A Compiler-Hardware Technique for Protecting Against Buffer Overflow Attacks

Eugen Leontie, Gedare Bloom, Olga Gelbart, Bhagirath Narahari and Rahul Simha
Department of Computer Science
The George Washington University
Washington, DC 20052
Contact: eugen@gwu.edu

Abstract: Buffer overflow attacks are widely acknowledged by computer security professionals to be one of the greatest threats to the security of computer systems. We present an integrated software-hardware approach to protect against buffer overflow attacks while minimizing performance degradation, software development time, and deployment costs. Our technique does not change the processor core, but instead adds a hardware module in the form of a Field Programmable Gate Array (FPGA) that sits between cache and memory and that is able to defend return addresses from buffer overflow attacks. Our solution exhibits neither the performance overhead of software solutions nor the CPU redesign costs of hardware solutions.

Keywords: Software security, Buffer overflow, Computer architecture, Embedded design

1 Introduction

Buffer overflow (overrun) attacks continue to be among the most prevalent form of attacks on computer systems [1]. A buffer overflow occurs when data is written outside of the bounds of its allocated memory buffer. A buffer overflow is a technique that exploits vulnerabilities in languages that lack bounds checking on memory accesses, allowing an attacker to write malicious content past the end of a reserved memory region. Stack memory, which holds automatic variables and function return addresses, is a common target for buffer overflow attacks. Attacks that target buffers allocated in stack memory are extremely powerful, because overflowing a stack-allocated buffer enables an attacker to overwrite the return address. If the return address is overwritten, the return from that function no longer preserves the application's original control flow, instead branching to an arbitrary location determined by the attacker. If the attacker supplies the address of a buffer containing malicious code, then the CPU will execute the injected malicious code with the same privilege level as the original application. Thus, a buffer overflow in a superuser privileged application enables an attacker to gain administrative privileges after a successful attack.

Due to their prevalence, buffer overflow vulnerabilities have received a lot of attention in the literature. Past work on preventing buffer overflow attacks in unsafe languages includes software-only approaches, code instrumentation techniques, and hardware-assisted approaches. (We provide a summary of past work in Section 6.) Static software-only approaches do not prevent run-time attacks and more complex overflow attacks. Code instrumentation techniques support runtime detection of attacks but rely on operating system primitives and incur high performance overheads. Hardware assisted approaches improve on software schemes and result in low performance overheads, but require architectural modifications, often to the CPU and instruction set, thereby requiring a significant buy-in from chip manufacturers.

In this paper, we present a combined software-hardware approach to protecting the function return address that does not require a redesign of the processor core. On the software side, the compiler adds instructions to every function call and return; these added instructions interface with a new secure hardware module called a *Guard*. The Guard, which sits between the cache and main memory, can be implemented as reconfigurable logic such as a field programmable gate array (FPGA) that augments the CPU functionality, or simply as a gateway chip that interfaces the CPU with the system bus. We model our simulations with the FPGA method. Note that recent work has shown the effectiveness of using FPGA architectures to address specific security threats. For example [2] discusses an FPGA based module for intrusion detection. As a potential architecture platform for our solution, systems such as Xilinx Corporation's Virtex Pro family [3] provide a CPU core with an on-chip FPGA logic which can be used to implement our solution.

Our approach can be summarized as follows (with details in Section 3). The return address is copied to a memory region in the hardware Guard that is inaccessible through any direct memory calls. This memory region is called the *return address (RA) stack*, and it is automatically managed by a modified compiler. The compiler-inserted instructions to manage the RA stack synchronize the state of the RA stack with the program stack, independent of code locality and caching policies. On each function call (CALL), the return address is stored in the RA stack, and at each function return (RET), the Guard checks if an overflow has occurred and ensures that the correct address is returned to the processor. Thus, the Guard maintains a copy of the valid return address and ensures that the correct address is always returned regardless of any buffer overflows.

Our solution is efficient, and incurs low development and deployment costs. Code modification occurs in the compiler tool-chain, therefore the programmer's work is not increased. The same compiler technique can be made to work with binary instrumentation to retrofit legacy code and libraries, as long as binary dissassembly can identify the code regions and there are sufficient registers available for the extra instructions. Our modular Guard does not require changing the CPU and is physically located beyond the cache boundaries to minimize impact on memory access time. We tested our system using a suite of benchmarks – the average performance overhead is a modest 7%.

Although we have achieved good performance, our results are validated in an embedded environment. In particular, our technique is applicable to a wide range of applications that are statically linked and single-threaded. Embedded systems, which typically do not have a complicated operating system, would benefit greatly from our technique by providing built-in security for these networked and ubiquitous embedded computing devices. Furthermore, embedded systems are often programmed using low-level procedural langagues to minimize performance overheads, leading to fewer compute cycles for an embedded device to perform its task, thus providing a longer battery life. Such languages avoid bounds checking, which incurs non-negligible overheads, and thus they provide the possibility for buffer overflow attacks, so hardware support for preventing a successful attack is desirable.

The rest of the paper is organized as follows. In Section 2 we describe how overflow attacks occur and introduce our technique. Section 3 presents the details. Performance results are given in Sec-

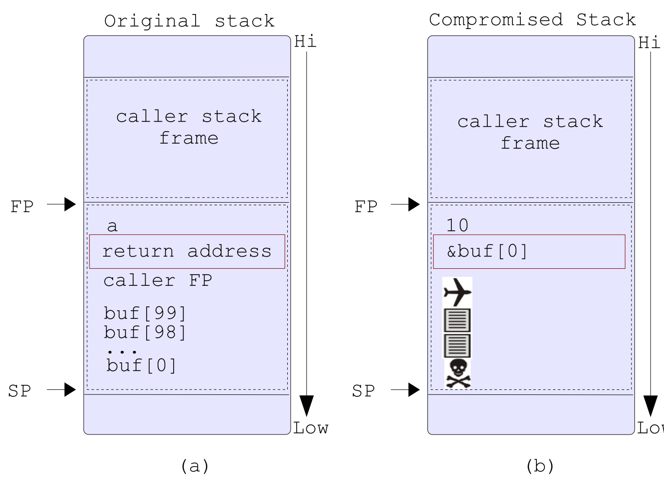

Figure 1: Stack layout for a function with a 100 words local buffer: (a) stack layout before attack; (b) the attacker's code is placed on the stack and the return address (RA) is overwritten to with the local buffer's start address

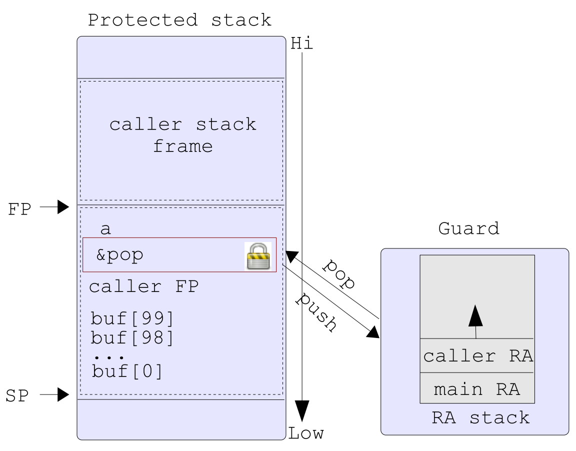

Figure 2: The push operation transfers the return address to the protected storage and the pop operation checks and restores the correct return address (see Section 3 for details)

tion 4, with discussions in Section 5 and related work in Section 6. Conclusions and future work are discussed in Section 7.

2 Overview

Buffer overflow attacks can be categorized based on where overflow data is written and how the attack vector affects the system. These categories are stack smashing, arc injection, pointer subterfuge, and heap smashing [4]. The original buffer overflow attack, as well as the majority of vulnerabilities reported, rely directly on stack smashing attacks [5] [6].

We first explain the vulnerability that enables stack smashing and then describe the stack smashing attack in more detail. Stack smashing is a simple concept that exploits the placement of the return address in close proximity to local function variables. The compiler determines the layout of these variables, which are stored on the program's stack segment. For each function call, the compiler allocates a memory region on the program stack. Function parameters, the caller's frame pointer, and the return address are copied on the stack. Stack space is also reserved for local (automatic) variables. Figure 1(a) demonstrates a stack layout for a simple function. A stack smashing attack proceeds in two steps, first an overflow that can change the return address and then controlling the execution to achieve the attackers intended actions.

The first part of a stack smashing attack is to cause an overflow to replace the return address value stored on the stack. Statically allocated buffers are extremely vulnerable to stack smashing attacks. If the attacker succeeds in providing the application an input data stream longer than a static buffer, the over-length input will write memory locations adjacent to the buffer. One of the locations vulnerable to overwriting contains the return address. Because the return address value is used for an implicit jump at the function return, the attacker can make arbitrary control flow redirections.

The second part of a stack smashing attack involves executing the attacker's code. In a simple scenario, the buffer that the attacker uses to overflow the stack contains executable code constructed by the attacker. By overwriting the return address value with the address of the buffer, upon the function return the attacker's code is executed; more complicated scenarios are also possible. Fig. 1(a) shows the stack layout during the execution of a function with vulnerabilities. An attacker first writes executable code into the buffer `buf` and continues writing past the original length (100 elements) allocated for the buffer. By overwriting the return address with the starting address of `buf`, the attacker diverts the program control flow such that the processor starts executing the injected code (Fig. 1(b)).

A simple defense mechanism is to prevent execution of data located in the stack memory region, thus preventing the injected code from executing. This is easily accomplished with non-executable memory regions, for example by using memory pages with no-execute (NX) permission bit or a Harvard architecture, in which data and instruction memory are explicitly separated. In response to such defense techniques, attackers have devised more complex attacks known as "return-to-libc", "return-oriented programming" [7, 8] or architecture specific attacks [39]. In the return-to-libc attack, attackers overwrite the return address with the location of a C library function, such as `exec()`, or `system()`. These functions accept parameters such as "/bin/sh", which will spawn a shell with the same privilege as the exploited application. A successful attack can potentially give the attacker the ability to run any commands on the vulnerable system with root privilege. Return-oriented programming involves creating a forged stack with a series of fake callers by injecting return addresses along with parameters to the stack. Thus, the attacker creates a sequence of operations that are executed by existing code based on the contents of the forged stack. Preventing the more complex variants of the stack smashing attack requires preventing the return address from being successfully overwritten, since the attack code is no longer injected onto the stack.

Our solution offers a compiler-hardware approach both to detect and to prevent overwriting of the return address. We store the return address in the Guard, and the compiler modifications are necessary to maintain the RA stack. The modified compiler adds instructions for each CALL and RET, as shown in Fig. 2. Our safeguards guarantee that if the return address on the program stack is overwritten, the hardware Guard detects the changed address and still provides the correct return address. Thus our solution provides both detection and prevention of stack smashing attacks. We discuss the details of the hardware and the added instructions in the next section.

3 Details: Protecting the Stack

In this section we describe the system architecture and implementation details for preventing and detecting stack smashing attacks. A simplified view of the functionality we implement to protect the return address is:

- Upon a function call, the return address is stored in the RA stack.
- Upon a function's return, the correct return address is provided by the RA stack.

In our solution, we store and manage the RA stack in a secure hardware component called a *Guard* which augments the existing processor-memory hierarchy, as shown in Figure 3. The additional

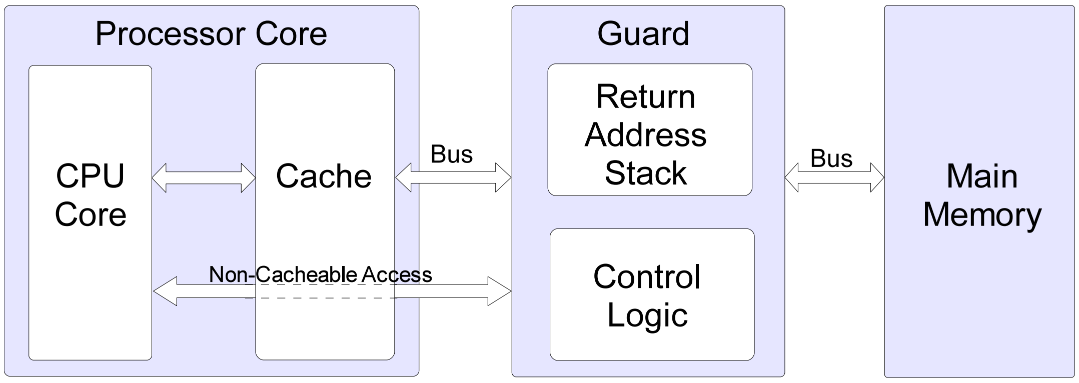

Figure 3: Processor with added hardware Guard. New functionality is implemented in the Guard, providing protection for unmodified CPU cores.

hardware is supported by compiler modifications which add new assembly-level instructions that can be implemented at the machine level using the existing ISA. A performance concern is that every function return needs to retrieve the correct return address, which is an access to the Guard (but not all the way to memory). We measure these performance concerns in our experiments, as described in Section 4.

The Guard implements part of the system bus architecture (e.g. the OPB [9] or HyperTransport [10]) so that it can monitor memory traffic for specific operations. Every instruction fetched by the CPU on an instruction cache (I-cache) miss goes through the Guard, thus exposing all instructions fetched by the CPU. Design limitations prevent deployment on devices which lack non-cacheable memory areas or which encapsulate the full memory hierarchy with the CPU. For such devices, or for better performance, the CPU core can be modified to support the Guard. Otherwise, the Guard can be implemented as reconfigurable logic in FPGA based processing cores or as a physical gateway on standard bus systems.

Introducing the Guard to an existing architecture does not require changing the processor's internal (pipeline) architecture. Indeed, the CPU does not need to be aware of the additional verification, which takes place outside the core boundary. An added benefit is that for reconfigurable cores, proprietary designs can remain black boxes.

The key to protecting the return address is that the Guard must be aware of both function calls and returns. We expose the CALL instructions by instrumenting the compiler to prepend each CALL with a memory-mapped "store to the Guard" instruction (`push_guard`) to push the function's return address to the Guard's RA stack. Similarly, another instruction (`notify_guard`) is inserted just before every RET to trigger the Guard verification. Pseudo-code for `push_guard` and `notify_guard` is given in Algorithms 1 and 2 respectively. The latency of these instructions is somewhere between a cache hit and a full memory access, potentially requiring an off-chip access to the Guard.

Algorithm 1 `push_guard`

```
Require: RET_Addr
Ensure: RET_Addr is stored on RA stack

  push(RET_Addr) to RA Stack
  if RA Stack is Full then
      Save RA Stack to Mem

  RET_Addr Register ← pop_ret_addr
  return
```

The `push_guard` instruction sends the correct return address (RET_Addr) to the Guard, which pushes the RET_Addr to its hardware stack. After the stack push, the Guard checks if the stack is reaching its capacity and will evict previous return addresses to memory if necessary. The instruction terminates by loading the pop_ret_addr into the processor's return address register.

Algorithm 2 `notify_guard`

```
Require: PC, FP
Ensure: Guard prepares RET_Addr

  RET_Addr ← pop() from RA Stack
  Prepare instruction: branch RET_Addr
  req ← wait_for_cachemiss() {req gets addr or timeout}
  while req <> pop_ret_addr do
      if invalid_mem_access(req, PC, FP) then
          Attack_Detected
      else if timeout_occurs then
          Attack_Detected
      else
          data ← fetch_from_mem(req)
          supply_to_cpu(data)
          req ← wait_for_cachemiss()

  supply_to_cpu(branch RET_Addr instruction)

  if RA Stack in Mem then
      Restore RA Stack from Mem
  return
```

The `notify_guard` instruction begins with the processor supplying the current program counter (PC) and frame pointer (FP). The Guard pops the correct return address (RET_Addr) from the top of its hardware stack and then prepares a branch instruction with branch target RET_Addr. Now the Guard must wait for the processor to issue a fetch for the instruction located at pop_ret_addr. Because the return can cause I-cache and data cache misses, a valid but bounded number of memory accesses may occur before the return finishes. Such a situation requires the Guard to validate memory accesses (using the PC and FP) and to have a timeout value. The timeout allows for execution time and data access times, and can change if the Guard observes valid cache misses. In the absence of an attack, the processor will request the instruction located at pop_ret_addr before the timeout. When that request occurs, the Guard will supply the crafted branch instruction, thus causing the processor to jump to the correct return address. As a cleanup operation, the Guard will check if some of its hardware stack should be restored from memory (due to previous eviction in `push_guard`).

Because the only modifications to the source code are made at CALL and RET locations, we may sometimes perform instrumentation directly on binaries, allowing for support of legacy applications and libraries distributed as binary images. Binary instrumentation is possible if sufficient registers are available to perform the `push_guard` and `notify_guard` instruction sequences, and if

```
foo:
        mov       ip, sp
        push_guard lr
           ;real return address(lr) is saved by Guard
        mov       lr, pop_ret_addr
        stmfd     sp!, {fp, ip, lr, pc}
          ;Regular function call preamble
          ;the address in lr will force a cache miss
        ...
        notify_guard(PC)
           ;signal that a function return is coming
        ldmea     fp, {fp, sp, pc}
           ;at this point pc contains pop_ret_addr
main:
        ...
        bl        foo
retAddr:
        ...

pop_ret_addr:
        b         retAddr
           ;since pc contains pop_ret_addr, we adjust
           ;it and jump to retAddr, the real address
           ;pop_ret_addr is mapped in the Guard space
```

Figure 4: Source-level instrumentation to add a cache miss on every function return (added instructions are in **bold**).

the code and data segments of the binary can be distinguished. In summary, the Guard contains the secure RA stack and necessary logic to examine and verify instruction addresses and memory loads. On every I-cache miss, the processor requests instruction blocks through the Guard. Both of the new instructions are implemented as memory-mapped operations, and we use non-cacheable addresses to force the Guard to process every function return. We next examine how push_guard and notify_guard are used by applications, and conclude this section with some extensions to provide protection for non-LIFO control flow.

3.1 Replacing the Return Address

The assembly code in Figure 4 shows an example of how the Guard inserts the correct return address. To guarantee an I-cache miss on a function's return, thus ensuring that the Guard verification takes place, we instrument the following steps:

1. The start of every function is modified to call push_guard, saving the real return address on the Guard's RA stack.
2. A non-cacheable address, called the pop_ret_addr, is selected and is written to the register holding the real return address.
3. Prior to the function's return, notify_guard is called, signaling the Guard that a function return is coming.
4. Within a bounded period of time, the CPU will issue a request for the instruction located at pop_ret_addr, which causes an I-cache miss.
5. Upon receiving the pop_ret_addr request the Guard will inject a branch instruction to the real return address which is located on the top of the RA stack.

The processor will thus jump to the correct address.

This scheme ensures that every function return is guaranteed to cause a cache miss, and that a correct return address (which was safely saved by the Guard) is restored, thus preventing buffer overflow attacks from redirecting control flow. The case of stack "spill-over" can easily be handled by the Guard without extra help from the operating system, as the overflow function return addresses can be stored encrypted in regular memory without having the operating system manage a separate secure memory. The Guard can also contain extra processing logic to handle non-LIFO situations, which we explain next.

3.2 Non-Lifo Control Flow

Performance optimizations and fast search algorithms sometimes use a non-lifo model of function returns. For example, a deeply nested tree search algorithm will use setjmp() and longjmp() to circumvent the long nested returns by returning directly to the search root when a result is found deep in the tree. setjmp() creates a buffer with the context of the current function, and longjmp() forces a return to that context, ignoring other contexts saved on the stack. Because this return mechanism will use the return value in the setjmp() buffer, and not a stack value, return address protection will trap this valid usage as an attack.

We solve the non-lifo problem by adding two memory mapped operations: push_setjmp and notify_lngjmp. setjmp calls push_setjmp to inform the Guard of its return address, and longjmp calls notify_lngjmp instead of notify_guard. The Guard reserves a special register (only one address is needed) and an index to keep the return address of the setjmp function, and replaces the setjmp buffer value with a pop_ret_addr to force a cache miss. The index stores the number of pops the hardware stack needs to execute in order to discard the return addresses circumvented by longjmp. Unfortunately, this re-instrumentation of setjmp()/longjmp() requires a recompilation of the program, so binary rewriting is harder to implement because the binary signature of setjmp() and longjmp() varies with the architecture, compiler, and libc version.

4 Experimental Results

To evaluate the performance overhead, we used the SimpleScalar [11] simulation suite and the gcc cross-compiler for the ARM processor. The benchmarks included programs from the MiBench [12] and Data Intensive Systems (DIS) [13], computationally intensive benchmarks commonly used to evaluate performance of architectures.

From MiBench [12] we used the following benchmarks: bitcount - tests the bit manipulation abilities of a processor by counting the number of bits in an array of integers; crc - checksum calculation for a file; dijkstra - an implementation of the graph algorithm for calculating the shortest paths between nodes; fft - Fourier transforms are used in digital signal processing to find the frequencies contained in a given input signal; sha - the standard secure hashing algorithm used in many security transactions; stringsearch search algorithm for given words in phrases using a case insensitive comparison algorithm; susan - an image processing suite with three variants–corners, edges, and smoothing. Field, pointer, transitive and update are data intensive benchmarks from DIS.

Because the protection technique operates on every function call, the performance penalty is highly dependent on the number of function calls for each program. Figure 5 shows the frequency of function calls with respect to other instructions for each of the benchmarks.

The performance penalty P per function call is

$$P_C = T_G + T_s,$$

where T_G is the access time to the Guard and T_s is the Guard's access time to the RA stack. These variables will change based on the speed of the CPU, the Guard, and the interconnect bus. In general, T_G will consume 1 bus cycle and 5 Guard cycles and T_s will consume 1 Guard cycle. How the Guard and bus frequency relate to the CPU frequency ultimately will determine the performance penalty. The performance penalty on a function return is

$$P_R = T_G + T_V + T_s,$$

where T_G and T_s are the same as before, and T_V is the time required by the Guard to verify the return address. T_V will consume 1 Guard

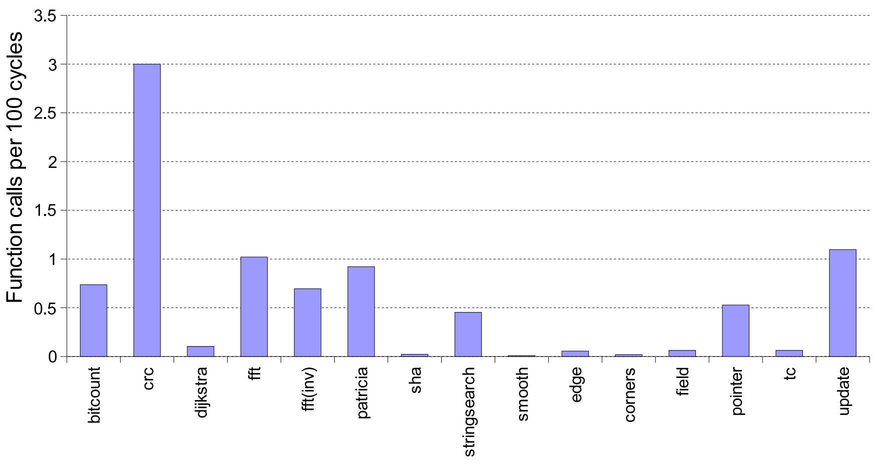

Figure 5: Frequency of function calls per benchmark

cycle. Finally, the overall performance penalty is

$$P = I + C_m + (N_C * P_C) + (N_R * P_R),$$

where I is the number of cycles consumed by added instructions, C_m is the number of cycles waiting on cache misses, N_C is the number of function calls, and N_R is the number of function returns. Thus P shows the extra execution cycles that are added to the baseline execution by the compiler (`push_guard` and `pop_guard`), the penalty of extra cache misses caused by the increased code size, and the verification penalty for each function call and return.

Our processor model has one level of instruction and data caches, each 32KB in size, 32-way associative, with 32-byte cache lines – changing cache size and associativity does not impact the overheads, so we only show results for a single fixed cache size. The simulator used was `sim-outorder`. The processor models an average embedded system: a 400 MHz CPU augmented with a 200 MHz FPGA (for the Guard), and an external bus and memory running at 100 MHz. Thus every Guard cycle takes 2 CPU cycles, and every bus cycle takes 4 CPU cycles.

Figure 6 shows performance penalties for each benchmark. On average, our scheme achieves a 7% performance penalty, with crc being a special case, containing the highest frequency of function calls and having a very high performance penalty.

5 Discussion

In this section, we describe some of the changes (or lack thereof) required to support systems that do not fit the model we assumed throughout this paper. Our scheme is general enough to work on most systems, although some parameters may need to be modified from system to system.

The `notify_guard` instruction is aided by the availability of the program counter (and frame pointer), but can still execute securely without access to either. At one extreme are processors that provide direct access to the control registers, such as the ARM and PowerPC, which are well suited for our approach. In such cases we can save and restore the original control flow by directly modifying control registers instead of using the artificial branch instruction generated by the Guard. At the other extreme are call and return processors, which do not give any other means of saving the control registers and need micro architecture level changes, defeating the non-intrusive nature of our design. In between are architectures such as the soft core Microblaze by Xilinx, which permit read only access to the control registers. This allows saving the return address, but requires the extra branch instruction for restoring the control flow.

The time between `notify_guard` and the request for `pop_ret_addr` will vary by architecture, and multiple layers of cache can provide an attacker with extra time to launch an attack, but will not allow any successful off-chip accesses, so the attack is limited. Some additional care is taken to guarantee tight timing restrictions between when `notify_guard` is issued and when the Guard expects a request for the `pop_ret_addr`. The `notify_guard` instruction passes the Guard the values of the program counter (PC) and frame pointer (FP) so that the Guard can verify subsequent memory requests as valid cache misses caused during the function return. If there is no request for `pop_ret_addr` in the bounded time, the Guard identifies that an attack happened and can halt execution. The timeout value is determined by the number of possible cache misses and cycle time required for the function return instruction to generate the request for new instructions at `pop_ret_addr`.

Systems that support execution reordering and multiprocessing may also require additional changes. Fortunately, in an out-of-order processor the memory mapped instructions are not executed until the speculative store buffer is allowed to commit, so the Guard is oblivious to the out-of-order execution and will not require any additional logic. However, OS modifications will be necessary for multi-threaded (and multiprocessing) environments, to interface with the Guard to allow for some secure administrative tasks, particularly to maintain context for the RA stack. Such an interface may require some changes to the timing mechanism in `notify_guard` to correct for OS overhead during context switches.

6 Related Work

A wide range of software-only approaches exist for buffer overflow prevention and detection that can be divided into static analysis tools and code instrumentation techniques. Some solutions in the first category are FlawFinder [14], RATS [15], PScan [16], BOON [17], ITS4 [18], MOPS [19], StackOFFence [20, 21], and PFD [22]. These approaches use source code analysis to detect potential vulnerabilities in the code before execution. The two main advantages of source code analysis are zero performance overhead at runtime and offline vulnerability detection. However, static methods do not protect from runtime attacks, and are prone to producing false positives. In the second category of software solutions, the best known are StackGuard [23] and ProPolice [24]. These approaches rely on the inability of an attacker to guess the value of a randomly generated "canary" value placed just before the return address on the stack. Unfortunately, more complex overflow attacks and format string attacks can reveal the value of the canary or change the return address without altering the canary.

Two code instrumentation techniques do support runtime detection of attacks and are particularly relevant because they both use a shadow stack protection mechanism similar to our RA stack. The

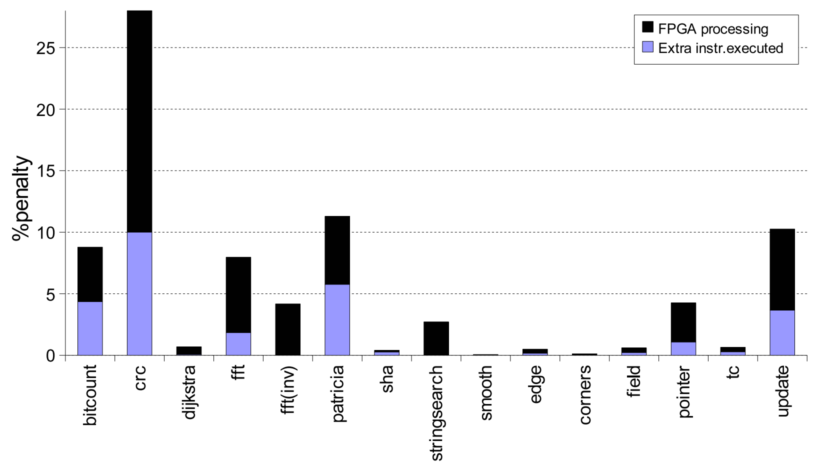

Figure 6: Performance penalty (%) per benchmark

first approach, the Return Address Defender (RAD) [25], performs source modification of each function's prologue and epilogue to include special instructions to store a copy of the return address in a particular area of the data segment. Protection of the shadow stack memory region is handled by the operating system. The second approach uses dynamic instruction stream editing (DISE) to perform the same modifications only using binary instrumentation [26]. The advantage of a scheme based on binary instrumentation is that source code does not need to be exposed to the protection mechanism. However, in order to make the RA stack read-only these schemes rely on OS primitives to manage page-table permissions which incurs high performance overhead. In some environments the software-only solution suffices, however, in the embedded world, the added overhead is unacceptable. Even worse, the OS and memory protection mechanisms might not even be available.

Hardware-assisted approaches to buffer overflow protection improve upon accuracy and performance of software-only schemes for dynamic attack detection by using a variety of techniques. The most closely related to our solution include approaches that shadow the return address in hardware by creating a return address stack or monitoring the location of the return address for any unauthorized modifications [27, 28, 29, 30, 31, 32, 33]. We will briefly review these closely related works. Other hardware-supported solutions tend to focus on protecting control-data in the general sense, including all branches and jumps, not just the return address [34, 35, 36, 37, 38].

Secure Return Address Stack (SRAS) [27] uses processor modifications, which includes ISA (Instruction Set Architecture) changes, additional logic, and storage space, to implement a shadow stack in hardware. Unlike the regular system stack, the shadow stack only holds return addresses. On a function call (CALL), the return address is pushed to the regular stack and the shadow stack. On a return (RET), the return address is popped from both stacks and compared. To handle function call nesting, the operating system is modified to handle secure overflow storage. The spill-over of the secure stack is stored in a special part of memory, which is accessible only to the kernel. The kernel is responsible for managing a separate spill-over secure stack for each process.

Several solutions for non-LIFO control flow are proposed for SRAS. Because `goto`s are generally considered a bad programming practice, they are not allowed. The `setjmp()/longjmp()` situation is handled in four ways. In the first case, non-LIFO flow is prohibited altogether. Second, if non-LIFO is detected then SRAS is disabled, allowing execution to continue without protection. Third, `setjmp()/longjmp()` executes, but only if the return is already somewhere on the secure stack; this requires the compiler to insert `sras_pus()` and `sras_pop()` functions. The fourth solution identifies the non-LIFO control flow situations dynamically and has separate handling and dynamic stack adjustments; this last approach works fine as long as all non-LIFO cases are known in advance (a new non-LIFO methodology would not be detected on the fly).

SmashGuard [28], like SRAS, uses a processor and operating system modification technique. The semantics of CALL and RET instructions are modified to store a function return address inside a special memory-mapped hardware stack and compared upon the function's return. All context switching and spill-over stack growth is handled by modifying the operating system to include additional security kernel functions.

The non-LIFO (`setjmp()/longjmp()`) situations are handled in SmashGuard through library rewriting, so that `setjmp()/longjmp()` is handled by a "jump-through-register" rather than regular jump. This creates stack inconsistencies upon return address verification, which are handled by popping addresses off the stack until reaching the correct address. If the end of the stack is reached, then an error or attack has occurred (i.e., the return address verification has failed), so the execution is halted. The drawback of this approach is that an attack, which forces a jump on an executable stack would succeed. Also, if a function is called multiple times in a loop, in order to pop the correct version of the function's return address off the stack–not just the first occurrence–SmashGuard also stores the corresponding value of the stack pointer.

Xu et al. [29] present a scheme that splits the stack into two pieces: the *control* stack, which holds the return addresses, and the *data* stack, which holds everything else. They propose a software solution, which involves compiler modification, and a hardware solution, which modifies the processor and semantics of CALL and RET. In the software-only solution, the compiler allocates and manages the additional control stack. During each function prologue, the compiler saves the return address to the control stack, which resides in memory securely managed by the operating system. The compiler restores the return address from the control stack onto the system stack in the function epilogue. Simulation yielded significant performance overheads, which led the researchers to a hardware solution.

The hardware redesign is very similar to SmashGuard and SRAS, although there are additional operating system changes to handle the two stacks. The processor is modified to change CALL and RET instructions to store and verify the function return addresses. However, even though an extra `jump_buf` structure is mentioned for `setjmp()/longjmp()` cases, the non-LIFO problem is currently not addressed, and neither is stack spill-over. Thus the authors claim no performance penalty for the hardware solution.

Secure Cache (SCache) [30, 31] uses cache memory to protect the return address, by providing replica cache lines in which the return address is shadowed. This way, a corrupted return address will have a different value in cache than its replica. By preventing the eviction of replica cache lines, the detection technique improves. In addition, the non-lifo cases do not cause a problem for SCache, because the return address is not explicitly checked against a known value. A drawback to using cache space for storing the return address is that the SCache approach is sensitive to cache parameters and behavior.

Kao and Wu [32] present a scheme to detect if the return address is modified without explicitly storing the good return address for verification. Two registers are added to store the current return address and the previous frame pointer. A valid bit that acts as an exception flag is also added. If a memory store is issued to either a return address on the stack or to the frame pointer, then the flag is raised to signal a violation. When the function returns, if the return address is to the local stack or to the current return address, then execution is halted. The authors claim that only two levels of function calls need monitoring. As with SCache, the non-lifo situations are not a problem in this scheme, because the return address is not explicitly checked; however, if non-lifo control flow is not detected, then a `longjmp` to a function that is exploited by a buffer overflow might remain undetected.

The existing hardware solutions for runtime return address protection naturally yield much smaller performance penalties than software-only approaches. To accomplish such improvement, processor modifications are made to include a hardware stack (or support for software shadow stack) and some additional processing logic. Our work avoids changing the processor and still exhibits low performance overhead.

7 Conclusions and Future Work

We designed and evaluated a scheme for guarding against buffer overflows that overwrite the return address on the stack. By using a small amount of unobtrusive hardware, we are able to achieve a number of advantages over prior solutions:

- No CPU or ISA modifications.
- Minimal OS modifications needed (only for context-switching support).
- Legacy and binary support via binary instrumentation.
- Support for non-LIFO control flow.

As hardware and software schemes become progressively more effective at protecting the return address, attackers are motivated to find other methods for diverting control flow. One method related to stack overflows is pointer subterfuge, which can be used multiple times to construct a complex attack that overwrites multiple pointer values to create corruption of function pointers or the jump table [40]. Future work will extend protection to attacks other than stack smashing and to general-purpose computing platforms, resulting in a more thorough and general protection mechanism.

Acknowledgment

This research is partially supported by NSF grants ITR-025207 and CNS-0934725, as well as FOSR grant FA9550-09-1-0194.

References

[1] CERT Coordination Center. http://www.cert.org.

[2] A. Das, D. Nguyen, J. Zambreno, G. Memik, A. Choudhary, “An FPGA-based intrusion detection architecture”, in *IEEE Trans. Information Forensics and Security*, vol.3, no.1, pp 118-132, 2008.

[3] Xilinx, inc. http://www.xilinx.com.

[4] Jonathan Pincus and Brandon Baker, “Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns”,*IEEE Security and Privacy*, pp. 20-27, Jul 2004.

[5] Aleph1, “Smashing the Stack for Fun and Profit,” in *Phrack Magazine*, vol. 7, issue 49-14, 1996.

[6] Benjamin A. Kuperman and Carla E. Brodley and Hilmi Ozdoganoglu and T. N. Vijaykumar and Ankit Jalote, “Detection and prevention of stack buffer overflow attacks”, *Communications of the ACM*, pp. 50-56, Nov 2005.

[7] H. Shacham, “The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)”, *In Proceedings of the 14th ACM Conference on Computer and Communications Security*,ACM, pp. 552-561, Oct 2007.

[8] Erik Buchanan, Ryan Roemer, Hovav Shacham and Stefan Savage ,“When good instructions go bad: generalizing return-oriented programming to RISC”,*CCS ’08: Proceedings of the 15th ACM conference on Computer and communications security*, ACM, pp. 27–38, 2008.

[9] Technical Reference, “On-Chip Peripheral Bus V2.0 w/OPB Arbiter Data Sheet”, available: http://www.xilinx.com.

[10] Hyper Transport Consortium, ”The future of High-Performance Computing: Direct Low Latency CPU-to-Subsystem Interconnect”. Whitepaper.

[11] T. Austin, E. Larson, D. Ernst, “SimpleScalar: An Infrastructure for Computer System Modelling”, *IEEE Computer*, pp. 59-67, Feb 2001.

[12] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, “MiBench: A free, commercially representative embedded benchmark suite”, *Proceedings of the* 4^{th} *IEEE Workshop Workload Characterization*, pp. 10-22, Dec 2001.

[13] J. Manke and J. Wu. Data-intensive system benchmark suite analysis and specification. *Atlantic Aerospace Electronics Corp*, 1999.

[14] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner, “Detecting Format-String Vulnerabilities with Type Qualifiers”, *In 10th USENIX Security Symposium*, Aug 2001.

[15] Rough Auditing Tool for Security (RATS), “Secure Software: Safer Source from Start to Finish”, available: http://www.fortifysoftware.com/security-resources/rats.jsp

[16] “PScan: A Limited Problem Scanner for C Source Files”, available: http://deployingradius.com/pscan/

[17] D. Wagner, J.S. Foster, E.A. Brewer, A. Aiken, “A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities”, *Network and Distributed System Buffer Overflow Symposium*, 2000.

[18] J. Viega, J.T. Bloch, T. Kohno, G. McGraw, “Token-based Scanning of Source Code for Security Problems”, *ACM Transactions on Information and System Security*, Vol. 5, No. 3, pp. 238-261, Aug 2002.

[19] D. Wagner, H. Chen, “MOPS: An Infrastructure for Examining Security Properties of Software”, *Proceedings of the* 9^{th} *ACM Conference on Computer and Communications Security*, pp. 235-244, 2002.

[20] B. B. Madan, S. Phoha and K. S. Trivedi, “StackOFFence: A Technique for Defending Against Buffer Overflow Attacks”, *International Conference on Information Technology: Coding and Computing*, 2005. (ITCC 2005), vol. 1, pp. 656–661, Apr 2005.

[21] Bharat B. Madan, Shashi Phoha and Kishor S. Trivedi, “Stack Overflow Fence: A Technique for Defending Against Buffer Overflow Attacks”, *Journal of Information Assurance and Security (JIAS)*, vol. 1, issue 2, pp. 129-136, 2006.

[22] Prattana Deeprasertkul, Pattarasinee Bhattarakosol, Fergus O’Brien, “Automatic detection and correction of programming faults for software applications”, *Journal of Systems and Software*, vol. 78, issue 2, pp. 101-110, Nov 2005.

[23] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, “StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks”, *Proceedings of the 7th Conference on USENIX Security Symposium*, 1998.

[24] Eto Hiroaki, Yoda Kunikazu, ProPolice,*Transactions of Information Processing Society of Japan*, vol.43, no.12, pp 4034-4041, 2002.

[25] T. Chiueh and F. Hsu, “RAD: A Compile-Time Solution to Buffer Overflow Attacks”, *Proceedings of the* 21^{st} *International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, Arizona, USA, Apr 2001.

[26] M. L. Corliss, E. C. Lewis, and A. Roth, “Using DISE to protect return addresses from attack” in *SIGARCH Comput. Archit. News 33*,pp 65-72, Mar 2005.

[27] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi, “Enlisting Hardware Architecture to Thwart Malicious Code Injection”, *Proceedings of the International Conference on Security in Pervasive Computing*, Boppard, Germany, pp. 237-252, 2003.

[28] H. Ozdoganoglu, T. Vijaykumar, C. Brodley, B. Kuperman, and A. Jalote, “SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address,” *Computers, IEEE Transactions on*, vol. 55, pp. 1271-1285, 2006.

[29] J. Xu, Z. Kalbarczyk, S. Patel, and R.K. Iyer, “Architecture Support for Defending against Buffer Overflow Attacks”, *Proceedings of the* 2^{nd} *Workshop Evaluating and Architecting System Dependability (EASY-2002)*, Oct 2002.

[30] K. Inoue, “Energy-security tradeoff in a secure cache architecture against buffer overflow attacks”, *SIGARCH Comput. Archit. News*, vol. 33, Issue 1, pp. 81-89, Mar 2005.

[31] K. Inoue, “Lock and Unlock: A Data Management Algorithm for A Security-Aware Cache”, *IEEE International Conference on Electronics, Circuits and Systems (ICECS’06)*, pp. 1093-1096, 2006.

[32] Wen-Fu Kao, S. Felix Wu,“Light-weight Hardware Return Address and Stack Frame Tracking to Prevent Function Return Address Attack”, *The 2009 IEEE/IFIP International Symposium on Trusted Computing and Communications (TrustCom-09)*, 2009.

[33] Zili Shao, Jiannong Cao, Keith C.C. Chan, Chun Xue, Edwin H.-M. Sha, “Hardware/software optimization for array & pointer boundary checking against buffer overflow attacks”, *Journal of Parallel and Distributed Computing*, Volume 66, Issue 9, Security in grid and distributed systems, pp 1129-1136, Sep 2006.

[34] J. R. Crandall and F. T. Chong. “Minos: Control data attack prevention orthogonal to memory model”, In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture* (Portland, Oregon). pp 221-232, Dec 2004.

[35] A. Smirnov and T. Chiueh. “DIRA: Automatic detection, identification and repair of control-data attacks”, In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb 2005.

[36] G. Suh, J. Lee, and S. Devadas. “Secure program execution via dynamic information flow tracking”, In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. Boston, MA., Oct 2004.

[37] J. Zambreno, A. Choudhary, R. Simha, B. Narahari, N. Memon. “SAFE-OPS: An approach to embedded software security”. *ACM Transactions in Embedded Computer Systems*, vol. 4, issue 1, pp 189-210, Feb 2005.

[38] M. Budiu, Ú. Erlingsson, M. Abadi. “Architectural support for software-based protection”, In *Proceedings of the 1st Workshop on Architectural and System Support For Improving Software Dependability* (San Jose, California). ASID ’06. ACM, New York, NY, pp 42-51, Oct 2006.

[39] A. Francillon and C. Castelluccia, “Code injection attacks on harvard-architecture devices” *In Proceedings of the 15th ACM Conference on Computer and Communications Security*, pp 15-26, Oct 2008.

[40] K. Piromsopa, and R.J. Enbody, “Defeating buffer-overflow prevention hardware”, *In 5th Annual Workshop on Duplicating, Deconstructing, and Debunking*, WDDD 2006.

Authors Biographies

Eugen Leontie received his M.S in 2005, focusing in Advanced Computer Architectures, from Politehnica University of Bucharest, Romania, where he also obtained his B.S in 2004. He is currently working towards his PhD degree at George Washington University, Washington, DC. His research interests include embedded system designs, secure hardware architecture, cryptography and software and information protection.

Gedare Bloom received the B.S. degree in computer science from Michigan Technological University, Houghton, MI in 2005, and is currently pursuing the Ph.D. degree in computer science at The George Washington University. His research interests include software security, system security, operating systems, computer architecture, and distributed systems.

Olga Gelbart (D.Sc.08) received her B.S. degree magna cum laude in 1997, M.S. degree in 1999 and D.Sc. degree in 2008 (all in Computer Science) from the George Washington University, Washington, DC. Her research interests include computer security and cryptography with the focus on compiler techniques and hardware-software techniques for software protection. She currently holds a position of Computer Scientist at the US Naval Research Laboratory, Washington, DC, where her latest work also includes security data visualization. Dr. Gelbart has been the recipient of several honors and awards, including membership in the Tau Beta Pi engineering honors society. She is a professional member of the ACM and the IEEE.

Bhagirath Narahari is a Professor of Computer Science at The George Washington University. He received the B.E. degree in electrical engineering from the Birla Institute of Technology and Science, Pilani, India in 1982, and the Masters and Ph.D. degrees in Computer Science from the University of Pennsylvania, Philadelphia, in 1984 and 1987 respectively. He is currently a Professor of Computer Science at The George Washington University, where he was the Department Chairman from 1999-2002. His research interests lie in the areas of embedded systems, computer architecture, software security, and parallel computing.

Rahul Simha is Professor of Computer Science at The George Washington University. He received his PhD in Computer Science from the University of Massachusetts in 1990. His research interests include embedded systems, compilers, languages, architecture, security and complex systems.