Abstract

VAIDYANATHAN, ANURADHA

Poseidon: Hardware Support for Buffer Overflow Attacks

(Under the direction of Dr. Gregory T. Byrd)

Stack smashing attacks were the most exploited security vulnerability in the past decade, according to CERT. Using a method called stack smashing, a malicious user overflows a buffer in the stack frame, overwriting critical stack state. The return address of the current function, which is saved in the function's stack frame, is overwritten when the buffer overflows. The new return address points to the attacker's code. So, when the function is exited, control is transferred to the attacker's code instead of back to the calling function.

A common way to prevent overflow-based stack smashing is to insert bounds checking code or insert sentinel values on the stack, but this requires recompilation. We propose a hardware-based method that does not require recompilation, based on the idea that an attack of this kind produces an unexpected return address. The processor maintains a separate hardware stack, called the shadow stack, and monitors the dynamic instruction stream for subroutine calls and returns. When a call instruction is retired, its return address is pushed on the shadow stack. When a return instruction is retired, the address at the top of the shadow stack is popped and compared to the target of the return instruction. If the addresses differ, then the conventional subroutine call/return semantics have been violated. This may truly be an attack, or it may be a legitimate program construct (e.g., setjmp()/longjmp()) that also violates call/return semantics. Legitimate

cases are distinguished from attacks by recording the stack pointer along with the return address at the time of a call: when a subroutine returns, the stack pointer appears consistent in the case of an attack but not in the case of setjmp()/longjmp().

There are three distinct parts to the evaluation of the usefulness and the practicality of this idea. The first part is identifying the *generality* of this solution. This means that we seek to answer the question: “Do we detect all forms of buffer overflow attacks without raising unnecessary false positives in the case of legal program constructs?” The second part is the actual design details of such a stack and the amount of state that needs to be recorded to facilitate the generality described above. The third part is the actual recovery mechanism that could take the form of exceptions raised that could be further handled by the Operating System. This thesis answers the generality and design questions in entirety while laying a solid basic understanding and initial set of experiments for the recovery scheme that could be utilized.

POSEIDON: HARDWARE SUPPORT FOR BUFFER OVERFLOW ATTACKS

by

ANURADHA VAIDYANATHAN

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

COMPUTER ENGINEERING
Raleigh
2002
Approved by

Dr. Gregory T. Byrd, Chair of Advisory Committee

Dr. Thomas M.Conte

Dr. Alex Dean

Dr. Frank Mueller

BIOGRAPHY

Anu Vaidyanathan was born on a fine sunny day, in a fine sunny city in India. After leading most of her life in blissful oblivion, she faced formal education with much ado. It has left her beaten and wondering if all the math she did for all these 10+ years is going to give her all the answers she needs.

ACKNOWLEDGEMENTS

I tried the verbose, three page thanks. I tried the silent, one-liner. I tried to convince the thesis editor that I didn't want one at all. In the end, I concluded, no effort (specially considering the celerity of this one) can be attributed to just me. I would like to thank mom and dad and Arun. These are people who will love me no matter what, who will try to blackmail me to get married one way or another (Im growing old, quit school, get real being a few of the lines) and who will pray for me and lean on me and hold my hand when the going gets tough. Thank you people. I would like to thank Karen and Terry Buxton for being my family out in NC, for motivating me to always do my own thing, to go run and play instead of code. And Marty and Liz for being supportive of all my bad hair/ no hair GI-Jane days and introducing me to concepts like lava lamps. I would like to thank Prof. Mark Bell at Purdue for being a shining example of all that is true and free. Him and his wife Cathy have been wonderful to me for no good reason really – true love and intellectualism are the two things I think about when I am with these people. I would like to thank Dr. Vijaykumar for being an enthusiastic professor and telling me about the wonderful land called Wisconsin. I would like to thank Dr. Babak Falsafi for singing songs from Sesame Street and being the one other person to roam the halls of the ECE building well after midnight in the quest for coke. These two people bought architecture alive for me. I would like to thank my wonderful frends: Aurora Tiffany-Davis, Zach Purser, Ben Welch, Stephanie Pruitt, Chloe Palenchar, Amber Moore, Jeanette, Joshua Starmer, Saurabh Dash and Peter Harrington for always being around to get me out of my blues, go drink a beer with or just plain drink coffee with and yell about Israel after a night of no sleep. I would like to thank my advisor Dr. Byrd for his faith in me and for treating me like his own kid. I would like to thank Dr. Jim Smith for putting up with me for no reason at all and joining the gang of anti-TV-nazis (Me, Josh Starmer, Dr. Bell and him should start something soon). I would like to thank Sandy Bronson for being my pseudo mom and listening to hours of ranting and raving. I would like to thank Eric Rotenberg for providing me an experience in graduate school no student can ever forget. Finally (phew!), I would like thank everyone else that I left out – maybe if you called me more often, Ill put you down on other such pieces of crap I write J

கற்றது கை மண்ணளவு கல்லாதது உலகளவு

ஔவையார்

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Smashing the stack is a popular exploit that is a consequence of a particular vulnerability in modern programs called buffer overflows. Semantic insufficiencies of popular programming languages such as C, coupled with shortcomings of operating systems such as UNIX*, serve as a conduit for a malicious user to gain control of a remote machine [8]. By overflowing stack-allocated buffers whose bounds have not been verified, the stack state of a process may be altered. The vulnerable buffer is filled with more entities than it was declared to handle, thereby overriding the return address [11]. The state that is of interest to an attacker is the return address of the program or subroutine containing the buffer that can be overflowed. By modifying the return address of a privileged process to point to code that the malicious user wishes to execute, typically spawning a shell in root mode, the remote computer is taken over. Many modern programs running on UNIX and Windows environments have been known to have buffers with inadequate bounds checking. Secure Shell [2], Rundll32.exe [12] and lprm [1] are a few examples of vulnerable programs – there are many more. While individual buffer overflow vulnerabilities are easy to circumvent with patches, the vulnerabilities are present among millions of lines of legacy code, most of them written in C, which continue to run as privileged daemons on many machines. This motivates a more generic mechanism for detecting and preventing attacks that are caused due to them. Current solutions take the form of a patch for a particular attack on a particular

* UNIX allows a program running in root mode superuser privileges

program, safe compilers, which insert sentinel values to maintain stack integrity [18] or kernel fixes that are specific to a certain architecture and operating system[29]. Patches are not exhaustive in their detection of these vulnerabilities, safe compilers require a massive effort at recompiling a lot of code, and specific fixes do not address all architectures and operating systems.

In this thesis, we present a hardware scheme that detects stack-smashing attacks without the need for recompilation, based on the observation that the attack produces an unexpected return address. Since the proposed scheme is made part of the pipeline, it is not specific to any particular combination of architecture/operating system. The processor maintains a separate hardware stack, called the shadow stack, and monitors the dynamic instruction stream for subroutine calls and returns. The returns of the subroutine calls usually match up with the calls themselves [16] and maintain LIFO ordering. When compiled for the Simplescalar ISA [24], subroutine calls are Jump-And-Link or Jump-And-Link-Register instructions (JAL/JALR), and returns are jump register instructions (JR) whose only source operand is register 31. Some architectures may not comply with this model and might have to be modified to accommodate detecting returns. The shadow-stack is modeled as a LIFO buffer that is updated with some state associated with the instruction at the retirement stage. This state would typically include the Next_PC for the call instructions and the return addresses of the return instructions along with the Stack Pointer at the time the instructions were dispatched into the pipeline. Upon seeing a return instruction, the shadow stack is popped and the recorded value of the return address is compared to the target of the return instruction. If the addresses differ, then the conventional subroutine call/return semantics have been violated.

There are two scenarios in which the address popped from the shadow stack does not match the target of the return instruction. First, this may truly be an attack. Second, the difference in return addresses could be caused due to encountering program constructs that also violate call/return semantics, such as setjmp()/longjmp(). Setjmp() and Longjmp() are program constructs that have to do with saving and restoring environment variables. Setjmp() typically saves the environment variables (such as stack-pointer etc.) in certain registers in the register file. Longjmp() reinstates those environment variables when it is executed. Legitimate cases are differentiated from attacks by recording the stack pointer along with the return address at the time of a call: when a subroutine returns, the stack pointer appears consistent in the case of an attack but not in the case of setjmp()/longjmp() or non-local returns. This is made clear in Fig. 3.3.

There are other legitimate cases that are not taken into account and are beyond the scope of this thesis. These would include very simple code generators for stack-machines, like Kaffe[30] simply generate stack-modifying code, i.e. lots of loads and stores. The proposed technique might run into problems with dynamically generated code because, the glue code which interfaces the compiler code with the generated code usually modifies the stack in several ways. Bintrans[32], for example, often uses subroutine jumps which never return, making the stack grow beyond all reasonable limits. Another problem area might be interpreters for languages with first-class continuations, like guile for scheme[33]. Most of them simply copy the stack whenever a continuation is created and switch to that stack when it is invoked, hence breaking the LIFO pattern. These might be overcome by recording more pertinent state onto the stack but, the answer to what that state might be is not answered here. Yet another consideration should be programs which implement their own thread switching, usually triggered by hand (cooperatively) or by an alarm signal (preemptively).

Many interesting design questions present themselves in the implementation of a shadow stack. We discuss the important issues in the actual implementation of a feature like the shadow stack in modern superscalar pipelines. First, the merits of updating the shadow stack at the retirement stage are discussed. Second, propagating information like the value of the stack pointer when an instruction is fetched is explained. We would prefer to avoid register-file reads (to access the stack pointer perhaps) at the retirement stage. Thus, the stack pointer would have to be recorded at the time of fetching a call instruction and passed down the pipeline. Third, the shadow stack needs to be able to raise exceptions when four different events occur. In order to facilitate these exceptions, the shadow stack needs to be visible to software. The state that needs to be visible is also discussed.

1.1 Contributions

This thesis makes the following contributions:

- Detection of this vulnerability with the aid of a hardware structure and the design issues for such a structure.
- Recovery mechanisms in the case of the detection of an attack.
- Legitimate program constructs that need to be addressed in order to make this scheme practical.

1.2 Organization of the thesis

Chapter 2 gives an overview of the buffer-overflow security vulnerability. The mechanisms to implement a shadow stack, detect a buffer-overflow attack, design issues to be considered and the recovery methods thereof are described in Chapter 3. Chapters 4 describes the experiments that were run as a part of this work in two different environments and presents rationale and results for the same. Some of these experiments include recording the size to which the shadow stack is typically seen to grow, the performance impact of having a shadow stack at the retirement stage of the pipeline and recording popular rewind lengths (explained later) for the shadow stack. Chapter 5 discusses related work. Chapter 6 talks about practicality issues for the proposed idea and the other considerations that we made as part of this work. Chapter 7 wraps up with conclusions and future work.

Chapter 2

Description of the vulnerability

We begin by describing how the stack of a vulnerable program can be smashed. We demonstrate, by means of simple examples, how arbitrary code can be placed within the memory space of the vulnerable program to be executed.

2.1 Smashing the stack

When an executable file is loaded, the text segment is loaded into memory first, followed by the data area, followed by the stack. The stack is a part of the Process Control Block. A typical PCB for UNIX is shown in Fig. 2.1.

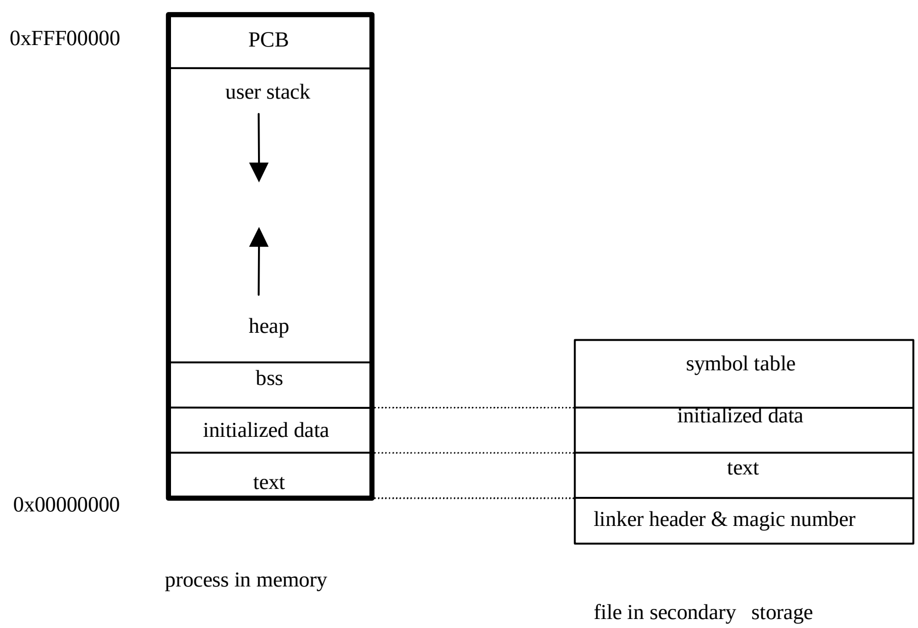
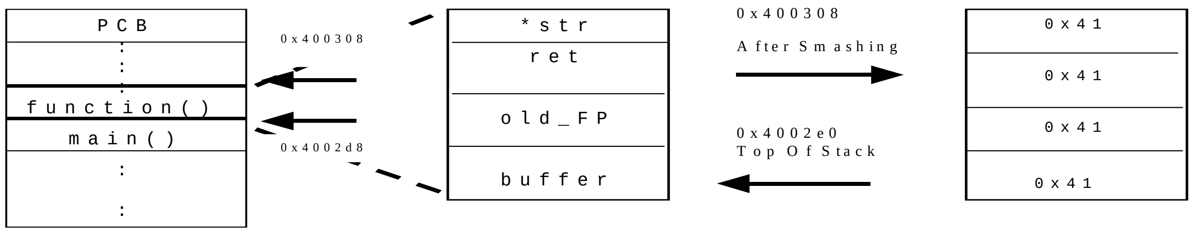


FIGURE 2.1. UNIX process in primary and secondary storage

The stack is composed of a series of logical stack frames, each corresponding to a subroutine. The stack is used for variables and control state that are pushed when a subroutine is called and popped when a subroutine returns. When a stack frame is pushed onto the stack, the stack grows downward in memory (Fig. 2.2a). The contents of the stack frame for individual subroutines are the parameters to the subroutine, the local variables and a return address to restore context to the previous stack frame (Fig. 2.2b). The sizes of the various fields of the stack are not the same, for example, buffer[] in this example is 256 bytes long but, we just show one entry representative of the field. Processors usually provide a stack pointer register (SP) that is used to point to the top of the stack [3]. SP contains the address of the top-most element of the stack. The bottom of the stack is at a known address and the size of the stack is dynamically adjusted at runtime. In addition to the stack pointer that points to the top of the stack, there also exists a frame pointer, FP (sometimes called a local base pointer, LB). This pointer addresses the base of the current frame. Assuming that stack growth is downwards in memory (as shown in Fig. 2.1), parameters have a positive offset with respect to the frame pointer and local variables have a negative offset. Local variables include buffers, so we need to be concerned about buffer overflows corrupting the stack.

EXAMPLE.1

```
void function(char *str) {
        char buffer[5];
        strcpy(buffer,str);
}
void main() {
        char large_string[256];
        int i;
        for(i=0;i<255;i++)
                large_string[i] = 'A';
function(large_string);
}
```



(a) Stack Frames in memory corresponding to different subroutines

(b) Stack Frame corresponding to `function()`

(c) `function()`'s Stack Frame after smashing

FIGURE 2.2. Stack state for Example 1.

2.2 Buffer Overflows

A buffer overflow is a consequence of adding more data to a buffer than it has been declared to handle. Example 1 illustrates a simple buffer overflow that causes a segmentation violation. A snapshot of the stack when this function is called is shown in Fig.2.2b. During execution, strcpy() copies the contents of *str which points to large_string[]into buffer[], until a null character is found in the source string. However, buffer[] is much smaller than large_string[]. A 256-byte buffer is being copied into a 5-byte string. After buffer, 251 bytes in the stack are overwritten. This includes the old_FP, ret, and even *str. Since large_string[] was filled with a series of A's (whose hex value is 0x41), the value of ret (or the return address) is now 0x41 (Fig.2.2c). This is external to the process space. When the function returns, it attempts to read the instruction at that location, leading to a segmentation violation. This demonstrates that the return address of a function can be overwritten via a buffer overflow. This common programming error can be exploited to execute arbitrary code, for example, code to spawn a shell from the process space of a privileged program. The next step is to use this vulnerability to execute arbitrary code. To demonstrate this process, we refer to Example 2.

2.3 Executing Arbitrary Code

EXAMPLE. 2a

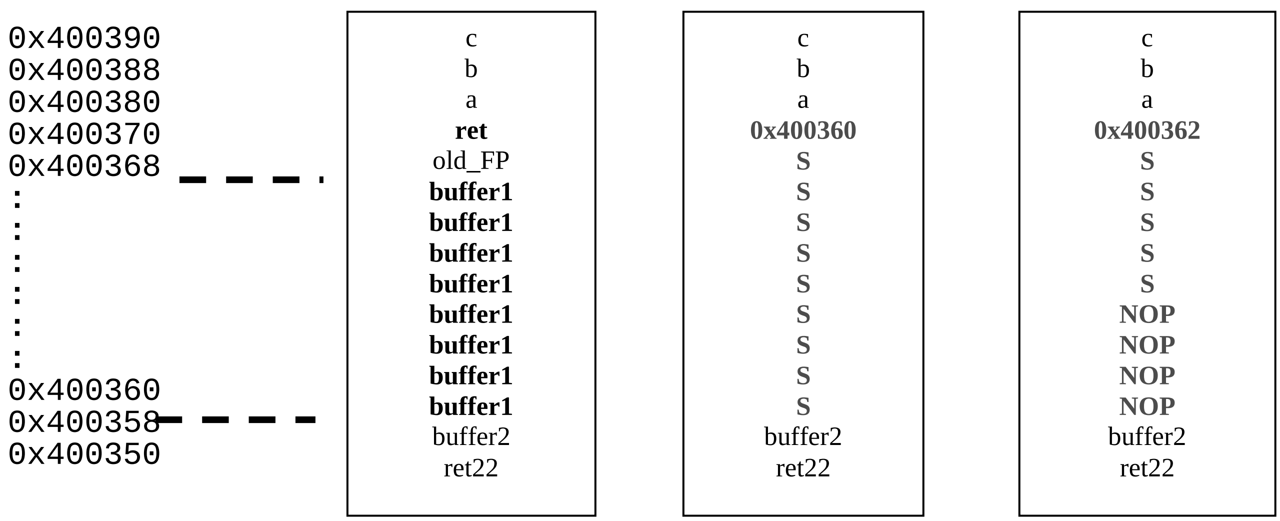
```
void function(int a, int b, int c)
{
   char buffer1[5];
   char buffer2[10];
   int *ret22;
}

void main()
{
   function(1,2,3);
}
```

EXAMPLE. 2b

```
void function(int a, int b, int c)
{
   char buffer1[5];
   char buffer2[10];
   int *ret22;
   ret22 = buffer1 + 12;
   // added 12 to buffer1[]'s
      //address
   (*ret22) += 8;
}

 void main()
{
   int x;
   x = 0;
   function(1,2,3);
   x = 1;
  printf("%d\n",x);
}
```



(a) Stack Frame for `function()` in Ex. 2a

(b) Stack Frame after overflowing `buffer1[]` to contain shellcode

(c) Overflowing `buffer1[]` with a shellcode padded with NOPs

FIGURE 2.3. Stack state for Example 2.

Upon examining the stack for Example 2a (Fig 2.3a), we see that just above buffer1[] lies the old frame pointer, before which lies the return address. Location of the return address is four bytes prior to the end of buffer1[]. But, buffer1[] is really two words long (5 bytes = 2 words) and therefore 8 bytes long. This places the return address 12 bytes away from the start of buffer1[]. The goal of the code shown in Example 2b is to modify the stack so that the assignment statement (x = 1;) after the function call will be jumped. To achieve this, 8 bytes are added to the return address. The modified code is shown in Example 2b. When this code is executed, we see 0 on the screen from the printf().

The next step (after changing the return address) would be to place arbitrary code to be executed, at that return address. Note that we need to place this “other code” (usually “exec(“/bin/sh”)”) in the address space of the program whose stack we have smashed. In order to do this, we could place that code in the buffer that we are overflowing (in our case, buffer1[]), and overwrite the return address so that it points back into that same buffer. Assuming that ‘S’ stands for the code we are trying to execute, the stack would now look as in Fig.2.3b.

2.4 Riding on the runway

What was shown above was a simple example. In most cases, we do not know that the value is going to be ‘8’, i.e., we do not know the exact offset at which the shellcode is to be placed (as this would involve guesstimating [9] using a debugger). To circumvent this issue, hackers employ what is called a runway. If the buffer could be added in some fashion with a series of NOP instructions, then roughly estimating the region of the code would be sufficient. Thus, when placing the program code in memory, the first few instructions are NOP instructions. Since most processors have a NOP instruction defined in their instruction set, this is not a problem. Referring to Example 2, we could pad the first few characters of the string that we are using to overflow buffer1[]with the NOP instruction. The stack in that case would look as shown in Fig. 2.3c.

A naïve method to detect a stack-smashing attack in the light of this information (i.e, the use of a runway to estimate where to place the mutant code) would be to monitor the dynamic instruction stream for a series of NOPs. In the case of these attacks, the number of NOPs is usually in the range of 200 or more [9]. This however is not a very robust detection mechanism for a couple of reasons. First, the number of NOPs can be arbitrary thereby causing difficulties for setting a lower limit of NOPs that can be encountered before raising an attack exception. Second, NOPs can arbitrarily appear in the code for a variety of reasons[25].

2.5 Attack Model

In this work, we assume that the attacks are primarily stack smashing attacks due to buffer overflow vulnerabilities in programs. This would mean that the attacker uses some vulnerable buffer in a pre-existing daemon or application to cause the attack. The way that would be accomplished is typically by forcing that buffer to hold more information than it has been declared to handle, thereby changing the return address of the stack associated with that program/process. The attacker is not assumed to have control over the binary or be able to modify the dynamic instruction stream in any way. He would also be incapable of modifying the register file.

Chapter 3

The Shadow Stack

This section discusses the design and implementation details of the Shadow Stack. This includes the basic operation of the stack, the detection mechanism used to detect a buffer overflow attack, the implementation issues that arise for both the detection mechanism and the design of the stack and finally the recovery schemes proposed. This discussion is based on the semantics of the Simplescalar ISA[24] with the binaries compiled using gcc.

3.1 Basic Stack Operation

To give some basic insight into the functioning of this stack structure, we discuss the way the stack might be modeled. In our experiments, the shadow stack is modeled as a LIFO structure that can carry out the following operations:

push() – pushes the entry onto the stack.

pop() – pops the entry at the top of the stack and sets the top of stack pointer to the next entry.

top() – examines the top of the stack.

full() – examines whether the stack has reached its upper-bound in terms of entries.

empty() – examines whether the stack has been emptied out.

clear() – clears the entire stack and purges all the entries.

Call instructions, either jump-and-link or jump-and-link-register (JAL/JALR) are made to push the return address (the next PC), onto the shadow stack. Return instructions, usually identified as an indirect jump-register (JR) instruction whose source operand is register 31, pop the top of the shadow stack. If returns are indirect jump instructions (JR) that use a particular register that other indirect jump instructions do not use, they are easily identified and can pop values from the shadow stack accurately. Some compilers do not store the return address in a specific register and/or use an explicit return instruction. These might have to be tweaked for detecting returns accurately. The fields of the shadow stack are shown in Fig 3.1.

Next_PC(for JAL and JALR)	Stack Pointer of JAL/JALR (r[29])

FIGURE 3.1 Fields of a shadow stack entry.

3.2 Detection Mechanism

In this section we present the general conditions used to detect a buffer overflow attack while distinguishing it from legitimate program constructs.

3.2.1 Record, Report and Monitor

In order to detect a stack-smashing attack, the following methodology is used. When a return instruction is encountered, the shadow stack is popped and we compare the target of the return instruction to the address that was popped off of the stack (which was put on the stack by the corresponding call). If the addresses match, the simulator continues to retire the next instruction. If the addresses differ, then the conventional subroutine call/return semantics have been violated. This is shown in the first comparison of Fig. 3.2. The addresses may differ due to an attack, or due to non-local returns, which are legitimate in certain program constructs. The call-return semantics of special program constructs such as setjmp()/calljmp() have been shown in Fig. 3.3. The double lines are indicative of subroutine boundaries as observed in the static binary. Note that in this case,

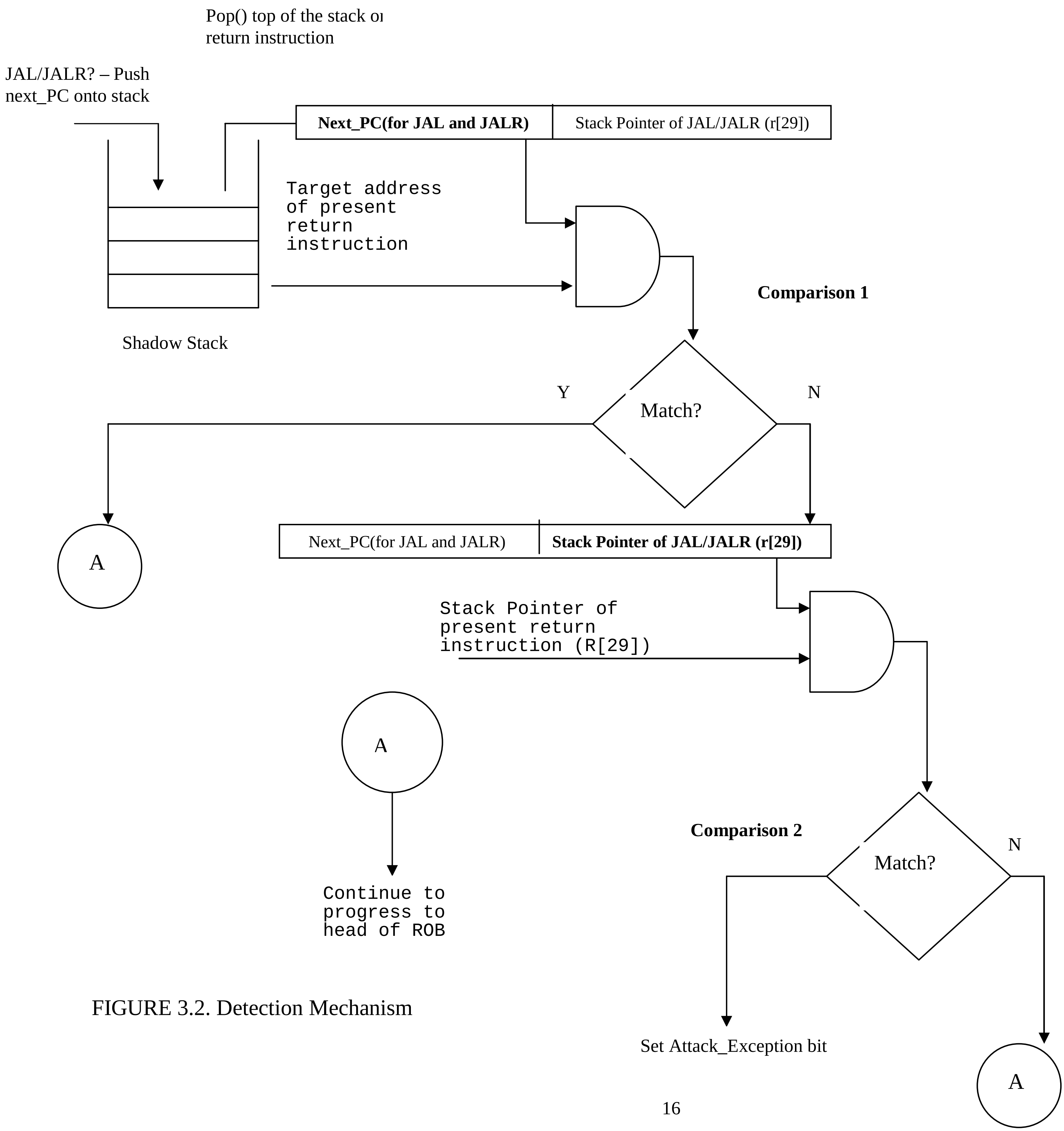


FIGURE 3.2. Detection Mechanism

the call to longjmp() will push the address 0x4003a8 onto the shadow stack as this is the next_PC. But when the return from longjmp() is executed, the return address is 0x400230. Thus, the return address comparison will fail.

Legitimate cases are distinguished from attacks by recording the stack pointer along with the return address at the time of a call. This is the second field of the shadow stack entry shown in Fig. 3.1. If a return instruction jumps to a target that differs from what was recorded on the shadow stack, we compare the current stack pointer (typically the contents of register 29 when the return instruction was fetched) to what was recorded on the shadow stack by the corresponding call instruction. This is shown in the second comparison of Fig. 3.2. If the stack pointers do not match, the case is detected as being a valid program construct, such as setjmp()/longjmp(). The stack pointers do not match in the case of setjmp() and longjmp() because, both of these are calls to special subroutines that save and restore environment. As we have already pointed out, jumps save the value of the stack pointer before jumping to the subroutine. When a longjmp() returns (to the point after a setjmp()), the value of the stack pointer has been modified (because longjmp() restores this state from where the setjmp() saved it) and this value of the stack pointer does not match up with what the call to the longjmp() routine saved on the shadow stack. This is exemplified in Fig. 3.3 with different values of stack pointers for calls. There is some state of the stack at this juncture that needs to be cleaned up.

3.2.2 Rewinding the Stack

Entries in the shadow stack that correspond to the call to the longjmp() function and calls and returns thereof (first two entries shown in Fig.3.4) need to be purged from the stack because the longjmp() does not execute these returns. Fig. 3.4 shows the shadow stack state corresponding to the sequence of calls detailed in Fig. 3.3 and indicates entries we have to purge. The JR instruction from longjmp() will have the stack pointer that corresponds to the last entry that we need to purge. This helps us to go into

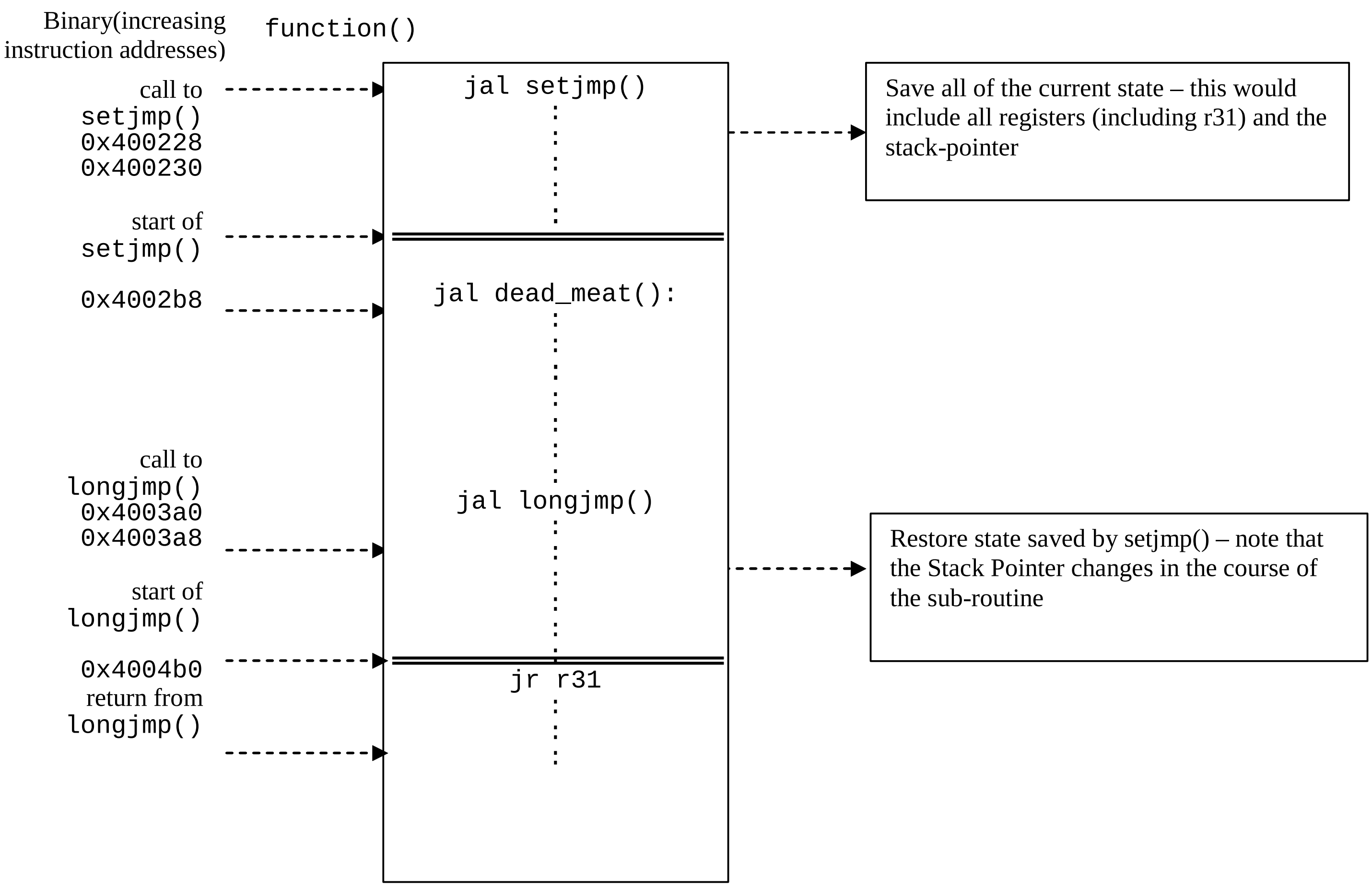


FIGURE 3.3. Subroutine boundaries and call/return semantics of setjmp()/longjmp()

the shadow stack and pop state off until the stack pointers of the return instruction from the longjmp() routine and the stack pointer of the entry at the top of the stack don't match.

If the stack pointers do match, in the second comparison part of Fig. 3.2, then the case is detected as an attack and an exception is raised. There are four cases in which an exception can be raised with implications on the shadow stack. These four cases are:

- Attack exception
- Underflow exception
- Overflow exception
- Context Switch exception (modeled as an overflow exception).

The four cases when there is an exception are detailed in section 3.4.

3.3 Implementation Issues

This work considers a couple of implementation issues for the shadow stack. The shadow stack is implemented in hardware to avoid recompilation of code and yet be able to detect stack-smashing attacks. The shadow stack is updated at the retirement stage as opposed to the return-address stack that is updated at the fetch stage [16] because, this is a security feature rather than being used for prediction. We also do not have to deal with extensive repair mechanisms for the shadow stack since we have an accurate picture of the sequence of calls and returns at the time of retirement. Recall that we need the stack pointer and the target address of return instructions to update the shadow stack. Assuming the compiler maintains the stack pointer in a specific register in the register-file, this implies that we need to read the register file, and propagate this information down the pipeline. While the target address for a return instruction is already latched in the source operand, the stack pointer needs some support for propagation. The registers would be read after the fetch stage and before the renaming stage. In the case of call and return instructions, in order to be able to record the stack pointer accurately, we have to create the illusion of having an extra source operand for these instructions. In order to avoid having to change the ISA, whatever value of rSP (where rSP is used to denote the register which is architected to hold the Stack Pointer) is read from the register file at the point of fetching these instructions, would be latched and sent down the pipeline along with the instruction. This description is discussed in section **4.1** where we talk about simulating this detection mechanism using a detailed, out-of-order timing simulator, simplescalar 3.0[24]. Since the operating system needs to save and restore state of the shadow stack, the shadow stack needs to be visible to the OS. Besides the entries shown in Fig. 3.1, other state associated with the shadow stack would include the top of stack pointer and perhaps a length register. An alternative to the length register would be to implicitly run the Full() function described in section 3.1 every cycle.

3.4 Exceptions

Since the shadow stack is implemented in hardware, a finite number of elements are represented in hardware. Though our simulator uses an unbounded stack, this is not feasible in a real implementation. The hardware structure would have an upper-limit on the number of entries. Once the upper-limit has been reached, the hardware raises an exception, called the overflow exception, which the operating system deals with. The way the operating system deals with such an exception is to save the current contents of the shadow stack into memory and clearing the shadow stack. If overflows are facilitated in this fashion, there could be underflows as well, in which case the current shadow stack needs to be refreshed with what was saved to memory, in a previous overflow. This would be the second exception, an underflow exception. There are two other situations when exceptions occur. These are the times when an attack is detected and when there is a context-switch. In the case of an attack having been detected, we have to alert the operating system and stop executing the current program. If an attack is detected, an exception bit could be set in the return instruction that detects an aberration and when that instruction reaches the head of the ROB, it would cause an attack exception. In the case of a context-switch, the current state of the shadow stack would have to be saved to memory. We could handle this context switch exception in a manner similar to the overflow exception. The impact of these exceptions, along with their impact on performance would be part of future work. The performance impact of having the shadow stack itself has been quantified by means of simple experiments in chapter 4.

Chapter 4

Simulation Methodology and Experiment Results

This section describes the simulation parameters used in the evaluation of the basic shadow stack in a functional simulator, and then in the implementation of the detailed design requirements in a detailed, out-of-order superscalar simulator. Some experiments were carried out in order to get an idea of some popular sizes of the stack in the case of a benchmark such as lisp. Lisp is the benchmark on which the representative experiments are run because it extensively uses setjmp()/longjmp() calls. For the stack sizes alone, we ran the simulations on a few other SPEC benchmarks, because these too are impacted by the presence of the shadow stack at the end of the pipeline. Other experiments were used to generate some typical rewind lengths for lisp. The results of these experiments have also been presented in this section.

4.1 Simulation Environment

The shadow stack was implemented in two environments. To identify the usefulness of the idea and make quantitative measurements such as popular sizes of the stack and rewind-lengths, discussed in the upcoming section, we used a fast-functional simulator. To explore design issues, we used a detailed, out-of-order, processor simulator, simplescalar 3.0[24].

4.1.1 Functional Simulation

A fast-functional simulator was used to run the experiments. Many examples from [9] were run on a Linux machine to understand the process of stack-smashing. A simple strcpy example was compiled for the simplescalar ISA using ssbig-na-sstrix-gcc compiler that comes with the simplescalar toolset [24]. When, the stack-smashing example was simulated, the simulator detected the attack accurately and simulation was terminated upon detection.

Another example which used the setjmp()/longjmp() constructs was simulated and the simulator did not exit incorrectly and completed running the binary. The stress test was applied with the benchmark lisp, from the SPEC95 suite, which is known to be intensive in its use of the setjmp()/longjmp() constructs. This benchmark ran to completion without exiting incorrectly as well.

The shadow stack was implemented using a stack structure from the standard g++ library. For purposes of simulation, we used two separate stacks for the return addresses for jumps and returns and the stack pointer for the same instructions. In the actual hardware, there would be just one stack with two fields, which are updated simultaneously each time a jump or return is retired. The stack pointer at the time of fetching the current call or return instruction is simply read from register 29 in the architected register file and stored as a field in the instruction. During retirement of these instructions, this value of the stack pointer is stored on the shadow stack (for calls) or used for comparison 2 in Fig. 3.2 (for returns). For detecting returns, we check to see if the instruction’s opcode is a JR (or jump-register) and if its source operand is register 31.

The size of the shadow stack was unbounded (dynamically allocated) in these experiments.

4.1.2 Timing Simulation

In the case of a timing simulator, the shadow stack was implemented as an array, with the current element that is the top of the array being held in a variable. There are modifications made to two stages in the simulator, the dispatch stage and the retirement stage. The values of the return addresses of a Jump-and-Link and Jump-and-Link-Register instructions are latched into the RUU entry at the time the instruction is dispatched. The RUU entry here refers to a standard data-structure that is used in the timing simulator to contain information about the instruction as it proceeds down the pipeline. Along with this, the value of the stack pointer is also latched in the RUU entry[ss3.0]. This is an easily accomplished task, as all we have to do is latch the value of PC+4 as the address the JAL or JALR is supposed to return to and the value of reg29, which is the value of the stack pointer. In the case of a return instruction (or a JR with a source register numbered 31), the value of reg31 and the stack pointer are latched into the RUU entry for that instruction. At the retirement stage, if the dynamic instruction being retired is a JAL or a JALR, the return address and the stack pointer are pushed() onto their respective stacks. On the other hand, if the instruction is a JR with one of the source registers being numbered 31, the topmost entry of the stack is popped and the check described in section 3.2 is carried out. If an attack has been detected, the simulator counts up the number of attacks and exits the simulation. The timing simulation was useful in order to gain insight into the performance degradation that the benchmark had to suffer because of this security feature.

4.2 Experiments and Results

4.2.1 Popular Stack Sizes and Rewind Lengths

The popular sizes of the shadow stack were measured and we constructed a histogram by implementing a very simple experiment in the fast-functional simulator. The histogram was built by incrementing a particular size bin every time the shadow stack was observed to be at that many number of elements in length. If a rewind were to occur, we decrement the stack size by that many elements before updating the size-histogram. The rewind lengths were also used to build a second histogram, the rewind-histogram.The rewind lengths have been tabulated. The histograms were generated accurately with sufficient checks for errant outliers. Though only lisp has setjmp()/longjmp() calls in it, we measured the popular sizes of the stack across benchmarks that did not have these calls as well. This was done in order to get an idea of how big the stack can potentially grow and whether the overflow and underflow exceptions can be minimized by sizing the stack in a particular way across workloads. The rewind lengths across benchmarks that did not have the setjmp()/longjmp() calls remained at 0. The results for this section have been presented in Table 4.2 and in Figs. 4.1 – 4.6 showing the actual size-histograms for all benchmarks and the rewind-histogram for lisp alone.

Benchmark	Most Popular stack size	Maximum Stack size observed
Li	37	90
Gcc	8	52
Go	8	14
parser	6	16
vortex	11	31
perl	38	50
m88ksim	5	11
jpeg	5	62

TABLE 4.1. Popular Stack Sizes.

STACK-SIZE HISTOGRAMS

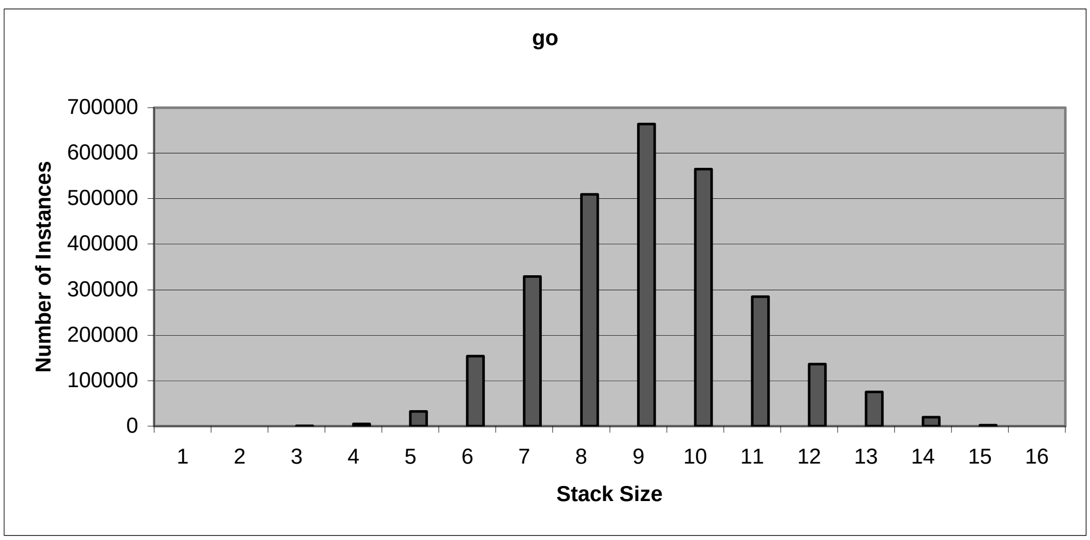


FIGURE 4.1. Stack Size Histogram for go.

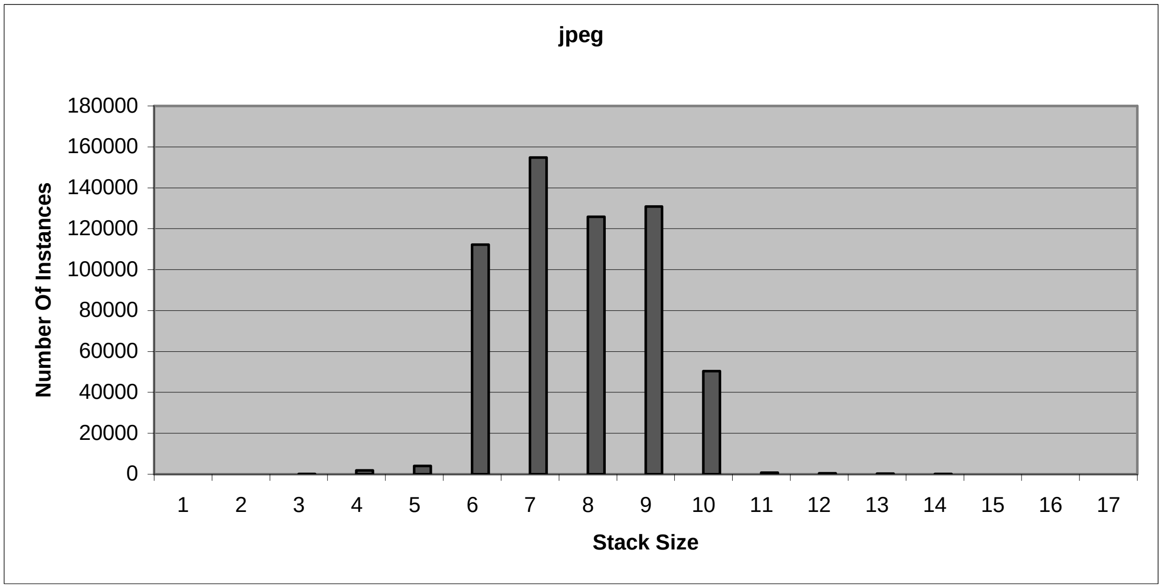


FIGURE 4.2. Stack Size Histogram for jpeg.

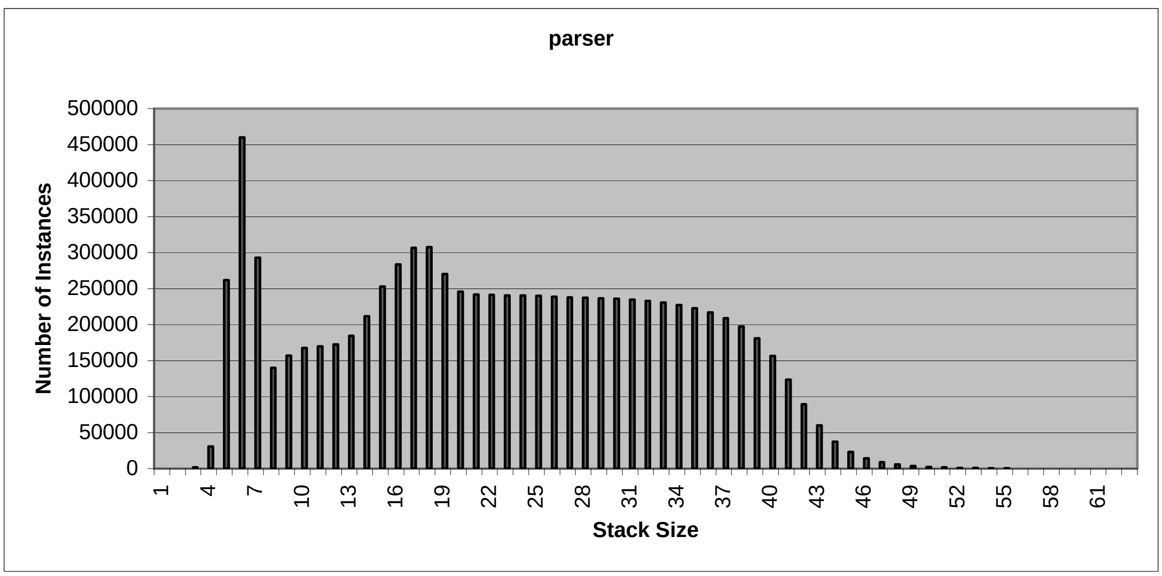


FIGURE 4.3. Stack Size Histogram for parser.

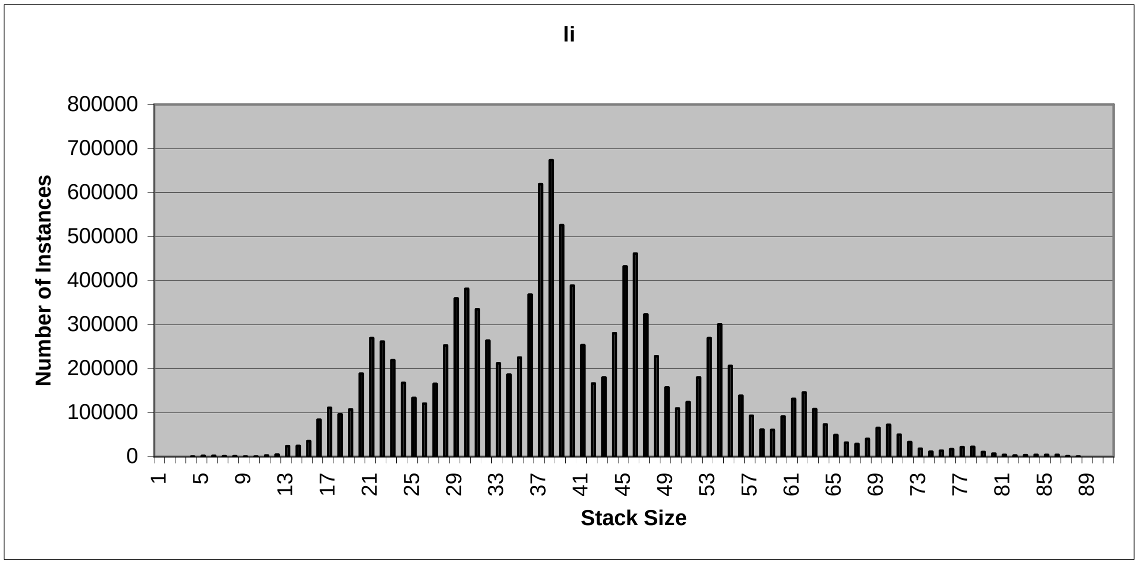


FIGURE 4.4. Stack Size Histogram for li.

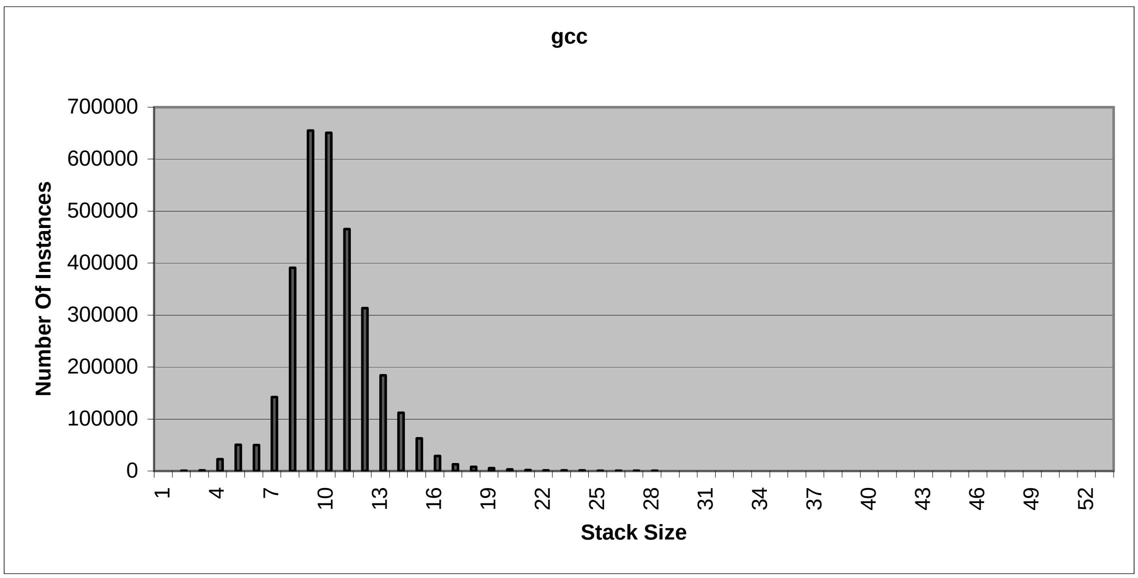


FIGURE 4.5. Stack Size Histogram for gcc.

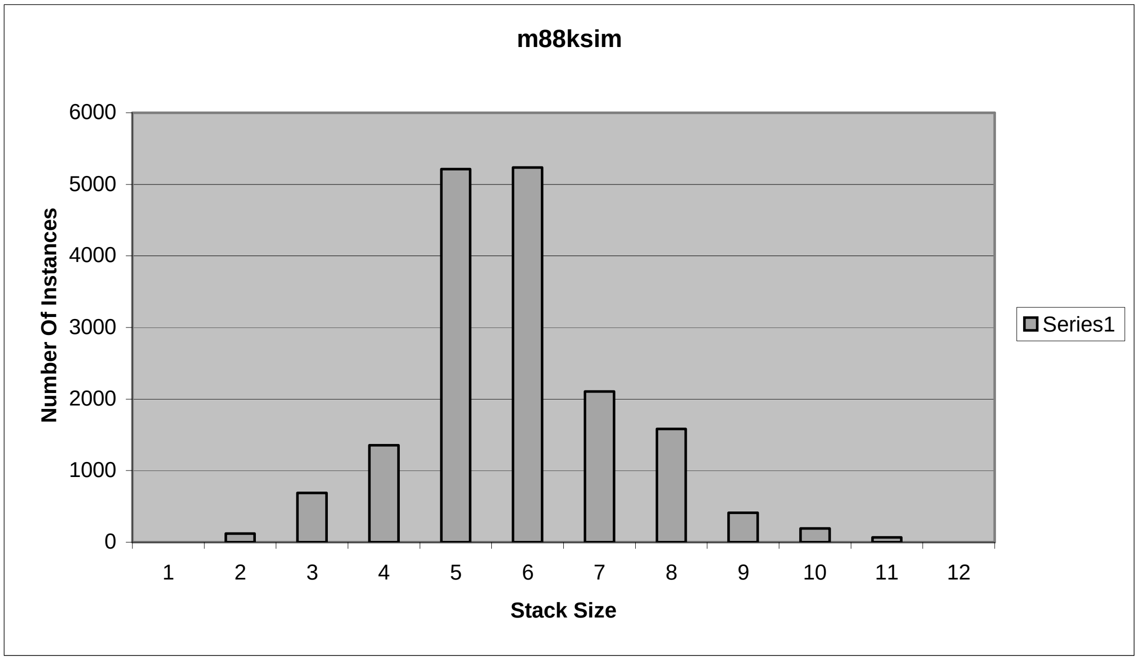


FIGURE 4.6. Stack Size Histogram for m88ksim

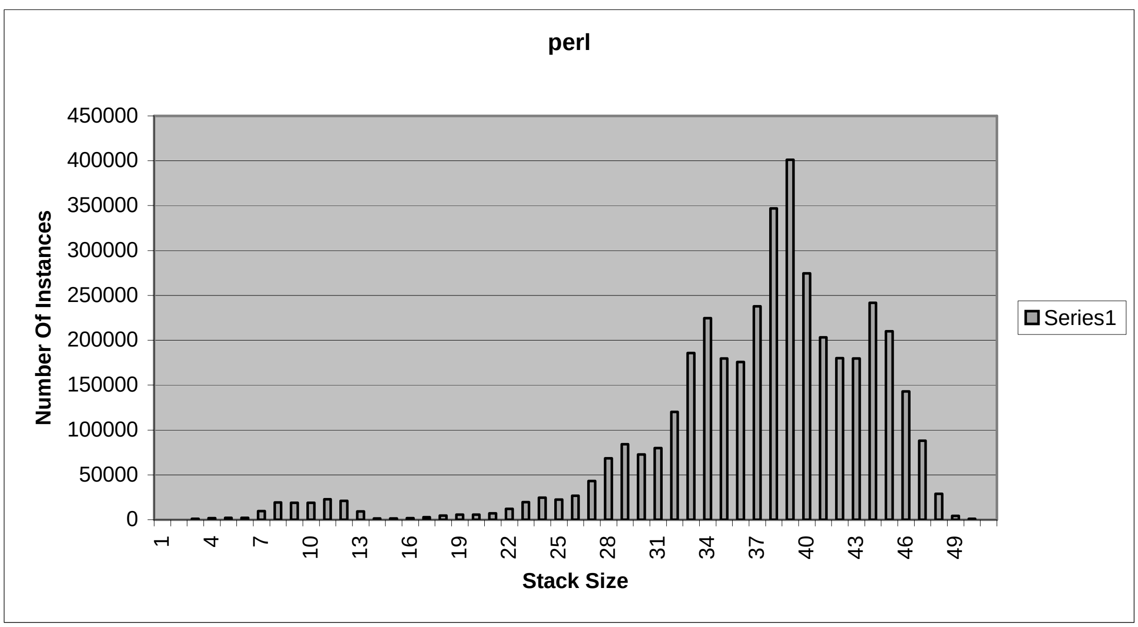


FIGURE 4.7. Stack Size Histogram for perl

REWIND HISTOGRAM

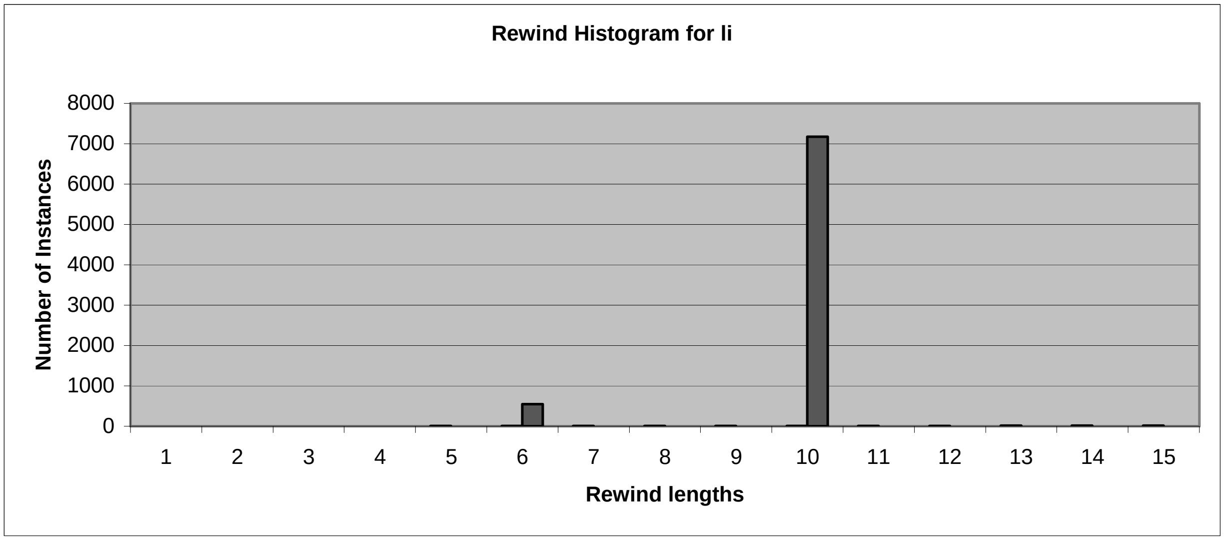


FIGURE 4.8. Rewind Length Histogram for li.

4.2.2 Performance degradation because of the Shadow Stack

A set of experiments were carried out with the timing simulator and the IPC with and without the shadow stack have been tabulated below. This feature, though used for security, will impact a performance penalty because the check in the retirement stage takes place before the instruction retires. We add a cycles delay for pushing an entry onto the shadow stack and one cycle for popping the entry and doing the first comparison of return addresses. There is one more cycle penalty if the second check needs to be carried out. If a rewind were to occur, we add a cycle to every entry we purge and compare. We find that the performance degradation varies between 1.5% to 9% on gcc, go, parser and jpeg and upto 12% on lisp (which is intensive in its use of setjmp()/longjmp()). The results have been tabulated in Table m.

Benchmark	Input dataset
li	7queens.lsp (queens 7)
gcc	gcc -O3 genrecog.i –o genrecog.s
go	9 9
parser	2.1.dict -batch
vortex	vortex.persons.in (persons.250,bendian.*)
perl	scrabbl2.pl (dictionary)
m88ksim	-c < ctl.in (dcrand.big)
jpeg	vigo.ppm

TABLE 4.1. Inputs to Benchmarks.

Benchmark	Original IPC	Modified IPC w/Shadow Stack	% Performance Degradation
li	1.4912	1.3702	12.1
gcc	0.9103	0.8877	2.26
go	0.9162	0.8990	1.72
parser	1.5549	1.4621	9.28
vortex	1.0717	1.0285	4.3
perl	1.0549	1.0147	4.0
m88ksim	1.4769	1.4286	4.8
jpeg	2.0959	2.0807	1.52

TABLE 4.2. Performance Degradation due to the Shadow Stack.

4.3 Discussion

From our experiments we observe that the stack does not seem to grow very large in most cases (amongst the benchmarks we have used). The occurrences of overflow and underflow exceptions therefore can be extrapolated from these numbers. If we perhaps modeled the stack to be a certain size based on the benchmarks we have run, we could minimize or eliminate these two exceptions. But, since these benchmarks are only from the SPEC suite, these numbers may or may not be accurate of other applications that have been known to have buffer overflow vulnerabilities which might have different stack behaviors. The rewind lengths are an interesting number because it gives us some indication of how many cycles are lost with the processor just rewinding the shadow stack. The lengths for lisp are either of 10 or 2 elements, but the rewinds occur very frequently (which is expected because of the frequency at which setjmp()/longjmp() occur). This translates as a performance penalty because when rewinds occur, that instruction holds up the subsequent instructions that are waiting to retire. Lisp shows a performance degradation of 12% when the shadow stack was implemented. In the case of the other benchmarks, the performance degradation was in the range of 1.5% to 9%. This was primarily due to the extra push() and pop() + compare operations at the retirement stage.

Chapter 5

Related Work

Since buffer overflows are a well-recognized vulnerability used to start stack smashing attacks, there have been solutions previously proposed in literature. This section discusses in brief, some of these solutions.

Writing correct code – Writing correct code is a very expensive proposition [14, 15]. This would involve getting rid of function calls like strcpy(), sprintf() etc. which do not check for the length of their arguments. Versions of the C standard library exist which complain when a program links to vulnerable functions like strcpy() and sprintf(). There are groups of people [7, 22, 23] that monitor the code by hand looking for security vulnerabilities. Fault injection tools have also been proposed [13] to pick out the really subtle bugs. In these methods, a buffer overflow is injected to test the code for robustness.

Non-Executable Buffers –Another approach would be to make data segments non executable. While this was popular in the past, recent dynamic optimization systems [25] rely on the ability to emit dynamic code into the program data segments to support various performance optimizations. This limits the possibility of making all parts of the data segment non-executable. An alternative would be to make the stack segment non-executable, while preserving program compatibility. Kernel patches have been made available for both Linux and Solaris [4]. There are a few exceptions to this, however, where executable code (in the case of Linux) needs to be placed on the stack, since Linux uses executable user stacks for signal handling.

Array Bounds Checking – Unlike non-executable buffers, array bounds checking completely stops buffer overflow attacks. If a buffer cannot be overflowed, the return address cannot be corrupted and no malicious code can be executed as a result. To implement array bounds checking, all the references to the array have to be checked [6].

Code Pointer Integrity Checking – In this approach, the tool detects whether a particular pointer has been corrupted before it is dereferenced [18, 19]. While the attacker may succeed in corrupting the code pointer, it is really of no use because he can never use that pointer. This method however does not perfectly solve the buffer overflow problem. Overflows that affect program components other than code pointers will still succeed in messing up the control flow to suit the attacker's ends.

The name of the stack itself was borrowed from Digital's FX!32 software [10], which used a shadow stack to record x86 return addresses and stack pointers to avoid an expensive hash-table lookup. When a subroutine call returned from the native Alpha code, the x86 return address was expected to be consistent with what was recorded on the shadow stack in the FX!32. If there was an inconsistency, the software trapped to the emulator. Return-address stacks have been proposed [16] for improving branch prediction accuracy. Our shadow stack is used as a security feature and is to our knowledge the first to provide such detection.

Chapter 6

Practicality Matters

6.1 Defining the scope of legitimacy

So far, we have discussed legitimate program constructs to be setjmp() and longjmp() that are allowed to pass through the integrity check without flagging an exception. These however are not extensive in their coverage of what might be all of the legitimate program constructs that should NOT be allowed to raise an exception. The second largest class of legitimate program constructs that should be allowed to pass through the checking phase on the shadow stack are the dynamic instructions that are generated as a result of JIT compilers. These would include very simple code generators for stack-machines, like Kaffe[30] that simply generate stack-modifying code. Others, like Cacao[31], generate code which looks very much like code generated by a traditional compiler. The proposed technique might run into problems with dynamically generated code, because the glue code which interfaces the compiler code with the generated code usually modifies the stack in several ways. Bintrans[32], for example, often uses subroutine jumps which never return, making the stack grow beyond all reasonable limits. Another problem area might be interpreters for languages with first-class continuations, like guile for scheme[33]. Most of them simply copy the stack whenever a continuation is created and switch to that stack when it is invoked, hence breaking the LIFO pattern. Yet another consideration should be programs which implement their own thread switching, usually triggered by hand (cooperatively) or by an alarm signal (preemptively).

6.2 The problem with workloads

This work has evaluated a simple, general methodology to detect stack smashing attacks. We have also laid the groundwork for a recovery scheme that can be used to implement this idea in a way that is transparent to the application, i.e, that lacks recompilation. The workloads that have been used to evaluate these ideas however are very specific. They include examples of code that are used to understand and demonstrate a stack smashing/buffer overflow attack and one benchmark out of the SPEC suite (lisp) and some other benchmarks to estimate the typical size of the shadow stack. This however is not an accurate coverage of all the vulnerable programs. There are many commercially available, vulnerable programs that we should evaluate. Examples of these would include ssh and apache, but the difficulty of compiling these for the simplescalar toolset has posed a major setback in running these workloads. Evaluation of these are important because the degradation in performance would be more accurately quantified in these cases as would the other experiments on popular rewind lengths and stack sizes. If a virtual machine were to be run, we could have even more interesting results and evaluate potential deadlock cases (multiple, successive exceptions etc.).

Chapter 7

Summary and Future Work

9 of 13 CERT advisories from 1998 involved stack smashing attacks [26] and at least half of 1999 CERT advisories involve the same [27]. A survey on the Bugtraq security mailing list [28] showed that approximately 2/3 of the respondents felt that buffer overflows are the leading cause of security vulnerability. This paper evaluates a hardware scheme that detects stack-smashing attacks without needing recompilation. The processor maintains a separate hardware stack, called the shadow stack, and monitors the dynamic instruction stream for procedure calls and returns. Calls push an entry onto the shadow stack which contains the stack pointer at the time the call was fetched and the expected return address of the call. Returns pop the shadow stack and employ a comparison using the shadow stack entry and their own values of the stack pointer and target address, to detect potential stack-smashing attacks. A series of design issues were considered which included managing the size of the shadow stack, making it visible to the operating system to enable four different kinds of interrupts, propagating necessary information down the pipeline to update the stack at retirement and implementing this hardware structure. By means of simulation, we demonstrate that the shadow stack accurately distinguishes between true attacks and legal program constructs which might not maintain the call-return semantics we expect, thereby providing another solution to the vulnerability of the decade.

Future work would include implementing the exceptions that we have talked about in detail. This would be done using a full-system simulator. The purpose of this study would be to quantify the occurrences of each of the exceptions while measuring their effects on performance. We also plan to look at the design issues detailed in the section 3.3 in greater detail. Propagating the information that is pushed on the shadow stack is an interesting issue in itself and we would like to explore methods to do that while keeping the implementation simple. Another issue we would like to consider is the behavior of the detection/recovery mechanism in the case of user-level threads

Bibliography

1. http://www.insecure.org/sploits/lprm.overflow.html.

2. http://www.wwdsi.com/demo/tutorials/SSH_vulnerabilities.html

3. www.intel.com

4. Solar Designer. Non-Executable User Stack.

http://www.false.com/security/linuxstack/.

5. Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In Proceedings of the Winter USENIX Conference, 1992.

6. Richard Jones and Paul Kelly. Bounds Checking for C.

http://www.ala.doc.ic.ac.uk/~phjk/BoundsChecking.html, July 1995.

7. M. Bishop and M. Dilger. Checking for Race Conditions in File Accesses. Computing Systems pp. 131-152. (Spring 1996)

8. Simson Garfinkel, Eugene Spafford. Practical UNIX and Internet Security. O'Reilly Associates. 1996.

9. Aleph One.Smashing the Stack for fun and profit. Phrack Magazine. Fall 1997.

10. Raymond J Hookway, Mark A. Herdeg.DIGITAL FX!32: Combining Emulation and Binary Translation. Digital Technical Journal. Vol 9. 1997.

11. Nathan P. Smith.Stack Smashing Vulnerabilities in the UNIX Operating System. Computer Science Dept., Southern Connecticut University, December 1997.

12. DilDog, The Tao of Windows Buffer Overflow, April of 1998.

13. Anup K. Ghosh, Tom O'Connor, Gary McGraw. An Automated Approach for Identifying Potential Vulnerabilities in Software. IEEE Symposium on Security and Privacy, Oakland, CA, May 3-6, 1998

14. Crispin Cowan, Calton Pu, and Heather Hinton. Death, Taxes, and Imperfect Software: Surviving the Inevitable. New Security Paradigms Workshop, Charlottesville, VA, September 1998.

15. Crispin Cowan and Calton Pu. Survivability From a Sow's Ear: The Retrofit Security Requirement.2nd Information Survivability Workshop, Orlando, Fl, October 1998.

16. K. Skadron, P.S. Ahuja, M. Martonosi, and D.W. Clark. Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms. In Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture, pp. 259-71, December 1998.

17. S.W. Smith, S.H. Weingart. Building a High-Performance, Programmable Secure Coprocessor. Computer Networks (Special Issue on Computer Network Security.) 31: 831-860. April 1999.

18. Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, Erik Walthinsen. Protecting Systems from Stack Smashing Attacks with Stack Guard. Linux Expo, Raleigh NC. May 1999

19. Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, Jonathan Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. System Administration and Network Security SANS, 2000.

20. Daniel J. Barrett, Richard E. Silverman. SSH The Secure Shell. May 2001.

21. J. Dyer, M. Lindemann, R. Perez, R. Sailer, S.W. Smith, L. van Doorn, S. Weingart. Building the IBM 4758 Secure Coprocessor. IEEE Computer. 34: 57-66. October 2001

22. Theo de Raadt et. Al. OpenBSD Operating System. http://www.openbsd.org.

23. Anon. Linux Security Audit Project. http://lsap.org

24. The Simplescalar architectural research toolset.

25. V. Bala and E. Duesterwald and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. PLDI, June 2000.

26. Steve Bellovin. Buffer Overflows and Remote Root Exploits. Personal Communications, October 1999.

27. Fred B. Schneider et.al. Trust in Cyberspace. National Academy Press, 1999. Committee on Information Systems Trustworthiness.

28. Aleph One. Bugtraq Mailing List. http://geek-girl.com/bugtraq/

29. StackGhost: Hardware Facilitated Stack Protection Mike Frantzen, *Cerias*, and Mike Shuey, *Engineering Computer Network*. USENIX 2001

30. www.kaffe.org

31. http://www.complang.tuwien.ac.at/java/cacao/

32. http://www.complang.tuwien.ac.at/schani/bintrans

33. http://www.gnu.org/software/guile/guile.html