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

H. Özdoğanoğlu[*], T. N. Vijaykumar[‡], C. E. Brodley[†],
B. A. Kuperman[§] and A. Jalote[¶]

[*]hilmi@alumni.purdue.edu, [‡]vijay@purdue.edu, [¶]jalote@purdue.edu
School of Electrical and Computer Engineering
Purdue University, West Lafayette, Indiana 47906–1285

[†]brodley@cs.tufts.edu,
Department of Computer Science
Tufts University, Medford, MA 02155

[§]kuperman@cs.swarthmore.edu
Department of Computer Science
Swarthmore College, Swarthmore, PA 19081

Abstract

A buffer overflow attack is perhaps the most common attack used to compromise the security of a host. This attack can be used to change the function return address and redirect execution to the attacker's code. We present a hardware-based solution, called SmashGuard, to protect against all known forms of attack on the function return addresses stored on the program stack. With each function call instruction, the current return address is pushed onto a hardware stack. A return instruction compares its address to the return address from the top of the hardware stack. An exception is raised to signal the mismatch. Because the stack operations and checks are done in hardware in parallel with the usual execution of instructions, our best-performing implementation scheme has virtually no [1] performance overhead. While previous software-based approaches' average performance degradation for the SPEC2000 benchmarks is only 2.8%, their worst-case degradation is up to 8.3%. Apart from the lack of robustness in performance, the software approaches' key disadvantages are less security coverage and the need for recompilation of applications. SmashGuard, on the other hand, is secure and does not require recompilation of applications.

Index Terms

Buffer overflow, function return address, hardware stack.

[1] Beacause we are modifying hardware, it is impossible to guarantee zero overhead without an actual hardware implementation.

I. Introduction

COMPUTER security is critical in this increasingly networked world. Attacks continue to pose a serious threat to the effective use of computers and often disrupt commercial services worldwide, resulting in embarrassment and significant loss of revenue. While techniques for protecting against the malicious attacks have been confined primarily to the domain of software, the increasing demand for computer security presents a new opportunity for hardware research. Recent examples are buffer overflow protection features employed in processors by AMD [1], Intel [2] and Transmeta Corporation [3]. These features can be activated by Windows XP's Data Execution Protection mechanism [4] to block any attempt to execute code from memory reserved for data only, i.e., the stack and the heap. These efforts, although not complete solutions to buffer overflows, are indications of the severity of the attacks and the inclination towards hardware based methods to find a solution.

In this paper, we describe such an opportunity. We propose microarchitectural support for automatic detection/prevention of what is perhaps the most prevalent vulnerability today: attacks on the function return address pointer. The most common example of an attack on the function return address pointer is a buffer overflow attack [5]. The Code Red [6] and Code Red II [7] worms of 2001, W32/Blaster [8] and W32/Nachi-A [9] worms of 2003 and Sasser [10] of 2004 all exploited such a vulnerability in Microsoft's IIS [11], Windows RPC [12] implementation and Local Security Authority Subsystem Service (LSASS) [10], respectively, to propagate themselves across the Internet. Although it is fairly simple to fix an individual instance of a buffer overflow vulnerability, it continues to remain one of the most popular methods by which attackers compromise a host. In 2002, buffer overflow vulnerabilities were in 10 of the 31 advisories published by CERT [13] and in 5 of the top 20 vulnerabilities compiled by the SANS Institute [14]. In 2003, 17 out of 28 advisories published by CERT [15] and 13 out of the top 20 vulnerabilities compiled by the SANS Institute [16], have been on buffer overflow vulnerabilities.

Buffer overflow attacks overwrite data on the stack and can be used to redirect execution by changing the value stored on the process stack for the return address of a function call. We propose SmashGuard, a hardware-based approach to detecting such attacks, in which we add a small hardware stack to the pipeline. With each function call instruction, the return address and the current stack

TABLE I

APPLICATIONS DETECTED VULNERABLE TO BUFFER OVERFLOW ATTACKS

Linux/UNIX	Bind, RPCs, Apache, SendMail, SNMP, SSH, NIS/NSF, OpenSSL
Windows	IIS, MS SQL, IExplorer, RPCs, MSDAC

frame pointer[2] are pushed onto the hardware stack. A return instruction compares its return address against the address from the top of the hardware stack. A mismatch indicates an attack, and raises an exception.

A. Contributions

SmashGuard is a novel hardware-based security technology which provides a combination of advantages that none of the software methods can provide alone. The advantages SmashGuard provides are robust performance, broad security coverage, application transparency and low implementation cost. We discuss each of these benefits in turn.

Robust Performance: Because the stack operations and checks are done within CPU instructions, and in parallel with the usual execution of call and return instructions, the best-performing SmashGuard implementation scheme incurs virtually no performance overhead.

Security Coverage: Many software solutions do not protect against all forms of attack on the return address pointer. For instance, they may fail to protect against attacks that overwrite the return address indirectly. In contrast, SmashGuard protects against all forms of attack on the return address pointer.

Transparency: Many software solutions' key disadvantage is the need for recompilation of the source code to protect the program. SmashGuard, on the other hand, is a hardware modification with a kernel patch that supports the hardware technology, and therefore protects all applications.

Low Implementation Cost: Finally, the cost of our solution is a modest 2-KB storage for the hardware stack and a 2-KB storage for an internal table used by our best-performing implementation scheme. Addition of storage buffers and modifications to the microarchitecture are basic steps of designing a new processor version. In addition, since the stack is accessed

[2] We explain why we store the Stack Pointer to properly handle `setjmp()/longjmp()` in Section IV-B.

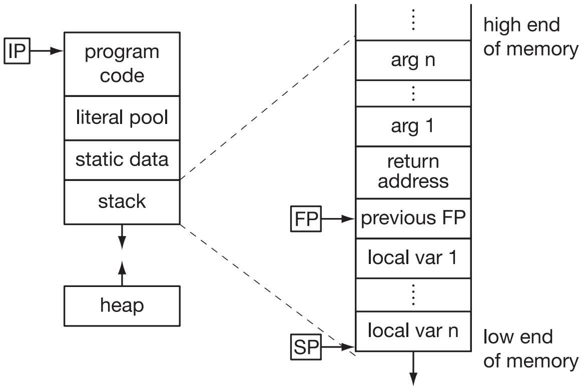


Fig. 1. Process Memory Organization and the Stack Layout

at instruction commit, which is not on the execution path of instructions, the critical path of the pipeline is not affected.

B. Paper Organization

In Section II we describe the vulnerability of the function return address and the different ways in which an attacker can exploit a vulnerability. In Section III we summarize related work to point out their strengths and weaknesses, both in terms of performance and functionality. Then in Section IV we describe our proposed hardware solution in detail. In Section V we present performance results. Finally, in Section VI we provide our conclusions and outline future extensions to our work.

II. Anatomy of an Attack

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

A. The Stack

Before describing the vulnerabilities and the attacks affecting the function return address, we first briefly review the memory organization of a process. On the left hand side of Figure 1, we show the five logical areas of memory used by a process. The text-only portion contains the program instructions, literal pool, and static data. The stack is used to implement functions and procedures, and the heap is used for memory that is dynamically allocated by the process during run time. During the *function prologue*, the function arguments are pushed on to the stack in reverse order, and then the return address is pushed onto the stack [3]. The return address holds the address of the

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

instruction immediately following the function call and is an address in the program code section of the process' memory space. The prologue finishes by pushing on the previous frame pointer followed by the local variables of the function. The function arguments, return address, previous frame pointer, and local variables comprise a *stack frame*. Because functions can be nested, the previous frame pointer provides a handy mechanism for quickly deallocating space on the stack when the function exits. During the *function epilogue*, the return address is read off of the stack and the stack frame is deallocated by moving the stack pointer to the previous stack frame.

B. Vulnerability of the Function Return Address

The return address in a stack frame points to the next instruction to execute after the current function returns (finishes). This introduces a vulnerability that allows an attacker to cause a program to execute arbitrary code. An attacker can overwrite the function return address with one of the exploit techniques explained in Section II-C.1 to redirect execution to the attacker's code. When the function exits, the program execution will continue from the location pointed to by the stored return address. On successful modification of the return address, the attacker can execute commands *with the same level of privilege* as that of the attacked program. If the compromised program is running as root, then the attacker can use the injected code to spawn a root shell, and take control of the machine. Recent exploits fall into the category of worms [6], [8], [10].

C. Exploiting the Vulnerability

There are several methods for overwriting the function return address and two targets to redirect execution. In this section, we describe different vulnerabilities that allow an attacker to overwrite the return address on the stack, possible targets to which to redirect execution, and how the attacker can inject the crafted exploit into the vulnerable code.

1) Overwriting the Return Address on the stack: **Buffer overflow attacks** are the undesirable side-effects of unbounded string copy functions. The most common examples from the C programming language are `strcpy()` and `gets()` which copy each character from a source buffer to a destination buffer until a null or newline character is reached, respectively. The vulnerability arises because neither checks whether the destination buffer is large enough to contain the source buffer's contents. If the destination buffer is a local variable (and therefore stored on the stack frame), then an attacker can exploit this vulnerability to overflow the buffer and overwrite a pointer on the stack or the return address. Note that for most architectures (e.g., x86, SPARC, MIPS) the stack grows down from high to low addresses, whereas a string copy on the stack moves up from low to high addresses. It is trivial to overflow a buffer to overwrite the return address because it is located above the local variables in that particular stack frame. There are two types of buffer overflow attacks to overwrite the function return address:

Type 1: A local buffer (character array) is filled in excess of its bounds (overflowed) to overwrite the return address on the stack, which is adjacent[4] to the local buffer; or

Type 2: A local buffer is overflowed to overwrite an adjacent pointer variable with a pointer to the return address on the stack. Then the return address is overwritten by an assignment to the pointer.

Format string attacks are relatively new and are thought to have first appeared in mid 2000 [17]. We provide a brief overview here, but for details the reader is referred to [17], [18]. Similar to a buffer overflow attack, format string attacks modify the return address in order to redirect the flow of control to execute the attacker's code. In the C programming language, format strings allow the programmer to format inputs and outputs to a program using conversion specifications. For example, in the statement `printf("%s is %d years old.",name,age)`, the string in quotes is the *format string*, `%s and %d` are conversion specifications, and `name` and `age` are the specification arguments. When `printf()` is called, a stack frame is created and the specification arguments are pushed on the stack along with a pointer to the format string. When the function executes, the conversion specifiers will be replaced by the arguments on the stack. The vulnerability arises because programmers write statements like `printf(string)` instead of the proper form: `printf("%s",string)`. The statements behave identically, unless `string` contains conversion specifiers. In that case, for each conversion specifier, `printf()` will pop an argument from the stack. For example, consider the following:

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

If the user calls `foo1()` with an argument string `"%08x.%08x"`, the function will pop two words from the stack and display them in hex format with a dot (.) in between. Using this technique, the attacker can dump the contents of the entire stack. The key to this attack is the `"%n"` conversion

[4] Note that the frame pointer is stored between the local variables and the return address on the stack.

specifier, which pops four bytes off the stack and writes the number of characters in the format string before `"%n"` to the address pointed to by the popped four bytes. An attacker can craft a format string with length (in bytes) equal to the address of the exploit code, with the last four bytes (a 32-bit address) identical to the address of the function return address on the stack followed by a final `"%n"`. When the format string is decoded by a `printf()`, [5] the number of bytes written thus far (this number is the address of the shellcode) will be written to the address popped off the stack which will be the address of the function return address. Note that length specifiers allow creation of arbitrarily long format strings without needing the string itself to be of equivalent length.

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

Integer Overflows: We find it valuable to mention integer overflows in our discussion of attacks on the return address because, even though they do not directly overwrite the function return address, they lead to other attacks (which are generally buffer overflows). The behavior of an integer overflow

```
int foo1(char *str, char *str2,
         unsigned int size, unsigned int size2) {
  char local[256];
  if((size + size2) > 256) { /*[a]*/
    return (-1);
  }
  strncpy(local, str, size); /*[b]*/
  strncpy(local + size, str2, size2);
  ...
  return (0);
}
```

Fig. 2. Integer Overflow Example

is undefined in ISO C99 standards, and most compilers ignore them. This becomes dangerous when the integer that is overflowed is used to calculate the size of a buffer or the index into an array. Unsigned integers do not overflow but wrap around to 0. The example in Figure 2 demonstrates a possible integer overflow attack that leads to a buffer overflow attack. An attacker can bypass the validation check at `[a]` and overwrite past the end of the `local` buffer with two large unsigned

[5] Format string attacks are possible with various `printf()` family of functions.

numbers in `size` and `size2` that result in a number smaller than 256 when added together. For a more detailed explanation, the reader is directed to [19].

2) Where to redirect control:: After an attacker overflows a buffer to overwrite the return address, there are two ways to redirect execution to compromise a host:

Shellcode: The most well-known method to redirect execution is to overwrite the return address with an address that points to a location in memory at which the attacker has placed an *exploit code*. Exploit code is a hexadecimal representation of machine instructions which most frequently either spawns a shell or is a worm. Even though placing the exploit code into the local buffer being overflowed is a common technique, the code can alternatively be placed above the return address on the stack, or in the heap. If the attacked program has root privilege, then when control is redirected to the injected exploit code, the code is executed with root privileges.

A buffer overflow usually contains both executable code as well as the address of where that code is stored on the stack. Frequently, this is a single string constructed by the attacker with the executable code first followed by enough repetitions of the target address that the return address is overwritten. This requires knowing exactly where the executable code will be stored or else the attack will fail. Attackers get around this by prepending a sequence of unneeded instructions (such as `NOP`) to their string. This creates a *ramp* or *sledge* leading to the executable code. Now the modified return address only needs to point somewhere in the ramp to cause a successful attack. While it still takes some effort to find the proper range, an attacker only needs a close guess to hit the target.

system() function: The second choice for redirecting execution is called the *return-to-libc* attack. It was invented to bypass protection methods that mark the stack as non-executable [20], which prevents execution of code on the stack. The return-to-libc attack eliminates the need for shellcode by redirecting execution to the `system()` call to create a shell. All the attack needs to do is copy the necessary arguments for the `system()` call onto the stack and change the return address to point to the library address of `system()`.

3) Methods of inputting the exploit code: There are three main ways of injecting malicious code into the vulnerable program. These are 1) user input, 2) network connection and, 3) environment variables. For example, a program might ask for a user or file name from standard input. If the program uses `gets()` then a sufficiently large user response could overflow the target buffer. An operating system might utilize a small buffer for the handling of ICMP echo packets (as they are normally quite small) and suffer an overflow if an attacker sends an unusually large packet. Similarly, if a program attempts to determine a user's home directory via the HOME environment variable, a malicious user might be able to cause an overflow by setting the value of the variable to be an unusually long value.

III. Related Work

Various tools and methods have been devised to stop these attacks with varying levels of security advantage and performance overhead. Solutions that trade off high levels of security for better performance prove incomplete and are eventually bypassed by attackers. On the other hand, high security solutions seriously degrade the system performance due to the high frequency of integrity checks and high cost of software-based memory protection. Another issue that diminishes the feasibility of these tools and methods is their lack of transparency to user applications. We have split the existing work into five groups: Static and Dynamic Analysis of Source Code, Modifications to the Executable, Modifications to the Compiler, Modification to the System Software, and Hardware solutions. A thorough list of all buffer overflow protection methods and tools is available from *The Buffer Overflow Page* [21].

A. Static (and Dynamic) Analysis of Source Code

Static analysis techniques try to identify potentially dangerous pointer de-references and unsafe function calls in the source code. Because detecting buffer overflow vulnerabilities statically is undecidable, these methods work on heuristics and therefore are neither sound nor complete. Several factors affect the inadequacy of static analysis: difficulty of bounds checking, pointer analysis, inter-procedural analysis, and unavailability of the program input at compile time. There is a collection of freely available auditing tools for C/C++ code, but Wilander et al. [22] reports that static analysis tools do not have a sufficiently low false positive rate to be of use to programmers, therefore they are merely used for security audits.

Wagner et al. [23] formulated the buffer overrun detection problem as an *integer constraint* problem and used graph theoretic techniques to solve the constraints. This technique has a high rate of false alarms, cannot handle pointers, double pointers or aliasing. The reported analysis time for 32K lines of C code is on the order of tens of minutes.

In a recent paper, Dor, et al. [24] combined all known types of static analysis methods to propose a tool, **CSSV**, for statically detecting buffer overflows. Using procedural analysis, CSSV in-lines the source code with annotations that have pre-, post-, and side-effect conditions (which they name "contracts"), analyzes pointer interaction, checks for runtime string manipulation errors with `assert()`, and finally performs conservative integer analysis. With respect to related work, a 93% drop in false alarm rate is reported however, manually writing contracts still renders a high implementation cost.

Larochelle and Evans [25] proposed a static analysis tool *built upon LCLint* with more expressive annotations. Annotations are the semantic comments that specify the highest index that can be safely written to and read from in a buffer. The annotations are used to detect inconsistencies between the code written and its expected behavior. This method does not detect all instances of vulnerabilities and has a high rate of false alarms.

Dynamic checks inserted by static analysis analyze the run-time contents of the variables during program execution. However, dynamic analysis is computationally more complex than static analysis and better results come with the price of increased computation time.

Haugh and Bishop [26] extended Wagner, et al.'s [23] method for dynamic execution. This method uses the STOBO tool to convert the vulnerabilities in the source code to the instrumented safe versions. The paper reports that this method compares favorably to ITS4 and Wagner, et al.'s original method in that it detects more vulnerabilities and has fewer false positives.

Yong and Horwitz [27] proposed a static analysis tool with dynamic checks to protect C programs from attacks via invalid pointer dereferences. The method has a low runtime overhead, no false positives, requires no source code modification and protects against a wide variety of attacks via bad pointer dereferences. The main idea is to use static analysis to detect *un-safe* pointers, and protect memory regions that are not legitimate targets of these pointers. This method maintains a mirror of the memory locations that can be pointed by *un-safe* pointers using one bit for every byte of the memory to specify whether each mirrored byte is write-safe, i.e., *legitimate*. The major drawback of this approach is that it doubles application run-time.

Toth and Kruegel [28] proposed abstract payload execution of HTTP requests to detect the NOP sledge which precedes the shellcode in most Type 1 buffer overflows. Toth and Kruegel report only a 1.4% increase in client contention rate and 2.9% decrease in client throughput. This method will only detect attacks that use a NOP sledge with the shellcode.

B. Modification of the Executable

Bhatkar et al. [29] proposed a method called **Address Obfuscation** that transforms the object file at link time (or the executable at load time) to 1) randomize the base addresses of stack, heap, and dynamically loaded libraries; 2) randomize the location of the routines and static data in executables; 3) permute the order of local variables on stack, static variables and routines in shared libraries and executables; and 4) insert random gaps in stack frames, between successive malloc buffers and between static variables. This method, which is very similar to PaX [30] except that PaX is a kernel patch, requires no change to the OS or to the compiler. Both of these methods are probabilistic methods that only harden, but do not eliminate, the attacker's chances of success. This method also imposes a process startup overhead.

Prasad and Chiueh [31] presents a static binary translation method that saves a redundant copy of the return address on the stack in the return address repository (**RAR**) at the function prologue, compares the saved return address with the original at the function epilogue, and flags an exception upon a mismatch. It is implemented by inserting a jump instruction in the prologue and the epilogue to jump to the corresponding code snippet and jump back to do the real prologue and real epilogue. The paper reports a 3% runtime performance overhead and 16K per process space overhead. This method is not secure because the RAR is protected with two *mine zones*[6] which makes this method vulnerable to Type 2 attacks.

C. Modification of the Compiler

StackGuard[7] [33], [34] is one of the earliest and most well-known compiler-based solutions. The additional code inserted at compile time places an integer of known value (called a canary) between the return address and the local variables on the stack at the function prologue. If a local buffer on the stack is overflowed, the attacker must overwrite the canary to reach the return address. StackGuard supports two types of canaries. The *random canary* method inserts a 32-bit random

[6] Mine zones are read only protected regions above and below the RAR to protect overflow into or out of the RAR.

[7] Microsoft also adopted a StackGuard-like mechanism in Visual C++.NET (v.7) [32].

canary after the return address in the function prologue, and checks the integrity of its value before using the return address at epilogue. The *terminating canary* consists of four string termination characters: `null, CR, -1`, and `LF`. Note that each one of these characters is a terminating value for at least one unbounded data copying function. If the attacker tries to overwrite the canary with the same terminating values, the overflow will never reach the return address because the string copy will be terminated at the canary.

As pointed out by Bulba and Kil3r [35], StackGuard only protects against Type 1 buffer overflows. In addition, it requires recompilation of the source code. Because it modifies the stack contents, programs dependent on the stack structure (e.g., debuggers) may no longer work. Finally, the random canary needs to be protected. For every function call and return instruction executed StackGuard must write the random canary to the stack and compare it on return. A varying performance overhead of 6-80% is reported in [33] which is a function of the ratio of the instructions required for the modified prologue and epilogue to the number of original function instructions.

StackShield [36] is a compiler modification that provides two different protection mechanisms for protecting the return address. *Global ret stack* implements a separate stack for the return addresses in a global array of 256 32-bit entries. For each function call, the return address is pushed onto both the program stack and the redundant global stack. On function return, the return address stored on the separate stack is used. *Ret range check,* a faster alternative to global ret stack, saves the return address of the currently executing function in a global long integer, and then compares it to the return address on the program stack when the function returns. This method has a low overhead, however it leaves the global ret stack and global return address vulnerable to Type 1 and Type 2 attacks. In addition, return addresses from the program stack and the redundant stack are not compared therefore, attacks are prevented but *not detected*.

Return Address Defender [37] creates a global integer array called the *Return Address Repository (RAR)* that holds the copies of the return addresses pushed on the stack. There are two versions of RAD that differ in the amount and cost of protection to the RAR. The first and less expensive method, *MineZone RAD* inserts two “minezones” above and below the RAR and marks them as read-only with the `mprotect()` system call. Any attempt by the attacker to overflow a buffer and overwrite the RAR would cause a trap and be denied by the OS. This method protects against Type 1 buffer overflows but can be defeated by Type 2. The second version of RAD, *Read-Only RAD* marks the entire RAR as read-only with `mprotect()` to achieve high security. This incurs a large overhead because during the function prologue, the RAR is marked as writable, the return address is saved into the RAR, and then the RAR is marked as read-only again. Similar to MineZone RAD, this method cannot prevent return-to-libc attacks which overwrite function pointers on the stack.

Chiueh, et al., report a performance degradation of 5-40% for Minezone RAD, and up to 1000% degradation for the more secure Read-Only RAD [37].

ProPolice [38] is a gcc extension that utilizes a mechanism similar to that in StackGuard, but with additional features. It adds some protection against Type 2 attacks by reordering the local variables stored on the stack such that the buffers are right before the canary and hence cannot be used in the same function's scope to overwrite a pointer. This tool was used to compile OpenBSD [39] and is part of its distribution. ProPolice requires recompilation of the source code, and like Stackguard, it modifies the stack contents, so, programs dependent on the stack structure may no longer work.

PointGuard [40] is a compiler technique to defend against attacks using pointers. A modification to gcc enables pointers to be encrypted with a per-process XOR key while in memory, and to be decrypted only when they are loaded into the registers. This technique requires recompilation of source code and incurs up to 21% slowdown on OpenSSL Speed benchmarks [41].

CRED (C Range Error Detector) [42] is a dynamic buffer overflow detector implemented as an extension of the GNU C compiler. CRED uses a bounds checking method that replaces every out-of-bounds (OOB) pointer value with the address of a special OOB object created for that value. Tested on twenty open-source programs, CRED claims to avoid deficiencies of previous dynamic buffer overrun detectors. CRED imposes 26% overhead and requires compilation of source code.

D. Modifications of the Library

FreeBSD Stack Integrity Patch (Libparanoia): Alexandre Snarski posted a patch to FreeBSD [43] in 1997 to check the integrity of the stack and later improved on the same idea and called it Libparanoia [44]. The patch modifies the insecure libc functions like `strcpy()` and `sprintf()` to kill the process if the destination buffer contains a stack frame pointer (FP).

Baratloo et al. [45]–[47] proposed two dynamically loadable library methods to protect against buffer overflow attacks. Neither of these methods require re-compilation unless the program is statically linked. The first method, **Libsafe** intercepts all calls to vulnerable library functions, such as `strcpy()` and `strcat()`, and executes their safe versions which implement the same functionality as the original but employ bounds checking to prevent buffer overflows. This method estimates the upper bound on the size of the buffer to be the end of the stack frame, so the return address can not be overwritten. Libsafe protects against Type 1 buffer overflows only since it still allows for overwriting a pointer or a function pointer in the local variables area of the stack which can be used to modify the return address. **Libverify**, on the other hand, is a run time implementation of StackGuard, which inserts function return address verification code at execution time via a binary re-write of the process memory instead of at compile time. Libverify also protects against only Type 1 buffer overflow attacks.

Baratloo, et al., report an average overhead of 15% for applications protected by Libsafe, Libverify, and StackGuard.

FormatGuard [48] is a patch to glibc that provides general protection against format bugs. FormatGuard uses particular properties of GNU CPP (the C PreProcessor) macro handling of variable arguments to extract the count of actual arguments. The actual count of arguments is then passed to a safe `printf()` wrapper. The wrapper parses the format string to determine how many arguments to expect, and if the format string calls for more arguments than the actual number of arguments, it raises an intrusion alert and kills the process. This method fails to protect against calls to `printf()` when the correct number of arguments is given but they are not of the expected types, i.e., if an integer is received when a double is expected. It also fails if the call to `printf()` is implemented via a function pointer or if the low level functions of `printf()` (e.g., `vsprintf()`) are called directly, or another I/O library is used. FormatGuard imposes 37% overhead on `printf()` calls which result in an 1.3% run-time overhead for their set of benchmarks [48].

E. Modifications of the Kernel/OS

The first kernel-based solution, **StackGhost** [49], is a patch to the OpenBSD 2.8 kernel under the Sun SPARC architecture. Frantzen and Shuey performed experiments on three methods for protecting the return address. The first two XOR the return address on the stack with a cookie before writing it on the stack and then XOR it again with the same cookie before the return address is popped off the stack. This method distorts any attack to the return address but does not detect it, therefore, another method is used to detect the attacks. In SPARC architecture, the memory is four byte aligned and the least significant (LS) two bits are always 0s. So, the two bits are set at the function prologue and verified to be set at the epilogue. If the attacker is not aware of this, they will inject a four byte aligned address in the return address and therefore the attack will fail. But, once the attacker figures this out, they can set the two LS bits of the address that they want to jump to, and then overwrite the return address with the modified address. The XOR cookie method comes with two flavors. XOR cookie per-kernel and XOR cookie per-process. Both of these methods (especially per kernel XOR cookie) are easily bypassable since the cookie can be figured out if the contents of the stack frame can be observed (e.g., using the method in the format string attacks as described in Section II-C) and the return addresses are extracted from the program binaries. Frantzen and Shuey report 17.44% overhead for per-kernel XOR cookie and 37.09% overhead for per-process XOR cookie.

To prevent execution of the shellcode on the stack, Solar Designer proposed the **Non-Executable User Stack**. This solution, a Linux kernel patch from the Openwall Project [20], can be bypassed with return-to-libc attacks or running the shellcode somewhere in memory other than in the stack, for instance the heap. To prevent return-to-libc attacks, this patch also changes the default addresses of the shared libraries in libc to contain a zero byte. It is difficult to overwrite the return address with a value that contains a zero byte (null) since a zero byte is a string terminator and terminates string copying functions. This method, which failed to pass Linus Trovald's approval [50] to be included in the linux kernel, prevents attacks where the shellcode is inserted in the stack and causes trampoline functions [51] and debuggers to fail.

PaX [30] is a kernel patch that includes two protection mechanisms. *NOEXEC* is a page based mapping mechanism which does not allow pages that are writable to also be executable. This prevents injection and execution of code in a process' address space.[8] Address Space Layout Randomization (ASLR) is a technique that randomizes the addresses of the libc functions (e.g., system), the function return addresses, the base of the stack and the heap. Although this method makes it harder for the attacker to predict the vulnerable memory addresses, it is fundamentally a probabilistic method which also incurs a process startup overhead.

F. Hardware Solutions

Independent of and concurrent to our proposal, there have been two recent attempts to provide a hardware solution.

[8] Write-Xor-Execute [52] method implements the same idea.

Xu, et al. [53] proposed two methods for protection of the function return address from being overwritten on the stack. *Split control and data stacks* protects the return address by storing it on the control stack, away from buffers in the data stack that can be overflowed to overwrite the return address. This approach can be implemented with either compiler or hardware support. The compiler implementation has up to 23% overhead in SPECINT benchmarks and 2% to 5% overhead for an FTP server. The hardware implementation eliminates this overhead, but would require an extra register and a change to the instruction set semantics. The authors assume one page of memory should be enough for every process and do not discuss memory management of the control stack. This method does not protect against Type 2 buffer overflows or format string attacks because the control stack is not protected. The second method, Secure Return Address Stack (SRAS), is a hardware-based approach that is implemented on top of the Return Address Stack (RAS). SRAS stores a redundant copy of the function return addresses in the processor to validate the return addresses on the stack. This method has three versions, *Speculative SRAS*, *Non-Speculative SRAS*, and *Non-Speculative SRAS with Overflow Handling*. Speculative SRAS incurs almost 100% overhead. Non-Speculative SRAS has fixed stack size and cannot handle deeply nested functions. Non-Speculative SRAS with Overflow Handling swaps the contents of the SRAS to the PCB of the process to handle overflows. Xu, et.al do not discuss context switch overhead and their `setjmp()/longjmp()` handling method requires the addition of a special instruction to rewind the SRAS.

Lee et al. [54] also proposed a hardware-based **Secure Return Address Stack** to protect against attacks on the function return address. Changes are made to the microarchitectural structure of the CPU to keep a copy of the return addresses for validation. This approach does not consider 1) register port contention due to validity checks of the return address, 2) issues of cleaning up the SRAS after branch mispredictions, or 3) program flow changes caused by functions like `setjmp()` and `longjmp()`. Its performance tests use a single-way, in-order-issue processor, which is outdated compared to modern wide, out-of-order-issue processors.

G. Safer C Language Compilers

There are several dialects of C that offer security measures employed in higher level languages such as Java, while maintaining the low level and efficient aspects of the C programming language. Enforcing type safety, providing better memory management, and array bounds checks are some of the security features employed in Cyclone [55], Safe C Compiler [56], and CCured [57].

These modified variants of C are not simple drop-in replacements. These language modifications require a programmer to change portions of the source code, often requiring some sort of indication where protection should be enabled (otherwise, the normal lack of bounds checking applies for compatibility). The reason for manual activation of bounds checking is that these projects self-report overheads on the order of 100% in some instances. Additionally, they suffer from the same drawbacks on legacy binaries as do other compiler modifications. Namely, they only protect newly compiled programs and do not protect system kernel, libraries, or existing binaries without recompilation.

H. Summary

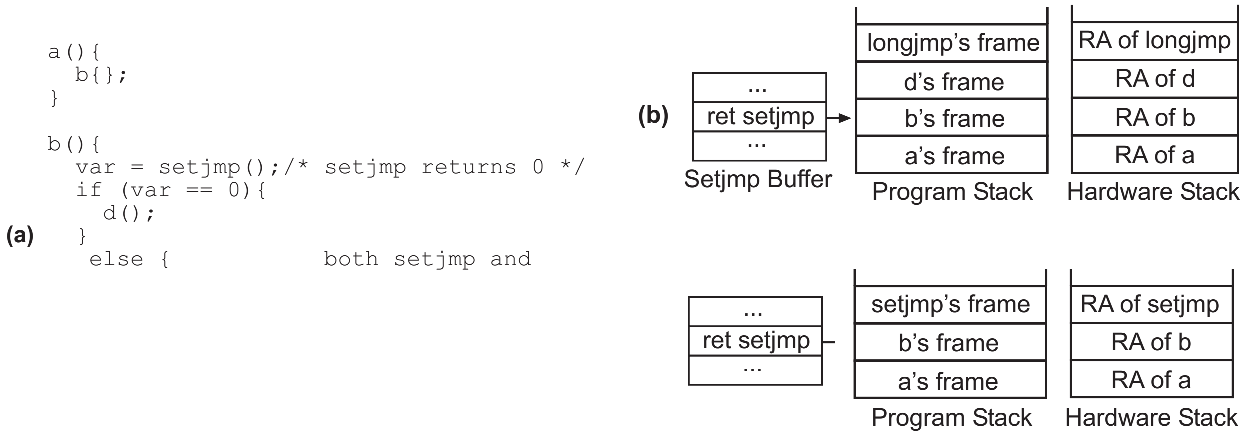
Solutions that trade off a high level of security for better performance are eventually bypassed by the attackers and prove incomplete. On the other hand high-security solutions seriously degrade the system performance due to frequent integrity checks and costly software based memory protection. An issue that diminishes the feasibility of these tools and methods is their lack of transparency to the application or to the operating system. Moreover, some earlier methods lack protection against Type 2 attacks. In our evaluation of our hardware-based approach we have elected to compare against StackGuard for several reasons. First, it is the approach that is most widely cited. Second, its mechanism for protecting the return address on the stack is found in the tools ProPolice and Libverify. Third, it is not architecture specific and is therefore portable, and fourth, it reports little overhead while maintaining security against the most prevalent type of attack on the return address pointer.

IV. SmashGuard: A Hardware Solution

In this section we present a hardware solution that is secure and inherently faster than the existing software methods. We elaborate on the complications we face with `setjmp()` and `longjmp()`, process context switches, and deeply-nested function calls, and how we solve them. Finally, we describe our microarchitecture and discuss hardware implementation issues.

A. Overview

Our approach, which we call SmashGuard, protects against attacks on return addresses by saving the return address in a hardware stack added to the CPU. With each function call instruction, the



```
a(){
  b{};
}

b(){
  var = setjmp();/* setjmp returns 0 */
  if (var == 0){
    d();
  }
   else {           both setjmp and
```

Fig. 3. Setjmp()/longjmp() example. (a) code snippet, (b) program stack and hardware stack just before `longjmp()` returns, (c) program stack and hardware stack just before `setjmp()` returns

return address and the stack frame pointer are pushed onto the hardware stack [9]. A return instruction pops the most recent pair of address from the top of the hardware stack and compares it to its return address. If a mismatch is detected between the two return addresses, then a hardware exception is raised. In the exception handler, the OS may employ a variety of policies based on the desired level of security (e.g., the process may be killed and a report may be sent to syslog).

This simple functionality is not sufficient to handle the problem of `setjmp()` and `longjmp()`. `setjmp()` and `longjmp()` circumvent the last-in first-out ordering of the program stack causing the hardware stack to become inconsistent with the program stack. As we explain in Section IV-B, we extend the hardware stack's functionality to enable it to maintain consistency.

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

[9] In the next section we explain why merely storing the return address on the hardware stack is not sufficient.

B. Handling `setjmp()` and `longjmp()`

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

Because `longjmp()` avoids going through the usual function return sequence, our hardware stack becomes inconsistent with respect to the program stack. In Figure 3(a), we show a code snippet where `a()` calls `b()` which in turn calls `setjmp()`. As is typical in programs using `setjmp()` and `longjmp()`, depending on `setjmp()`'s return value *var*, `b()` may or may not call `d()`. `d()` calls `longjmp()`. During execution, `a()` calls `b()` and `b()` calls `setjmp()`. `setjmp()` saves a snapshot of `b()`'s current register state and a copy of its own return address in a buffer. `setjmp()` then returns with a return value of 0 causing `d()` to be called. `d()` calls `longjmp()` which uses `setjmp()`'s buffer to restore `b()`'s register state. `longjmp()` uses the saved return address in the buffer which is `setjmp()`'s return address to return to `b()` with a return value of 1, allowing `b()` to return to `a()`.

In Figure 3(c), we show the program stack when `setjmp()` is about to return. We see that the hardware stack is consistent with the program stack. We also see that the buffer holds `b()`'s state and `setjmp()`'s return address. In Figure 3(b), we show the program stack when `longjmp()` is about to return. At this point, the program stack will collapse down to `b()`'s frame, and `longjmp()` will return to `setjmp()`'s return address using the buffer. Because the return address is coming from the buffer and not the program stack, `setjmp()`'s return address does not exist anywhere – certainly not at the top, nor anywhere below – in the hardware stack, which tracks only the program stack. If nothing is done, SmashGuard would compare the hardware-stack top, which is `longjmp()`'s return address into `d()`, against `setjmp()`'s return address, and a mismatch would result.

Because the relevant return address simply does not exist in the hardware stack, we propose that `longjmp()` use an *indirect-jump* (i.e., jump-through-register) instruction to return, rather than use a return instruction [10]. Because an indirect-jump instruction will not trigger SmashGuard's check, `longjmp()` will be allowed to return without a mismatch. The program stack and hardware stack are not consistent yet: the program stack holds frames for `b()` and `a()`, but the hardware stack holds the return addresses of `longjmp()`, `d()`, `b()`, and `a()` (see Figure 3 (c)). When `b()` returns, a mismatch would result.

However, unlike the previous mismatch situation, the required return address (i.e., `b()`'s return address) exists in the hardware stack – only not at the top. Therefore, we propose that upon a mismatch SmashGuard keep popping the hardware stack until either a match occurs, or the bottom of the stack is reached in which case the mismatch exception is raised. If a return address is modified due to an attack, none of the addresses on the hardware stack would match and the bottom of the stack will be reached. Therefore, no attack will go undetected. Because the only way for the bottom of the stack to be reached is due to an attack, SmashGuard will never raise a false alarm.

There are two more complications remaining. First, if `b()` is called multiple times before `longjmp()` is called, then the hardware stack would hold multiple instances of `b()`'s return address. In that case, the popping of the hardware stack would stop at the first instance of `b()`, which may not be the instance that executed the `setjmp()`. To identify the correct instance, we propose to *store the return address and the stack pointer*, instead of just the return address, in the hardware stack. Now calls push the two values onto the hardware stack, and returns compare both the return addresses and the stack pointers. Using the stack pointer is guaranteed to identify the correct instance because 1) the stack pointer holds a unique value for each instance, and 2) the stack pointer value is the same when a function call and the corresponding function return occurs.

Second, because we require `longjmp()` to return using an indirect-jump instruction and not a return instruction, returns from `longjmp()` are not processed within SmashGuard. Therefore, an attack on the return address stored in the `setjmp()` buffer (via some buffer overflow attack that somehow overflows into the `setjmp()` buffer) would go undetected. To avoid this problem, we propose that writers of `setjmp()` and `longjmp()` library code protect the return address stored in the buffer using schemes similar to StackGuard (e.g., place random numbers around the return address and check their integrity before using the return address in the `longjmp()`). Because this code is library code and not application code, we retain application transparency.

[10] This is a library modification.

Now we explain the solutions proposed by the other approaches described in Section 3 to `setjmp()` and `longjmp()`. Techniques, such as StackGuard, that do not store a copy of the return address stack need not do anything special for `setjmp()` and `longjmp()`. RAD's solution is to continue to pop return addresses off of their stored table of return addresses until a match is found. The problem with this approach is that it is possible that the modified return address value exists somewhere further down on the hardware stack, causing execution to continue without detecting the problem. As has been pointed out before, failing to stop execution is no worse than the current situation where no check is being made, but this answer is unsatisfactory.

The hardware solution proposed by Lee, et al. [54] lists four ways to handle `setjmp()` and `longjmp()`, none of which retain both security and the functionality of the code. On the two extremes the authors suggest either prohibiting `setjmp()/longjmp()` or disabling the hardware stack protection for programs that contain `setjmp()/longjmp()`. An intermediate solution, is to introduce new user-mode (i.e., non-privilege mode) instructions `sras_pop` and `sras_psh` (SRAS is the name of their proposed hardware solution) to make the hardware stack consistent after a `longjmp()`. They propose injecting these instructions either at compile time or at runtime. However, a malicious user could use the instructions to tamper with the hardware stack itself, possibly compromising security.

C. Handling Deeply-Nested Function Calls and Process Context Switches

Because our solution is the same for deeply-nested calls and context switches, we describe these issues together. The hardware stack may fill up for programs with deeply-nested function calls. A 2-KB stack holds 512 32-bit addresses (e.g., x86) or 256 64-bit addresses (e.g., Alpha). To handle nested function calls deeper than 512 (256) smashguard raises a hardware-stack-overflow exception, which copies the contents of the hardware stack to the program's Process Control Block (PCB) where it is saved at context switch. The PCB includes a stack of stacks and every time a stack is full, it is appended to the previous full stack. Another exception, hardware-stack-underflow, will be raised when the hardware stack is empty to copy in the last saved full-stack from the PCB. These exceptions are not a performance concern because we expect them to be infrequent. Indeed, in our experiments with the SPEC2000 benchmarks, our 2-KB stack was sufficiently large such that no overflows occurred.

A context switch requires saving process state, requiring that we 1) copy out the hardware stack of the running process either to the PCB or a memory location pointed by a special pointer in the PCB, and 2) copy in the hardware stack of the scheduled process. To handle both the above scenarios without adding any special instructions to the instruction set, we employ memory mapping (similar to memory-mapped I/O), so that regular load or store instructions can be used to read and write the stack in these scenarios. We map a part of the address space to the hardware stack. A regular load or store access to this part translates to a read or write access to the hardware stack, much as memory-mapped I/O devices are read and written. Recall that I/O devices are protected from direct access by user-level code via virtual memory protection. Similarly, direct access to the hardware stack is forbidden by virtual memory protection. Thus, only the OS can read or write the memory-mapped stack, and the OS does so to handle both scenarios. Because the saving and retrieving of the hardware stack from memory is handled by the kernel our method is secure.

Although SmashGuard increases the state that needs to be saved and restored at context switches, we expect this overhead to be small. In typical interactive desktop environments, modern operating systems target about 1% overhead for context switches due to time slice expiration. For a 10-20 milliseconds time slice, the context switch overhead (i.e., time spent in the OS to switch from one process to another) is about 100-200 microseconds. Copying our 4-KB (512 64-bit words) hardware stack will require about 1000 instructions (a pair of load and store instructions for each word), which may take around 2000 cycles (assuming a conservative 0.5 instructions per cycle). At 1 GHz, this copy adds 2 microseconds to the context switch time of 100-200 microseconds, or about 1-2% of context switch time. With a 10-20 millisecond time slice, copying adds about 0.01-0.02% overhead to wall clock time. In more context-switch-intensive environments (e.g., interrupt-intensive embedded systems), the copying overhead will be higher.

D. Implementation

In this section, we describe three implementation schemes that allow different trade-offs between implementation difficulty and performance. We explain our implementations in terms of an Alpha-like RISC architecture that places the return address of a call instruction in a *link register*. This link register may be either an implicit register that is hard-coded in the instruction set, or a register explicitly-specified in the call instruction. The return instruction uses a *return address register* –

either the implicit register or an explicitly-specified register to return.

SmashGuard modifies call instructions to push the link register and the stack pointer register onto the hardware stack. Recall that both are needed to accommodate `setjmp()/longjmp()` (see Section IV-B). Return instructions pop off the hardware stack and check the return register and the stack pointer against the popped values. Because modern processors execute instructions out of program order and speculatively under branch prediction, call and return instructions may be executed under misspeculation and out of program order. Consequently, pushing on and popping off the hardware stack at the time of execution of call and return instructions is not reliable. Doing so would require that we clean up the hardware stack on mispredictions. To avoid this complication, we push on and pop off the hardware stack when call and return instructions commit, which occurs in program order and after all outstanding speculations are confirmed. However, there is one main difficulty: call and return instructions do not carry the needed register values – the link register and the return address register – with them to the commit point. The link register is written to the register file when the call instruction executes, and the return address register value is used by the return instruction when it executes, well before commit. Certainly the instructions do not carry the stack pointer to the commit point. There are two options: 1) obtain the register values from the pipeline during instruction execution, or 2) obtain the values from the register file at instruction commit.

For the first option, we use a table, called the return address table (RAT), into which call and return instructions place the register values. The values are read from the RAT upon instruction commit and pushed on the hardware stack, or compared against the top of the stack. To avoid complications in matching instructions to their RAT values, we make the RAT as large as the active list (or the reorder buffer, which is used to hold all in-flight instructions until commit), so that instructions can easily find their register values simply by using their active list pointers. Because the RAT is accessed using the active list pointers, misprediction – rollbacks of the active list – automatically roll back the RAT. This advantage does not exist if we had used the hardware stack itself to hold speculative values, because rolling back the active list, which is a queue, is not similar to rolling back the hardware stack, which is a stack.

The only issue now is that call and return instructions need to read the stack pointer register value (from the register file or bypass paths), an action that is not taken in conventional pipelines. This extra read, however, is not a problem because calls and returns read at most one source operand (a call-through-register reads the call target from a register), implying the stack pointer can be read in place of the non-existent second source operand. The link register value is computed by calls and can be pulled off from wherever it is computed (e.g., the execute stage).

Because the RAT is invisible to the software, like the hardware stack, this scheme is secure. Because the number of in-flight instructions is not large (e.g., 300 instructions) and because call and return instructions are relatively infrequent, the RAT need be neither large (e.g., a 2-KB RAT would suffice) nor support high bandwidth. Because this option results in virtually no performance degradation, we call this scheme *No-Stall*.

If the RAT does not fit the constraints of a specific pipeline implementation, designers may pursue the second option of reading the values from the register file. This option raises two issues: 1) because of register renaming, we cannot access the physical register file with the architectural register specifiers, and 2) the register file needs to be accessed by all committing call and return instructions, which may contend with instructions in the register read stage of the pipeline.

We address each of these issues in turn. Call and return instructions have to carry the required physical register specifiers to the commit point. It would seem that carrying the required values themselves instead of the specifiers is a better option. However, there are two advantages with the specifiers: 1) the specifiers are smaller than the values (e.g., 8-bits versus 64-bits), and 2) in modern pipelines, instructions already carry the previous physical register specifier mapping the architectural destination register to the commit point, so that the previous physical register may be freed. Therefore, the wires and control circuitry needed to carry specifiers already exist; we simply need one additional specifier to be carried.

The only remaining complication is register port contention. Because adding extra register file ports is expensive, and because call and return instructions are not frequent enough to cause significant contention, we propose two schemes to handle contention: a conservative scheme called *Complete-Stall* and a more aggressive scheme called *Partial-Stall*. In the Complete-Stall scheme, we completely stall issue in a cycle in which a call or a return instruction commits. The rationale is that it may be hard to design a select logic that accounts for register port requirements of committing call and return instructions, in addition to the usual resource requirements of instructions waiting to be selected. The select logic is usually on the critical path of the clock, and such additional requirements may impact clock speed. In the Partial-Stall scheme, the select logic stalls only those instructions as are needed to free up the requisite number of ports for the committing calls and returns.

E. Implementation Cost

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

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

When introducing new hardware, a key cost factor to avoid is the introduction of new instructions to the instruction set. New instructions imply an implicit cost in future implementations that must support the new instructions (in their original form) for compatibility reasons. Because SmashGuard introduces a hardware stack, we have to ensure that the stack does not imply new instructions. If the hardware stack were completely invisible to software (e.g., the hardware caches are usually invisible to the user-level code, unless the code optimizes for cache performance), then the stack will not require new instructions. In our approach the hardware stack is invisible to software except for context switches and when the call depth exceeds the stack size. In the later case, an exception is raised and the exception handler copies the stack to locations in memory owned by the OS.

F. Issues Raised by Multithreading

Some modern processors implement Simultaneous MultiThreading (SMT) [58] which simultaneously executes multiple threads on a single pipeline. Multiple threads sharing a single hardware stack in SmashGuard may make the effective size of the stack too small. Because SMT already provides as many copies of certain hardware resources (e.g., rename tables, load/store queue, active list) as the number of threads, SmashGuard's hardware stack can also be replicated. Second, kernel-level multithreading does not cause any problems for SmashGuard because the threads are switched in and out by the OS and the hardware stack can be saved and restored as part of the context switch. Third, process migration in multiprocessor systems does not cause any problems. Conventional systems explicitly migrate some of the process state such as register and TLB contents, SmashGuard's hardware stack can also be migrated explicitly.

However, user-level multithreading is problematic for SmashGuard because multiple user-level threads would share the hardware stack, but call and returns from the threads would interleave in arbitrary order, and not LIFO. Because user-level threads do not go through the OS for invocation, suspension, and resumption, an OS-driven context switch cannot be used to share the hardware stack among the threads. The option of providing a large number of stacks in hardware is not attractive either because the number of stacks needed would be large (e.g., 256) to avoid restricting user-level threading. One option is to allow threads the same hardware stack by (statically or dynamically) partitioning the stack and accessing the stack based on a thread identifier (id). The thread id is maintained by the thread library in a register and the thread id allows each thread to access its part of the hardware stack. Any overflow or underflow would be handled as before. Another option is to disable SmashGuard and use software-based solutions for user-level multithreaded code. Finally, certain synchronization primitives such as coroutines may be difficult to accommodate in SmashGuard. Coroutine calls may be done in one thread and returns in another thread, and it may be hard to synchronize the hardware stacks of the two threads. Here again, an option is to disable SmashGuard and use software-based solutions for coroutine-based code. In both of these cases recompilation is not an issue because the user code is available.

TABLE II
HARDWARE PARAMETERS

Processor	4-way issue, 128-entry window, 64-entry ld/st queue (10 cyc br. penalty)
Branch Prediction	8K/8K/8K hybrid, 128-entry RAS, 4-way 8K BTB
L1 Cache L2 Cache	64K 2-way 2-cycle I/D (lockup free and pipelined) 2MB 8-way 14-cycle (lockup free and pipelined)
Main Memory	Infinite capacity, 80 cycle latency split transaction 32-byte wide bus
Hardware Stack	512-entry Hardware Stack

V. Experiments and Results

In this section, we evaluate the performance of SmashGuard and StackGuard, a software based protection mechanism, on a common execution-driven simulation infrastructure for a modern, high-performance processor.

TABLE III

Call frequency per 1k instructions (CF), maximum call depth (CD), and base IPCs using 4-way (I4) and 8-way (I8) issue widths for Integer and Floating Point benchmarks

Int	bzip	crafty	gap	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	vpr
CF	4.24	6.89	3.37	2.41	3.34	6.06	14.08	9.71	8.13	13.33	16.67
CD	12	32	77	61	12	22	238	65	17	31	18
I4	1.9	1.8	2.0	1.7	1.9	0.7	1.7	1.4	1.5	1.9	2.2
I8	2.1	2.2	2.5	1.8	2.1	0.7	1.8	1.6	1.7	2.3	2.8
FP	ammp	applu	apsi	art	mesa	mgrid	swim	wupwise			
CF	4.83	∼0	2.11	0.34	7.35	∼0	∼0	8.33			
CD	17	13	20	12	15	16	13	14			
I4	2.2	2.6	1.9	1.6	2.3	2.5	2.8	2.0			
I8	2.5	3.1	1.9	1.7	2.9	2.7	3.8	2.3			

A. Methodology

We modified the SimpleScalar-3.0 simulator [59] to model two of our three schemes of SmashGuard – Partial-Stall and Complete-Stall. We do not report No-Stall because it incurs almost no performance overhead. Table II shows the base system configuration parameters used throughout the experiments, unless specified otherwise. We simulate both 4- and 8-way out-of-order issue superscalar processors augmented with a 512-entry hardware stack for SmashGuard. We modified *gcc-3.0.3* to port StackGuard to the Alpha architecture. The ported version of StackGuard modifies the prologue and epilogue code of the compiled functions to include the terminating canary (see Section III-C). In Figure 4 we show the eight extra instructions inserted by our StackGuard patch. The prologue code places the terminating canary (0x000aff0d) on the program stack, and the epilogue code loads the canary from the stack and compares it to the original. If there is mismatch, the function `attack_handler()` is called.

We compiled the benchmarks on an Alpha machine running Tru64 using the original gcc and the StackGuard port. The original gcc's binaries are used by the SmashGuard runs. Because the StackGuard port to handle C libraries is not available, we compiled only the benchmark code with the StackGuard port and used the standard C libraries. Accordingly, our simulator samples performance only in the application functions and not the library functions. We ran the SPEC2000 benchmarks shown in Table III. We used f2c to covert fortran-77 benchmarks (applu, apsi, equake, mgrid, sixtrack, swim, and wupwise) to C. We did not simulate Fortran-90 (facerec, fma3d, galgel,

```
function entry:
<old prologue>
  ldah  $1, 11     #move canary into reg
  lda   $1, -243($1)
  stq   $1, 0($30)
<old prologue>
  ...
<function body>
  ...
  ldq   $1, 0($30) #load canary from stack
  ldah  $2, 11     #move canary into reg
  lda   $2, -243($2)
  cmpeq $1, $2, $3 #compare
  bne   $3, $Label
  call  $26, attack_handler
$Label:
  <old epilogue>
  ret
```

Fig. 4. StackGuard's extra instructions

and lucas) and C++ (eon) benchmarks as doing so would require implementing StackGuard in Fortran-90 and C++ compilers. While the total number of instructions executed by SmashGuard and StackGuard are different, the number of call/returns are the same in SmashGuard and StackGuard. Therefore, we ran each benchmark for the same number of call instructions in each case for fair comparison. We skipped 20 million calls and ran 10 million calls for all the integer programs (bzip, crafty, gap, gcc, mcf, parser, perlbmk, twolf, vortex, and vpr) and for three floating-point programs (ammp, mesa, and wupwise). The rest of the floating-point programs have such low call frequency that we had to simulate fewer calls to avoid inordinately extending our simulation time. We skipped 1 and 0.5 million calls and ran 1 and 0.5 million calls for apsi and art, respectively. Programs applu, mgrid, and swim make virtually no application calls. We do not show results for equake and sixtrack because they make only library calls.

B. Functionality Results

To verify that our hardware modifications can actually detect changes in the program return address, we created a binary for the Alpha that overwrites one of its own local buffers and executed it in the simulator. We were limited to self-attacking code because SimpleScalar only supports single process execution. Our hardware modification was able to detect that the return address value being

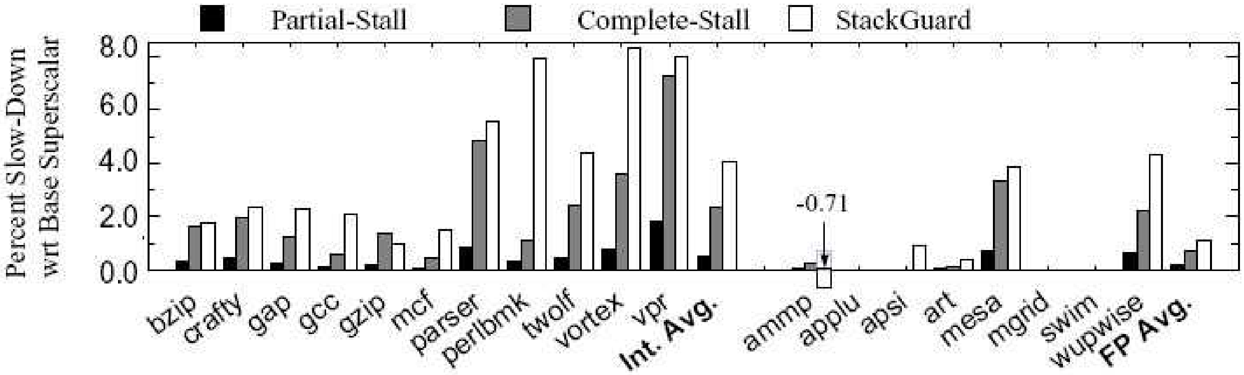


Fig. 5. Results for 4-way issue superscalar

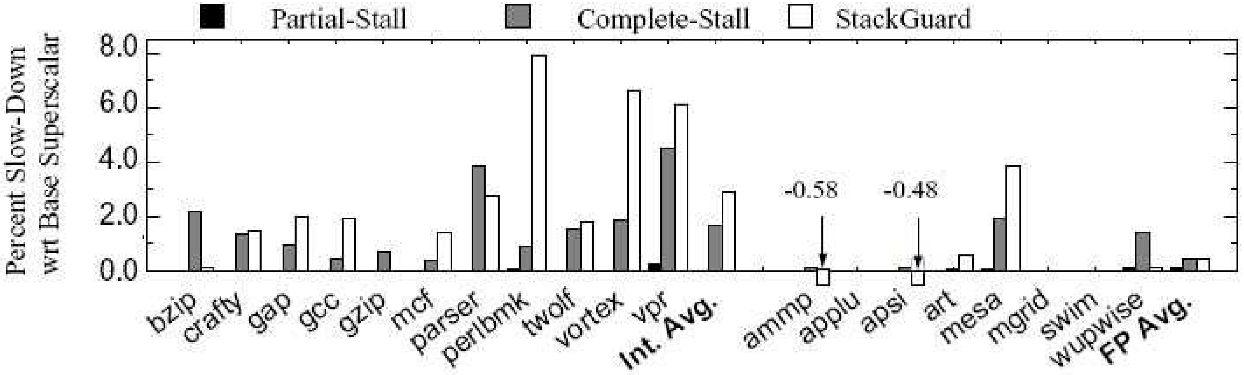


Fig. 6. Results for 8-way issue superscalar

pulled from the stack did not match the value stored in the hardware stack.

C. Performance Results

In this section, we compare SmashGuard and StackGuard to a conventional superscalar with no support for buffer overflow detection. Figure 5 and Figure 6 show our results for issue widths of 4 and 8, respectively. In both graphs, the Y-axis gives the percent slowdown with respect to the base superscalar processor of equal issue width, and the X-axis shows our benchmarks starting on the left with the integer programs bzip through vpr, followed by the average for the integer programs, the floating-point programs ammp through wupwise, ending with the average for the floating-point programs. The left bars show SmashGuard using the Partial-Stall scheme, the middle bars show SmashGuard using the Complete-Stall scheme and the right bars show StackGuard. The figures do not show the No-Stall scheme because it does not incur any more stalls than the base superscalar (i.e., No-Stall has virtually zero percent degradation).

A striking trend in both Figure 5 (4-way issue) and Figure 6 (8-way issue) is that the integer programs incur more performance degradation than floating-point programs, which incur little degradation. If a program's call frequency is low, then both SmashGuard's and StackGuard's overhead are incurred less frequently. This trend is corroborated by Table III, where we see that the integer programs' call frequencies are generally higher than those of the floating-point programs. The exceptions are *mesa* and *wupwise*, which have modestly high call frequencies. Because these programs have high instruction-level-parallelism indicated by their high IPC (instructions per cycle), SmashGuard's overhead of stalled issue is hidden by the parallelism. Because the floating-point programs' degradations are negligible, we do not discuss them further.

Focusing on the SmashGuard numbers (left and middle bars), we see two trends. First, as expected, the Partial-Stall scheme (left bar) performs better than the Complete-Stall scheme (middle bar), on both 4-way issue (Figure 5) and 8-way issue (Figure 6) processors. With 4-way issue, Partial-Stall and Complete-Stall incur 0.5% and 2.4% average degradation, respectively, for the integer programs. Partial-Stall's worst-case degradation is 1.8% for vpr and has less than 1% degradation for the rest of the programs. Complete-Stall, on the other hand, incurs more than 4% degradation for mcf, parser, vortex, and vpr. The relatively large degradations are not surprising because these programs have not only high call frequency leading to high overhead, but also low IPC with less ability to hide the overhead (Table III).

As we increase the issue width from 4 to 8, Partial-Stall incurs almost no degradation while Complete-Stall still incurs 1.2% average degradation. Because there are more free issue slots in a 8-way issue processor than a 4-way issue processor, both schemes' overheads are hidden.

Now, we focus on the StackGuard numbers (right bar). We see that StackGuard's average degradation is worse than that of Partial-Stall, and comparable to that of Complete-Stall on both 4-way issue and 8-way issue processors. StackGuard incurs a 2.8% and 1.8% average degradation on 4-way issue and 8-way issue processors, respectively. However, for perlbmk, vortex and vpr, StackGuard incurs more than 8% and 6% degradation on 4-way issue and 8-way issue processor, respectively. High call frequency and low IPC of these programs have the same negative effect on StackGuard's performance as SmashGuard's performance. Like SmashGuard, StackGuard incurs less degradation when the issue width was increased from 4 to 8. On the 8-way issue processor, apsi and wupwise unexpectedly improve in performance (i.e., negative degradation) with StackGuard. This improvement is the result of a pathological interaction between StackGuard's extra instructions and the branch predictor, causing an accidental improvement in the prediction accuracy.

Finally, the call depths listed in Table III show that the programs do not exceed the depth of 238 (parser), indicating that a 512-entry hardware stack is sufficient to avoid most stack overflow exceptions in SmashGuard.

VI. Conclusions and Future Work

This paper has proposed a novel microarchitectural support to protect against attacks that overwrite the return address on the process stack to redirect execution. We have provided a hardware stack that securely handles both Type 1 (buffer overflows) and Type 2 (attacks through a pointer) attacks on the return address. The key contributions of this paper are:

Complete Solution: We have designed a complete solution, which handles `setjmp` and `longjmp` as part of the hardware solution, and handles hardware stack overflow/underflow and process context switches with a small modification to the OS.

Trade-offs: We have proposed three implementation schemes that allow different trade-offs between implementation difficulty and performance.

Detailed Performance Analysis: We have performed a detailed performance analysis comparing the most-frequently applied software solution, StackGuard, to SmashGuard on a common simulator for a high-performance processor.

Our best-performing implementation, No-Stall, incurs virtually no performance degradation but has the small implementation cost of a 2-KB table. We compared the other two implementations (Complete-Stall and Partial-Stall) to StackGuard. Our experiments show that StackGuard performs comparably to Complete-Stall but StackGuard is less robust than Partial-Stall. For an 8-issue processor, while StackGuard incurs only slightly less average degradation than Partial-Stall, StackGuard's worst-case degradation is 8% whereas Partial-Stall incurs less than 0.5%. Moreover, StackGuard requires application recompilation and does not protect against Type 2 attacks.

With every passing day, the number of attacks on systems connected to the Internet increases [60]. Attacks are increasingly automated, attack tools are much more sophisticated, and there have been attacks on the critical infrastructure of the Internet [61]. SmashGuard provides a robust solution to one of the most prevalent attacks of today.

REFERENCES

[1] AMD. (2004) AMD chips include new buffer overflow protection. [Online]. Available: http://www.computerweekly.com/Article127571.htm

[2] Intel. (2001) Execute disable (XD) bit. [Online]. Available: http://www.intel.com/business/bss/infrastructure/security/xdbit.htm

[3] T. Corporation. (2004) AntiVirusNX Technology. [Online]. Available: http://www.transmeta.com/efficeon/antivirusnx.html

[4] Microsoft. (2004) Microsoft Windows XP SP2 Data Execution Prevention. [Online]. Available: http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx

[5] Aleph1, "Smashing the stack for fun and profit," *Phrack Magazine*, vol. 7, no. 49, Nov. 1996. [Online]. Available: http://www.phrack.org/show.php?p=49&a=14

[6] CERT Coordination Center. (2001, June 19) CERT Incident Note IN-2001-08 Code Red Worm Exploiting Buffer Overflow In IIS Indexing Service DLL . [Online]. Available: http://www.cert.org/incident_notes/IN-2001-08.html

[7] ——. (2001, August 6) CERT Incident Note IN-2001-09 Code Red II: Another Worm Exploiting Buffer Overflow In IIS Indexing Service DLL . [Online]. Available: http://www.cert.org/incident_notes/IN-2001-09.html

[8] ——. (2003, August 11) CERT Advisory CA-2003-20 W32/Blaster worm. [Online]. Available: http://www.cert.org/advisories/CA-2003-20.html

[9] Sophos Virus Analysis. (2003, August 19) W32/Nachi-A. [Online]. Available: http://www.sophos.com/virusinfo/analyses/w32nachia.html

[10] ——. (2004, May 1) W32/Sasser. [Online]. Available: http://www.eeye.com/html/research/advisories/AD20040501.html

[11] CERT Coordination Center. (2001, June 19) CERT Advisory CA-2001-13 Buffer Overflow In IIS Indexing Service DLL. [Online]. Available: http://www.cert.org/advisories/CA-2001-13.html

[12] ——. (2003, July 16) CERT Vulnerability Note VU 568148 Microsoft Windows RPC vulnerable to buffer overflow. [Online]. Available: http://www.kb.cert.org/vuls/id/568148

[13] ——. CERT Coordination Center Advisories for 2002. [Online]. Available: http://www.cert.org/advisories/#2002

[14] SANS Institute. SANS/FBI Top 20 List, The Twenty Most Critical Internet Security Vulnerabilities, 2002. [Online]. Available: http://www.sans.org/top20/oct02.php

[15] CERT Coordination Center. CERT Coordination Center Advisories for 2003. [Online]. Available: http://www.cert.org/advisories/#2003

[16] SANS Institute. SANS Top 20 List, The Twenty Most Critical Internet Security Vulnerabilities, 2003. [Online]. Available: http://www.sans.org/top20/

[17] Scut. (2001, September) Format String Vulnerabilities. [Online]. Available: http://teso.scene.at/articles/formatstring

[18] T. Newsham. (2000, September) Format string attacks. [Online]. Available: http://www.lava.net/~newsham/format-string-attacks.pdf

[19] Blexim, "Basic integer overflows," *Phrack Magazine*, vol. 11, no. 60, December 2002. [Online]. Available: http://www.phrack.org/show.php?p=60&a=10

[20] S. Designer. (2001, January) Linux kernel patch from the openwall project: Non-executable user stack. [Online]. Available: http://www.openwall.com/linux/README

[21] The SmashGuard Group. (2003) SmashGuard Website. [Online]. Available: http://www.smashguard.org/

[22] J. Wilander and M. Kamkar, "A comparison of publicly available tools for static intrusion prevention," in *Proc. of the 7th Nordic Workshop on Secure IT Systems*, Karlstad, Sweden, November 2002, pp. 68–84.

[23] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, “A first step towards automated detection of buffer overrun vulnerabilities,” in *Network and Distributed System Security Symposium*, San Diego, California, February 2000, pp. 3–7.

[24] N. Dor, M. Rodeh, and M. Sagiv, “CSSV: Towards a realistic tool for statically detecting all buffer overflows in C,” in *Proc. of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, San Diego, California, June 9-11 2003, pp. 155–167.

[25] D. Larochelle and D. Evans, “Statically detecting likely buffer overflow vulnerabilities,” in *Proc. of the 10th USENIX Security Symposium*, Washington D.C., August 2001, pp. 177–190.

[26] E. Haugh and M. Bishop, “Testing C Programs for Buffer Overflow Vulnerabilities,” in *Proc. of the Network and Distributed System Security Symposium*, 2003. [Online]. Available: http://seclab.cs.ucdavis.edu/papers/HaughBishopNDSS2003.ps

[27] S. H. Yong and S. Horwitz, “Protecting C Programs from Attacks via Invalid Pointer Dereferences,” in *Proc. of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, Helsinki, Finland, September 2003, pp. 307–316.

[28] T. Toth and C. Kruegel, “Accurate Buffer Overflow Detection via Abstract Payload Execution,” in *Proc. of the 5th International Symposium on Recent Advances in Intrusion Detection*, Zurich, Switzerland, 2002. [Online]. Available: http://www.infosys.tuwien.ac.at/Staff/chris/doc/2002_08.ps

[29] S. Bhatkar, D. C. DuVarney, and R. Sekar, “Address obfuscation: An efficient approach to combat a broad range of memory error exploits,” in *12th USENIX Security Symposium*, Washington, D.C., August 2003, pp. 105–120.

[30] The PaX Team. (2001) PaX. [Online]. Available: http://pageexec.virtualave.net/

[31] M. Prasad and T. Chiueh, “A Binary Rewriting Defense against Stack based Buffer Overflow Attacks,” in *Usenix Annual Technical Conference, General Track*, San Antonio, TX, June 2003, pp. 211–224.

[32] Microsoft. (2001) Visual c++ option to tighten security. [Online]. Available: http://archive.devx.com/security/bestdefense/2001/mh0301/mh0301-1.asp

[33] 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 *Proc. of the 7th USENIX Security Conference*, San Antonio, TX, January 1998, pp. 63–78.

[34] C. Cowan, S. Beattie, R. F. Day, C. Pu, P. Wagle, and E. Walthinsen, “Protecting Systems from Stack Smashing Attacks with StackGuard,” in *Proc. of the 5th Linux Expo*, Raleigh, NC, May 1999. [Online]. Available: http://www.cse.ogi.edu/DISC/projects/immunix/lexpo.ps.gz

[35] Bulba and Kil3r, “Bypassing StackGuard and StackShield,” *Phrack Magazine*, vol. 10, no. 56, May 2000. [Online]. Available: http://www.phrack.org/show.php?p=56&a=5

[36] Vendicator. (2001, January) StackShield: A “stack smashing” technique protection tool for Linux. [Online]. Available: http://www.angelfire.com/sk/stackshield/download.html

[37] T. Chiueh and F. Hsu, “RAD: A Compile-Time Solution to Buffer Overflow Attacks,” in *Proc. of the 21st International Conference on Distributed Computing Systems (ICDCS ’01)*, Mesa, AZ, April 2001, pp. 409–.

[38] H. Etoh. (2003, April) GCC extension for protecting applications from stack-smashing attacks. IBM Research. [Online]. Available: http://www.trl.ibm.com/projects/security/ssp/

[39] (2003, April) The OpenBSD Project. [Online]. Available: http://www.openbsd.org/

[40] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, “Pointguard: Protecting pointers from buffer overflow vulnerabilities,” in *Proc. of the 12th USENIX Security Symposium*, Washington, D.C., August 2003, pp. 91–104.

[41] Various. OpenSSL. [Online]. Available: http://www.openssl.org/

[42] O. Ruwase and M. S. Lam, “A practical dynamic buffer overflow detector,” in *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS 2004)*, February 2004, pp. 159–169.

[43] A. Snarskii. (1997) FreeBSD Stack Integrity Patch. [Online]. Available: ftp://ftp.lucky.net/pub/unix/local/libc-letter

[44] ——. (2000, April) libparanoia. [Online]. Available: http://www.lexa.ru/snar/libparanoia/

[45] A. Baratloo, T. K. Tsai, and N. Singh, “Libsafe: Protecting Critical Elements of Stacks,” Bell Labs, Lucent Technologies, Murray Hill, NJ, Tech. Rep., December 1999. [Online]. Available: http://www.bell-labs.com/org/11356/libsafe.html

[46] A. Baratloo, N. Singh, and T. Tsai, “Transparent run-time defense against stack smashing attacks,” in *Proc. of the USENIX Annual Technical Conference*, San Diego, California, June 2000, pp. 251–262.

[47] T. Tsai and N. Singh, “Libsafe 2.0: Detection of Format String Vulnerability Exploits,” Avaya Labs, Avaya Inc., Basking Ridge, NJ 07920, Tech. Rep. ALR-2001-019, August 2001. [Online]. Available: http://www.research.avayalabs.com/techreport/ALR-2001-019-paper.pdf

[48] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier, “FormatGuard: Automatic protection from printf format string vulnerabilities,” in *Proc of the 2001 USENIX Security Conference*, Washington, D.C., August 2001, pp. 191–200.

[49] M. Frantzen and M. Shuey, “StackGhost: Hardware facilitated stack protection,” in *Proc. of the 10th USENIX Security Symposium*, Washington, D.C., August 2001, pp. 55–66.

[50] L. Torvalds. (1998, Aug.) Reply to non-executable stack patch. [Online]. Available: http://old.lwn.net/1998/0806/a/linus-noexec.html

[51] GNU Compiler Collection Internals. [Online]. Available: http://gcc.gnu.org/onlinedocs/gccint/Trampolines.html

[52] (2003, April) The OpenBSD 3.3. [Online]. Available: http://www.openbsd.org/33.html

[53] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer, “Architecture Support for Defending Against Buffer Overflow Attacks,” presented at the Workshop on Evaluating and Architecting System dependability (EASY-2002), San Jose, California, U.S.A., October 2002.

[54] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi, “Enlisting Hardware Architecture to Thwart Malicious Code Injection,” in *Proc. of the International Conference on Security in Pervasive Computing (SPC-2003)*, Boppard, Germany, March 2003.

[55] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, “Cyclone: A Safe Dialect of C,” in *Proc. of the 2002 USENIX Annual Technical Conference*, Monterey, CA, June 2002, pp. 275–288.

[56] T. Austin, S. Breach, and G. Sohi. (1994, June) Safe C Compiler (SCC). [Online]. Available: http://www.cs.wisc.edu/~austin/scc.html

[57] G. C. Necula, S. McPeak, and W. Weimer, “CCured: Type-Safe Retrofitting of Legacy Code,” in *Proc. of the ACM Symposium on Principles of Programming Languages*, Portland, OR, January 2002, pp. 128–139.

[58] D. M. Tullsen, S. J. Eggers, and H. M. Levy, “Simultaneous multithreading: maximizing on-chip parallelism,” in *Proc. of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995, pp. 392–403.

[59] T. Austin. (2001) SimpleScalar LLC. [Online]. Available: http://www.simplescalar.com/

[60] CERT Coordination Center. CERT coordination center statistics 1988-2002. [Online]. Available: http://www.cert.org/stats/cert-stats.html

[61] ——. CERT coordination center incident and vulnerability trends. [Online]. Available: http://www.cert.org/present/cert-overview-trends/

Hilmi Özdoğanoğlu is an Embedded Software Engineer at Broadcom Corporation. He received his masters degree from the school of Electrical and Computer Engineering at Purdue University in 2004. He graduated with a Bachelors degree from Louisiana State University in 2000. His research interests are Host-Based Computer Security Systems and Intrusion Detection Systems.

Carla E. Brodley is a professor in the Department of Computer Science at Tufts University. She received her bachelors degree from McGill University in 1985 and her PhD in computer science from the University of Massachusetts in 1994. From 1994-2004, she was on the faculty of the School of Electrical Engineering at Purdue University, West Lafayette, Indiana. Prof. Brodley's research interests include computer security, machine learning and knowledge discovery in databases. She has worked in the areas of intrusion detection, anomaly detection in networks, hardware support for security, classifier formation, unsupervised learning and applications of machine learning to remote sensing, computer security, and content-based image retrieval of medical images.

T. N. Vijaykumar joined the faculty of the School of Electrical and Computer Engineering in 1998 after completing his Ph.D. at the University of Wisconsin-Madison. His research interests are in computer architecture, VLSI microarchitectures, processor and memory hierarchy, and compiler optimizations. At Purdue, Vijaykumar investigates speculative threading, transient-fault tolerance, and low-power techniques in high-performance microprocessors.

Benjamin A. Kuperman is a visiting assistant professor at Swarthmore College in Pennsylvania. He earned his masters and doctorate from the Department of Computer Sciences at Purdue University in 1999 and 2004. While there, he was a researcher in the Center for Education and Research in Information Assurance and Security (CERIAS) for 5 years and was affiliated with COAST before that. His main areas of research are on host based computer security monitoring systems and OS level audit systems.

Ankit Jalote is a masters student in the Department of Electrical and Computer Engineering at Purdue University. He completed his bachelors in Computer Science and Engineering from Indian Institute of Technology, Kanpur (India) in 2002. His research interests are Computer Architecture and Security.