6375 966C

This is to certify that the
thesis entitled

Simulating Hardware-level Buffer-overflow Protection

presented by

Matthew R. Fletcher

has been accepted towards fulfillment
of the requirements for the

Master of Science	degree in	Computer Science and Engineering

[illegible]
Major Professor's Signature

April 29, 2005
Date

SIMULATING HARDWARE-LEVEL BUFFER-OVERFLOW PROTECTION

By

Matthew R. Fletcher

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science and Engineering

2005

ABSTRACT

SIMULATING HARDWARE-LEVEL BUFFER-OVERFLOW PROTECTION

By

Matthew R. Fletcher

In the realm of computer security, buffer-overflow attacks have been a serious problem for over 15 years. Beginning with the original attack against the **fingerd** service, hundreds of programs, of many different natures, have been targeted and exploited by buffer-overflows.

The vast majority of buffer-overflow attacks work by targetting both return addresses and function pointers. A hardware-level solution, *Secure Bit*, has been proposed to help protect the integrity of addresses. Secure Bit works by modifying the semantics of a processor's instructions such that an address can be verified before jumping to it. Secure Bit is also meant to be transparent, in that it does not affect the regular operation of the processor. The goal of this thesis is to study the architectural impact of Secure Bit on a simulated processor using the SimpleScalar toolset. The conclusion is that Secure Bit indeed provides protection from buffer-overflow attacks without disrupting the normal execution of the processor.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the last decade, online security has become a major issue in both the field of Computer Science and in the eye of the public. Identity theft, denial of service attacks, and Internet worms have severely affected not only systems administrators, but also ordinary end users. Internet worms have been particularly bad since they have affected the most users and are the easiest to spread. Unknowing users can spread worms by simply opening email attachments, installing non-trustworthy software, or running an unpatched operating system. Many times worms only cause a headache for a single user, but sometimes, they can cost millions of dollars to companies dependent on stable and secure systems.

Worms and other forms of attack operate by targeting software flaws that exist primarily at the programming level. A technique called the "buffer-overflow attack" exploits the fact many programming languages do not perform automatic bounds checking on data buffers. Using this technique, an attacker can force a program to execute code it was never meant to—code that is invariably malicious. The first major Internet worm, the Morris Worm of 1988, was the result of a buffer-overflow attack against the `fingerd` service provided by many UNIX systems [1]. Since then, buffer-overflow attacks have become a well-known problem [2], resulting in the infamous

MS Blaster [3], Sasser [4], and Apache Slapper [5] worms. Despite the publicity surrounding and attention placed on buffer-overflow, to this day it remains the most prevalent form of attack.

Various software techniques have been developed to prevent buffer-overflow attacks; however, each technique has its weaknesses. That is, they either induce a significant penalty in a program's performance, do not protect against all common forms of buffer-overflow, or have been defeated by crackers [6, 7, 8]. *Secure Bit* [9] is a new hardware protection scheme that promises to both prevent buffer-overflow and remain completely transparent to programs. Support for Secure Bit requires only minimal modifications to the compiler, operating system, and processor.

The goal of this research is to understand the impact of Secure Bit on a simulated processor. While simple in theory, does the technique require substantial modifications to the hardware? Does it create harmful pipeline hazards or memory latencies that slow down the system? And most importantly, does Secure Bit truly protect against buffer-overflow attacks? These are all important questions that must be answered before Secure Bit becomes a viable solution.

To this end, we use the SimpleScalar processor simulation software [10] to analyze Secure Bit. The SimpleScalar system is designed to be easily extensible and understandable, yet still provide detailed and useful statistics. This makes SimpleScalar an ideal platform for Secure Bit simulation.

This thesis makes several contributions:

- We describe, the buffer-overflow problem and classify the most commonly used type of software exploit: the address-corrupting buffer-overflow attack. Then, we describe how this type of attack can be prevented using the Secure Bit address protection scheme.

- We identify the relevant components of SimpleScalar and detail the modifications required to correctly implement Secure Bit. We find the amount of modification required to be minor.

- Using the modified simulators, we show how Secure Bit protects addresses and stops programs from executing malicious code. We also show that Secure Bit has almost no affect on the system's performance.

The remainder of this thesis is organized as follows. Chapter 2 further describes how buffer-overflow works and how Secure Bit can stop this kind of attack. Chapters 3 and 4 discuss the organization and architecture of SimpleScalar and the work done to implement Secure Bit into the simulators. Chapter 5 demonstrates correct Secure Bit functionality in SimpleScalar, and then charts the performance of Secure Bit. We conclude with a summary of the results and recommendations about how to best integrate Secure Bit into a processor.

Chapter 2

Background

The "buffer-overflow attack" is a general term used to describe a class of various attacks that share something in common—they all operate by overwriting valid data with malicious data. Data targeted by buffer-overflow attacks can be anything, such as stored passwords, but the vast majority of attacks target stored program addresses. By changing an address, an attacker forces a program to mistakenly execute deleterious code instead of the program's own code. We call this class of attacks "address-corrupting" buffer-overflow attacks. This chapter will focus on characterizing precisely what address-corrupting buffer-overflow attacks are and how some variations of them work. A new hardware-based solution to the problem, Secure Bit, is also described.

2.1 Buffer-overflow Attacks

Most programming languages allow a programmer to allocate a buffer of data, whether it be on the stack or on the heap. Unfortunately, many languages such as C and C++ do not provide any sort of automatic bounds checking on these buffers. Further, bounds checking mechanisms (performed by either the runtime environment or the programmer) can sometimes be fooled. Buffer-overflows take advantage of this weakness—by overflowing a buffer with carefully constructed data, an attacker can manipulate the execution of a program. As mentioned previously, although there are several variations of the buffer-overflow attack, most of them specifically target addresses used in the regular operation of a program. Usually, these are function return addresses saved on the stack; however, they can be other addresses, such as those used by a function pointer.

2.1.1 The Stack

Every program running on a system divides its memory into several logical regions. Two of these regions are the data and stack regions. In memory, the data region grows from lower addresses to higher addresses, while the stack region grows from higher addresses to lower addresses (it is possible for the two areas to collide, but this is rare in modern virtual memory systems). The data region, usually called the heap, is used by the programmer to dynamically allocate and deallocate objects and data to be processed. The stack, on the other hand, is generally used as a temporary "scratch pad" maintained by the processor and compiler to facilitate the execution of the program. As a program executes, stack memory is constantly rewritten as functions are called and return.

Whenever a function is called, the compiler creates a *frame* for it on the stack. A frame can consist of:

1. The parameters passed to the callee function.
2. The return value of the callee function.
3. The return address of the caller function. This is the address where execution will resume when the callee function returns.
4. The saved registers of the caller function. The data in registers must be saved or else a new function may change them.

5. The frame pointer of the caller function. The frame pointer is the address of a function's frame on the stack.

6. The local variables used by the callee function.

The contents, the order, and even the names of objects in a frame depend heavily on the architecture and compiler being used; however, the above order is typical.

Consider the following function `func1`. Figure 2.1 lists the code and Figure 2.2 illustrates a possible layout of `func1`'s frame on the stack. First comes the `count` parameter, followed by the return address, the frame pointer, and the two local variables `i` and `name` (an integer is 4 bytes on most systems and a char is 1 byte, so `name` is twice as large as `i`). Note that the buffer `name` is located at a lower memory address than the return address. The order in which objects appear on the stack is significant and is the basis of buffer-overflow attacks.

```
void func1(int count)
{
   char name[8];
   int i;

   for (i = 0; i < count; i++) {
      printf("Enter your name: ");
      gets(name);
      printf("You entered %s.\n", name);
   }

   return;
}
```

Figure 2.1: A simple function

The method in which frames and return addresses are setup vary depending on the specifics of the architecture. The Intel x86 architecture defines the `CALL` instruction, which is capable of automatically saving the return address and setting up the called function's frame. Similarly, the `RETURN` instruction will destroy the

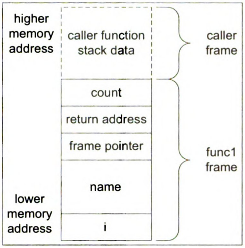

Figure 2.2: `func1` stack frame layout

frame and continue execution at the saved return address. RISC-like architectures leave the task of managing function calls to the compiler. When a function must be called, a Jump and Link instruction (`JAL`) is issued. Jump and Link will jump to the new function and place the return address in predefined register (usually register 31). It is then up to the compiler to save the return address on the stack. When the function is finished, the return address is loaded, jumped to, and execution continues. In either architecture, while the return address is saved on the stack, it is vulnerable to buffer-overflow attacks.

2.1.2 Return Address Attack

The most basic and easiest form of a buffer-overflow attack is called *stack-smashing*. In many programs, it is common for functions to make use of local buffers stored on the stack. These buffers are typically used to hold text, such as user input or an incoming request for a web server. If the programmer or system does not verify that the input data will fit in the buffer, the buffer will be filled until its bounds are reached. At that point, any extra input may cause a memory access fault in the best case, or begin overwriting program data in the worst case. As described above, other data on the stack could be local variables, parameters, or stored memory addresses. Stored addresses include frame pointers, function pointers, and most importantly, function return addresses.

This means an attacker can intentionally overflow an unprotected buffer and overwrite a return address with whatever address is desired. Generally, the new address points to the attacker's own malicious code, which is "injected" at the same time the buffer is overflowed. Figure 2.3 illustrates how a stack-smashing attack might work. In this example, the attacker overflows the buffer and blindly writes the address of the beginning of the buffer to each memory location, including the real return address. We call this stack *smashing* since there is little precision involved in how the stack's memory is overwritten—all nearby memory locations are simply overwritten with the malicious address. The attacker has also included, as input to the buffer, injected code that will now execute when the function attempts to return. Injected code generally does something like spawn a root shell or manipulate some of the program's data.

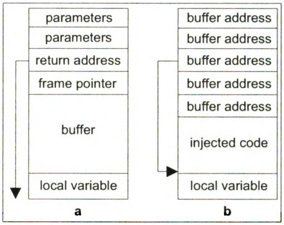

Figure 2.3: Buffer-overflow attack before (a) and after (b) stack-smashing

2.1.3 Function Pointer Attack

Protecting against first-generation stack-smashing (i.e. the method described above) is not terribly difficult, and several techniques exist to prevent it. Unfortunately, these techniques have either been defeated, incur a significant performance penalty, or focus on protecting only return addresses [6, 7, 8]. Other addresses, such as local function pointers stored on the stack, are still vulnerable to stack-smashing. Function pointer attacks are generally much more difficult to accomplish, since a function must not only define both a function pointer and buffer, but they must also appear with the function pointer following the buffer on the stack. Nevertheless, function pointer attacks are still possible. Figure 2.4 shows the similarity between function pointer attacks and regular stack-smashing—if the conditions are right, a function pointer attack works the same as a stack-smashing attack.

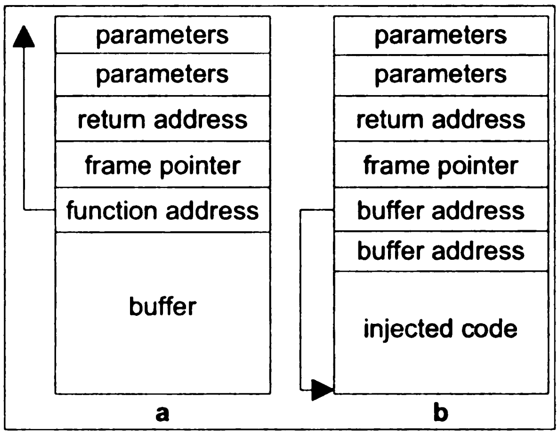

Figure 2.4: Function pointer smash attack before (a) and after (b)

2.1.4 Other Attacks

Attackers have developed several variations of stack-smashing. Instead of simply overwriting a segment of memory on the stack, attackers manipulate pointers or exploit memory alignment within data structures, modifying only a return address. Favorable conditions for these more precise attacks are rarer than conditions needed for simple stack-smashing; unfortunately, they still exist.

Other attacks include heap smashing and arc injection. Heap smashing works by corrupting addresses used internally by dynamic memory management routines, while arc injection takes advantage of addresses in well-known system libraries (usually `libc`) [11]. In all cases, these more sophisticated attacks maintain the spirit of stack-smashing—they each target addresses and disrupt the expected flow of execution in a program.

2.2 Proposed Solution: Secure Bit

Addresses are the primary target for all forms of address-corrupting buffer-overflow attack. The processor inherently trusts the addresses given to it during the execution of a program. After all, it makes little sense for a programmer or compiler to compromise its own program (especially programs running with elevated privileges). This level of trust is precisely what attackers exploit. To solve this problem, it must be assured that addresses are trustworthy at all times. A new technique, Secure Bit, has been proposed [9] to protect the integrity of *all* addresses. By protecting stored addresses, Secure Bit promises to guard against every type of address-corrupting buffer-overflow attack.

The mechanism behind Secure Bit is relatively simple. For every word of memory, there exists a corresponding Secure Bit. Each Secure Bit acts as a simple locking mechanism for its word. Whenever a word of memory contains an address, the address is considered valid and safe if the Secure Bit is "set." If the Secure Bit is "clear," the address is compromised and no longer trustworthy. Secure Bits are managed automatically by the processor and operating system, and are transparent to the user.

The processor is primarily responsible for manipulating Secure Bits (the operating system's role is outside the scope of this thesis—see [9] for more details). Figure 2.5 depicts the mechanism of Secure Bit. When a function is called, the return address is placed in some memory word on the stack. The Secure Bit corresponding to this memory word is set. Later, when the function is ready to return, the value of the Secure Bit on the memory word containing the return address is tested. If the Secure Bit is set, the return address is valid and execution continues. If the Secure Bit is clear, a buffer-overflow has compromised the program. The CPU generates a Secure Bit fault and execution of the program halts. How can the Secure Bit be cleared? Every memory write operation clears the Secure Bit for the destination memory word. Thus, if an attacker overflows a buffer and attempts to overwrite the return address, the memory word holding the return address will have the Secure Bit cleared by the write operation. Then, when the processor attempts to return, it will see the Secure Bit has been cleared and halt execution. This is how return addresses are protected by Secure Bit.

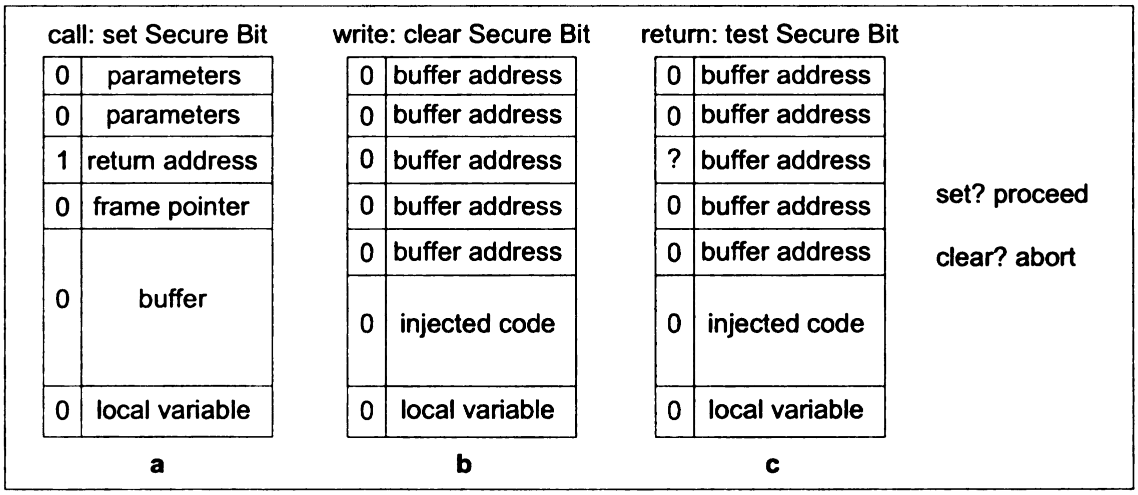

Figure 2.5: Secure Bit operation at function call (a), after buffer-overflow (b), and function return (c)

Buffer-overflow attacks targeting function pointers can also be prevented by Secure Bit. As with return addresses, memory words on the stack holding function pointer addresses are locked by a Secure Bit. In this case, the compiler is responsible for executing a new instruction, SBITSET, which will set the Secure Bit on the memory word containing the function pointer. When the function pointer is later used, the Secure Bit value is tested and execution aborts if an attacker has changed the address via buffer-overflow. Because the compiler is in control of the setup and usage of function pointers, it is not difficult to have it also insert the new instruction supporting Secure Bit. Figure 2.6 illustrates that Secure Bit protection of function pointers is very similar to protection of return addresses. Secure Bit is capable of maintaining the integrity of all addresses, whatever their use may be.

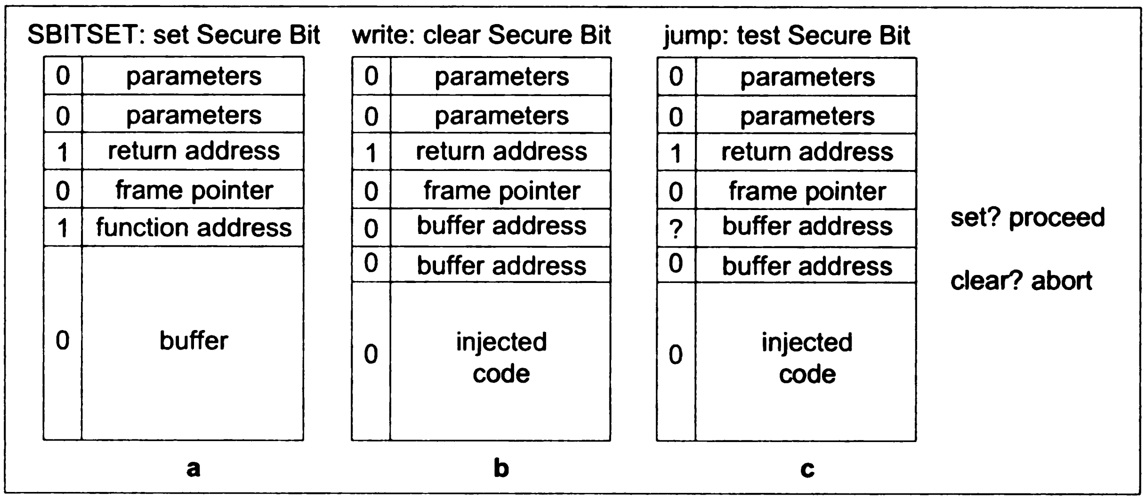

Figure 2.6: Secure Bit operation at SBITSET (a), after buffer-overflow (b), and function pointer jump (c)

There are two methods for storing Secure Bits. One method is to add an extra bit onto every memory word, much like how a parity bit is present for every memory byte. Another way is to store Secure Bits in a separate, dedicated memory unit accessible only to the processor. Whenever a memory word is accessed, the address is mapped to the necessary bit in Secure Bit memory. Assuming a system where a word is defined as 4 bytes, one byte of Secure Bit memory is needed for every 32 bytes of main memory. Secure Bits will also be cached by the processor in a dedicated Secure Bit cache, like the instruction and data caches. Either way, Secure Bit values are accessed in parallel with main memory access. In our research, we logically think of

Secure Bit as being present with every memory word, but actually implement Secure Bit via the dedicated memory unit.

Other issues surrounding Secure Bit, including proof, performance, virtual memory concerns, and other types of address protection, are analyzed in [9]. This thesis focuses on simulating Secure Bit return address and function pointer protection in the SimpleScalar processor simulator.

Chapter 3

SimpleScalar

3.1 Overview

The SimpleScalar tool set [10] provides detailed system simulation at the processor level and is used by dozens of groups to research computer architecture related topics. Secure Bit has been successfully implemented, simulated, and analyzed in SimpleScalar. This chapter will begin by introducing the SimpleScalar distribution, then follow up with more in-depth descriptions of the components relevant to Secure Bit. Implementation and analysis details are left to Chapter 4 and Chapter 5, respectively.

3.1.1 Architecture

SimpleScalar is designed to support any architecture one is willing to simulate, but is structured such that RISC-like architectures are favored. The SimpleScalar distribution ships with two architectures, which include the Alpha and PISA architectures. Other research groups have developed the components necessary to simulate other architectures, such as ARM [12] and PowerPC [13].

The PISA architecture (Portable Instruction Set Architecture) included in the distribution is not a real architecture—it is solely for research purposes and is based on MIPS [14]. SimpleScalar also comes with precompiled benchmarks, SPEC95 programs, and a special version of the GCC compiler and supporting libraries for PISA. The research in this thesis uses PISA because it is clean, simple to understand, and easily extensible.

3.1.2 Organization

SimpleScalar is organized such that components fall into one of two categories: core modules and simulators. Core modules include the simulated registers, simulated cache, and instruction semantics. SimpleScalar also distributes several sample simulators. One of the supplied simulators, sim-safe, executes programs as simply and safely as possible, while sim-outorder implements a relatively full-featured processor. Users are encouraged to modify the provided simulators, or create new simulators if needed. All simulators require some of the modules, such as the memory and register modules, but the branch prediction module does not need to be included if it not used. For example, sim-safe uses only a few core modules, while sim-outorder includes most of them. The various core modules form something like an Application Programming Interface (API)—they are meant to only be included on an "as-needed" basis.

3.2 Internals

To help fully understand the decisions made during Secure Bit's implementation (Section 4.1), we further characterize the internal structure of the SimpleScalar system and describe how the various pieces interact.

3.2.1 Core Modules and Simulators

SimpleScalar is divided into core modules and simulators. Although not all modules were relevant to Secure Bit, several did require modification. The affected modules were:

- `pisa.h` — Defines system specifications such as memory word size, register file sizes, and processor fault types.
- `pisa.def` — Large file of preprocessor macros that define the PISA instructions along with their semantics.
- `regs.c` — Supports the operations needed to manipulate the simulated registers.
- `memory.c` — Allocates simulated memory, supports memory access instructions used by simulated programs, and translates between simulated and host memory.
- `cache.c` — Supports simulated caching operations.

Along with the modules, the following simulators are provided in the distribution:

- sim-safe — Simulator that "safely" executes instructions one at a time without any features like branch prediction, cache, or dynamic scheduling.
- sim-fast — Similar to sim-safe, but ignores some things, like exceptions, to run a program straight through as quickly as possible.
- sim-profile — Captures statistics used to profile a program.
- sim-eio — Allows execution via program trace.
- sim-bpred — Implements several types of branch prediction algorithms and provides branch prediction statistics.

- sim-cache — Allows a user to specify an arbitrary cache hierarchy and provides caching statistics.
- sim-outorder — The most complete simulator, as it implements dynamic scheduling of instructions in an out-of-order pipeline, as well as branch prediction and caching.

Each of these simulators was adapted to support Secure Bit. The sim-safe simulator was used for simple testing, with sim-outorder used for analysis. The rest were not used regularly, since sim-outorder subsumes the functionality of them (and sim-fast completely ignores exceptions, including Secure Bit faults!).

Interaction between the core and simulators is done largely through preprocessor macros. This is done so that the core does not need to know details about the simulator and vice versa. For example, one simulator could name the memory region "mem" while another names it "memory." Or, a file defining a completely different architecture can be used by the same simulator. Macros abstract the machine-dependent details away so that the core modules and simulators may work independently.

3.2.2 Instruction Set

The ability to easily modify or create new instructions is an important aspect of SimpleScalar. Although the system supports several architectures, the PISA instruction set shipped with the distribution is, in particular, designed to be simple to understand and easily extensible. To this end, PISA provides two methods with which to create new instructions. One method is to define a completely new instruction—the instruction set encoding leaves free opcodes available for use by new instructions. The advantage of this is obvious: the designer may define a new instruction to use any format, input dependences, output dependences, semantics, etc. desired. The disadvantage is that the special PISA assembler must be modified to understand the new instruction and properly encode it into the binary.

To alleviate the burden of modifying the assembler, the instruction set encoding has also left the upper 16 bits of each instruction unused. The extra bits, called "annote" bits, are available to designers for any use. For example, the presence of annote bits may indicate an instruction should be interpreted differently; or, annote bits could be used by the compiler to give branch prediction hints to the architecture. The advantage of annote bits is that the PISA assembler is already aware of them, so adding annote bits to an instruction is trivial. The disadvantage is that an existing instruction must be overloaded—the bits do not allow you to redefine the instruction's inputs, outputs, and the pipeline hazards it creates. In other words, to add a new instruction using annote bits, an existing instruction that is a similar match, in terms of inputs and outputs, must already exist.

The syntax for annotating an instruction is very simple. Take, for example, the `MUL` instruction. The annotated `MUL/A` will set annote bit 1, while `MUL/E` will set bit 5. Annote bits can also be set in batch to a specific value, e.g. `MUL/4:3(3)` sets bits 3 and 4 to the value 3. Just like accessing memory and registers, an instruction may test for the presense of annote bits by using preprocessor macros, which simply return the result of a bitwise AND operation. If the bits are not set, the instruction executes as normal, but if the bits are set, it may act much differently. Overall, manipulating annote bits to redefine the semantics of an existing instruction is a quick-and-easy way to add a "new" instruction to the ISA.

3.2.3 Pipeline

The sim-outorder simulator is the only simulator which executes through a fully out-of-order execution pipeline. It is modeled after a classic, five stage RISC pipeline [15], although some of the names are different. Its five stages and their tasks are:

1. Fetch: Instructions are fetched.

2. Dispatch: Instructions are decoded and executed. Branches are predicted.

3. Issue: Issue as many instructions as possible. Instructions may be issued when resources are available and dependences are fulfilled.

4. Writeback: Instructions release any resources consumed. Mispredicted branches are cleaned up.

5. Commit: Instructions are removed from the pipeline. Store operations are committed to memory.

The primary difference between the simulated pipeline and a real processor is that instructions are executed as soon as they are decoded, whereas execution would normally have its own stage. This has no affect on Secure Bit. On the other hand, Section 3.2.4 details another significant departure from a real processor that does affect the Secure Bit implementation. Regardless, sim-outorder's pipeline is still relatively straightforward.

3.2.4 Speculative Execution

Since sim-outorder is the only simulator which provides both an out-of-order execution pipeline and branch prediction, it has the unique capability to issue instructions between branch prediction and branch resolution. These instructions are known as *speculated* instructions, since the processor is speculating along a predicted path. Normally, issued instructions in the pipeline operate against speculative registers and memory until the final commit stage, at which point they are saved to architected registers and memory. This mechanism allows the processor to safely issue instructions along the speculated path. If the branch was correctly predicted, the results are saved; if the branch was mispredicted, the results in speculative registers and memory are thrown away and the correct instructions are executed.

Speculative instructions are handled a bit differently in sim-outorder than expected. The following are the steps taken by sim-outorder in regards to speculation:

1. During the fetch stage, if an instruction is a branch, then the branch is predicted and new instructions are fetched from the predicted path.

2. In the dispatch stage, the simulator resolves the branch and decides if the prediction was correct or incorrect. If correct, execution proceeds as normal. If the branch is mispredicted, then the simulator goes into "spec mode." All new instructions from the wrong path are marked as "mis-speculated."

3. Finally, when the mispredicted branch reaches the writeback stage, it is finally resolved. The simulator backs up through the pipeline, destroys mis-speculated instructions, and begins fetching down the correct path.

Since the program is being run through a simulator and not a real processor, the simulator can "cheat" and decide early if a branch was correctly predicted or not. Instructions along an incorrect execution path are then clearly marked as mis-speculated.

When the simulator is in spec mode, memory and register accesses behave differently. Any instructions that are executing during spec mode should not, under correct behavior, even be executing, and they must certainly not have any permanent affect on architected registers or memory. The simulator enforces this by taking a copy-on-write policy with respect to mis-speculated instructions. Consider, for example, a mis-speculated memory instruction. If the instruction is a load, the simulator allows it to read from architected memory. On the other hand, if the instruction is a store, the data is written into special speculative memory. Any other mis-speculated memory instructions that reference the same memory address will then use the speculative memory. The same behavior is also true for registers. When the mispredicted branch is eventually resolved, the mis-speculated instructions, speculative registers, and speculative memory are destroyed and the processor resumes regular execution. Figure 3.1 illustrates speculation in the sim-outorder pipeline. Instructions that were never meant to execute have no adverse affect on architected registers and memory.

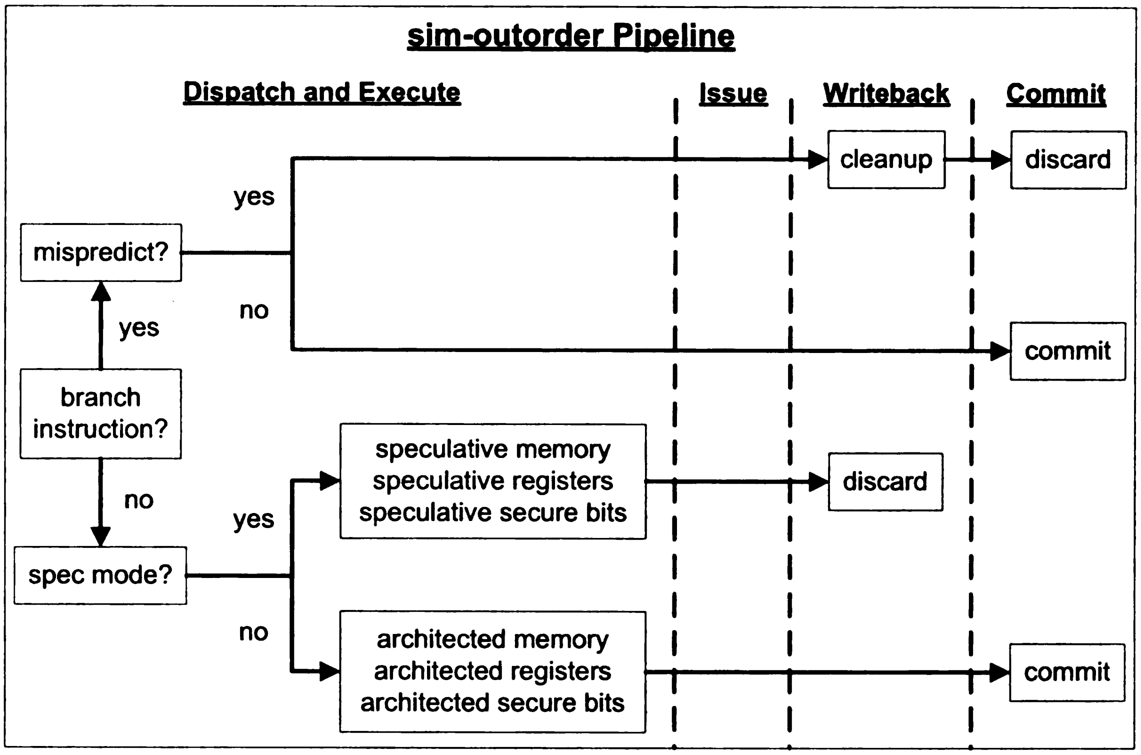

Figure 3.1: Mis-speculated instructions work against speculative registers and memory (fetch stage not shown)

3.2.5 Caching

Both sim-cache and sim-outorder make use of SimpleScalar's cache functionality. The cache module allows a user to define caches using almost any legal configuration. Simulators then use the `cache_access` function whenever a cache access is appropriate; `cache_access` simulates the access using the given configuration. Two results are produced. First, statistics about cache hits, misses, etc. are gathered. Second, the latency (in cycles) that would be caused by the access is returned to the simulator.

Simulator	Cache Type	Total Size	Block Size	Sets	Assoc.	Algorithm
sim-cache	L1 Data	8 KB	32	256	1	LRU
	L1 Instruction	8 KB	32	256	1	LRU
	L2 Unified	256 KB	64	1,024	4	LRU
sim-outorder	L1 Data	16 KB	32	128	4	LRU
	L1 Instruction	16 KB	32	512	1	LRU
	L2 Unified	256 KB	64	1,024	4	LRU

Table 3.1: Configuration of caches in sim-cache and sim-outorder

By default, the sim-cache and sim-outorder simulators make use of level 1 instruction, level 1 data, and level 2 unified caches (see Table 3.1 for default configurations). sim-cache uses cache only for statistics gathering.

Like sim-cache, sim-outorder uses cache to collect statistics. However, sim-outorder also uses the calculated latency of every cache access. At certain stages during the pipeline (specifically, during fetch, issue, and commit), cache is accessed and the latency is recorded for various statistics. More importantly, though, latency is used in the issue stage to stall an instruction in the pipeline while its operands are loaded from cache (although, the simulator does not actually copy any data to and from the simulated cache—it simply acts like it would). Doing this allows sim-outorder to account for memory access delays in the simulated pipeline.

Caching in the simulators may be extended by a designer in two ways. First, completely new, independent caches can be created and used. If new caches are used, the designer specifies the default configuration and cache heirarchy in the simulator code. Then, through command line arguments, the caches can be adjusted by the user to the desired configuration. The other way to extend the simulators' caching is to use the pre-existing "user data" field associated with each cache block as metadata. User data may be used for whatever extra information the designer may wish to associate with the cache block.

3.2.6 PISA Stack Operation

In terms of stack operation, the PISA architecture is true to its RISC nature. Function calls and returns are composed of several simple instructions, instead of complex, atomic instructions such as x86's `CALL` and `RETURN`. The compiler is responsible for managing the instructions and stack memory used in functions calls [16].

The following steps are performed by the compiler to call a function, execute it, and then return:

- The `JAL` or `JALR` instruction changes the program counter to the address of the new function and places the return address in register 31. `JAL` is used for regular function calls; `JALR` for function pointer function calls.
- The stack pointer is adjusted such that enough space on the stack is available to the new function (for the return address, frame pointer, local variables, etc.).
- The address in register 31 is saved on the stack.
- The old frame pointer value is saved on the stack.

The function then continues until it is ready to return. Upon return, the compiler performs these steps:

- The saved return address is loaded into register 31.
- The old frame pointer value is restored.
- The stack pointer is adjusted back to what the old function expects.
- The `JR` instruction changes the program counter to the address held in register 31, the return address.

Table 3.2 lists the five instructions used during function calls and returns and their purpose. Because addresses are temporarily loaded and stored in registers during

Instruction	Name	Purpose	Call or Return?
`JAL`	Jump and Link	Changes the program counter to the specified target address and loads the return address into register 31	Call
`JALR`	Jump and Link Register	Similar to `JAL`, but the loads target address from a register	Call
`LW`	Load Word	Copies the value held in a memory word to a register	Return
`SW`	Store Word	Copies the value held in a register to a memory word	Call
`JR`	Jump Register	Changes the program counter to the target address held in a register	Return

Table 3.2: Five PISA instructions, their purpose, and which operation (call or return) they are used for

a PISA function call, the Secure Bit solution described in Section 2.2 must be extended slightly—each register must also be protected by a Secure Bit. Nevertheless, the required changes to SimpleScalar are minimal.

Chapter 4

Implementation

4.1 Design Decisions

Overall, SimpleScalar is quite flexible when it comes to extensibility. In many cases, such as adding new instructions, the system can be extended in more than one way. This means that a few major decisions had to be made about the best way to implement Secure Bit into the system.

The first, and probably most significant, decision made was whether to store the actual Secure Bits as either a single bit appended to every word of memory, or as a separated, dedicated memory region. Adding a single bit to every word in memory is how we think of Secure Bit, so implementing it in this fashion is conceptually the simplest. Unfortunately, when it comes to actually adding an extra bit, it is not so simple. Adding the "odd bit" presents many problems including the need to adjust memory bus width, primitive data type size (the simulators store a word using type `unsigned int`—precisely 32 bits—how can an an extra bit be added to that?), and cache block sizes. In general, adding an extra bit to every memory word would require substantial modifications to any component of the simulator accessing memory.

Because of this, it makes more sense to implement Secure Bit using the Secure Bit memory unit. Secure Bit memory is one large, contiguous area of memory dedicated solely to Secure Bits. Whenever a Secure Bit access takes place, the memory word's address is mapped to the Secure Bit in Secure Bit memory, and the bit is tested, set, or cleared. Access to Secure Bit memory occurs in parallel with main memory access, so there is no slowdown in waiting for both accesses to happen sequentially. Not only is this scheme much easier to add into SimpleScalar, but it still allows us to logically think of Secure Bit existing as an extra bit on every word. We need only remember that there is some mapping going on between the memory address and the actual Secure Bit (Section 4.2.1.1).

Another important decision was about how to best handle the caching of Secure Bits. Section 3.2.5 describes two possible ways to extend the caches: either create new caches, or make use of the "user data" associated with each cache block. In this case, user data would hold the Secure Bits associated with the words held by that particular cache block. From a programming perspective, implementing new caches is much easier, as existing code can easily be reused to create Secure Bit caches; new code to manipulate user data is not nearly so straightforward.

As it turns out, the previous choice to place Secure Bit data into a dedicated memory unit makes the cache decision easy. Had we chosen to add a Secure Bit onto every word of memory, manipulating the user data of each cache block is the preferred solution. The Secure Bits for a cache block exist with that block, much like how the Secure Bit for each memory word exists in memory along with it. However, since we will implement Secure Bits as their own memory region, it makes the most sense to create new Secure Bit caches that access the dedicated memory. All accesses to Secure Bit memory go through Secure Bit cache, just as all accesses to main memory go through instruction and data cache.

Recall from Section 2.2 that the new Secure Bit `SBITSET` instruction can be used to protect memory locations holding function pointer addresses. When a function pointer address is loaded to a register, the compiler can insert the SBITSET instruction to set the Secure Bit on that register. The store instruction that saves this value to the stack, for use later on, will in turn set the Secure Bit on that memory address. This way, function pointers are protected from buffer-overflow attacks.

Two choices were available for adding the new SBITSET instruction. One is to take an existing PISA instruction and use one of its annote bits to flag it as an SBITSET instruction. If the annote bit is not set, we execute the instruction as normal; otherwise, execute it as SBITSET. This is possible because the PISA instruction set is easily extensible.

An alternate approach is used by Piromsopa and Enbody [9] in previous Secure Bit research. In [9], Secure Bit is added to the Intel x86 architecture—an architecture that leaves almost no room for new instructions. Piromsopa and Enbody solve this problem by giving special meaning to bogus LOAD instructions. Whenever a LOAD instruction with the source memory address of -1 is encountered, a new processor mode, "sbit mode", is toggled on or off. If an instruction copies to a Secure Bit value from one location to another and sbit mode is off, the value is copied. If sbit mode is on, however, the Secure Bit is *always* set.

Thus, to set the Secure Bit on memory holding a valid function pointer address, the following steps are taken:

1. A bogus LOAD instruction is issued and enables sbit mode.
2. A STORE instruction copies the address value to the function pointer's memory location. Since sbit mode is on, the Secure Bit on the memory is set.
3. A second bogus LOAD instruction is issued to disable sbit mode.

Using this technique, Piromsopa simulates the new SBITSET instruction without actually adding a new instruction, thereby overcoming a limitation of the x86 instruction set.

The question then is: is it better to create the new `SBITSET` instruction, as the original Secure Bit scheme specifies, or is it better to maintain consistency with previous Secure Bit research? We chose to go ahead and create the `SBITSET` instruction by using one of the annote bits for the `AND` instruction. `AND` was not chosen for any particular reason; it simply has the same input and output dependences we might expect from the `SBITSET` instruction. Although it breaks consistency with previous research, functionally, both the new instruction and Piromsopa's approach are equivalent. In fact, it is worth noting that overloading the meaning of the `LOAD` instruction by using a bogus memory value is almost exactly the same as overloading the meaning of a PISA instruction by setting its annote bits.

In summary, at several points during the Secure Bit implementation, various options were available and significant decisions were made. To fully integrate Secure Bit into SimpleScalar, we made the following choices:

1. The general Secure Bit technique described in Section 2.2 will be implemented.
2. Secure Bits will be stored in a separate, dedicated region of memory.
3. Secure Bits will be cached in their own, separate cache.
4. The `SBITSET` instruction will be created by using an annote bit to overload the meaning of the `AND` instruction.

The next section describes, in detail, the specific changes made to to the simulators.

4.2 Secure Bit Modifications

In support of Secure Bit, several modifications were made to the components described in Chapter 3.

4.2.1 Registers and Memory

The changes made to registers and memory, compared to other components, were minor. Register operations are defined in the files `regs.h` and `regs.c`. The `regs_t` structure is a simple register file containing data to represent the integer, floating point, and other simulated registers. An array of boolean values, `regs_sbit`, was added to the end of the structure to denote a Secure Bit for each register. The three functions to set, clear, and test a register's Secure Bit were also created.

Memory structures and functions are located in `memory.h` and `memory.c`. The `mem_t` structure represents simulated main memory and corresponding statistics tracked during simulation. A new structure, `sbit_t`, was created to depict Secure Bit memory and statistics and then added to `mem_t`.

4.2.1.1 Virtual Memory—Secure Bit Mapping

The size of the `sbit_t` Secure Bit region is 2^{26} bytes (64 megabytes). SimpleScalar presents a 31-bit address space to simulated programs; Secure Bits for the entire range must be present. Since a PISA word is defined as 4 bytes in size, one byte of Secure Bits can protect 32 bytes of virtual memory. Thus, 2^{31} bytes of memory require 2^{26} bytes of Secure Bits.

Whenever a Secure Bit is accessed, the 31-bit virtual address must be mapped to a 26-bit Secure Bit memory address. Figure 4.1 illustrates the mapping from one address to another; each of the set, clear, and test Secure Bit functions added to `memory.c` use this mapping. The upper 29 bits of a virtual address represent the memory word. Of these 29 bits, the lower 3 bits are used to index the byte containing the Secure Bit. The code listing for the `mem_test_sbit` function in Figure 4.2 demonstrates this mapping in code. First, a counter statistic is updated. Then, the two lowest order bits are removed from the address, the Secure Bit is indexed and, in this case, tested. Finally, the Secure Bit value is returned.

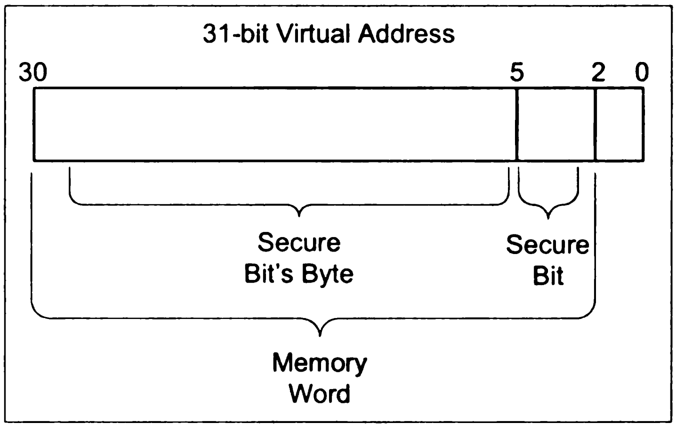

Figure 4.1: Mapping 31-bit Virtual Address to 26-bit Secure Bit Address

```
int
mem_test_sbit(struct mem_t *mem, /* memory space */
              md_addr_t addr)    /* virtual address */
{
  mem->sbits.sbit_test++;
  addr = addr >> 2;

  return (mem->sbits.sbits[addr >> 3] & (1 << (addr & 0x07)));
}
```

Figure 4.2: The mem_test_sbit Secure Bit function

4.2.2 Instruction Set

4.2.2.1 Existing Instructions

Once Secure Bit memory was in place, extra functionality was added to each of the instructions involved in function calls, returns, and function pointer jumps (previously listed in Table 3.2). The following additions were made:

1. `JAL` – Sets the Secure Bit for register 31.

2. `JALR` – Tests the Secure Bit for the register holding the destination address. If the Secure Bit is set, the address is jumped to and register 31's Secure Bit is set. If clear, the simulator aborts with a Secure Bit fault.

3. `SW` – Tests the source register's Secure Bit. If it is set, the destination memory's Secure Bit is set. Otherwise, the memory's Secure Bit is cleared.

4. `LW` – Tests the source memory's Secure Bit. If it is set, the destination register's Secure Bit is set. Otherwise, the register's Secure Bit is cleared.

5. `JR` – Checks if the destination register is register 31. If so, it tests register 31's Secure Bit value. If set, the jump proceeds. If clear, the simulator aborts with a Secure Bit fault.

6. All instructions that write to a register—except `LW` and `SBITSET` (see below)—clear the Secure Bit on the register.

4.2.2.2 New Instruction

To create the new `SBITSET` instruction, annote bit A of the `AND` instruction was used to indicate what operations the instruction should execute. Whenever an `AND` instruction is executed, annote bit A is tested. If the bit is not set, the instruction executes like a regular `AND`. However, if the bit is set, the instruction works like `SBITSET` would and sets the Secure Bit on the destination register.

Both the old and new semantics for the instructions above are defined as preprocessor macros in the file pisa.def (along with a Secure Bit fault type added to pisa.h for use by JR and JALR). For example, Figure 4.3 is the definition of the SW instruction. The first portion copies the register to memory, while the second portion tests the register's Secure Bit and then sets or clears the memory's Secure Bit. The other instructions' changes are all similar to this.

```
#define SW_IMPL
  {
    word_t _src;
    enum md_fault_type _fault;

    _src = (word_t) GPR(RT);
    WRITE_WORD(_src, GPR(BS) + OFS, _fault);
    if (_fault != md_fault_none)
      DECLARE_FAULT(_fault);

    if (TEST_REG_SBIT(RT))
      SET_MEM_SBIT(GPR(BS) + OFS);
    else
      CLEAR_MEM_SBIT(GPR(BS) + OFS);
  }
```

Figure 4.3: The Secure Bit SW instruction

4.2.3 Simulators

The final components of SimpleScalar that required modifications were the simulators themselves. Aside for sim-cache and sim-outorder, the changes to the simulators were minor. For the other simulators, the only additions were definitions for the Secure Bit preprocessor macros used by the instruction set. The macros are straightforward: they simply call the Secure Bit register and memory functions. Figure 4.4 shows the relationship between the instructions and simulators. Once the macros were defined, the necessary changes to these simulators was complete.

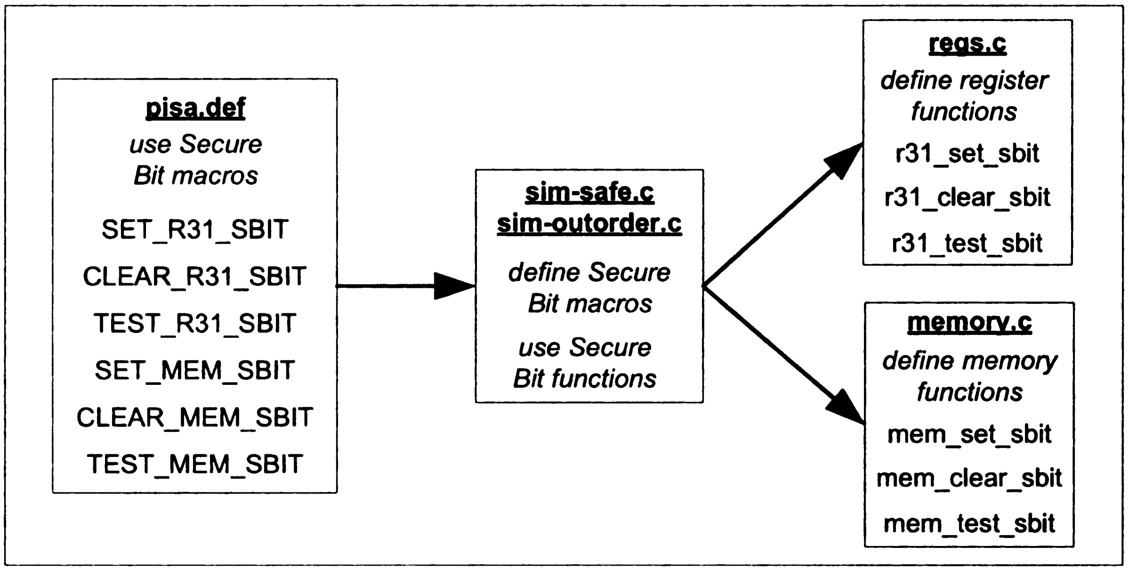

Figure 4.4: Interaction between PISA instructions, simulators, and Secure Bit functions

4.2.3.1 Caches

Like main memory, Secure Bit memory should also be cached, since Secure Bits are stored in dedicated, off-chip memory. Thus, levels 1 and 2 Secure Bit caches were added to both simulators. By default, both Secure Bit caches use the same configurations as the data caches (see Table 3.1). Addresses protected by Secure Bit (return addresses and function pointers) always exist in either the stack or heap, which are part of the data segment of memory. Thus, Secure Bit cache corresponding to instruction cache is not necessary.

Since sim-cache uses caches only for statistics gathering, adding Secure Bit cache access was straightforward. Mimicking data cache access, whenever a Secure Bit memory access macro is used, the `cache_access` function is called and the access is accounted for. Unfortunately, adding Secure Bit cache access to sim-outorder was not so straightforward, but still relatively simple. Recall that during the issue stage, cache latency is used to decide how long (in cycles) an instruction should be stalled while its operands are loaded. In the default sim-outorder, this number is determined by the level 1 data cache access time. To account for Secure Bit cache, the Secure Bit cache latency is calculated by `cache_access`. Since data and Secure Bit cache access should be concurrent, the larger of the two latencies is used as the overall latency. This way, any slowdown due to Secure Bit is taken into consideration. The impact of Secure Bit cache on simulated processor time is further discussed in Chapter 5.

4.2.3.2 Speculation

The final modification made to SimpleScalar was in the sim-outorder simulator. As described in Section 3.2.4, the simulator enters speculative mode whenever a branch is mispredicted. Architected registers and memory are protected, in that mis-speculated instructions work only against special speculative registers and memory.

Without doing the same for Secure Bit registers and memory, valid Secure Bit data will be clobbered by mis-speculated instructions. To accommodate these bogus instructions, several changes were made to sim-outorder:

- Speculative register Secure Bits were created.
- Speculative Secure Bits were added to speculative memory.
- New functions to read and write speculative Secure Bits were created.
- The preprocessor macros that call Secure Bit register and memory functions were modified to check for speculative mode.

If the simulator enters speculative mode, Secure Bit instructions operate using special speculative Secure Bit functions rather than the regular functions. In this way, Secure Bit data is safe from mis-speculated instructions.

Despite the amount of detail given in this chapter, the overall changes to SimpleScalar are minimal. In fact, more time was spent in understanding the system rather than undertaking a major overhaul. With the modifications made to registers, memory, instruction set, and simulators, the Secure Bit protection scheme was fully integrated into SimpleScalar.

Chapter 5

Analysis and Results

In this chapter, we analyze the results of our tests of the newly modified Secure Bit SimpleScalar simulators. The goal is twofold. First, we show that the modified simulators execute programs correctly—whether they suffer from an address-corrupting buffer-overflow attack or not. Second, we demonstrate how performance of the sim-outorder simulator is not hampered by Secure Bit memory and cache access. In other words, at the hardware level, Secure Bit is shown to be both functional in terms of security and transparent in terms of performance.

5.1 Test Programs

Three types of programs are used to demonstrate functionality and transparency. The different types are described here; Table 5.1 lists the programs themselves.

1. Six PISA-compiled benchmark programs ship with the SimpleScalar distribution. These programs cover integer math, floating-point math, branch-intensive code, etc. The primary purpose of the programs is for regression testing; known "good" outputs for the programs are available and can be compared against the output of programs run through new or modified simulators. If the regres-

Name	Type	Description
anagram	SimpleScalar	Finds anagrams for words
test-fmath	SimpleScalar	Tests basic floating-point math operations
test-math	SimpleScalar	More sophisticated floating-point math
test-llong	SimpleScalar	Tests `long long` data type operations
test-lswlr	SimpleScalar	Tests basic string output
test-printf	SimpleScalar	More sophisticated string output
compress	SPEC	Data compression
gcc	SPEC	C compiler
go	SPEC	Plays the game Go against itself
attack1	Created	Simple stack-smashing attack
attack2	Created	Simple function pointer attack

Table 5.1: Programs used during Secure Bit SimpleScalar testing

sion tests pass, the simulators are very likely to be correct (especially when the algorithm is known to be correct [17]).

2. Several PISA-compiled SPEC programs are available for use, and they can be used for either performance benchmarking, regression testing, or both. The compress, go, and gcc programs are used since sample inputs for them were freely and easily available from the SimpleScalar website [18].

3. Several specially-created programs were compiled with the special gcc PISA compiler. The programs are very small and simple, and suffer from address-corrupting buffer-overflow attacks.

The programs that ship with SimpleScalar and the SPEC programs are used primarily for regression testing. None of these programs normally suffer from buffer-overflow attacks, so Secure Bit operations should not affect these programs in any way. On the other hand, the homemade programs do suffer from buffer-overflow, so they are used to show how the regular simulators can be damaged by buffer-overflow, while the modified simulators catch the error.

The SPEC programs are also used in Section 5.3 to benchmark the performance of Secure Bit memory and cache in various configurations.

5.2 Secure Bit Functionality

The first and most important question about the Secure Bit version of SimpleScalar is "does it work?" To work properly, the modified simulators must:

1. Allow regular execution of programs that do not have a buffer-overflow problem.
2. Halt programs that have an address corrupted by a buffer-overflow attack when they attempt to jump to the new address.

5.2.1 Regular Execution

As previously mentioned, both the sample programs which ship with SimpleScalar and the three SPEC programs do not suffer from an address-corruption problem. After the simulators were modified to support Secure Bit, all regression tests passed—Secure Bit did not interfere with regular execution of these programs.

5.2.2 Stack-smashing Attack

Figure 5.1 is a code listing for the homemade program attack1. This program is meant to represent the most typical kind of buffer-overflow attack: stack-smashing via an unsafe library call accepting user input. The program and attack work as follows:

1. Lines 5–6: The buffer `attack` contains several copies of the function `func2`'s address. The address is stored in reverse order to accomodate little-endianess.
2. Lines 8–12: Function `func2` is the arbitrary code the attack will jump to. It prints a message indicating the function was executed before exiting.
3. Lines 14–23: Function `func1` uses the `name` buffer, which overflows during the attack.

4. Lines 25–30: The `main` function prints out `func2`'s address, calls `func1`, and prints a status message before exiting.

```
 1:     #include <stdio.h>
 2:     #include <string.h>
 3:
 4:     // func2's address is 0x004001f0
 5:     char attack[] = "\xf0\x01\x40\x00\xf0\x01\x40\x00"
 6:                     "\xf0\x01\x40\x00\xf0\x01\x40\x00";
 7:
 8:     void func2()
 9:     {
10:        printf("You Lose!\n");
11:        exit(1);
12:     }
13:
14:     void func1()
15:     {
16:        char name[4];
17:
18:        printf("Enter your name:\n");
19:        // gets(name);
20:        memcpy(name, attack, 16);
21:
22:        printf("Returning to main.\n");
23:     }
24:
25:     int main(int argc, char* argv[])
26:     {
27:        printf("func2 address %p\n", &func2);
28:        func1();
29:        printf("Back to main.\n");
30:     }
```

Figure 5.1: attack1, a stack-smashing attack

To begin, `main` calls the `func1` function. `func1` represents a function responsible for receiving user input and placing it into a buffer. In this case, it is a user's name, but could be something like a mail server accepting an SMTP command. Line 19 calls `gets`—a library function notorious for being unsafe. Since `gets` performs no bounds checking, overflowing the destination buffer is trivial (`fgets`, which *does* limit the input size, should always be used instead of **gets**). In the case of attack1, we simulate inputting an attack string by using `memcpy` on line 20 to copy 16 bytes (four words) of the input buffer into the `name` buffer. The input buffer, **attack**, has been carefully constructed to smash the address of `func2` onto four words of the stack. One of the smashed words is the return address; so when `func1` attempts to return, it instead jumps to `func2`. `func2`, in turn, prints a message and ends the program.

To understand more precisely what is happening to `func1`'s stack frame, examine Figure 5.2, the assembly code used for `func1`.

1. Line 2: 32 is subtracted from the stack pointer, thereby allocating 32 bytes on the stack for `func1`.

2. Line 3: The value in register 31, the return address, is stored at the word 28 bytes above the stack pointer.

3. Line 4: The old frame pointer is stored at the word 24 bytes above the stack pointer.

4. Line 5: The frame pointer is set to the same value as the stack pointer.

5. Lines 6–7: The first `printf` call ("Enter your name:") is setup and performed.

6. Lines 8–11: `memcpy` is setup and called. Line 8, which is setting up a parameter for `memcpy`, indicates that the word for the `name` buffer is stored at 16 bytes above the stack pointer.

7. Lines 12–13: The second `printf` call ("Returning to main.") is done.

8. Line 15: The stack pointer is set to the same value as the frame pointer.

9. Line 16: The return address is loaded from the stack to register 31.

10. Line 17: The value of the old frame pointer is restored.

11. Line 18: 32 is added to the stack pointer, effectively destroying the stack frame for func1.

12. Line 19: func1 finishes by jumping to the address in register 31.

```
 1:     func1:
 2:             subu    $sp,$sp,32
 3:             sw      $31,28($sp)
 4:             sw      $fp,24($sp)
 5:             move    $fp,$sp
 6:             la      $4,$LC1
 7:             jal     printf
 8:             addu    $4,$fp,16
 9:             la      $5,attack
10:             li      $6,0x00000010      # 16
11:             jal     memcpy
12:             la      $4,$LC2
13:             jal     printf
14:     $L2:
15:             move    $sp,$fp
16:             lw      $31,28($sp)
17:             lw      $fp,24($sp)
18:             addu    $sp,$sp,32
19:             j       $31
20:             .end    func1
```

Figure 5.2: PISA assembly code for attack1's `func1` function

It is important to note the locations of values on the stack. Relative to the stack pointer, `name` is 16 bytes higher, the stored frame pointer is 24 bytes higher, and the return address is 28 bytes higher. When `memcpy` is called and told to copy 16 bytes, `name` is overflowed and the values of the stored frame pointer and return address are replaced with the address of `func2`. At the same time, the Secure Bit values for these memory locations are cleared. As the function finishes, `func2`'s address is loaded into the frame pointer and register 31, then `func1` unintentionally jumps to `func2`.

Figures 5.3 and 5.4 show sample output of the regular and modified sim-outorder simulator, respectively. Clearly, the regular version is compromised by the stack-smashing attack. However, since the return address's Secure Bit value was cleared during the attack, the modified sim-outorder throws a fault (fault 8—Secure Bit fault). The modified sim-outorder also executes fewer instructions, since the simulator halts the program immediately; the regular sim-outorder inadvertantly executes additional code for the `printf` and `exit` system calls.

```
sim: command line: ./sim-outorder /home/matt/research/attack1

sim: ** starting performance simulation **
func2 address 0x4001f0
Enter your name:
Returning to main.
You Lose!

sim: ** simulation statistics **
sim_num_insn   15197 # total number of instructions committed
```

Figure 5.3: Vanilla sim-outorder cannot prevent stack-smashing attack

```
sim: command line: ./sim-outorder /home/matt/research/attack1

sim: ** starting performance simulation **
func2 address 0x4001f0
Enter your name:
Returning to main.
fatal: non-speculative fault (8) detected @ 0x004002f8

sim: ** simulation statistics **
sim_num_insn   12982 # total number of instructions committed
```

Figure 5.4: Modified sim-outorder aborts upon Secure Bit fault

5.2.3 Function Pointer Attack

Another program, attack2, is listed in Figure 5.5 and demonstrates a function pointer attack. The program works almost exactly the same as attack1, except `func1` attempts to use a function pointer to call `func3` before returning. On line 22, the

```
#include <stdio.h>
#include <string.h>

// func2's address is 0x00400250
char attack[] = "\x50\x02\x40\x00\x50\x02\x40\x00"
                "\x50\x02\x40\x00";

void func3()
{
   printf("Here is func3.\n");
}

void func2()
{
   printf("You Lose!\n");
   exit(1);
}

void func1()
{
   char name[4];
   void (*fptr)();

   fptr = func3;
   printf("Enter your name:\n");
   // gets(name);
   memcpy(name, attack, 12);

   printf("Calling func3\n");
   fptr();
   printf("Returning to main.\n");
}

int main(int argc, char* argv[])
{
   printf("func2 address %p\n", &func2);
   func1();
   printf("Back to main.\n");
}
```

Figure 5.5: attack2, a function pointer attack

function pointer is defined; and on line 24, it is set to the address of `func3`. Line 30 makes a function call via the function pointer; unfortunately, the `memcpy` call on line 27 has overflowed the `name` buffer and changed the value of the function pointer to `func2`'s address. Figure 5.6 shows the output of running attack2 through the regular sim-outorder.

```
sim: command line: ./sim-outorder /home/matt/research/attack2

sim: ** starting performance simulation **
func2 address 0x400250
Enter your name:
Calling func3
You Lose!

sim: ** simulation statistics **
sim_num_insn   14787 # total number of instructions committed
```

Figure 5.6: The vanilla sim-outorder is also susceptible to function pointer attacks

To prevent function pointer attacks, we use the new Secure Bit `SBITSET` instruction to mark loaded addresses as valid function pointer addresses. Figure 5.7 lists attack2's assembly code. Only the relevant portions are shown, namely, the instructions corresponding to attacks2's lines 24 and 30.

```
1:          la     $2,func3
2:          add/a  $2,$2,$2    # simulates SBITSET instruction
3:          sw     $2,24($fp)
4:          ...
5:          ...
6:          lw     $16,24($fp)
7:          jalr   $31,$16
```

Figure 5.7: attack2 assembly instructions for function pointer setup and function pointer call

The function pointer is setup by loading the address of `func3` into register 2 on line 1. On line 2, the annotated `AND` instruction is used to set the Secure Bit on register 2, thereby marking it a valid function address. The function pointer's data and Secure Bit values are then stored at 24 bytes above the stack pointer.

Lines 6 and 7 are where the function is actually called. The address, along with its Secure Bit value, is loaded from the stack into register 16. Then, JALR tests the Secure Bit for the register and either continues execution (Secure Bit is set) or aborts with a Secure Bit fault (Secure Bit is clear). In the case of attack2, the memcpy function has been maliciously used to overflow a buffer and to overwrite the function pointer with func2's address. In Figure 5.8, it is clear that the modified sim-outorder correctly aborts before performing an unsafe function pointer call.

```
sim: command line: ./sim-outorder /home/matt/research/attack2

sim: ** starting performance simulation **
func2 address 0x400250
Enter your name:
Calling func3
fatal: non-speculative fault (8) detected @ 0x00400368

sim: ** simulation statistics **
sim_num_insn   12416 # total number of instructions committed
```

Figure 5.8: The modified sim-outorder does not fall to a function pointer attack

In summary, the modified simulators not only pass the SimpleScalar regression tests, but also use Secure Bit to avert address-corrupting attacks that the vanilla simulators cannot. Regular stack-smashing attacks targeting return addresses are automatically prevented, while function pointer attacks are stopped by simply inserting a single SBITSET instruction during the function pointer's setup. Programs that do not suffer from buffer-overflow attacks still execute normally, while those which do suffer are prevented from causing harm.

5.3 Secure Bit Performance

In the previous section, we have seen how the modified SimpleScalar simulators provide the Secure Bit functionality we expect. Here, we show how Secure Bit remains transparent to programs in terms of cache and memory access latencies.

Recall from Section 3.2.5 how cache and memory access times affect sim-outorder. When load and store operations are executed, a function is called to simulate cache access. The function uses the defined cache and memory configurations to calculate the latency (in cycles) the memory access would cause, which is then used to stall the pipeline appropriately. Secure Bits are stored in memory, so accessing them also causes latency that should be taken into consideration (see Section 4.2.3.1).

In this section, the SPEC compress, go, and gcc programs are run under various cache configurations; both data and Secure Bit caches are tested. A variety of performance statistics are recorded and charted, and we find that Secure Bit caches much smaller than data caches are more than sufficient to maintain pipeline performance and efficiency.

5.3.1 Methodology

SimpleScalar defines caches using four parameters: block size, number of sets, set associativity, and replacement algorithm. The configuration settings for a cache ultimately affect its size and performance. Table 5.2 lists the default parameters for level 1 data and level 2 unified caches in sim-outorder. By default, Secure Bit levels 1 and 2 cache use the same settings.

In monitoring performance of the caches, we analyze three statistics gathered by the simulator:

1. Miss Rate: The miss rate of the cache. Lower is better.

2. Cycles: The total number of simulated cycles consumed during the program's

Level	Total Size	Block Size	Sets	Assoc.	Algorithm
L1	16 KB	32	128	4	LRU
L2	256 KB	64	1,024	4	LRU

Table 5.2: Default data and Secure Bit cache parameters in sim-outorder

execution. Lower is better.

3. IPC: The Instructions Per Cycle, which measures efficiency of the pipeline and is a good overall performance metric. Higher is better.

Miss rate is considered so as to measure how often latency due to memory access is introduced into the pipeline. If the requested memory is in L1 cache, the latency is only 1 cycle. However, a miss in L1 causes a 6 cycle latency to check L2, and a miss in L2 causes an 18 cycle delay to bring data in from memory. A higher miss rate will increase the number of cycles needed to execute a program, which in turn lowers the overall IPC of the pipeline. Our experiments measure the performance of smaller and smaller Secure Bit caches and compare them against the default; finding a good ratio of data cache to Secure Bit cache (where Secure Bit cache configuration does not substantially lower performance) is the goal.

5.3.2 Results

To begin, Table 5.3 shows the performance of level 1 data and Secure Bit caches, in default configurations, for the three SPEC programs. Similarly, Table 5.4 lists the performance of level 2 data and Secure Bit caches. Data cache numbers are taken from the regular sim-outorder, not the modified version supporting Secure Bit.

Although the miss rates for level 1 Secure Bit cache seem unnaturally low, the results still make sense. Recall that portions of the stack are constantly being rewritten as functions are called and return. One Secure Bit byte can protect 32 bytes of memory, so a cache block of 32 Secure Bit bytes protects a continuous, 1 KB area of memory. Since most function calls and returns manipulate the same region of the

	L1 Data			L1 Secure Bit		
Program	Miss Rate	Cycles	IPC	Miss Rate	Cycles	IPC
compress	0.0469	47028415	1.7132	0.0001	47028415	1.7132
gcc	0.0149	303513264	0.9213	0.0000	303523290	0.9213
go	0.0096	623730884	0.8789	0.0000	623730884	0.8789

Table 5.3: Default level 1 data and Secure Bit cache results

	L2 Data			L2 Secure Bit		
Program	Miss Rate	Cycles	IPC	Miss Rate	Cycles	IPC
compress	0.0995	47028415	1.7132	0.1573	47028415	1.7132
gcc	0.0416	303513264	0.9213	0.0836	303523290	0.9213
go	0.0118	623730884	0.8789	0.5183	623730884	0.8789

Table 5.4: Default level 2 data and Secure Bit cache results

stack, only the occasional function with a large buffer on the stack or a recursive function will cause Secure Bit cache misses.

The miss rates for level 2 Secure Bit cache, especially for go, are much higher than may be expected. As it turns out, there are so few level 2 accesses that many of them naturally miss. In go's case, there were only a total of 492 level 2 accesses—over half of the accesses were simply populating the cache. Thus, with these cache parameters, the relatively high miss rate of level 2 Secure Bit cache is insignificant. Overall, the results in Tables 5.3 and 5.4 indicate that a Secure Bit cache equal in size to data cache is probably excessive.

Next, we adjust the parameters of level 1 Secure Bit cache. Table 5.5 demonstrates the affect of decreasing block size, number of sets, and set associativity on miss rate and IPC. Figures 5.9, 5.10, and 5.11 chart the results for compress, gcc, and go, respectively.

Block Size									
	compress			gcc			go		
	Miss	IPC	Diff.	Miss	IPC	Diff.	Miss	IPC	Diff.
32	0.0001	1.7132	—	0.0000	0.9213	—	0.0000	0.8789	—
16	0.0025	1.7132	0.00%	0.0002	0.9213	0.00%	0.0000	0.8789	0.00%
8	0.0102	1.7126	0.04%	0.0009	0.9212	0.01%	0.0001	0.8789	0.00%

Number of Sets									
	compress			gcc			go		
	Miss	IPC	Diff.	Miss	IPC	Diff.	Miss	IPC	Diff.
128	0.0001	1.7132	—	0.0000	0.9213	—	0.0000	0.8789	—
64	0.0024	1.7131	0.00%	0.0002	0.9213	0.00%	0.0001	0.8789	0.00%
32	0.0106	1.7107	0.15%	0.0008	0.9212	0.01%	0.0004	0.8788	0.01%
16	0.0224	1.7026	0.62%	0.0034	0.9205	0.09%	0.0083	0.8773	0.18%
8	0.0363	1.6868	1.57%	0.0117	0.9170	0.47%	0.0750	0.8516	3.21%

Set Associativity									
	compress			gcc			go		
	Miss	IPC	Diff.	Miss	IPC	Diff.	Miss	IPC	Diff.
4	0.0001	1.7132	—	0.0000	0.9213	—	0.0000	0.8789	—
2	0.0030	1.7130	0.01%	0.0006	0.9211	0.02%	0.0004	0.8788	0.01%
1	0.0172	1.7010	0.72%	0.0122	0.9126	0.95%	0.0185	0.8724	0.75%

Table 5.5: Results from varying block size, numbers of sets, and set associativity of level 1 Secure Bit cache. Miss rate, IPC, and percent difference in IPC from the default are recorded (the first line of each section repeats the default configuration).

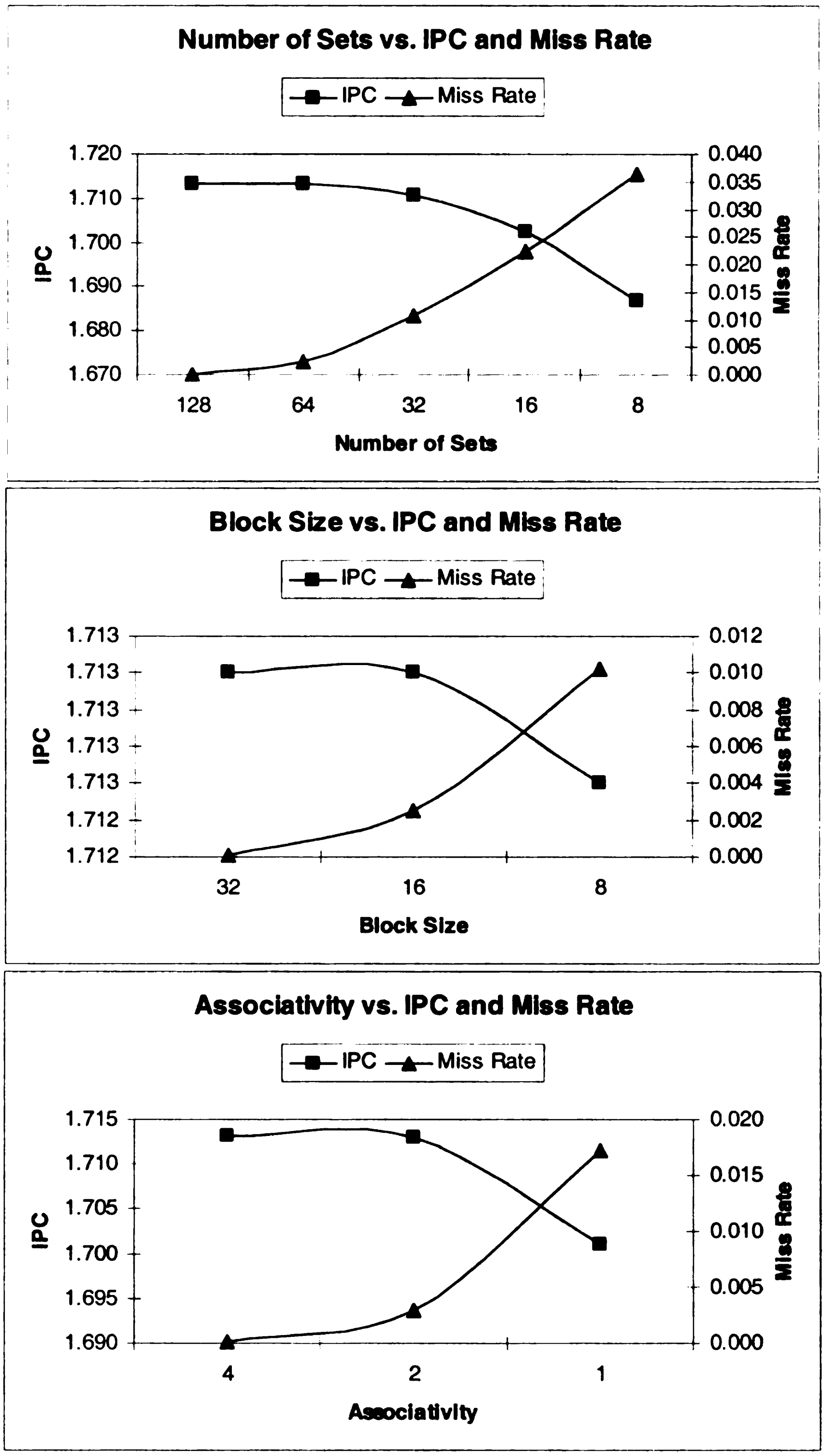

Figure 5.9: compress – Secure Bit level 1 – Varying block size, number of sets, and set associativity

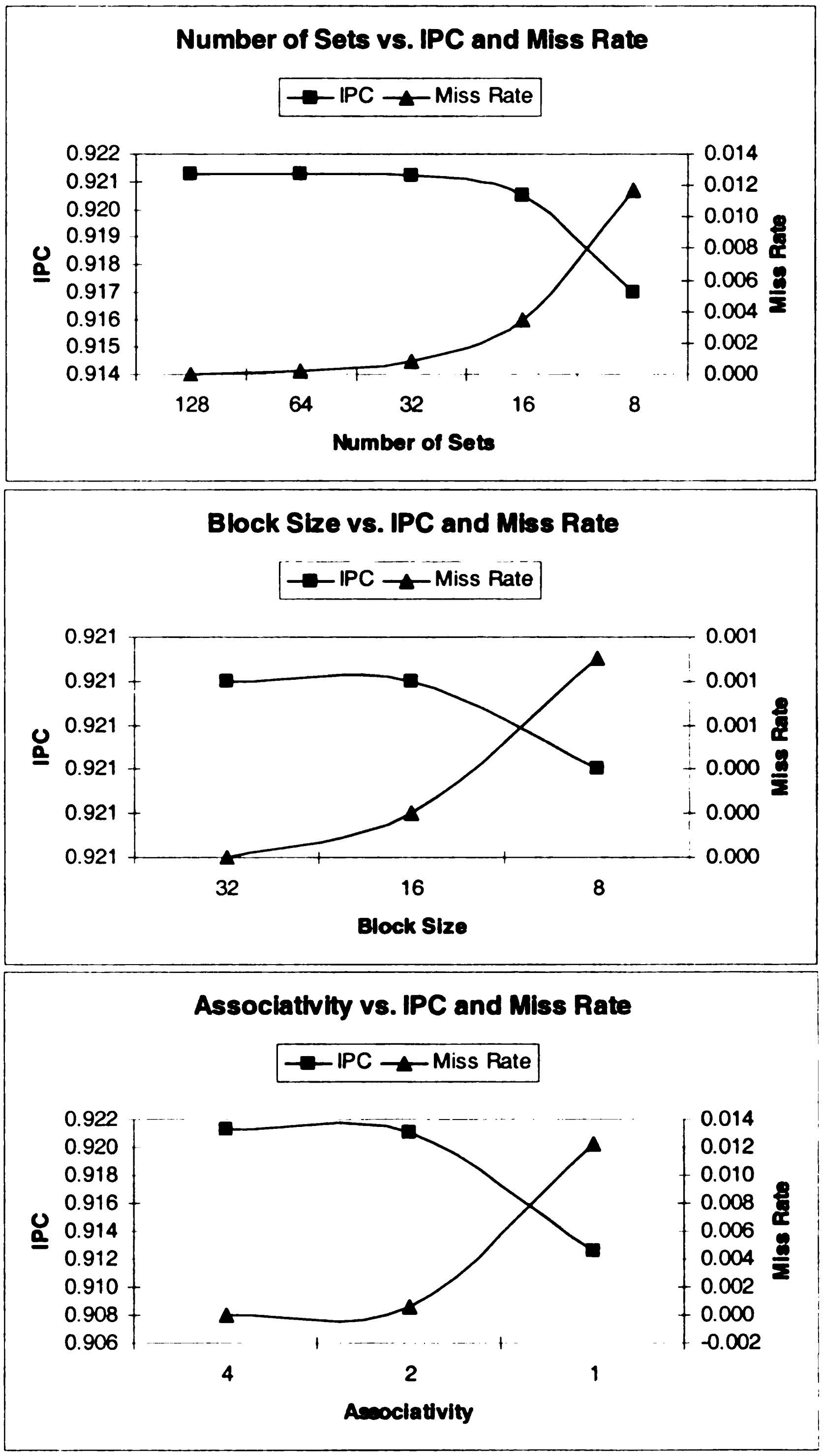

Figure 5.10: gcc – Secure Bit level 1 – Varying block size, number of sets, and set associativity

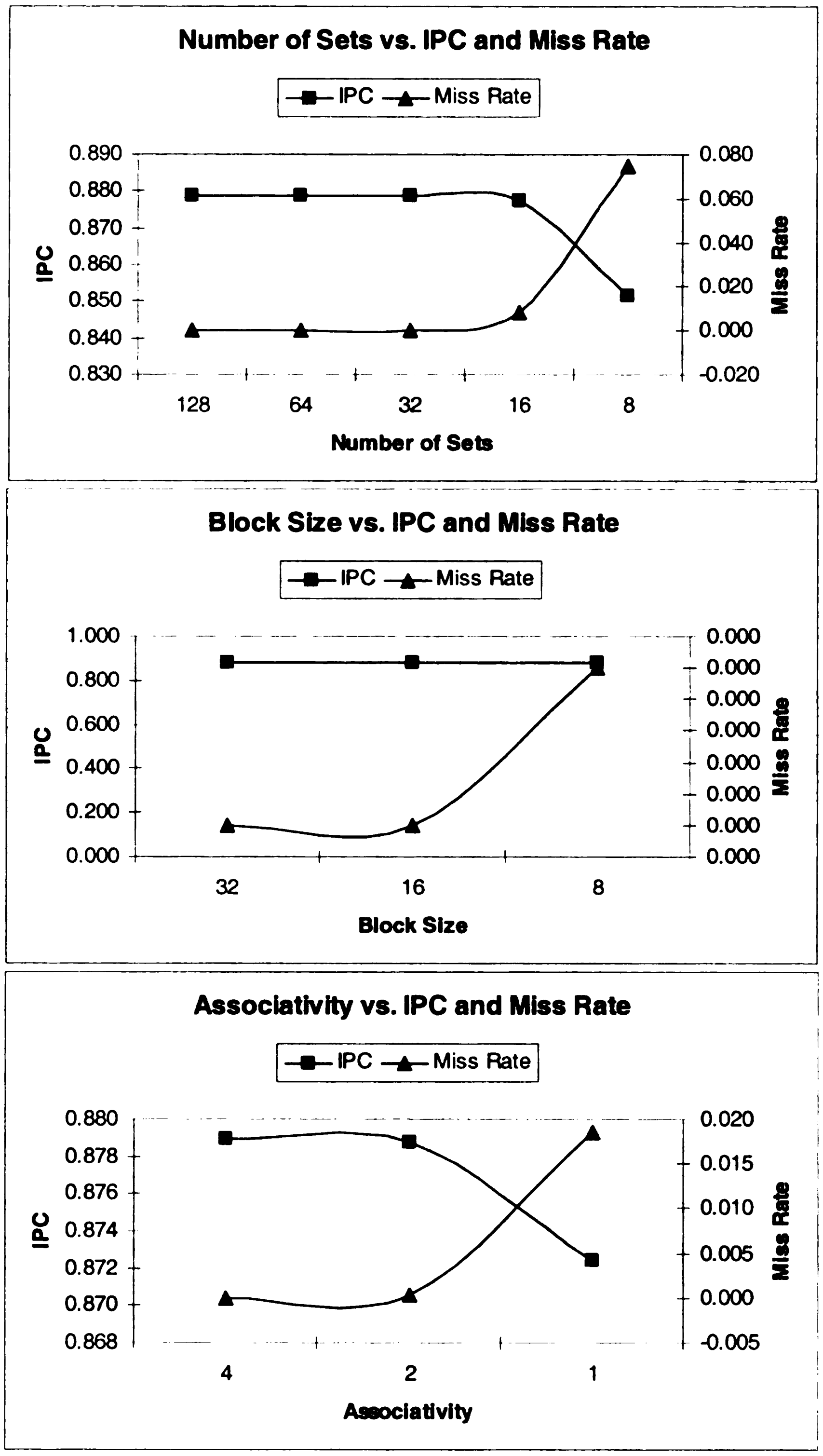

Figure 5.11: go – Secure Bit level 1 – Varying block size, number of sets, and set associativity

It is clear from the table and charts that configuring level 1 data and Secure Bit caches to the same size is indeed wasteful; even with a half-sized Secure Bit cache, the performance difference is almost zero.[1] Secure Bit cache sizes one-quarter and one-eighth the size of data cache incur a more noticeable performance decrease, although they are all less than 1%. Setting the level 1 Secure Bit cache to one-quarter or one-eighth the data cache size appears to give the best size and performance tradeoff.

Block Size									
	compress			gcc			go		
	Miss	IPC	Diff.	Miss	IPC	Diff.	Miss	IPC	Diff.
64	0.0007	1.7107	—	0.0056	0.9212	—	0.0022	0.8788	—
32	0.0013	1.7107	0.000%	0.0102	0.9212	0.00%	0.0039	0.8788	0.00%

Number of Sets									
	compress			gcc			go		
	Miss	IPC	Diff.	Miss	IPC	Diff.	Miss	IPC	Diff.
1024	0.0007	1.7107	—	0.0056	0.9212	—	0.0022	0.8788	—
512	0.0007	1.7107	0.00%	0.0056	0.9212	0.00%	0.0022	0.8788	0.00%
256	0.0007	1.7107	0.00%	0.0056	0.9212	0.00%	0.0022	0.8788	0.00%
128	0.0007	1.7107	0.00%	0.0085	0.9212	0.00%	0.0022	0.8788	0.00%
64	0.0020	1.7107	0.00%	0.0455	0.9212	0.00%	0.0030	0.8788	0.00%

Set Associativity									
	compress			gcc			go		
	Miss	IPC	Diff.	Miss	IPC	Diff.	Miss	IPC	Diff.
4	0.0007	1.7107	—	0.0056	0.9212	—	0.0022	0.8788	—
2	0.0007	1.7107	0.00%	0.0057	0.9212	0.00%	0.0022	0.8788	0.00%
1	0.0064	1.7105	0.01%	0.0090	0.9212	0.00%	0.0025	0.8788	0.00%

Table 5.6: Results from varying block size, numbers of sets, and set associativity of level 2 Secure Bit cache. Miss rate, IPC, and percent difference in IPC from the default are recorded (the first line of each section repeats the default configuration).

The final experiment tests the level 2 Secure Bit cache configuration. In this experiment, we configure level 1 Secure Bit cache to the suggested one-quarter data cache size by using 32 sets (running tests with level 1 Secure Bit cache size set the same as data cache size results in so few level 2 accesses that the data are unreliable). Table 5.6 lists the values obtained by adjusting the level 2 Secure Bit cache size down

[1]The SimpleScalar FAQ indicates programs may execute more or less instructions between tests as the environment changes. The difference between full-sized and half-sized Secure Bit cache results may be accountable solely to specific environmental conditions.

IPC			
	compress	gcc	go
Regular	1.7132	0.9213	0.8789
Secure Bit	1.7107	0.9212	0.8781
Percent Difference	0.15%	0.01%	0.09%

Table 5.7: Regular sim-outorder, with 16 KB level 1 and 256 KB level 2 data caches, vs. Secure Bit sim-outorder, with 4 KB level 1 and 16 KB level 2 Secure Bit caches

from the level 2 data cache size.

Level 2 Secure Bit cache, amazingly, has almost no influence on overall IPC. In only one case, where set associativity is 1, does the cache configuration lower IPC; even in this case, performance drops a miniscule 0.01%. For many configurations, the miss rate increases, while the IPC does not change. In these situations, some other latency (perhaps a data cache miss?) in the pipeline masks the miss penalty. In other words, although the cache misses, there is no enduring consequence. Since Level 2 Secure Bit cache has such little influence on performance, we recommend a very small cache.

To conclude, Table 5.7 presents the overall IPC of compress, go, and gcc running on the both the regular version of sim-outorder and the Secure Bit sim-outorder. The Secure Bit simulator uses a level 1 Secure Bit cache one-quarter the size of level 1 data cache (32 sets, 32 byte block size, 4 set associativity) and a level 2 Secure Bit cache one-sixteenth the level 2 data cache size (64 sets, 64 byte block size, and 4 set associativity).

5.4 Recommendations

Based on the results of the previous section, we propose three possible Secure Bit cache configurations:

1. At Secure Bit's introduction (Section 2.2), an alternative approach proposed that, instead of maintaining separate Secure Bit memory, a single bit is added

to every data word in memory. In this case, no caching is necessary as the Secure Bit is implicitly loaded and stored alongside the data word.

2. Level 1 Secure Bit cache set to one-quarter the data cache size and level 2 Secure Bit cache set to one-sixteenth the data cache size results in a less than 1% performance decrease.

3. Even smaller Secure Bit caches—one-sixteenth or less the data cache size—may be used for a very modest performance decrease (an average 1.75%).

It is up to the system designer to decide which configuration is most appropriate for the system. A general-purpose processor may be able to spare the extra cache to retain performance, while an embedded system may be willing to sacrifice some speed in order to save on cache size. In either case, Secure Bit poses virtually no performance penalty on the system, and remains completely transparent to executing programs.

Chapter 6

Conclusion

Buffer-overflow attacks have been a very serious and persistent problem in recent years. They alone have led to dozens of security breaches in end-user applications, Internet services, and operating systems themselves. Using the hardware-level Secure Bit protection scheme, buffer-overflow attacks which have corrupted a stored program address are correctly and efficiently identified before the program causes damage.

In this thesis, we have focused on simulating Secure Bit at the processor level using the SimpleScalar software. Throughout the implementation and analysis of Secure Bit, we have reached several achievements and discoveries:

1. Secure Bit protection of return addresses and function pointer addresses was integrated into the SimpleScalar simulators.

2. Secure Bit operates as expected. Namely, programs not attacked by buffer-overflow execute normally, while those with an address corrupted by buffer-overflow are halted.

3. Secure Bits must be cached just like data; however, Secure Bit cache size may be much smaller (one-quarter to one-sixteenth) than data cache size and impose a less than 1% performance penalty.

Using Secure Bit, developers can be much more confident that their software is safe from malicious users. With program addresses protected, an attacker's primary attack vector is no longer available. Of course, keeping security in mind during development is the best way to prevent attacks; however, Secure Bit can help alleviate at least some of the burden.

6.1 Future Work

Although the Secure Bit technique is extremely effective at preventing address-corrupting buffer-overflows, at this time, it is unable to stop buffer-overflows that attack other kinds of data. For example, buffer-overflow has been used to overwrite passwords instead of addresses. In many of these cases, a security-concious developer should be able to prevent the problem through better programming habits.

Nevertheless, protection of arbitrary data would still be invaluable. Unfortunately, extending Secure Bit both to protect other kinds of data and remain completely transparent to the programmer would probably not be possible. Processor support (i.e. the research done in this thesis) for this already exists—the `SBITSET` and a new `TESTSBIT` instruction could be used by libraries or system calls. However, some sort of interface between the developer and processor's Secure Bit features is necessary, since developers would need to be able to define, set, and test Secure Bit protected data. Perhaps Secure Bits for data regions could be manipulated at the library or system call level, not unlike how process synchronization is managed. Then, developers can "lock" valid data with Secure Bits and test them before using the data.

Bibliography

[1] E. H. Spafford, "The internet worm program: An analysis," Purdue University, West Lafayette, IN 47907-2004, Tech. Rep. CSD-TR-823, 1988.

[2] AlephOne, "Smashing the stack for fun and profit," *Phrack Magazine*, vol. 7, no. 49, Nov. 1996.

[3] Microsoft Corporation. Malicious Software Encyclopedia: Win32/Msblast. [Online]. Available: http://www.microsoft.com/security/encyclopedia/details.aspx?name=Win32%%2fMsblast

[4] Microsoft Corporation. Malicious Software Encyclopedia: Win32/Sasser. [Online]. Available: http://www.microsoft.com/security/encyclopedia/details.aspx?name=Win32%%2fSasser

[5] CERT. CERT Advisory CA-2002-27 Apache/mod_ssl Worm. [Online]. Available: http://www.cert.org/advisories/CA-2002-27.html

[6] D. Litchfield. (2003, Sept.) Defeating the stack based buffer overflow prevention mechanism of Microsoft Windows 2003 Server. [Online]. Available: http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf

[7] P.-A. Fayolle and V. Glaume. (2002) A buffer overflow study: Attacks & defenses. [Online]. Available: http://downloads.securityfocus.com/library/report.pdf

[8] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," in *Proceedings of the 10th Network and Distributed System Security Symposium*, San Diego, California, February 2003, pp. 149–162.

[9] K. Piromsopa, M. R. Fletcher, and R. J. Enbody, "Secure Bit: Hardware, buffer-overflow protection," Michigan State University, Tech. Rep. MSU-CSE-04-48, Nov. 2004.

[10] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: an infrastructure for computer system modeling," *IEEE Computer*, vol. 35, no. 2, Feb. 2002.

[11] J. Pincus and B. Baker, "Beyond stack smashing: Recent advances in exploiting buffer overruns," *IEEE Security & Privacy*, vol. 2, no. 4, pp. 20–27, July/Aug. 2004.

[12] T. Austin. an ARMload of SimpleScalar/ARM. [Online]. Available: http://www.eecs.umich.edu/~taustin/code/arm/ANNOUNCE.ARM

[13] K. Sankaralingam, R. Nagarajan, S. W. Keckler, and D. Burger, "SimpleScalar simulation of the PowerPC instruction set architecture," The University of Texas at Austin, Tech. Rep. TR2000-04, 2000.

[14] C. Price, *MIPS IV Instruction Set*, 3rd ed., MIPS Technologies, Inc., Sept. 1995. [Online]. Available: http://techpubs.sgi.com/library/manuals/2000/007-2597-001/pdf/007-2597-%001.pdf

[15] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann Publishers, 2003, appen. A.

[16] *System V Application Binary Interface*, 3rd ed., The Santa Cruz Operation, Inc., Feb. 1996.

[17] K. Piromsopa and R. Enbody, "Buffer overflow: the fundamentals," Michigan State University, Tech. Rep. MSU-CSE-04-47, Nov. 2004.

[18] T. Austin. SimpleScalar benchmarks. [Online]. Available: http://www.simplescalar.com/benchmarks.html