA-32.

In the Intel IA-32 architecture [13], 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 transfers 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 space allocated for buffers, variables, and function arguments storage.

3.1 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 against 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 requires modifying the GNU C Compiler, *gcc* [12], on Linux platform. The compiler allocates space for the control stack and manages the control stack pointer variable `csp`. Saving the return address on 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. An 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 [3] team

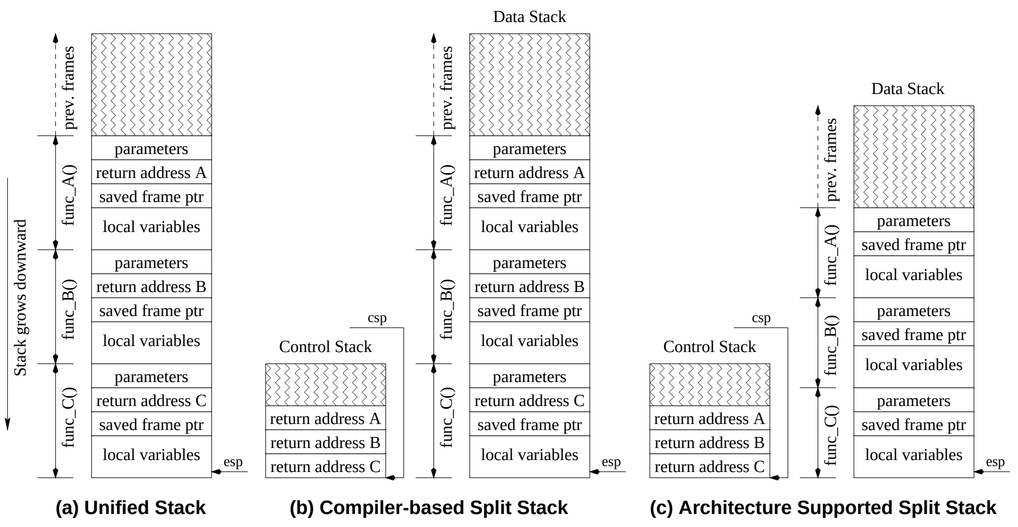


Figure 4. The Split Stack Scheme

in their distribution package [2]. 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 runs *setuid* as *root*. Without any protection, these exploits result 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.

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)	0.117	0.123	5.43%
ftpd (1MB)	0.127	0.130	2.38%

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

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 the FTP server, wu-ftp-2.6.0 [20]. 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 using the modified compiler. The overhead of the integer benchmarks ranges between 0.01% and 23%. Performance of the FTP serever is obtained by measuring the end to end delay as seen by 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 on I/O which hides the overhead due to the split stack implementation.

The measured overhead is 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.

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

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

3.2 Split stack with 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 explore changing 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 transfers 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 and write operations for adjusting the memory variable `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 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 architecture-based approach is that the latter preserves only a single copy of the return address stored on the control stack and hence is more space efficient.

Employing the proposed 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.

3.3. Discussion

The advantage of the compiler-based approach is that it does not require changes in the processor, the disadvantage is the introduced runtime overhead. The architectural approach does not incur performance overhead and is transparent to application programs, however, it requires changing the instruction set semantic and adding a new register into the processor. Next, we discuss several system implementation issues that are not addressed in our current solution.

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	Maximum 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. Maximum Function Call Depth

Handling Multi-threading and Longjmp. 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 and transparently allocate the required space for the application programs. *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.

It should be noted that StackShield [4] implements the similar idea as our split stack approach. By changing the assembler, StackShield provides similar support for stack buffer protection. We distinguish our work with StackShield in the following respects: (1) our implementation changes the compiler while StackShield changes the assembler; directly modifying the compiler, the optimizer can more efficiently optimize the application code (including the code for the split stack operation); (2) the proposed architectural support for implementing the split stack eliminates the overhead of software solution overhead and does not require access to the source code.

4. Secure Return Address Stack (SRAS)

In this section, we propose the Secure Return Address Stack (SRAS). In the split stack approach discussed in the previous section, return addresses are stored in the control stack in memory to prevent them from being changed. In contrast to the split stack approach, SRAS does not try to

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

prevent overwriting a 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] 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, the return address is pushed onto the RAS. When the 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%) [17]. 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 when the return address has been tampered with. In normal operations, RAS mis-predictions 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 mis-prediction 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/mis-prediction occurs; (2) *Non-speculative SRAS* that operates as new RAS at the commit stage of processor pipeline and eliminates RAS mis-prediction 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 and Non-speculative SRAS

The mis-prediction 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 mis-prediction occurs, an exception is raised so that the operating system can handle such a mismatch and determine whether the exception is due to mis-prediction or buffer overflow. The exception handler can use a table of all valid return points or the stack trace of the current process to make this decision. If a buffer overflow has occurred, 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 indirect memory references to trace the process stack. Currently, a penalty of 500 cycles[4] is associated with each mis-prediction exception. The performance overhead of this scheme is evaluated in the simulator. For RAS size of 64, except for *bzip2* and *gzip*, most of the applications experienced significant performance degradation, exceeding 100% for some applications. The main reason for the high overhead is the speculative update of the RAS at the fetch stage. The higher the RAS prediction rate, the lower the overhead.

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 speculatively executed instruction. Incorrect mismatches (mismatches that are not due to an attack) in this scheme can occur due to RAS overflow—such mismatches unnecessarily trigger the exception code. Simulation results with a SRAS size of 64 entries show that 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.

4.2 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. A perfect SRAS can be achieved by handling the overflow situation. 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 mis-prediction and (2) overflow due to too many levels of nested function calls. To eliminate pollution 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. 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

[4]Let's assume the average call depth of a process is 25 levels, which is a pessimistic value for most programs as can be seen from Table 2. Tracing of each level requires read of the saved frame pointer for that level. The memory access latency in our simulator is 18 cycles, thus the overhead is about 500 cycles (25*18=450)

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.

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 access latency (the default value in the SimpleScalar simulator is 18 cycles). This formula of penalty calculation is pessimistic in associating n cycles of with every transfer 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 transferred in the 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.

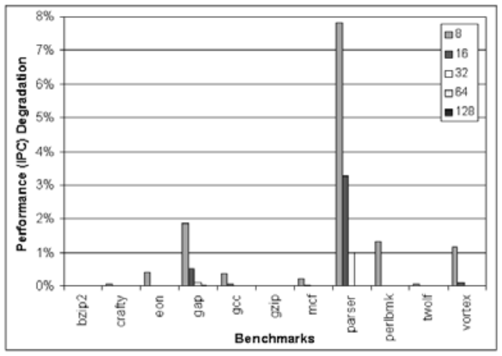


Figure 5. Performance Degradation of Non-speculative SRAS with Overflow Handling

Performance degradation of the SRAS is evaluated for SPECINT 2000 benchmarks. Figure 5 shows the performance (IPC, instructions per cycle) degradation due to the operation of the SRAS for various SRAS sizes. Most benchmark programs exhibit a small amount of runtime overhead when SRAS size is much smaller than their maximum call depth. Only *gap* and *parser* exhibit overhead when SRAS is 32 since their maximum call depth is 362 and 62. 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. We believe that a SRAS size of 32 or 64 entries shall incur minimal performance overhead for most applications.

4.3 Discussion

As with the split stack approach, there are certain system implementation issues that affect the SRAS. Most noticeably, 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. We discuss the related research in this section and compare our work when appropriate.

Static Analysis Several commonly used tools, such as Lint [14], 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. [18] 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 checks 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. [15] proposed to use special comments (*annotations*) in program source code as a heuristic to infer and detect vulnerabilities in C programs. All these methods produce undesirable false positives and false negatives and require access to source code.

Compiler Extensions for Runtime Detection. This class of solutions inserts code for runtime detection at compile time. The instrumentation is done by changing existing compilers. StackGuard [8] place a *canary* (a random number) on the stack at function entry and check it when function returns to detect overflow. It bases on the assumption that tampering of the canary implies tampering of return address. StackShield [4] modifies the assembler to implement the similar idea as our split stack approach. We have discussed the difference between our work and StackShield in Section 3.3.

Operating System Kernel Patches. By changing part of the operating system kernel, it is possible to prevent or detect certain types of buffer overflow attacks. The Non-executable Stack Linux Patch [16] prevents execution of malicious code on the stack by making the stack non-executable. This approach is transparent to application programs and offers zero performance overhead. However, the *return-into-libc* [19] attack completely defeats this scheme by overwriting the return address and transfers execution control to the heap or shared libraries.

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. Libsafe [3] intercepts function calls to shared C Library and conducts frame-pointer based boundary checking. These approaches can 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 application internal function calls.

6 Conclusion

This paper proposes and evaluates two approaches (the split stack and SRAS) for detecting stack buffer overflow attacks. The split stack approach separates control and data stack to prevent the function return address from being overwritten. It is implemented and evaluated by changing the GNU C compiler gcc. The compiler implementation showed slight performance overhead (e.g., 2% for ftp server). We also showed that the technique can be implemented with architectural supported that incurs no overhead and does not require source code access. The SRAS approach uses a redundant copy of return addresses inside the processor and detects an attack when a return instruction is being retired. Simulation results show that a maximum overhead of 0.02% with a SRAS size of 64 entries. Future work will address design implications of integrating the proposed solutions with actual processors. We also plan to further enhance the proposed techniques to handle a larger class of attacks, such as heap buffer overflow.

References

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

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

[3] 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.

[4] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack Magazine*, 56(5), May 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] Intel Corporation. *Intel Architecture Software Developer's Manual, volume 2, Instruction Set Reference*, 1999.

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

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

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

[17] 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.

[18] 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.

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

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