ARCHITECTURES FOR RUN-TIME VERIFICATION OF CODE INTEGRITY

by

MILENA MILENKOVIC

A DISSERTATION

**Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
The Shared Computer Engineering Program of
The University of Alabama in Huntsville
The University of Alabama at Birmingham
to
The School of Graduate Studies
of
The University of Alabama in Huntsville**

HUNTSVILLE, ALABAMA

2005

In presenting this dissertation in partial fulfillment of the requirements for a doctoral degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this dissertation.

__________________________ ______________

(student signature) (date)

DISSERTATION APPROVAL FORM

Submitted by Milena Milenkovic in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering and accepted on behalf of the Faculty of the School of Graduate Studies by the dissertation committee.

We, the undersigned members of the Graduate Faculty of the University of Alabama in Huntsville and the University of Alabama in Birmingham, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering.

Committee Chair

____ (Date)

____

____

____

____

Department Chair

____

College Dean

____

Graduate Dean

____

ABSTRACT

The School of Graduate Studies
The University of Alabama in Huntsville

Degree Doctor of Philosophy

College/Dept. Engineering/Electrical and Computer Engineering

Name of Candidate Milena Milenkovic

Title Architectures For Run-Time Verification of Code Integrity

With the exponential growth of the number of interconnected computing platforms, computer security becomes a critical issue. As software continues to grow in size and complexity, so does the number of security vulnerabilities: According to the US-CERT Coordination Center, the number of vulnerabilities reported has grown from 171 in 1995 to 4,129 in 2002. One of the major security problems is the execution of unauthorized and potentially malicious code. This problem can be addressed at different levels, from more secure software and operating systems, down to solutions based on hardware support. The majority of the existing techniques tackle the problem of security flaws at the software level, lacking generality, often inducing prohibitive overhead in performance and cost, or generating a significant number of false alarms. On the other hand, a further increase in the number of transistors on a single chip will enable integrated hardware support for functions that were so far restricted to the software domain. Hardware-supported defense techniques have the potential to be more general and more efficient than solely software solutions. This dissertation proposes new architectural extensions to ensure trusted program execution in both high-end and embedded computing platforms. The eight proposed techniques have low performance overhead, low hardware complexity, and minimal or no compiler support.

Abstract Approval: Committee Chair ____

Department Chair ____

Graduate Dean ____

ACKNOWLEDGMENTS

"I thank you for making this day necessary."
Yogi Berra

I wish to express my deepest gratitude to many persons who made my PhD dissertation possible.

I thank my advisor, Dr. Emil Jovanov, for his constant support and guidance. I will be always grateful to Dr. Reza Adhami, Chair of the Electrical and Computer Engineering Department, for encouraging me to pursue PhD studies, and for providing me with financial support through the teaching assistantship during the Spring 2002 semester. Many thanks to Dr. Jeff Kulick for inspiring talks about malicious attacks and hardware-supported security. Thanks go out also to other members of my committee, Dr. Gary J. Grimes, Dr. Peter Slater, Dr. B. Earl Wells, and Dr. Seong-Moo Yoo, for their invaluable feedback. Dr. Yale Patt from the University of Texas in Austin first introduced me to the wonderful philosophy of Yogi Berra, and has been a source of inspiration for all my research work. Last but not least, this dissertation would not be possible without my husband, Dr. Aleksandar Milenkovic, who encouraged me to pursue the PhD research, collaborated with me on secure architectures, and provided me an inspirational work environment in the LaCASA laboratory.

I also thank my family in Serbia, for their unconditional love and support.

TABLE OF CONTENTS

Page

LIST OF FIGURES VIII

LIST OF TABLES X

CHAPTER

1 INTRODUCTION 1

1.1 Background and Motivation 1

1.2 Existing Techniques for Defense Against Code Injection Attacks 2

1.3 Architectures For Instruction Block Signature Verification 3

1.4 Main Contributions 4

1.5 Dissertation Outline 4

2 SOFTWARE VULNERABILITIES AND CODE INJECTION ATTACKS 5

2.1 Stack-Based Buffer Overflow Attacks 5

2.2 Heap-Based Buffer Overflow Attacks 6

2.3 Format String Attacks 7

2.4 Integer Error Attacks 8

2.5 Double free() attacks 8

2.6 A Format String Attack Example 9

3 EXISTING TECHNIQUES FOR DETECTION AND PREVENTION OF CODE INJECTION ATTACKS 12

3.1 Static Software-Based Techniques 12

3.2 Dynamic Software-Based Techniques 17

3.3 Defense Techniques With Hardware Support 31

3.4 Other Related Work 38

4 PROPOSED ARCHITECTURES FOR INSTRUCTION BLOCK VERIFICATION..........................41

4.1 Basic Mechanism of Proposed Techniques..41

4.2 Taxonomy of Proposed Techniques...44

4.3 Details of SIGCE Techniques 50

4.4 Details of SIGCT Techniques 57

4.5 Details of SIGB Techniques 61

4.6 Discussion 70

5 EXPERIMENTAL METHODOLOGY 74

5.1 Evaluation of Proposed Techniques...74

5.2 ELF Format..75

5.3 Secure Installation of Files in ELF Format 78

5.4 SimpleScalar Simulator 79

5.5 SimpleScalar Modifications 82

5.6 Custom-Made Trace-Driven Simulator...82

5.7 Simulator Parameters 86

5.8 Benchmarks 89

6 EVALUATION RESULTS...94

6.1 SIGC Evaluation 94

6.2 SIGB Evaluation 124

7 CONCLUSION 134

REFERENCES...137

LIST OF FIGURES

Figure Page

2.1 An illustration of a buffer overflow attack on the stack 6

2.2 An illustration of the use of the %n format character 7

2.3 Allocated and free memory chunk organization, GNU C library malloc() 9

2.4 An example of a vulnerable program 10

2.5 Malicious input and the corresponding output for the above program, and the stack content (SP – stack pointer) 11

4.1 Mechanism for trusted instruction execution 42

4.2 An implementation of a 4-bit MISR 43

4.3 Processor components 43

4.4 Taxonomy of proposed instruction block verification techniques 45

4.5 Modification of executable code 47

4.6 Qualitative assessment of signature verification techniques in the performance overhead - hardware complexity design space 49

4.7 SIGCED: Signature verification control flow 51

4.8 SIGCED: Instruction Block Signature Verification Unit 53

4.9 SIGCEK: Signature verification control flow 54

4.10 SIGCEK: Instruction Block Signature Verification Unit 55

4.11 SIGCEV: Signature verification control flow 56

4.12 The content of an I-cache line with the SIGCEV technique 57

4.13 SIGCTD: Signature verification control flow 59

4.14 SIGCTK: Signature verification control flow 60

4.15 Instruction streams. 61

4.16 SIGBEV: An example of the original and the protected code 63

4.17 SIGBEV: Instruction Block Signature Verification Unit 64

4.18 SIGBEV Procedures 64

4.19 SIGBTK: Instruction Block Signature Verification Unit 67

4.20 SIGBTK Procedures 68

4.21 SigTable access using segment approach 69

5.1 Linking and execution view of an ELF file 76

5.2 The main simulator loop body in the sim-outorder simulator 80

5.3 Pseudo-code for the trace-driven SIGB simulator 84

5.4 Pseudo-code for the function verify_signature() 85

6.1 SIGC: embedded processor configuration, I-cache line 128B, 32-bit bus, slow core 110

6.2 SIGC: embedded processor configuration, I-cache line 128B, 64-bit bus, slow core 111

6.3 SIGC: embedded processor configuration, I-cache line 128B, 32-bit bus, fast core 112

6.4 SIGC: embedded processor configuration, I-cache line 128B, 64-bit bus, fast core 113

6.5 SIGC: embedded processor configuration, I-cache line 64B, 32-bit bus, slow core 114

6.6 SIGC: embedded processor configuration, I-cache line 64B, 64-bit bus, slow core 115

6.7 SIGC: embedded processor configuration, I-cache line 64B, 32-bit bus, fast core 116

6.8 SIGC: embedded processor configuration, I-cache line 64B, 64-bit bus, fast core 117

6.9 SIGC: high-end processor configuration, I-cache line 128B 118

6.10 SIGC: high-end processor configuration n, I-cache line 64B 119

6.11 SIGBTK: Number of S-cache misses as a function of S-cache size I-cache size: 32K 128

6.12 SIGBTK: Number of S-cache misses as a function of S-cache associativity I-cache size: 32K ... 129

6.13 SIGBTK: Number of memory accesses per 1M instructions due to S-cache misses with the segmented binary search, (128-set, 2-way) S-cache, and 32K I-cache 130

LIST OF TABLES

Table Page

3.1 Static software-based techniques 14

3.2 Techniques that instrument code to verify run-time bounds 18

3.3 Attack-specific techniques 19

3.4 "Safe dialects" of C 19

3.5 Obfuscation techniques 19

3.6 Program monitoring techniques 20

3.7 Techniques with hardware support 32

4.1 Pros and cons of different techniques 49

5.1 Some common ELF file sections 77

5.2 Descriptions of .c files used by SimpleScalar simulators 81

5.3 Simulator parameters for the embedded processor configuration 87

5.4 Simulator parameters for the high-end processor configuration 88

5.5 Description of benchmarks from embedded domain 90

5.6 Benchmark code size and executed instructions for embedded systems 91

5.7 Description of SPEC CPU2000 benchmarks and the size of precompiled Alpha binaries 92

5.8 The size of SPEC CPU2000 benchmarks when compiled with the ARM gcc compiler 93

6.1 Base: I-cache misses per 1000 instructions in embedded processor configurations 96

6.2 Base: CPI in embedded processor configurations, slow core, memory bus 32 bits 97

6.3 Base: CPI in embedded processor configurations, slow core, memory bus 64 bits 98

6.4 Base: CPI in embedded processor configurations, fast core, memory bus 32 bits 99

6.5 Base: CPI in embedded processor configurations, fast core, memory bus 64 bits 100

6.6 Base: I-cache misses per 1M executed instructions in high-end processor configurations 101

6.7 Base: CPI in high-end processor configurations 101

6.8 SIGCEV: I-cache misses per 1000 executed instructions in embedded processor configurations 107

6.9 SIGCEV: I-cache misses per 1M executed instructions in high-end processor configurations 108

6.10 S-cache misses per 1000 executed instructions in high-end processor configurations 108

6.11 S-cache misses per 1000 executed instructions in embedded processor configurations 109

6.12 Percentage of file size increase for SPEC CPU2000 benchmarks 121

6.13 Percentage of file size increase for benchmarks from the embedded domain 122

6.14 Percentage of file size increase for SPEC CPU2000 benchmarks 123

6.15 SIGBTK: Number of I-cache misses and signature verifications per 1M instructions 126

6.16 SIGBTK: Number of S-cache misses per 1M instructions 127

6.17 SIGBEV: Number of I-cache misses and signature verifications per 1M instructions 131

6.18 Number of basic blocks and percentage of file size increase 133

CHAPTER 1

INTRODUCTION

"The art of war teaches us to rely not on the likelihood of the enemy's not coming, but on our own readiness to receive him; not on the chance of his not attacking, but rather on the fact that we have made our position unassailable."

Sun Tzu, "The Art of War"

With the exponential growth of the number of interconnected computing platforms, computer security becomes a critical issue. Today's society now more than ever relies upon computers and networks. Networked computing platforms make up the fabric of society's infrastructure; in fact, ubiquitous accessibility and interconnectivity are the driving forces in our modern economy, education, entertainment, medicine, transportation, and the military. Unfortunately, by connecting a computer system to the Internet or a local network, we expose its vulnerabilities to potential attackers. Failing to resist attacks can incur significant direct costs as well as costs in lost revenues and opportunities. The utmost importance of system security is further underscored by the increased complexity of high-end systems as well as the expected proliferation of diverse Internet-enabled, low-end embedded systems -- ranging from home appliances, cars, and sensor networks to personal health monitoring devices.

1.1 Background and Motivation

A very large group of malicious attacks on applications running on general-purpose processors consists of different techniques that impair the software integrity, by injecting and then executing the malicious code instead of regularly installed programs. The most widely known type of such attacks is so-called stack smashing, where an attacker overflows a buffer stored on the stack with a malicious code

sequence and replaces a valid return address with the malicious code address [1]. In addition, various other examples of attacks exist, such as heap overflow [2], format string attacks [3], and attacks exploiting integer errors [4] or dangling pointers [5]. The number of reported software vulnerabilities has grown from 171 in 1995 to 4,129 in 2002, according to the United States Computer Emergency Readiness Team Coordination Center (US-CERT/CC) [6].

Applications targeting embedded systems may suffer from the same vulnerabilities as applications running on general-purpose platforms. For example, one recent Cyber Security Bulletin from US-CERT reports multiple buffer overflow vulnerabilities in a Bluetooth connectivity program for Personal Digital Assistants (PDAs) [7]. Another US-CERT Cyber Security Bulletin indicates an emerging trend of mobile phone viruses [8]. As the communication and computation capabilities of smart phones, PDAs, and other embedded systems continue to grow, so will the number of malicious attacks trying to exploit code vulnerabilities.

1.2 Existing Techniques for Defense Against Code Injection Attacks

The multitude of code injection attacks prompted development of a large number of predominantly software-based counter-measures. Static software techniques rely on formal analysis and/or programmers' annotations to detect security flaws in the code, and then leave it to the programmers to correct these flaws. However, the use of these techniques has yet to become a common programming practice. Moreover, they fail to discover all vulnerabilities, suffer from false alarms, or put an additional burden on programmers. On the other hand, dynamic software techniques augment the original code or operating system to detect malicious attacks and to terminate attacked programs, or to reduce the attacker's chances of success. Though effective, these techniques can result in significant performance overhead and usually require program recompilation, so they are not readily applicable to legacy software.

Current trends in both hardware and software make us believe that dedicated processor resources should be used to ensure software integrity, consequently improving computer system security [9]. Software techniques by themselves are unlikely to counter all attacks, since more complex applications have potentially a larger number of defects, computing systems are becoming more diverse, and time-to-market constraints severely limit testing time. On the other hand, a further increase in the number of

transistors on a single chip will enable integrated hardware support for functions that so far have been restricted to the software domain. A form of hardware protection from buffer overflow has already found its way into mainstream general-purpose processors, AMD's Athlon-64 and Intel's Itanium [10]. Several recent research efforts propose hardware-supported techniques to prevent unauthorized changes of program control flow. Most of these techniques focus only on stack smashing or have significant performance overhead, or do not thoroughly explore the implications of implementation choices. We believe that there is a need for a new hardware security layer to prevent the whole class of code injection attacks.

1.3 Architectures For Instruction Block Signature Verification

This dissertation proposes and evaluates new architectural extensions to ensure trusted program execution in high-performance and embedded computing platforms at minimal cost, power overhead, and performance loss. We propose several new techniques that share a common mechanism: Instruction blocks are signed using secret hardware keys during the secure program installation process, and signatures are stored with the code. During program execution, signatures are recalculated from instructions and compared to the stored signatures. If the two values do not match, the program cannot be trusted and should be terminated. The proposed techniques differ in type of protected instruction blocks, signature placement in the address space, signature placement in the physical memory, and signature handling after the verification.

Hardware-supported techniques have the potential to provide trusted program execution with lower overhead in performance and overall power consumption than techniques relying solely on software. Instead of vulnerability-specific solutions, the proposed architectures offer protection from a whole class of vulnerabilities that allow execution of a malicious code. Moreover, since with our mechanism each program requires a secure installation process, viruses cannot penetrate the system without explicit permission by the user. The proposed mechanism does not require significant processor changes and can be implemented even as a separate co-processor; it is cost-effective and requires no changes in legacy source code; several considered techniques do not require compiler support, while others require minimal compiler support. In addition, encrypted instruction block signatures protect the code from software tampering, and enable fault detection in error-prone environments such as Space.

1.4 Main Contributions

The main contribution of this dissertation is a proposal of a novel hardware-supported mechanism for defense against code injection attacks. The proposed mechanism is based on run-time verification of instruction block signatures. We give taxonomy, detailed design, and performance evaluation for eight implementations of this mechanism.

Another contribution is an extensive survey of related work. The survey encompasses a wide range of software and hardware solutions proposed to counter malicious code injection attacks. Finally, the work on this dissertation resulted in a number of extensions for SimpleScalar simulator and a custom-made trace-driven simulator.

1.5 Dissertation Outline

The rest of this dissertation is organized as follows. Chapter 2 gives an overview of security vulnerabilities that may be exploited by code injection attacks.

Chapter 3 gives a survey of software-based static and dynamic defense techniques, and hardware-supported techniques. Several related fault-tolerant techniques and anti-tampering techniques are also mentioned. The proposed mechanism for run-time instruction block verification is explained in Chapter 4. This chapter also includes taxonomy of proposed architectural extensions, detailed descriptions of each technique, and a discussion of implementation challenges and limitations.

Chapter 5 describes experimental methodology used in this dissertation. It gives a short description of execution-driven simulator SimpleScalar and the modifications we made. It also describes a custom-made trace-driven simulator. The ELF format and various benchmarks used for evaluation are also described in this chapter, as well as metrics used for evaluation and simulator parameters fixed for all experiments.

Chapter 6 presents evaluation results and discusses them for a wide set of benchmarks. Finally, Chapter 7 states conclusions and indicates future research possibilities.

CHAPTER 2

SOFTWARE VULNERABILITIES AND CODE INJECTION ATTACKS

"We made too many wrong mistakes."
Yogi Berra

A successful code injection attack must achieve two goals: it must inject the malicious code sequence, and it must change the value of a code pointer to point to the address of the injected code. The most common software vulnerabilities that can be exploited by code injection attacks are input buffers without boundary checks, both on the stack and on the heap; functions from *printf* family accepting input arguments as format strings; and errors related to dynamic memory allocation, such as freeing an already freed pointer. Attacks exploiting these vulnerabilities are rather complex and require deep understanding of underlying architecture, operating system, and the application under attack. In this chapter we give a short description of each vulnerability and the corresponding attack, and provide a detailed walk-through example for format string vulnerability.

2.1 Stack-Based Buffer Overflow Attacks

The mechanism of stack-based buffer overflow attacks, so-called "stack smashing," is probably the most widely known code injection mechanism [1, 11, 12]. Figure 2.1 illustrates one such attack: a function accepts untrustworthy values into a local buffer, which is stored on the stack. In most architectures, the direction of the stack growth is opposite to the direction of memory address growth, so if we overflow a buffer on the stack over its limits, we can overwrite any location on the stack in the address

space after the beginning of the buffer. One such location holds the return address of the vulnerable function. Hence, if that function does not verify whether the length of the input exceeds the buffer size, an attacker might overflow the buffer to insert the malicious code and overwrite the return address with the address of the malicious code.

Stack smashing is probably the most exploited code injection attack, since an attacker knows that a return address is somewhere near the local function variables, and only needs to probe to find its exact location.

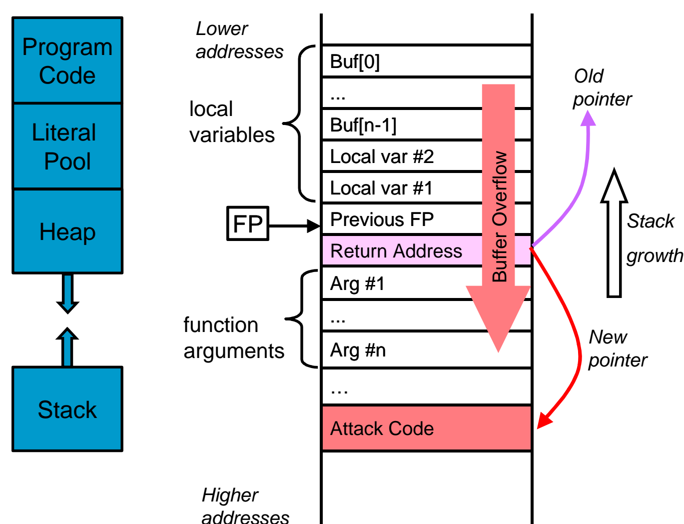

Figure 2.1 An illustration of a buffer overflow attack on the stack

2.2 Heap-Based Buffer Overflow Attacks

Buffer overflow vulnerabilities are not limited to buffers on the stack. Heap-based buffer overflow attacks are another attack category that exploits buffer overflows [2]. Let us assume that a buffer is stored on the heap in relative proximity to a code pointer, e.g., a function pointer. If that buffer accepts input without length verification, it may overwrite the function pointer with the address of attacker's choice, where the same or other buffer overflow attack stored the malicious code.

2.3 Format String Attacks

Format string attacks exploit the ability of functions from the *printf* family to actually accept an input argument as a format string, for example by writing *printf(string)* instead of *printf("%s", string)* [3, 13]. The *printf* function interprets its first argument as a format string, and scans the format string looking for special format characters such as *%d*, *%x*, *%s*, etc. These characters specify the type of arguments to be retrieved from the stack and the corresponding output format. If an attacker can pass format strings to the *printf* function, he or she can exploit the *printf* mechanism to read the content of any memory location. This is a so-called *read attack*, which may be used to gain knowledge to mount the actual *write attacks*, such as stack smashing.

Allowing user-defined format strings also enables write attacks, based on the *%n* format character. If *%n* is encountered in the format string, the number of characters output before *%n* is stored at the address passed as the next argument. Figure 2.2 shows one such example: in the first *printf* call, the number of characters output before *%n* is 6: 4 digits of x, a comma, and a space. The value 6 is stored in the variable *pos*, as shown in the memory layout. The use *%n* actually results in storing the number of characters that should have been output, not the actual count of characters that were output. For example, the *snprintf* function writes no more than *size* characters to the string *str*, where both *str* and *size* are arguments of *snprintf*. Let us assume that the size of a string buffer *buf* is 20. Then the function call `snprintf(buf, sizeof(buf),"%.100d%n",x,&pos)` will store 100 in the variable *pos*. An intelligent use of *%n* enables the attacker to write any value to almost any address in the program space.

```
int main(){
 int pos, x=1389, y=20044;

 printf("%d, %n%d\n", x, &pos, y);
 printf("The offset was %d\n", pos);
}
```

Output

```
1389, 20044
The offset was 6
```

Address	0	1	2	3	
	…				
0xbffffacc	4c	4e	00	00	y
0xbffffad0	6d	05	00	00	x
0xbffffad4	06	00	00	00	pos

Figure 2.2 An illustration of the use of the %n format character

2.4 Integer Error Attacks

Various integer errors that can compromise the system safety include unsigned integer overflow and underflow, precision error, and integer comparison [4]. A code injection attack cannot be based only on integer errors, but these errors can enable another type of attacks.

For example, overflow of an unsigned integer actually causes storing by modulo: a one-byte unsigned *char* variable can hold values 0-255, so the value 256 will be stored as 0, 257 as 1, etc. Let us assume that such a variable is used for buffer allocation and that it is overflowed. For instance, we want a buffer of 257 bytes, but the variable that controls dynamic buffer allocation is an *unsigned char*. Instead of 257 bytes, the size of the buffer is 1 byte. A "safe" function that stores data in the buffer may even verify whether the output exceeds the allowed values. However, the allowed value is 257 bytes, and the allocated size is only 1 byte. Here is an opportunity for a heap-based buffer overflow as explained before.

2.5 Double free() attacks

To understand the principle of double *free()* vulnerabilities, we must first understand how dynamic memory allocation and deallocation work. Dynamic memory management information is usually kept together with the actual allocated memory chunk. Figure 2.3 shows the fields in allocated and free memory chunks, when using the GNU C library [14]. An allocated chunk has *prev_size*, *size*, and *data* fields. The *prev_size* field defines the size of the previous chunk if it is free (i.e., not allocated), or it belongs to data field of the previous chunk. The *size* field defines the size of the current chunk and also includes some status bits. The actual data is stored in the *data* field, and the pointer to data *mem* is what is returned by *malloc()*. When a memory chunk is freed, it is linked to a doubly linked list of all free chunks of the similar size, so it also has fields *fd* (pointer to a chunk forward in the list) and *bk* (pointer to a chunk backward in the list). If one of its physical neighbors is free, these two chunks are merged into one larger chunk. The linked list of free chunks is ordered by size, so that a chunk with the same size as some chunk in the list is inserted before that chunk, and all relevant forward and backward fields are set accordingly. The list is re-linked when a chunk is allocated by using a macro *unlink()*.

When a chunk C1 is freed twice, its *bk* and *fd* fields both point to itself, if there was no merging to a larger chunk between two calls to *free()* [5, 15]. When a program now needs to allocate a chunk of the same size as C1, the C1 will be unlinked from the list of free chunks and user data will be written in the field *data*. However, since C1 points to itself, it will not be really unlinked, so first eight bytes of *data* are actually *bk* and *fd*. Hence, an attacker might overwrite the *fd* and *bk* with addresses of his/her choice.

Next time when a chunk of C1 size is requested, *unlink()* will cause the content of *bk* field to be written at the address stored in *fd* plus offset of 12 bytes. One option is to overwrite a return address with the address of injected code in the data field. Therefore, exploiting the double *free()* vulnerability the attacker may satisfy both conditions for code injection attacks – inject the code and change a code pointer.

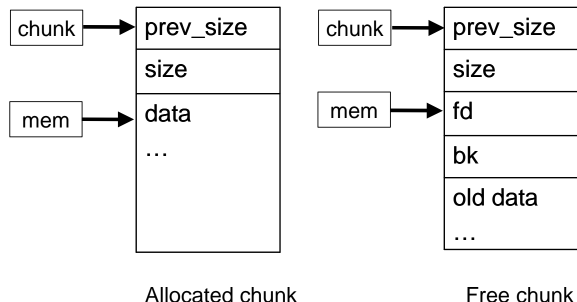

Figure 2.3 Allocated and free memory chunk organization, GNU C library malloc()

2.6 A Format String Attack Example

Figure 2.4 illustrates the effect of format string attack on a variable named *x*, which is set to 1 and not changed by any of the program instructions. The code in Figure 2.4 is slightly modified code from [3]. This attacks exploits the use of *snprintf()* with no specified format string. If the input argument argv[1] does not contain format characters, it is treated as a string, and its *sizeof(buf)* characters are simply copied to the string buffer *buf*. However, any format character will cause a value to be popped from the stack and stored in the *buf*. If we know the address of the variable *x*, we can change the value of *x* by using an input string that includes both the address of *x* and the *%n* format character. One way to do it is to store the address of *x* at the beginning of the buffer *buf*, and to include enough format string characters so that the argument for *%n* is read precisely from the *buf* beginning.

```
#include <stdio.h>

int main(int argc, char **argv){
  char unsigned buf[96];
  int x, y;

  if(argc != 2) exit(1);
  x=1; y=2;
  snprintf(buf, sizeof buf, argv[1]);
  buf[sizeof(buf) - 1] = 0;
  printf("buffer (%d): %s\n", strlen(buf), buf);
  printf("x is %d/%#x (@ %p)\n", x, x, &x);
  printf("y is %d/%#x (@ %p)\n", y, y, &y);
  printf("buffer[3:0]: %2x%2x%2x%2x\n",
       buf[3], buf[2], buf[1], buf[0]);
  return 0;
}
```

Figure 2.4 An example of a vulnerable program

Let us assume that the *x* is stored at the address `0xbffff8cc` (Figure 2.5). "Above" *x* on the stack are stored *y* and some other three values, and the *buf* starts below *x*. The input string `\xcc\xf8\xff\xbf.%08x.%08x.%08x.%08x.%08x%n` will cause the following to happen. First, `0xbffff8cc` will be stored in *buf[0:3]*. Then, five integer values will be popped from the stack and stored to *buf*, starting from the location below the stack pointer SP (this is the stack pointer when *snprintf()* starts to execute). Finally, the number of stored characters as specified by the format string will be stored to the address popped from the stack, i.e., the address of *x* stored at the beginning of the *buf*. Since the format string specifies writing of 49 characters before *%n* (4+9*5), the value of *x* will be changed to 49, as seen in the program output. Note that we used perl for input, since the address of *x* in hexadecimal form could not be specified when program was executed interactively under RedHat 7.0 Linux.

Input

```
perl -e 'system
"./fmtme","\xcc\xf8\xff\xbf.%08x.%08x.%08x.%08x.%08x%n"'
```

Output

```
buffer (49): Ìøÿ¿.420069e8.4212a2d0.bffffaa0.00000002.00000001
x is 49/0x31 (@ 0xbffff8cc)
y is 2/0x2 (@ 0xbffff8c8)
buffer[3:0]: bffff8cc
```

```
snprintf(buf,96,
"\xcc\xf8\xff\xbf.%08x.%08x.%08x.%08x.%08x%n");
```

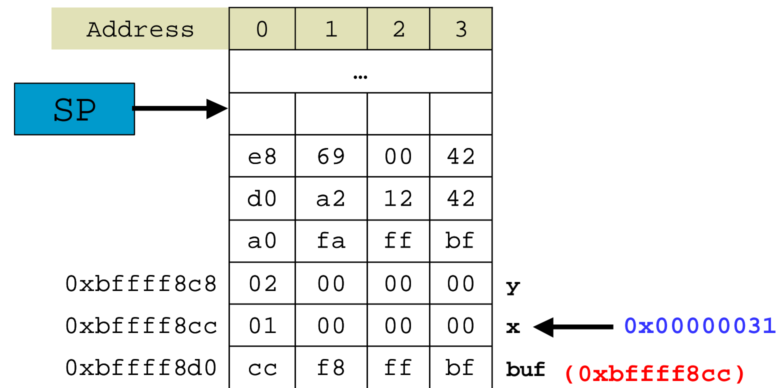

Figure 2.5 Malicious input and the corresponding output for the above program, and the stack content (SP –stack pointer)

CHAPTER 3

EXISTING TECHNIQUES FOR DETECTION AND PREVENTION OF CODE INJECTION ATTACKS

"Don't always follow the crowd, because nobody goes there anymore; it's too crowded."
Yogi Berra

Techniques for countering code injection attacks can be classified in two broad categories: those that are completely software-based and those that require some hardware support. The software techniques can be further classified into static techniques and dynamic techniques. Static software-based techniques try to find possible security vulnerabilities in the code, so they can be corrected before the release version of the code. Dynamic software-based techniques augment the code so that in run-time an attack can be detected, prevented, or made very difficult, depending on a particular technique. Younan et al. survey a large number of software-based techniques [15], and Cowan et al. give a qualitative assessment of several buffer-overflow defenses [16, 17]. More recent hardware-aided techniques are less often studied. The goal of this chapter is to give an up-to-date survey of existing techniques for prevention and detection of code injection attacks.

3.1 Static Software-Based Techniques

Static code analysis can find a significant number of security flaws and suggest where changes in the code should be made. However, the problem of static analysis is generally undecidable [18], so it is virtually impossible to discover all vulnerabilities in any given program by automated static analysis alone. Completely automated tools for detection of security-related flaws must choose between precise but not

scalable analysis and lightweight analysis that may produce a lot of false positives and false negatives. The need for precise automated analysis can be alleviated if programmers manually add specially formulated comments about program constraints, but such techniques put an additional burden on programmers. Moreover, one can argue that adding program constraints may be as error-prone as programming. Table 3.1 lists static software-base techniques aimed to discover potential security defects, including the possibility of code injection.

While the simple Unix utility *grep* can be used to find some of the known security vulnerabilities in the code [19], it is not able to make distinction between safe and unsafe use of potentially vulnerable functions, nor it can assign rank to security warnings. For example, if we want to find format string vulnerabilities, the *grep* search for *printf()* function will give all *printf()* instances, and the majority of instances will be safe.

Several tools are developed to be essentially a smart *grep*. Viega et al. developed a token-based scanning tool called It's The Software, Stupid! Security Scanner (ITS4) [20-22]. ITS4 breaks a source file into tokens and then compares tokens against a vulnerability database. The analysis results can be further refined by checking the parameters of string functions and race conditions. The report severity is reduced for function calls with constant string parameters. On the other hand, a heuristic for race condition check increases report severity if it discovers a related race condition. Willander developed a similar open-source tool called Flawfinder [23]. Flawfinder checks for Unicode constant strings, which may further reduce the number of false alarms. Another similar tool is RATS (Rough Auditing Tool for Security) [24]. DeKok developed a tool called PScan (a limited problem scanner for C source files) [25]. PScan looks only for format string vulnerabilities and it gives a warning if a format string is not a constant value.

Table 3.1 Static software-based techniques

Technique	Description
ITS4 [20-22], Flawfinder [23], RATS [24]	Marks potential vulnerabilities by comparing parsed code to a vulnerability database
PScan [25]	Scans code for format string vulnerabilities
BOON [26, 27]	Automatically detects potential string buffer overflow vulnerabilities
Buffer Overrun Tool [28]	Automatically detects buffer overflow vulnerabilities by using linear programming
ARCHER [29]	Automatically detects memory access errors
Splint [30-32]	Finds potential vulnerabilities if annotated by programmers
Propagation of tainted qualifiers [33]	Detects format string vulnerabilities by using a special qualifier for untrustworthy data
CSSV [34]	Detects all string manipulation errors in the code annotated by contracts
Compiler extensions using metacompilation [35]	Detects security errors by using programmer-written metacompiler extensions
Eau Claire [36]	Detects security errors by using error specifications and an automatic theorem prover
UNO [37]	Detects several types of errors, plus user-defined properties

Wagner et al. propose a tool for automated detection of code that might cause overflow of string buffers and introduce a tool prototype called Buffer Overrun detectiON (BOON) [26, 27]. The problem of string buffer overflow is formulated as an integer constraint problem: a string buffer is modeled as a pair of integers, one for the current buffer length and another for the allocated size, so the tool needs to verify whether the maximum length is not greater than the allocated size. The BOON's analysis is flow-insensitive and context-insensitive. Flow-insensitive means that the order of statements is ignored, and context-insensitive means that calls to a same function from different places are not treated in different way. The authors admit they sacrificed precision in order to have a scalable tool. BOON produces a relatively high number of false positives, e.g., 40 out of 44 generated warnings for *sendmail* program are false positives.

Ganapathy et al. propose a similar approach to BOON, but with more precise pointer analysis and context-sensitivity [28]. A code understanding tool is first used to generate abstract syntax trees (AST) for program expressions and points-to information, and this data is used to generate linear constraints. Constraints that cannot be solved by linear programming are then removed from the set of all constraints (variables that get an infinite value and uninitialized constraint variables). The rest of constraints are solved using two solvers based on linear programming. Finally, heuristics are used to decide whether a particular buffer can be overflowed. This approach still lacks flow-sensitivity. The authors also note that modeling constraints in terms of pointers to buffers instead of buffers can lead to false negatives.

Xie et al. proposed another tool for automatic detection of memory access errors, named ARCHER (ARray CHeckER) [29]. ARCHER uses interprocedural, flow-sensitive and context-sensitive data-flow analysis: C source code is first parsed into AST trees and transformed to a canonical representation with reduced number of syntactic constructs. The canonical representation is then used to generate a control-flow graph (CFG) for each function and an approximate program call graph. This call graph is traversed bottom-up: for each function call, the corresponding CFG is traversed using randomized depth-first search and the ARCHER solver module is called to evaluate conditional expressions and verify whether memory accesses are unsafe. The tool gives warnings for unsafe memory accesses. Although ARCHER can discover more errors and give less false positives when compared to BOON, it still cannot reliably discover all memory access errors. It does not handle C string operations and does not track function pointers.

The need for precise automated analysis can be alleviated if programmers add specially formulated comments about constraints. Larochelle and Evans propose one such tool called Splint [30-32]. Splint is an extension of LCLint, an annotation-assisted lightweight static checking tool, developed by Evans et al. [38]. Function preconditions and postconditions can be stated using *requires* and *ensures* clauses. Within these clauses programmers may specify minimum and maximum buffer indices that can be read or written to: *maxSet*, *minSet*, *maxRead*, and *minRead*.

Shankar et al. propose a tool for detection of format string vulnerabilities [33], built on the top of *cqual*, an extensible type qualifying framework for language C [39]. The authors propose an additional

C qualifier, *tainted*, for data that cannot be trusted. The tool then analyzes how tainted data propagates through the program and gives a warning if tainted data is used as a format string.

In a recent study, Dor et al. propose a tool for detection of all string manipulation errors with very few false positives, CCSV (C String Static Verifier) [34]. While the previous work by the same authors discussed a similar algorithm with certain limitations [40], CCSV is able to find all such errors. It can handle all C constructs, including multi-level pointers, multidimensional structures, and pointer arithmetic. However, this approach requires that the potentially vulnerable functions are annotated with so-called contracts, including pre-conditions, post-conditions, and potential side effects. CCSV reports an error when a specified post-condition is not guaranteed to hold. The authors also propose algorithms for automated strengthening of post- and pre-conditions, reducing the burden placed on the programmer, but at the cost of increased imprecision.

Instead of annotating the code, a programmer can write compiler extensions that describe potential security errors. Ashcraft and Engler propose to use the metacompilation approach to look for security errors in the code [35]. With metacompilation a programmer can easily add a high-level checking rule to the compiler. The authors use belief inference approach to detect incomplete rule specifications. A range checker extension is used to demonstrate the metacompiler approach. The range checker finds errors in the Linux kernel code where the integer data from untrustworthy sources is used without first being checked.

Chess proposes another code checker, named Eau Claire [36]. Similar to the metacompilation approach, Eau Claire requires specifications of security vulnerabilities it is supposed to find. For each function, the function code and security specifications are translated to a series of verification conditions, which are then used as an input to an automatic theorem prover. A disproved theorem means that the corresponding function violates its security specifications.

Holzmann proposes a code checker named UNO [37]. The author extends an open-source C parser to generate control-flow graphs and check the code for the use of uninitialized variables, nil-pointer dereferencing, and out-of-bound array indexing. UNO can also check for user-defined properties, where property definitions consist of actions and queries.

The authors of the static code analysis techniques rarely give quantitative comparisons of their techniques with other approaches. Wilander compared five static tools, ITS4, Flawfinder, RATS, BOON,

and Splint [41]. His test cases are based on 20 vulnerable functions from the ITS4 database and consist of 21 safe and 23 unsafe function calls with possibilities of buffer overflow and string format errors. Not surprisingly, "smart *grep*" techniques generated over 50% false positives and very few false negatives. Zitser et al. compared ARCHER, Boon, Splint, UNO and a commercial tool Polyspace C Verifier [42]. Test cases are based on known buffer overflow vulnerabilities and the corresponding code patches extracted from real applications, since the compared techniques were not able to process the complete code of *sendmail* and similar vulnerable programs. Splint and Polyspace were able to find a significant number of errors, but all tools gave a very high number of false warnings.

3.2 Dynamic Software-Based Techniques

Precise static analysis and high coverage testing techniques can reduce the number of security vulnerabilities, but they can rarely solve all potential problems before the code is released. Dynamic software techniques aim to prevent or detect attacks in run-time. We can distinguish several groups of these techniques. The largest group encompasses techniques that automatically add run-time checks for security vulnerabilities to code (Table 3.2); some of these techniques are designed only for testing purposes, or target only one type of attack (Table 3.3). Several "safe dialects" of language C prevent code injection attacks by restricting the use of unsafe constructs, static analysis, run-time checks, and changes in memory management (Table 3.4). Various obfuscating techniques make vulnerability exploits more difficult (Table 3.5). Another group consists of various monitoring techniques (Table 3.6). Finally, some portions of the memory address space can be made non-executable with operating system support, thus preventing the execution of injected code stored at those addresses. Most of dynamic software techniques require program recompilation, so they are not readily applicable to legacy software. These techniques essentially increase the number of executed instructions, so they incur a significant performance overhead.

Table 3.2 Techniques that instrument code to verify run-time bounds

Technique	Description
bcc [43]	Extended pointers are used to checks bounds on pointer dereferences and array accesses
RTCC [44]	Similar approach as bcc
Safe-C [45]	Safe pointers enable detection of both spatial and temporal memory access errors
Guarding [46]	Run-time checking is decoupled from the original computation
Backward-compatible bounds checking [47]	Unchanged pointer representation
Type-assisted run-time checks [48]	Run-time checks are based on type information stored in "mirror" memory
Type-assisted dynamic buffer overflow detection [49], TIED+LibsafePlus [50]	Target only buffer overflows
Fail-Safe ANSI-C Compiler [51]	A memory-safe implementation of full ANSI-C
CRED [52]	Detect buffer overflows of user-supplied string data
Optimized bounds checking using metadata [53]	Information about pointers is kept separated from the pointers
Boundless memory blocks [54]	Allows program to continue after an out-of-bound write, by storing it in a hash table
Appropriate location bits [55]	"Unsafe" pointers may point only to locations designated as appropriate in "mirror" memory
Purify [56], STOBO [57], detection of input-related security faults [58], SFI [59]	Testing tools

Table 3.3 Attack-specific techniques

StackGuard [60], StackShield [61], RAD [62, 63], SSP [64]	Defense against stack smashing
Libsafe, Libverify [65]	Defense against stack smashing implemented in libraries
HEALERS [66, 67]	Detects heap-based buffer overflows
Modified dlmalloc() [68]	Protects dynamic allocation information
FormatGuard [13]	Detects format string attacks

Table 3.4 "Safe dialects" of C

Vault [69, 70]	Enables resource management protocols in source code; supports region-based memory management
Cyclone [71, 72]	"Unsafe" C features replaced by 'safe" extensions; supports garbage-collection or region-based memory management
CCured [73]	Additional pointer types: safe, sequence, dynamic; garbage-collection
Control-C [74, 75]	Restricts dynamic memory allocation and pointer arithmetic; region-based memory management

Table 3.5 Obfuscation techniques

ASLR [76, 77], TRR [78]	Randomizes base addresses of memory regions
Randomization of system call mappings [79]	System call mappings are randomized in linking time or before loading using binary rewriting
Code relocation [80]	Randomizes the order of variables and routines, and uses random stack frame padding
Code randomization [81]	Scrambles each byte of code
PointGuard [17]	Encrypts code pointer values

Table 3.6 Program monitoring techniques

Technique	Description
Monitoring systems calls behavior [82], [83], [84], [85], [86]	An attack is detected when the monitored system call sequence deviates from the expected one
Monitoring performance register values [87]	An attack is detected when the monitored program deviates from its performance signature
Janus [88]	Untrusted applications are executed within a process-tracing framework, which allows or denies system call execution
Reference monitor [89]	Critical system calls are instrumented with access control tests
Program shepherding [90]	Security policies are enforced by monitoring control flow transfers

Bcc, a source-to-source translator for inserting boundary checks was proposed as early as 1983, by Kendall [43]. The source code is transformed so that a checking function is called on each pointer dereference and array access. These function calls are to a separate run-time package. Checking functions verify whether an array access is within the array bounds, a null pointer is dereferenced or pointer access is not properly aligned; pointer arithmetic operations are checked for overflows. Pointers are converted to pointer structures: one such structure contains lower and upper bounds of the object that pointer is pointing to. Bcc also adds function wrappers to vulnerable functions. The reported slowdown is about 30 times.

The run-time checking compiler (RTCC) implements bcc as a part of the compiler front-end, in order to reduce its execution overhead [44]. It also gets rid of some bcc checks as too restrictive or unlikely errors, e.g., pointer arithmetic overflow. Since the size of a pointer structure is three times larger than the size of a "normal" pointer, a "fat" pointer must be reduced to its normal size before being passed to a system call. RTCC solves this problem by encapsulating all system calls so that the boundary information is added or removed as needed. Encapsulation wrapper also verifies that character string arguments are terminated with a null character within bounds. C libraries are recompiled with RTCC, so there is no need for encapsulation of library calls. RTCC-compiled code runs about 10 times slower then original code. A similar project is called Bounded Pointers [91].

Austin et al. propose a source-to-source translator technique called Safe-C that detects not only the spatial memory errors such as accesses outside an object's bounds, but also the temporal errors, such as

accesses outside an object's lifetime [45]. To achieve this level of detection, Safe-C extends pointer representation even more than bcc/RTCC: a safe pointer structure consists of pointer value, base address, size, storage class, and capability. Storage class can be Heap, Global, or Local; it is used to detect errors in pointer deallocation. Capability is used to detect temporal errors. When a safe pointer is allocated, it is assigned a unique capability value, which is stored into an associative table and deleted from the table after pointer deallocation. Global objects and invalid pointers have special capability values. Calls to *malloc()* and *free()* are performed through function wrappers, which set/destroy capability values. Some checks can be avoided either by compile-time or run-time optimization. Even with optimization, Safe-C can incur a significant performance overhead (up to 6 times for considered benchmarks), so it is still not suitable for release software.

With the guarding technique, Patil and Fischer try to reduce the performance overhead of bound checking by decoupling run-time checking from original computation [46]. This approach creates objects called guards with similar properties as safe pointers in Safe-C [45]. Source-to-source translation adds guard arguments to functions with pointer arguments. The authors argue that programs with run-time checks are mostly used to find errors, and not to perform actual computations. Hence, they propose to reduce the code by deleting computations not relevant to guarding and to run such program before or after the original program. If a program is running on a multiprocessor system, checking is performed by a shadow process executing on an idle processor, thus further reducing the overhead. In this case the main process is slowed down up to 10%, due to interprocess communication.

Jones and Kelly propose a run-time bounds checking technique that is backward compatible, i.e., instrumented programs can be linked with uninstrumented libraries [47]. This is achieved by not changing the representation of pointers. Information needed for bounds checking is not kept as a pointer extension, but in a separate objects table. Object list is stored as a splay tree, which is a binary tree where frequently used nodes migrate towards the top. The authors report 5-6 times slowdown for most considered programs.

Loginov et al. propose a checking technique based on type information [48]. The type of each object can be unallocated, uninitialized, integral, real, or pointer; each element of structures and arrays has its own type tag. Type information is stored in a "mirror" of memory used by the program, so that each

byte of used memory has a corresponding 4 bits in the "mirror," describing object's type and size. The goal of this technique is to be used in debugging, since the reported slowdown can be more than 100 times.

Lhee and Chapin propose a technique that detects only buffer overflows, so it has lower performance overhead than previously described techniques [49]. Automatic and static buffers are described by an additional data structure generated by a compiler extension, and information about dynamically allocated buffers is kept in a table. Range checking is performed by functions in a shared library. This approach cannot detect the overflow of buffers allocated with *alloca()* and variable-length automatic arrays. A similar recent solution that works with binary files is proposed by Avijit et al. [50]. The authors propose a buffer overflow defense based on the use of two tools, TIED (Type Information Extractor and Depositor) and LibsafePlus. TIED extracts buffer information from a binary file compiled with –g option and writes in a new ELF section; this information is used by wrapper functions provided in LibsafePlus. Maximal reported execution slowdown is 2.4.

Oiwa et al. propose the Fail-Safe ANSI-C compiler, which fully supports ANSI C [51]. This approach is also based on extended pointer representation. A "fat" pointer is described by the base address of a memory region, offset in that region, and a cast flag. If a pointer has its cast flag set, it may refer to a value of different type than the pointer's static type. In the proposed implementation both pointers and integers occupy two machine words, one word for base and cast flag bit and another for offset. A value of an integer is stored in the offset field. This approach enables casting from a pointer to an integer and back to a pointer. The reported slowdown is up to 8 times.

Ruwase and Lam propose the C Range Error Detector (CRED), which detect buffer overflows with lower overhead than previous techniques [52]. CRED is implemented on the top of the technique presented by Jones and Kelly [47], with several improvements. It allows program manipulations of out-of-bounds addresses that do not result in buffer overflows, by creating an out-of-bound object (OOB) for every out-of-bound address value in a special OOB hash table. The performance overhead is reduced by verifying only user-supplied string data. The resulting approach detects all buffer overflows in tests described by Wilander and Kamkar [92], with maximum overhead of 130%.

Xu et al. propose a more efficient technique for detection of both temporal and spatial memory errors [53]. This approach does not handle customized memory management functions, cast of integers to

pointers, and cast of pointers to structures to pointers of structures of unrelated type. Pointer-related information (metadata) is kept separated from the pointer, unlike various fat pointer techniques. Metadata is similar to information kept in Safe-C [45]. Average performance slowdown with various optimizations is 2.21 times, with maximum slowdown 3.37 times. Optimizations include splitting metadata into header and info structures, eliminating unnecessary operations on the stack capability store, and converting metadata structures to individual variables.

Most defense techniques cause programs to abort execution when a buffer overflow is detected. Rinard et al. propose an approach called boundless memory blocks, which prevents harmful effects of buffer overflows, but allows programs to continue execution [54]. The values of out-of-bounds writes are stored in a hash table, so they can be read by out-of-bounds reads. In order to limit the amount of memory occupied by the out-of-bound writes, the hash table is implemented as a fixed size LRU cache. The checking scheme is based on techniques proposed by Jones and Kelly [47] and Ruwase and Lam [52]. Reported slowdown ranges from negligible for Apache HHTP server processing requests to 8.9 times for composing mail in Pine.

Yong and Horwitz propose a technique that keeps track of all locations that may be pointed to by an unsafe pointer in a memory "mirror" [55]. Each memory byte has one bit mirror tag indicating whether it belongs to appropriate or inappropriate locations. Unsafe pointers and locations they can legitimately point to are determined by static analysis. The location tag is set to appropriate when that location is allocated, and reset after deallocation. Write operations via unsafe pointers and *free()* are instrumented to verify the appropriate tag. Maximum reported slowdown is 8.02.

High overhead of most bounds-checking techniques limits their use in release code versions, but they can be successfully used for testing. Several techniques are designed particularly for testing purposes. Widely used commercial testing tool Purify may detect security vulnerabilities related to memory access errors, such as heap-based buffer overflows [56]. Haugh and Bishop propose a testing tool called STOBO (Systematic Testing of Buffer Overflows) which detects potential buffer overflows during tests with regular data [57]. STOBO generates one type of warnings when both the source and destination are statically allocated, and another type when the destination is dynamically allocated; it reports an error when source is dynamically allocated, and destination statically allocated. Larson and Austin propose a tool for detection

of input-related security faults, which also does not require “unsafe” test data [58]. All external input variables and derived variables are shadowed with a state variable: e.g., an integer is shadowed by a variable that stores the lower and upper variable bounds, and a string shadow variable encompasses maximum possible size of the string and null character information. Bounds are adjusted by control decisions (e.g., loops) and arithmetic operations. The tool generates an error report if any of values within the bounds causes can jeopardize security. Ghosh et al. propose to apply software fault injection (SFI) to discover potential security flaws [59].

Some dynamic techniques focus on only one type of attack targets: return addresses on the stack [60-65], format strings [13], or dynamically allocated memory [66-68]. Cowan et al. propose a compiler extension named StackGuard which detects or prevents changes of the return address on the stack [60]. With StackGuard detection, the function prologue places a dummy value, the so-called canary, between the return address and the rest of the stack. The canary is verified in the function epilogue before return execution. A buffer overflow attack that overwrites the return address must also overwrite the canary, so an attack is detected if the value of the canary has changed. The canary value may be randomized to prevent attack strings to overwrite it with the original value; however, even randomization does not prevent a write buffer overflow attack following a read attack. The overhead of canary mechanism is 125% for the worst-case function call, so it is very low for complete applications. StackGuard prevention of return address change is based on the debugging tool MemGuard, which protects values by marking the corresponding virtual pages as read-only and then emulating writes to non-protected values on those pages. Even with an optimization that uses Pentium debug registers to protect only last four return addresses, this approach has a significant slowdown.

StackShield also protects from stack smashing [61]. It applies two methods, Global Ret Stack and the Ret Range Check. In Global Ret Stack, return addresses are copied to a dedicated array in function prologue and restored from that array in function epilogue. The number of protected nested function calls is limited by array size. With Ret Range Check, a return address is copied to a global variable at the beginning of data segment. Newer versions of StackShield can also detect overwriting of function pointers. StackShield modifies assembly files, although it may be part of compiler chain.

Chiueh and Hsu propose the Return Address Defender compiler patch (RAD) [62]. The RAD technique is similar to the StackShield: return addresses are copied to the Return Address Repository (RAR) in the data segment. To protect the RAR from being overwritten by attackers, the authors propose two RAD implementations, MineZone RAD and Read-Only RAD. With MineZone RAD, the RAR area is in the middle of a global array, with the array beginning and end set as read-only areas by *mprotect*() system call. A buffer spilling into RAR will cause a trap, but MineZone will not prevent attacks writing directly into the RAR. With Read-Only RAD, the whole RAR is read-only except when return addresses are written into it. Read-Only RAD completely protects the RAR, but at the price of increased overhead for set/remove of read-only protection. MineZone RAD increases the execution time of two considered benchmarks 1.02 and 1.3 times, and Read-Only RAD 18 and 43 times. RAD handles the *setjmp()/longjmp()* issue in the following way: if the address on the top of the RAR does not match the return address, addresses are popped from the RAR until the correct address is found or the RAR bottom is reached. Prasad and Chiueh propose a way to implement RAD as a binary rewriting technique [63].

Etoh and Yoda propose a stack-smashing defense compiler extension called the Stack Smashing Protector (SSP) [64]. SSP places a pseudo-random guard value on the stack to protect a return address and the corresponding frame pointer, similar to the canary in StackGuard. In addition, SSP reorders local variables so that buffers are placed after pointers. It also protects pointers in function arguments by copying them to an area preceding local buffer variables. To reduce overhead, SSP instruments only functions that have string buffers as arguments or local variables. For three considered applications, SSP overhead ranges from 0 to 4%, while StackGuard overhead for same applications is 0-8%. SSP cannot prevent certain types of buffer overflows: for example, a buffer may overflow into a pointer variable if both are part of the same structure, since the order of structure elements cannot be changed.

One limitation of StackGuard and similar techniques is that they require source or assembly code. Baratloo et al. propose a transparent run-time defense against smashing attacks that works with precompiled binaries [65]. The transparent defense is based on two dynamically loadable libraries, libsafe and libverify. Libsafe implements “safe” versions of functions which can cause buffer overflows, such as *strcpy()*. The size of buffers in those functions is limited by the size of the corresponding stack frame, so they can never overflow beyond the frame pointer. Libverify protects all return address as the StackGuard

does, but canary code is completely contained within the library. Both libsafe and libverify rely on preload feature of ELF libraries to load with processes that need protection. For each function in a protected process, the *_init()* function of libverify copies function code to heap, and replaces first instruction in original function and last instruction in the copy with jumps to wrapper entry/exit routines. The entry wrapper writes a canary value on the canary stacks and jumps to function copy, and the exit wrapper verifies the canary. The canary value is the return address itself, as in StackShield. The canary stack is protected by read-only regions like MineZone RAD. Libverify has slightly larger performance overhead than StackShield.

Wilander tested StackGuard, StackShield, ProPolice (an old name for SSP), Libsafe, and Libverify with 20 buffer overflow benchmarks [92]. Although all these techniques effectively protected return addresses, they were not able to detect/prevent other buffer overflow attacks, such as buffer overflow on the heap. The best technique, ProPolice, missed 9 of 20 attacks.

Fetzer and Xiao propose transparent defense against heap smashing attacks by using a dynamically loadable C function wrapper called HEALERS [66]. HEALERS wrapper intercepts C functions that could be used to write to the heap and performs boundary checking of function arguments. Wrapper for *malloc()* records position and size of allocated memory in an internal table, and wrapper for *free()* deletes the corresponding table entry. The overhead of HEALERS is up to 10% for considered applications. The authors later extended the HEALERS toolkit to automatically discover problems in C libraries using automated fault injection experiments and to support flexible wrapper generation [67].

Heap-based buffer overflows may target memory management information, which is stored at the beginning of each memory chunk. Robertson et al. propose to protect this information by storing a canary value when a chunk is allocated, and verifying it when the chink is freed [68]. The canary is the checksum of the chunk header seeded with a global random value, initialized during process startup. The proposed approach is implemented as a library patch for glibc library. Memory allocation functions in glibc are implemented using *dlmalloc*, so the authors needed to modify only this routine. For the worst-case microbenchmark, the execution slowdown is 28%. Performance impact for real applications is negligible.

Another tool targeting only one class of attacks is FormatGuard, proposed by Cowan et al. [13]. FormatGuard is a library patch for protection from *printf()* format string attacks. It counts the number of

actual arguments presented to printf and compares is with the number of arguments specified in the format string. If the format string specifies more arguments than *printf()* receives, FormatGuard aborts the program. Although this technique was able to detect most format string exploits known at the time, it cannot defend against attacks in which the number of actual arguments is not less than specified, and does not detect calls to *printf()* via pointers.

Toth and Kruegel propose a completely different approach for run-time detection of code injection attacks, for Internet services applications [93]. Any injected code must be a part of a client request, so the authors propose abstract execution of payload in client requests before requests are serviced. Abstract execution of a byte sequence determines its maximum executable length (MEL). If MEL for a request is beyond a specified threshold, that request is dropped by the system.

Several researchers proposed “safe dialects” of C language. “Safe dialects” restrict the use of C language constructs that can be sources of security vulnerabilities. In addition, the corresponding compilers use static analysis to prove that the program is safe or to abort compilation. “Safe dialects” may provide C extensions for programmer annotations and/or insert run-time checks. They also may replace C dynamic memory allocation/deallocation mechanism with automated garbage collection or region-based memory management.

DeLine and Fähndrich designed Vault programming language [69, 70]. Vault allows programmers to describe domain-specific resource management protocols, which are then enforced by the compiler. Hence, memory-related errors such as dangling pointers can be discovered in compile time. Vault extends the type of a value with a type guard predicate, which specify conditions when that value can be used. These conditions relate to so-called keys, compile-time tokens representing run-time resources. In the simplest case, a type guard is true when the corresponding key is part of the global state. Function types have preconditions and postconditions. Vault has primitives for region-based memory management, where objects are individually allocated from a region (a named subset of the heap), but the region is deallocated as a whole.

A safe C dialect named Cyclone by Jim et al. is designed to prevent safety violations [71, 72]. Cyclone compiler uses static analysis to insert run-time checks. If the compiler cannot guarantee the program safety even with the checks, it does not perform compilation. Cyclone supports programmer

annotations such as hints to static analysis or enforced bounds checking. Cyclone restricts C features that might violate safety, but adds additional features that provide the same functionality in a safe way. For example, pointer arithmetic is permitted only on "fat" pointer structures. Instead of using *free()*, programmers may reclaim heap space either by using garbage collector, or region-base memory management. The worst reported slowdown for Cyclone program with garbage collection and bound checking is 2.85.

CCured technique, proposed by Necula et al., also inserts run-time checks based on static analysis [73]. CCured introduces additional pointer types: a pointer is safe, sequence, or dynamic. Pointer type is determined by programmer annotations or by static analysis. Different pointer types require different run-time checks. For example, a pointer is marked as sequence if it is used for array access. Sequence pointers require null pointer checks and bounds check when dereferenced or cast to safe pointers. Like Cyclone, CCured ignores *free()*, but it implements only automatic garbage collection. The worst reported slowdown is 2.44.

Kowshik at el. propose another "safe" C dialect named Control-C, for real-time control systems and other embedded programs [74, 75]. The main goal of Control-C is to guarantee safety by static-analysis only, without adding run-time checks and without programmer annotations. This goal is achieved by restricting dynamic memory allocation and array operations, and providing type safety. Memory allocation is region-based, restricted to a single dynamic region at a time. Type safety requires strong typing of all variables, assignments, expressions, and functions. It also forbids casts between pointer and other types, pointer arithmetic, and the use of uninitialized variables. Control-C implementation assumes a low-level typed virtual instruction set and system support to trap accesses to a range of reserved addresses.

Vulnerability exploits can be made more difficult by various obfuscating techniques. For example, the PaX kernel patch includes the feature named Address Space Layout Randomization (ASLR) [76, 77]. At task creation time, ASLR randomizes base addresses of memory regions such as code/data segments, heap, libraries, and stack. Chew and Song [79] propose three randomizing methods: randomizing of system call mappings, changing library entry points, and randomizing stack placement. First two methods are implemented by binary rewriting in linking time or before loading. Xu et al. propose Transparent Runtime Randomization (TRR) [78]. TRR randomly relocates stack, heap, shared libraries,

and the global offset table (GOT), in load-time. Bhatkar et al. expand the idea of address obfuscation with permutation of the order of variables/routines and generation of random padding between the objects [80]. The implemented prototype includes base address randomization of stack, heap, DLL, text, and data segments. It also applies random stack frame padding. Performance overhead is negligible if code relocation is performed at link-time, and up to 21% if performed dynamically at load-time. Another option is to randomize the code: Barrantes et al. propose a randomized instruction set emulator (RISE) which scrambles each byte of the program code in load time using pseudorandom numbers [81]. PointGuard by Cowan et al. keeps address pointer values encrypted in memory and decrypts them only before loading into CPU registers [17]. In the implemented prototype, pointer values are encrypted by XOR with a key.

Most dynamic software-based techniques require the access to source code, since the compiled and linked code version does not contain enough information, unless it is compiled with a debug option. DuVarney et al. propose SELF, a security extension for ELF binaries [94]. SELF extends the ELF format with an extra section with information about address, size, and alignment requirements of each code and static data item. The goal of this approach is to provide information necessary for binary transformations such as address obfuscation, and yet to reduce the number of details present in debugging sections that may be used for reverse engineering.

Several researchers suggest intrusion detection by monitoring the system calls of a program [82], [83], [84], [85], [86]. If the system call sequence for a particular program deviates from a normal behavior, an intrusion is suggested. The normal program behavior is obtained either by profiling, or by encoding the specification of expected behavior using a special high-level specification language. If profiling is used, false positives may be generated when a rarely used region of the code is executed. A specification-based approach, on the other hand, is as error prone as the coding process itself. Finally, although a malicious code is very likely to encompass a system call, an attack may be potentially devised with the same call sequence as the vulnerable program, or may inflict some damage even without system calls. Another profiling approach by Oppenheimer and Martonosi suggests using the values of performance monitoring registers to verify whether the program deviates from its performance signature [87]. For example, execution of injected code will change the memory reference profile of the attacked program.

Goldberg et al. propose Janus, a secure user-level environment that restricts system calls from untrusted applications [88]. Janus utilizes process-tracing facilities available in some operating systems. Untrusted applications run as child processes, which are stopped at each system call that might impede security. Policy modules specify which system calls are allowed to continue execution, and which get an abort signal. System calls can have a fixed security policy (always allow/deny), or the policy can be specified in a configuration file and can be dependant on system call arguments. In run-time, configurable policies are stored in a dispatch table structure. If *write()* and *read()* system calls are always allowed, performance overhead of Janus is negligible. However, Janus cannot be applied to applications with system calls that may be exploited by attackers.

Similar approach is proposed by Bernaschi et al. [89]. Instead of user-level tracing, the authors propose a kernel extension, based on the concept of operating system reference monitor. OS reference monitor decides which system calls can be executed, according to predefined access rules. This concept is implemented by instrumenting system calls that might be misused in a buffer overflow attack with access control tests. The access rules are stored in the Access Control Database (ACD). For each instrumented system call, this database specifies which processes are allowed to execute instrumented system calls and with which arguments. The authors also implemented special system calls for ACD access. For considered applications this approach has a negligible overhead.

Kiriansky et al. propose an approach named program shepherding, where execution of malicious code is prevented by monitoring all branch instructions [90]. Instead of instrumenting the code, the authors propose to use the runtime binary interpreter for runtime introspection and optimization (RIO). Program shepherding encompasses three techniques: restricted code origins, restricted control transfers, and uncircumventable sandboxing. All code pages are write-protected, so a basic block can be executed only if it is copied to RIO code cache from a write-protected page. If code and data share a page, program shepherding makes a write-protected copy of the page, and basic blocks are read from the protected copy. Restricted control transfers means that an arbitrary policy can be applied to each type of branch instructions, e.g., a return instruction must jump after the corresponding call, or library code can be executed only through declared entry points. Sandboxing is used for restrictions not covered by the first two techniques, e.g., to detect *execv()* system calls . Sandboxing is also used to prevent an application from

changing RIO's data. For the considered set of security policies and SPEC CPU2000 benchmarks, the slowdown is up to 1.7 times under Windows and up to 7.6 times under Linux operating system.

Code injection attacks assume that the memory segment with injected code is executable. Therefore, one defense technique is to make some memory portions permanently or temporarily non-executable. PaX offers non-executable memory pages [76, 77]. However, the IA32 architecture does not support non-executable pages, so PaX uses two techniques circumvent this issue, based on the IA32 paging or segmentation logic. The first technique is based on the split data and instruction translation look-aside buffer (TLB), which is implemented in all Intel CPUs since the Pentium. The pages whose execution should be prevented are marked as requiring supervisor access, so application accesses to those pages result in a data TLB page fault. The page fault handler then decides whether an access is an instruction fetch or a regular data access. For data accesses, the user/supervisor bit in the corresponding page table entry is temporarily cleared. Another technique is to divide the virtual address space in two halves. Application code and data are mapped to one half, and instructions are mirrored in the other; instructions can be executed only from the instruction space. The paging-based PaX approach can have significant performance overhead, and the segment-based approach reduces the available virtual memory space. PaX has support for stack-executable code and can be turned off for each particular application. PaX is incorporated into several operating systems with security features, such as Adamantix and Hardened Gentoo [95]. Non-executable heap and stack pages are also supported in RedHat [77].

3.3 Defense Techniques With Hardware Support

Some of the performance overhead of purely software-based dynamic techniques may be reduced with hardware support. Table 3.7 lists techniques with hardware support that can be used for full or partial defense from code injection.

A large portion of existing attacks targets return addresses on the stack, so several hardware-supported techniques protect only from stack smashing. The first such technique was proposed by Xu et al. [96]. The main idea is to use separate stacks for data and control information, so any overflow of a buffer stored on the stack can overwrite only other local data, and not any return addresses.

Table 3.7 Techniques with hardware support

Technique	Description
Split control and data stack [96]	Protects against attacks on function return addresses by keeping separate stacks for data and addresses
Secure Return Address Stack [96], [97], SmashGuard [98], Reliable Return Address Stack [99]	Protects against attacks on function return addresses by keeping a copy on the hardware return address stack
DISE [100]	Protects against attacks on return addresses by keeping a copy on the secure return address stack on the heap
SCache [101]	Reduces the possibility of success of attacks on return addresses by replicating cache lines where return addresses are stored
HSAP [102]	Protects against attacks on function return addresses by preventing writes on stack after frame pointer; makes difficult attacks on function pointers by encoding jump addresses
Hardware and binary modification support for code pointer protection [103]	Protects code pointers against buffer overflow by encoding jump addresses
HAT [104]	Protects from buffer overflow by keeping track of pointer size, allocation, deallocation, and liveness
SPEF [105]	Protects code integrity by transforming code blocks according to the encrypted transformation-invariant block value
Randomized instruction set [106]	Protects code integrity by randomizing underlying system's instructions
Data tagging [107]	Prevents control flow transfer based on data tagged as spurious
Minos [108]	Prevents control flow transfer base on low integrity data
Instruction block signatures [109], [110], [111]	Protects code integrity by verifying the signature of executing instruction blocks

This approach can be implemented as software-only, by modifying compiler to write/read return addresses on the control stack in prologue/epilogue of each function, allocate control stack space, and manage the control stack pointer. For considered benchmarks the performance overhead is from 0.01 to 23.77%. This overhead is due to extra memory accesses for saving and restoring return addresses: e.g., saving of a return address requires a read from the "regular" stack, a write to the control stack, and two memory operations for control stack pointer update. If this approach is implemented with hardware

support, the performance overhead can be completely avoided. The required processor modifications are relatively simple: changed implementations of call and return instructions and an additional register for the control stack pointer. The split stack technique prevents return addresses from being overwritten with very small additional hardware complexity and no performance overhead. However, it does not protect from other attacks, such as heap smashing.

Most modern processors already have a hardware resource that can be used for protection of return addresses: the Return Address Stack (RAS), used to predict a target address for return instructions in the pipeline fetch stage. Xu et al. propose three RAS extensions under the common name Secure RAS (SRAS) [96]. The first such extension keeps SRAS lookup in the fetch stage, so an exception to the operating system is raised both when a return address has been overwritten by a stack smashing attack, and when it was just mispredicted due to RAS speculative update or RAS overflow. The operating system decides why the addresses on the "regular" stack and the SRAS do not match, by keeping the trace of stack accesses and valid return points. This technique has a very large performance overhead: with 64-entry RAS, some applications are slowed down for more than 100%. Another option is to move SRAS lookup to the pipeline commit stage. In this case there are no SRAS mispredictions due to speculation, so the maximum observed performance overhead is 4%. Finally, the third option also eliminates mispredictions due to overflow, by keeping a part of the SRAS in a memory data structure. Just like the split stack technique, the SRAS is relatively simple to implement and its third option has a very small performance overhead. On the other hand, it protects only return addresses.

Similar efforts expand the idea of the SRAS [97], [98], [99]. The advantage of all these techniques is small performance and complexity overhead, and minimal or no code changes. Independently of Xu et al., Lee et al. propose a structure called also Secure Return Address Stack (SRAS) [97]. This SRAS is not en extension of RAS used for return address prediction, but rather a completely separate structure. Hence, a mismatch between an address stored on the SRAS and the "regular" stack can be due only to a successful stack smashing attack. The hardware support for this technique includes the SRAS and modified implementations of call and return instructions. An OS exception is raised in the case of a SRAS overflow/underflow. An exception handler writes/reads a half of the SRAS entries to the memory space accessible only to the OS kernel. The kernel maintains separate SRAS overflow areas for

different processes. While this approach successfully protects return address from buffer overflow, it prevents the so-called non-LIFO control flow, where a return address does not have to be located on the top of the stack. The use of *setjmp()* and *longjmp()* functions is one example of non-LIFO control flow. This problem can be solved with additional instructions that will push and pop addresses directly to the SRAS, or even turn off the SRAS protection. These instructions can be inserted in the code by compiler or a disassembling program, or even in run-time.

Ozdoganoglu et al. propose a solution called the SmashGuard [98]. The core of this technique is also a separate hardware stack for return addresses; it differs from previous such techniques in solving the issue of *setjmp()/longjmp()*. The authors propose that the *longjmp()* function should use an indirect jump instead of *return* instruction, so its return address is not read from the stack. At each call, both return address and "regular" stack pointer are stored on the secure stack. At return, the secure stack is searched for the corresponding pair. This technique leaves unprotected the return address of the *longjmp()*, but since both *setjmp()* and *longjmp()* are library functions, they could be protected using software methods. To avoid problems with speculative execution, the secure stack is accessed in the commit pipeline stage. The values of return address register and link register are saved in an additional processor resource, called RAT (return address table). Another option is to completely or partially stall the processor and read the required values from the register file. The implementation with complete stalling degrades performance up to 7% for considered benchmarks executing by a 4-way superscalar processor; the worst case with partial stall is about 2% degradation.

Another recent technique based on a secure hardware stack is called Reliable Return Address Stack (RRAS) [99]. The authors pair the RRAS with a structure called Address Pair Table (APT), which stores the entry and exit point of all active functions. This solution is able to handle non-LIFO control flow. Entry repetition in the RRAS for recursive functions is avoided using 3-bit tags.

The secure stack does not have to be implemented in hardware: using a technique called Dynamic Instruction Stream Editing (DISE), the 'shadow" stack is kept in a protected area on the heap [100]. DISE is a one-to-many instruction macro expander with programmable rewriting rules: to protect return addresses from the attack, call and return instructions are dynamically rewritten in the runtime to write/verify data from the shadow stack. When a call instruction is executed, both the current stack pointer and the return

address are saved on the shadow stack. When the return address on the top of the shadow stack differs from the top of the "regular" stack, the shadow stack is searched for the matching (stack pointer, return address) pair. The shadow stack is protected either by XORing its entries with a random value selected at the application start (DISE/XOR), or by testing all stores and allowing only DISE-expanded code to store data to shadow stack memory segment (DISE/MFI). Since DISE is implemented in hardware, it does not require code changes and does not introduce additional instruction cache misses. However, expanded instructions can significantly degrade performance. With a 4-way superscalar processor and considered benchmarks, the worst-case performance overhead is more than 15% for DISE/XOR and more than 30% for DISE/MFI.

The redundancy of return addresses can be achieved not only by duplicating stack entries, but by replicating cache lines with return addresses, as Inoue proposes in the SCache technique [101]. When a return address store is executed, this technique writes it to one or more cache line replicas, depending on implementation. Replicas are stored in the same cache set as the original cache line (the master line). Cache lines have a one-bit replica flag. A buffer overflow attack can overwrite only the master line, while values in the replicas are preserved. When a return-address load is executed, one of the replicas is randomly selected and the value of the replicated address is compared to the corresponding master line value. If the two values do not match, the SCache detects an attack and terminates the executing program. If there are no replicas due to the cache replacement policy, the SCache generates an indicator of potentially unsafe address. This technique cannot protect all return addresses, but the percentage of protected addresses is relatively high. With SPEC CPU2000 integer benchmarks, a 16KB L1 SCache with 4 ways and 32B line protects more than 98% return addresses with up to three replicas, and more than 94% with one replica. The worst-case performance overhead is 1.1%.

The main drawback of techniques discussed so far is that they provide protection from only one type of attack. A successful buffer overflow attack can overwrite not only return addresses on the stack, but any function pointer. To protect both return addresses and function pointers, Shao et al. propose the Hardware/Software Address Protection (HSAP) technique [102]. The HSAP consists of two complementary techniques: protection against stack smashing and protection of function pointers. The stack smashing is prevented by denying any writes to the memory if the write address is equal to or larger

than the value of the current stack frame pointer. The address check is performed in an additional pipeline stage before the write stage, so this approach has very low performance overhead. The authors also propose to make it difficult for potential attackers to change the values of function pointers to point to addresses of their choice. During code compilation, each function pointer assignment instruction is preceded by an XOR instruction, which XORs the function address with a key from a special register. This key is randomly generated for each process. An additional instruction, secure jump (*sjmp*), is used when a function is called using the value of a function pointer. The authors do not provide details about performance overhead of function pointer protection. One disadvantage of this approach is that it requires the change of the code. More important issue is the level of protection of this technique. Some processes run for a very long time, so an attacker might be able to discover the value of the key used for XOR: a read buffer overflow attack can be used to read the encrypted address value, and then a simple XOR with the "real" address will reveal the key.

Tuck et al. propose to protect code pointers from both read and write buffer overflow attacks by encrypting them [103], similarly to the software technique PointGuard [17]. By code pointers the authors refer both to return addresses on the stack and function pointers. Code pointers are encrypted and decrypted using special instructions: encrypt-store and decrypt-load. The authors propose three levels of encryption: XOR with a secret key, XOR with a value from random permutation table, or a Feistel network. While this technique does not prevent so-called replay attacks, the third encryption level offers very good protection against read attacks. However, the Feistel encryption/decryption latency is 40 or 80 processor cycles, depending on the implementation. This latency can be partially hidden for call and return instructions. Decrypted values of function pointers are cached in the L1 cache memory and protected from overwriting by a special cache bit. The performance overhead is up to 30%.

Other software techniques such as dynamic validity checking of augmented pointers can also be partially implemented in hardware. An approach presented by Keen et al. [104] combines static code analysis and instrumentation with dynamic run-time checking using a hardware structure, the Hardware Accelerated Table (HAT). Pointer validity checking has two components, spatial and temporal. Spatial component verifies whether the value of the pointer is within the bounds of the corresponding object, and temporal component verifies whether the object is alive. Temporal verifications use a hash table, and hash

table operations are the largest contributor to the overhead of the purely software implementation. The authors propose to implement the hash table and the relevant hash table operations in hardware (HAT). Hash table find, insert, and remove can be implemented as new instructions (GenHAT), or performed by a specialized checking engine (SpecHAT). With a small set of benchmarks, the SpecHAT reduces the overhead of the software technique for up to 3 times. The maximal observed overhead is 6% for SpecHAT and 12% for GenHat. The main disadvantage of this method is that it relies on correct instrumentation of source code.

Kirovski et al. propose the Secure Program Execution Framework for intrusion prevention (SPEF) [105]. The underlying idea is that a program executable can have different representations that produce the correct program behavior. Possible code transformations include instruction scheduling, basic block reordering, branch-type selection, and register permutation. During installation, a transformation-invariant (TI) hash value is calculated for each instruction block and is encrypted using a secret processor key. The encrypted hash value defines the transformation of the instruction block. During execution, the verifier component calculates the TI hash for every instruction block that is fetched after an instruction cache miss. It then encrypts the hashed value, and verifies whether the obtained transformation is equal to the actual code. If there is no match, an abort signal is sent to the processor. This solution successfully prevents execution of injected code, but at the cost of relatively significant performance overhead, up to 25% for MediaBench applications running in embedded systems. The overhead of encryption can be reduced with a TI cache; maximum overhead of the SPEF technique with a TI cache with twice as many entries as the I-cache is about 17%. Another disadvantage of the SPEF is that it must be customized for different platforms and instruction sets.

One possible defense against code injection is to encrypt complete program code. A technique proposed by Kc et al. “randomizes” the code of each user-level process [106]. Randomization is performed by XOR of a memory block with a key or by bit-transposition, also based on a key. The key is stored in an encrypted form in the program file header. This approach does not work with dynamically loaded libraries. Instruction blocks are decrypted in the fetch-decode pipeline stage, so this technique incurs a significant overhead.

Suh at al. propose to tag all data coming from "the outside world" (e.g., I/O channels) as spurious and to prevent execution of any control transfer instruction if the target address depends on spurious data [107]. This approach may generate some false positives, since the target address may be input-dependant, for example in switch constructs. Generally, input data can propagate to a target address through a series of calculations, so this technique requires a relatively complex data dependency analysis. A similar approach, Minos, augments every memory word and registers with an integrity low/high bit [108]. The integrity bit is set by the kernel when that memory word is written and determines the trust the kernel has in the data. The trust is propagated using a low-water-mark integrity policy with two rules: a subject can modify an object of same or less integrity, and when a subject reads an object of low integrity, its integrity also becomes lower. The low trust data cannot be used for control transfers.

The code integrity in run-time can be successfully protected if all instruction blocks are signed with a cryptographically secure signature. In run-time the actual signature is verified against the calculated signature. The signature mismatch means that there is change in the original code. We did preliminary research on protection of basic blocks and cache blocks using signatures [109], [110]. Drinic et al. also propose to sign all cache blocks and to verify signatures in run-time on a cache miss [111]. With this approach, a 16-byte instruction block signature is obtained by encrypting the instruction block using a 128-bit Rijndael cipher, and then XOR-ing the 16-byte sub-blocks. The overhead of Rijndael decryption implemented in hardware can be hidden if the instructions in an instruction block can be reordered in such a way that critical instructions such as stores are executed after decryption delay time.

3.4 Other Related Work

The problem of detection of code injection attacks can be related to the problem of detection of control flow hardware faults using fault-tolerant techniques, and valuable lessons can also be learned from techniques for detection of software tampering.

Mahmood and McCluskey's study from 1988 surveys various techniques for concurrent error detection using watchdog processors [112]. One of the discussed techniques for control flow checking is actually based on verification of basic block signatures. However, the approach presented in this dissertation does not require a dedicated watchdog processor, and focuses rather on seamless integration on

verification mechanism into existing processor architecture. Moreover, the signatures in our mechanism are also protected from read attacks.

Wilken and Chen propose a control flow error detection mechanism with reduced number of embedded signatures [113]. Ohlsson and Rimen propose another signature placement technique that does not require the knowledge of the program control flow graph [114]. In a more recent study, Kim and Somani propose a technique for checking the integrity of instructions and their sequencing from fetch to commit point [115]. Oh et al. propose a software-based technique for control-flow checking using assigned signatures, by embedding in code both signatures and instructions for error detection [116].

Joseph and Avizienis propose a virus protection technique using an extended Program Flow Monitor, which verifies basic block signatures [117]. However, the paper does not include any implementation details or evaluation. Davida et al. discuss various possible granularity levels for blocks protected by signatures, from whole program files to individual instructions [118].

Various techniques for tamper-resistant software aim to protect software integrity from potentially hostile operating system and other programs running on the same processor, in order to preserve the originally installed code and to prevent software piracy. To support copy and tamper resistant software, Lie et al. propose an approach called XOM (eXecute Only Memory) [119, 120]. The XOM main idea is memory in certain cases can be only executed, and not read or written, so that one program cannot read instructions or data of another program. This goal is achieved by keeping programs encrypted in off-chip storage, and tagging instructions and data with a program ID in on-chip storage. Each XOM processor has a public/private key pair, with private key unknown to the user. When the user installs an application, XOM generates a symmetric key to encrypt it and appends the symmetric key encrypted with the private key. During execution, instructions are decrypted when stored in the cache memory.

Collberg and Thomborson survey software tools for software protection against reverse engineering, software piracy, and tampering [121]. Reverse engineering can be made more difficult by obfuscation, where a program is transformed to an equivalent form that is harder to understand. Software piracy may be countered with watermarking, which embeds copyright in code. Tampering may be countered with tamper-proofing code embedded in the original application.

Horne et al. propose a software technique to improve run-time tamper resistance [122]. The authors propose an implementation of self-checking, where a program checks itself for modification while it is running. Embedded testers calculate hash values of large code portions and verify them against original values. A similar approach called oblivious hashing is proposed by Chen et al. [123]. Oblivious hashing protects the executed code only.

Software does not have to be protected only from tampering by "hostile" host machine. Mobile applications can also be changed in transit by malicious attackers. Jochen et al. propose a framework named StEgo-CRYpto Tamper detection (SECRYT) [124], which enables validation of mobile applications.

The Trusted Computing Group (TCG) is an industry-standards organization, with the goal of creating specifications of various hardware-supported security features [125]. TCG specifications describe hardware support for various high-level security features, such as storing passwords and digital certificates in hardware, protecting online transactions, and providing authentication between systems and networks. Our techniques provide a low-level security, so they can be integrated into TCG models.

CHAPTER 4

PROPOSED ARCHITECTURES FOR INSTRUCTION BLOCK VERIFICATION

"Before you build a better mousetrap, make sure you have some mice out there."

Yogi Berra

In this chapter we introduce the basic mechanism common to all proposed techniques for instruction block verification. Next, we present the taxonomy of different techniques and discuss techniques' pros and cons. Finally, we describe the implementation details of each technique and discuss various related issues.

4.1 Basic Mechanism of Proposed Techniques

All proposed techniques for instruction block verification share the same basic mechanism (Figure 4.1) and require minimal or no compiler support. The basic mechanism encompasses two phases: a secure program installation and program execution. During the secure installation process, signatures are calculated for each instruction block and added to the program binary. A signature is obtained in the following way: All instructions in the instruction block pass through a Multiple Input Signature Register (MISR). A MISR is essentially an array of D flip-flops with linear feedback coefficients (Figure 4.2). A new value of the *i*-th MISR bit is calculated as an XOR function of the *i*-th bit of an incoming instruction, the (*i-1*)-th MISR bit, and possibly some other MISR bits. Linear feedback connections are determined by a secret processor key hidden in hardware. The result of the final MISR calculation is then encrypted using Advance Encryption Standard (AES) [126], also with a secret hardware key, which can be different from

the MISR key. For each new protected block, the MISR is initialized to the same value, which is also secret.

Signatures are verified in parallel with program execution using a dedicated hardware resource called the Instruction Block Signature Verification Unit (IBSVU). The IBSVU encompasses registers for buffering instructions and signatures, support for AES decryption, MISR, and control logic. Without loss of generality, let us consider a processor with only the first level of instruction (L1I) and data (L1D) caches (Figure 4.3). To simplify abbreviations, the L1I cache is denoted further as the I-cache. Since the I-cache is a read-only resource, instruction block signatures are verified only on I-cache misses. Fetched instructions pass through a MISR register with the linear coefficients that are equal to the linear coefficients used during secure installation. Concurrently with MISR calculation, the AES block decrypts the signature fetched from memory. Hence, the decryption time is partially or completely overlapped with the instruction block fetch phase. The decrypted signature is compared to the final MISR calculation: If the two values match, the instruction block is properly installed and can be trusted. If the values differ, the instruction block includes injected code or it is not properly installed, so a trap to the operating system is asserted. The operating system than aborts the process whose code integrity cannot be guaranteed and possibly audits the event.

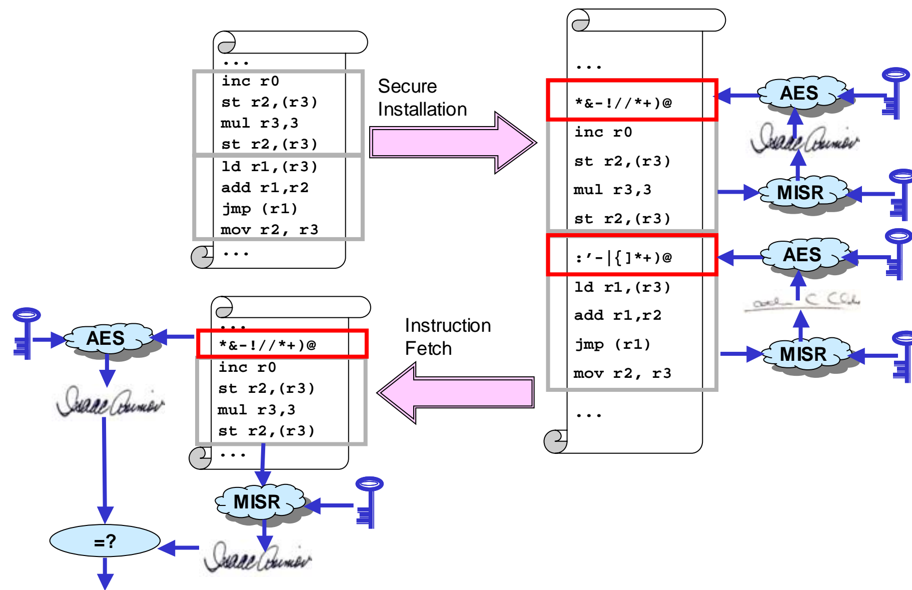

Figure 4.1 Mechanism for trusted instruction execution

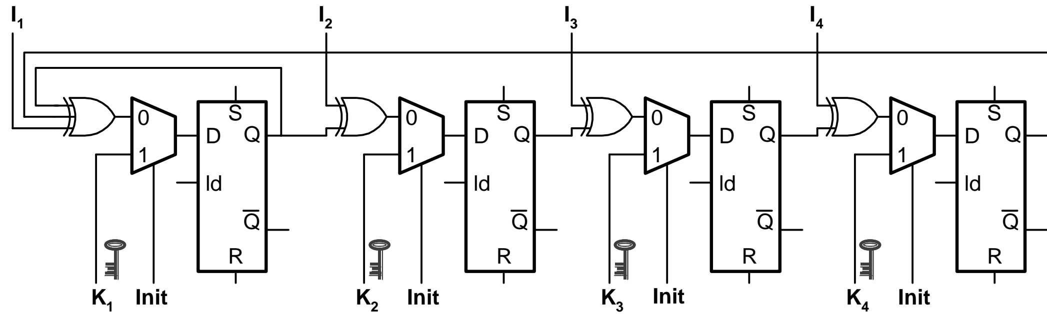

Figure 4.2 An implementation of a 4-bit MISR

A computing system might be designed to run only in the protected mode where all instruction blocks must be signed, as described above. However, some applications do not need instruction block protection. For example, some components of the operating system may not accept external inputs from untrustworthy channels and thus are not in danger from code injection attacks. Such programs may be installed without signatures and executed in the unprotected mode. The information about required execution mode is added to the program header.

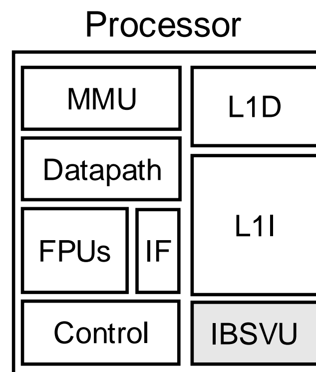

Figure 4.3 Processor components

Legend: MMU – Memory Management Unit, IF – Instruction Fetch Unit, FPUs – Floating Point Unit(s), Control – Control Unit, L1D – Level 1 Data Cache, L1I – Level1 Instruction Cache, IBSVU – Instruction Block Signature Verification Unit.

4.2 Taxonomy of Proposed Techniques

Instruction block verification techniques can be classified according to the following criteria:

- Type of protected instruction blocks,
- Signature placement,
- Signature handling after verification,
- Signature visibility to the I-cache.

The taxonomy of instruction block verification techniques is given in Figure 4.4. The name of a verification technique starts with SIG, and the rest of the name specifies the categories to which the technique belongs. For example, the SIGCED technique protects a code block with the size equal to the size of an I-cache line (C), with signatures embedded in the code (E), and disposed after verification (D).

A protected block can be of variable or fixed size. With variable-size blocks, one signature protects a logical code unit such as a basic block or an instruction stream (dynamic basic block). A *basic block* is a straight-line code sequence with no branch instructions out except at the exit and no branch instructions in except to the entry. An *instruction stream* or a *dynamic basic block* is a sequential run of instructions from the target of a taken branch to the first taken branch in sequence. With fixed-size blocks, one signature protects a physical code unit of the size equal to the size of one or more I-cache lines.

Verification techniques can be further classified depending on the placement of signatures in a binary file. A signature can be embedded in the code, i.e., placed before or after the instruction block it protects. Another option is to store all signatures in a separate table, i.e., a separate code section.

After verification, a signature can be discarded or stored in a dedicated resource called the signature cache (S-cache). The S-cache's number of entries and organization differ from the I-cache in order to keep decrypted signatures fetched from memory even when the corresponding instruction blocks are evicted from the I-cache. The S-cache may reduce the performance and energy overhead of run-time signature verification at the price of increased hardware complexity.

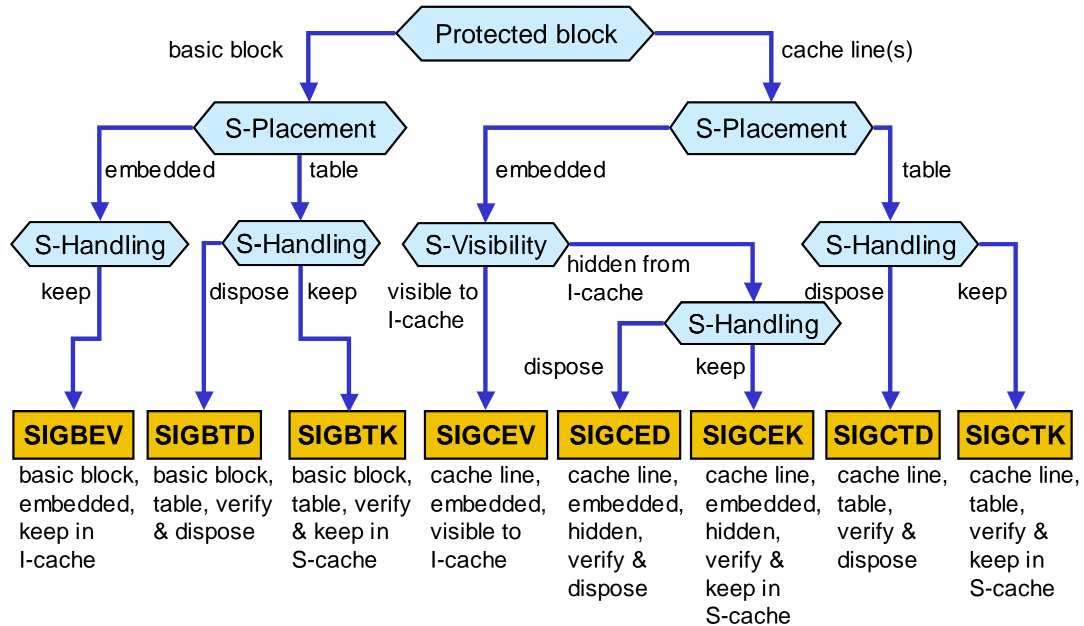

Figure 4.4 Taxonomy of proposed instruction block verification techniques

Basic block protection techniques differ slightly from the basic mechanism described above: A signature cannot be calculated in parallel with the fetch stage of the pipeline, since the end of a basic block is not known in that stage. The calculated signature is known only after the last instruction in the corresponding basic block has been decoded, so the signature verification for one basic block is done in parallel with execution of the following basic block.

The basic block protection technique with embedded signatures must keep signatures together with the instructions in the I-cache, since embedded signatures cannot be extracted from the code in the fetch stage without decoding [110]. The name of this technique in our taxonomy is SIGBEV. With the SIGBEV technique, the instruction decoder must be able to tell the difference between a signature and a regular instruction. This can be achieved by reserving one instruction bit for the signature flag, or by using a special opcode that indicates to the decoder that instruction words that immediately follow the current word represent a signature.

Basic block protection techniques with signatures stored in the separate code section work in the following way. When the instruction decoder detects the end of a basic block that caused at least one cache miss, the signature of that block must be fetched from the signature code section, decrypted, and compared

to the calculated signature. These techniques can be classified depending on whether a decrypted signature is kept in a dedicated S-cache after verification (SIGBTK) [109], or it is disposed of (SIGBTD).

Basic block protection techniques require compiler support, since disassembling generally cannot extract the basic block list from the executable code with 100% accuracy [63]. However, the required support is relatively simple: With SIGBT techniques, the program compilation process only needs to generate a list of all basic blocks in the code and to append it to the executable (Figure 4.5). With the SIGBEV technique, embedded signatures are converted to no-ops in the decode stage, so they are visible only to the dedicated signature verification unit and not to the rest of the processor core. However, instruction addresses will change due to embedded signatures, so the installation process must recalculate all target addresses. Hence, the list of basic block must also include target addresses.

Compiler support is not necessary for techniques protecting instruction blocks of a fixed size (Figure 4.5). Moreover, the signatures can be verified in parallel with the fetch stage, since the exact placement of signatures and protected blocks is known in advance. Similarly to the basic block protection techniques, embedded cache line signatures are not visible to the processor, i.e., the processor is aware only of the executable code. This invisibility is achieved with the use of a relatively simple address translation, so that the processor “sees” instruction addresses as if there were no embedded signatures. The address translation can be done before or after the instructions are stored in the I-cache; that is, the signatures can be hidden from the I-cache or visible to it (SIGCEV in our taxonomy). If signatures are hidden from the I-cache, they can be disposed of after verification (SIGCED) or kept in the S-cache (SIGCEK). As in the previously described techniques, cache line signatures placed in a separate code section can be discarded after verification (SIGCTD) or kept in the S-cache (SIGCTK).

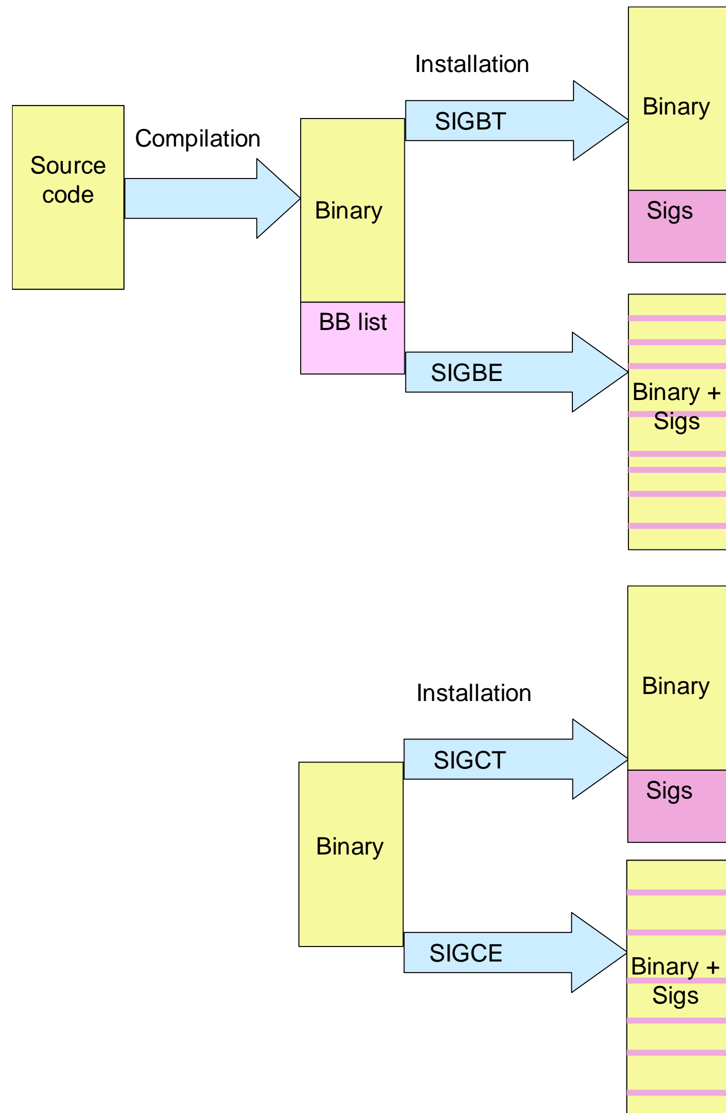

Figure 4.5 Modification of executable code

Table 4.1 illustrates the most important pros and cons of the proposed techniques. Relevant parameters include the need for compiler support; hardware complexity, i.e., the estimated area on the chip required by a particular technique; the projected performance overhead, based on delays that a technique introduces to program execution; applicability of a technique in systems without cache memory; and the requirement to change the instruction set architecture (ISA).

As explained before, the techniques protecting the basic blocks (SIGBEV, SIGBTD, and SIGBTK) require some compiler support, whereas the techniques protecting cache lines (SIGC) are applicable to the already compiled code. However, the SIGCE techniques (protected cache lines with embedded signatures) may benefit from compiler support. The branch target addresses change due to embedded signatures, so either a compiler recalculates all target addresses, or address translation is done in hardware. In this dissertation we evaluate the SIGCE techniques that use hardware address translation.

All proposed techniques require a relatively simple processor modification: a dedicated processor resource for signature verification, the IBSVU (Figure 4.3). Techniques that keep signatures in the S-cache require additional on-chip area, so they are marked as having *Medium* hardware complexity in Table 4.1: SIGBTK, SIGCEKT, and SIGCTK.

The overhead of fetching a signature from the memory and its decryption is avoided if a signature is found in the S-cache, so techniques with the S-cache have a low projected performance overhead. The SIGBEV and SIGBTD techniques have potentially higher performance overhead than the corresponding SIGC techniques. With the SIGBEV, embedded basic block signatures may reduce the number of cache hits, leading to a medium performance overhead. With the SIGCTD, the access function of the signature table in memory is relatively simple, whereas with the SIGBTD a more complex hash function must be used to access a table of basic block signatures, thus adding additional latency. Figure 4.6 illustrates the qualitative assessment of signature verification techniques in the performance overhead - hardware complexity design space.

The advantage of the basic block protection techniques is that they can be used in systems without cache memory. Another advantage is that only instructions that are executed are verified, whereas only a portion of instructions in a cache line might be really needed. However, if protected blocks of a fixed size correspond to the prefetch buffer size and not to the cache line, they can also be used in a cache-less system. All techniques but one, the SIGBEV, do not require the change of the processor instruction set.

Table 4.1 Pros and cons of different techniques

	Compiler support	Hardware complexity	Projected performance overhead	Applicable without cache	ISA change
SIGBEV	Yes	Low	Medium	Yes	Yes
SIGBTD	Yes	Low	Medium	Yes	No
SIGBTK	Yes	Medium	Low	Yes	No
SIGCEV	No; may be used	Low	Low to medium	No	No
SIGCED	No; may be used	Low	Low to medium	No	No
SIGCEK	No; may be used	Medium	Low	No	No
SIGCTD	No	Low	Low to medium	No	No
SIGCTK	No	Medium	Low	No	No

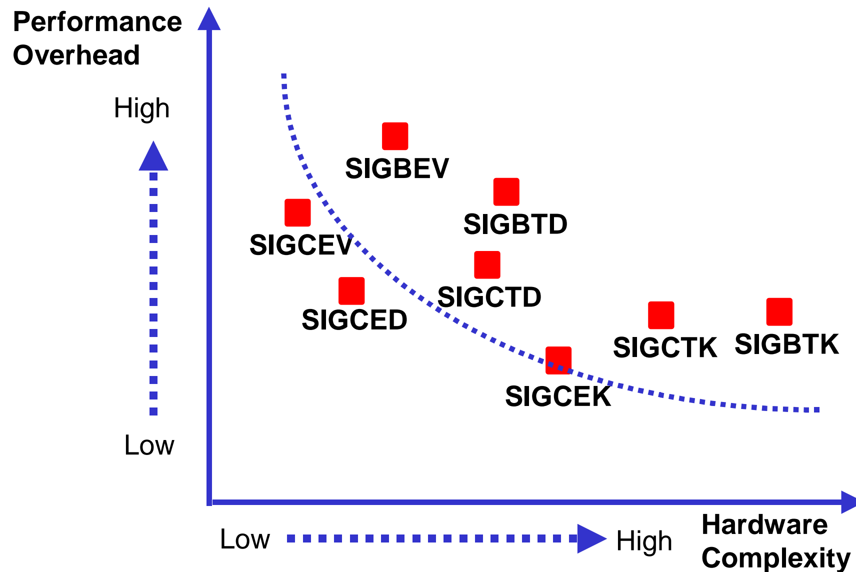

Figure 4.6 Qualitative assessment of signature verification techniques in the performance overhead - hardware complexity design space

4.3 Details of SIGCE Techniques

In this section we explain details of three SIGCE techniques. These techniques are

- SIGCED – signatures are invisible to the I-cache and discarded after verification;
- SIGCEK – signatures are invisible to the I-cache and kept in the S-cache;
- SIGCEV – signatures are visible to the I-cache.

We assume that all three techniques do not use compiler support, i.e., the original binary is modified during the secure installation process only by inserting signatures and necessary padding. If the last instruction block is shorter than the cache line, it is padded by instructions that do not change the state of the processor. If the code with embedded signatures is larger than a page size, it must be padded so that no instruction block is split between two pages. This padding is necessary for each page but the last.

4.3.1 SIGCED

The flow of the instruction fetch process is shown in Figure 4.7. The value of the program counter (*PC*) is used to access the I-cache. Note that without loss of generality we assume that the I-cache is indexed by virtual addresses and it is virtually tagged. This is a frequent case in embedded processor caches, for example in Intel's Xscale processor [127], and also in some high-end processors, for example Alpha 21264 [128]. In the case of a cache hit, the instruction is fetched from the I-cache and there is no need for instruction verification. In the case of a cache miss, we need to calculate the address of the instruction block to be fetched in the virtual memory. The instruction block address has changed because of signature embedding and added padding. If the padding is not necessary, i.e., one memory page can be completely filled with the protected instruction blocks and corresponding signatures, the true virtual address *tPCtemp* can be calculated as in Equation (4.1). The value *SigSize* is the size of the signature, *BlockSize* is the size of the protected block, and *TextBase* is the starting address of the text segment for a given program.

$$tPCtemp = PC + SigSize \cdot \left(\frac{PC\text{-}TextBase}{BlockSize} + 1 \right). \quad (4.1)$$

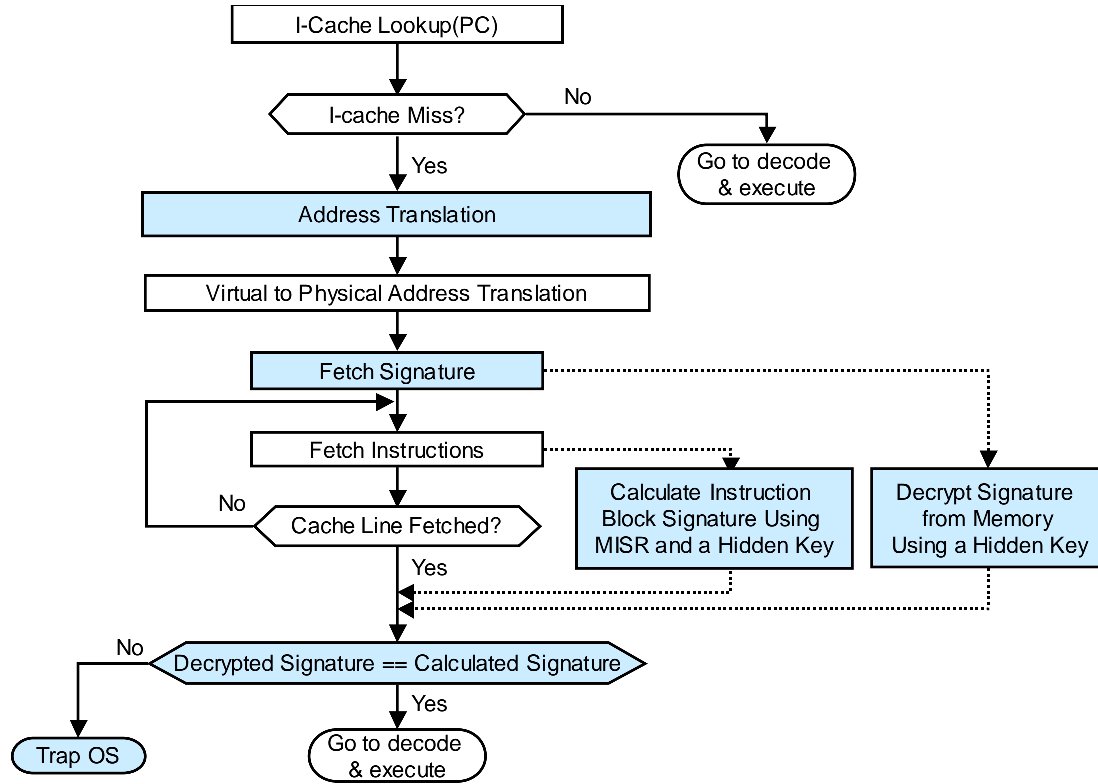

Figure 4.7 SIGCED: Signature verification control flow

Legend: Dotted lines indicate parallel tasks: AES decryption and MISR calculation are done concurrently with instruction block fetch. Shaded blocks indicate steps needed to support instruction block verification.

The size of the padding *PagePad* is given in Equation (4.2), with *PageSize* denoting the size of a virtual memory page. The final true address *tPC* can be calculated as in Equation (4.3).

$$PagePad = PageSize \bmod (BlockSize + SigSize)\ , \tag{4.2}$$

$$tPC = tPCtemp + \frac{tPCtemp\text{-}TextBase}{PageSize\text{-}\ PagePad} \cdot PagePad\ . \tag{4.3}$$

For example, consider a case where the I-cache line is 128B, the signature size is 16B, the page size is 4096B, the *TextBase* address is 131072, and the value of the *PC* of the instruction to be fetched as seen by the processor is 135200. In the original code without signatures, the size of a page is equal to the size of 32 instruction blocks. In the signed code, the size of a protected block together with its signature is 144B. Hence, 28 signed blocks can fit in one page, filling 4032 out of 4096B. Since one instruction block

cannot be split between two pages, the code must be padded so that the remaining 64B in a page are unused. All instruction blocks must have the same size, so if the last instruction block in a binary is shorter than the I-cache block, it is padded with randomly chosen instructions that do not change the state of the processor.

When a correct virtual address is calculated, the translation look-aside buffer (TLB) is accessed for virtual to physical address translation. In all considered SIGCE techniques a signature is inserted into the code just before the corresponding protected instruction block, so the signature can be fetched first. While instructions of a protected instruction block are being fetched, the signature is decrypted using a key hidden in the hardware. Each fetched instruction passes through the MISR register, and the final MISR output is compared to the decrypted signature. If the calculated and the decrypted signature differ, a trap to operating system is asserted; otherwise, the instructions proceed with execution.

If the time needed to fetch a cache line from memory is greater than or equal to the decryption time, there is no decryption performance overhead. The MISR calculation is completely overlapped with instruction fetch, so there is no MISR overhead either. Since in the I-cache the instruction addresses of the protected code are equal to the corresponding instruction addresses in the original code without signatures, the number of I-cache misses for the protected code is the same as for the original code. Hence, the performance overhead of the SIGCED technique is due only to the additional number of processor cycles during instruction fetch. Let t_{SigLat} be the additional latency due to signature verification mechanism. Then t_{SigLat}(SIGCED) is the sum of the time needed for address translation t_{Trans} and time needed to fetch a signature from the memory, $t_{SigFetch}$, as shown in Equation (4.4). The *MemBusWidth* value is the width of the data bus between memory and the I-cache in bytes. The t_{Dbus} value is the time needed for one data bus transfer.

$$t_{SigLat}(SIGCED) = t_{Trans} + t_{SigFetch} = t_{Trans} + \left\lceil \frac{SigSize}{MemBusWidth} \right\rceil * t_{Dbus} \ . \tag{4.4}$$

The signature verification is done by the Instruction Verification Unit (IBSVU), as illustrated in Figure 4.8. Signature bytes are stored only in the SIGM buffer and then decrypted, while instruction bytes

go both to the MISR logic and to the I-cache. An internal IBSVU signal, *sig*, controls the path of data from the data bus.

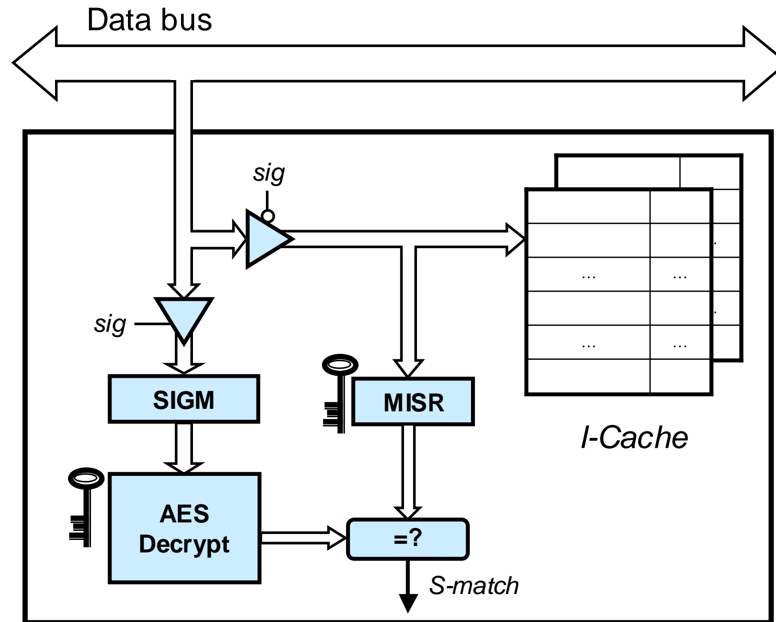

Figure 4.8 SIGCED: Instruction Block Signature Verification Unit

Legend: SIGM – buffer for storing the signature fetched from memory, sig – signal indicating whether data on the data bus are signature or instruction words, S-match – signal indicating whether the calculated and original signature match.

4.3.2 SIGCEK

A portion of the SIGCED overhead can be avoided if signatures are not discarded after verification, but kept in a dedicated cache-like IBSV resource – the S-cache. Figure 4.9 shows the flow of the instruction fetch process for the SIGCEK technique. With this technique, an I-cache lookup is performed concurrently with the corresponding S-cache lookup. In the case of an I-cache miss, the instruction block signature is fetched only if it is not found in the S-cache. Otherwise, if the decrypted signature is in the S-cache, the signature latency t_{SigLat} in Equation (4.4) is reduced to the number of cycles needed for address translation t_{Trans}. The SIGCEK technique has the potential to reduce not only the performance overhead of instruction signature verification, but also to reduce the power overhead due to

signature decryption, since a cached signature is already decrypted. Figure 4.10 shows a block-scheme of the IBSVU for the SIGCEK technique. A decrypted signature from the S-cache or a decrypted signature fetched from memory is compared to the final output of the MISR logic.

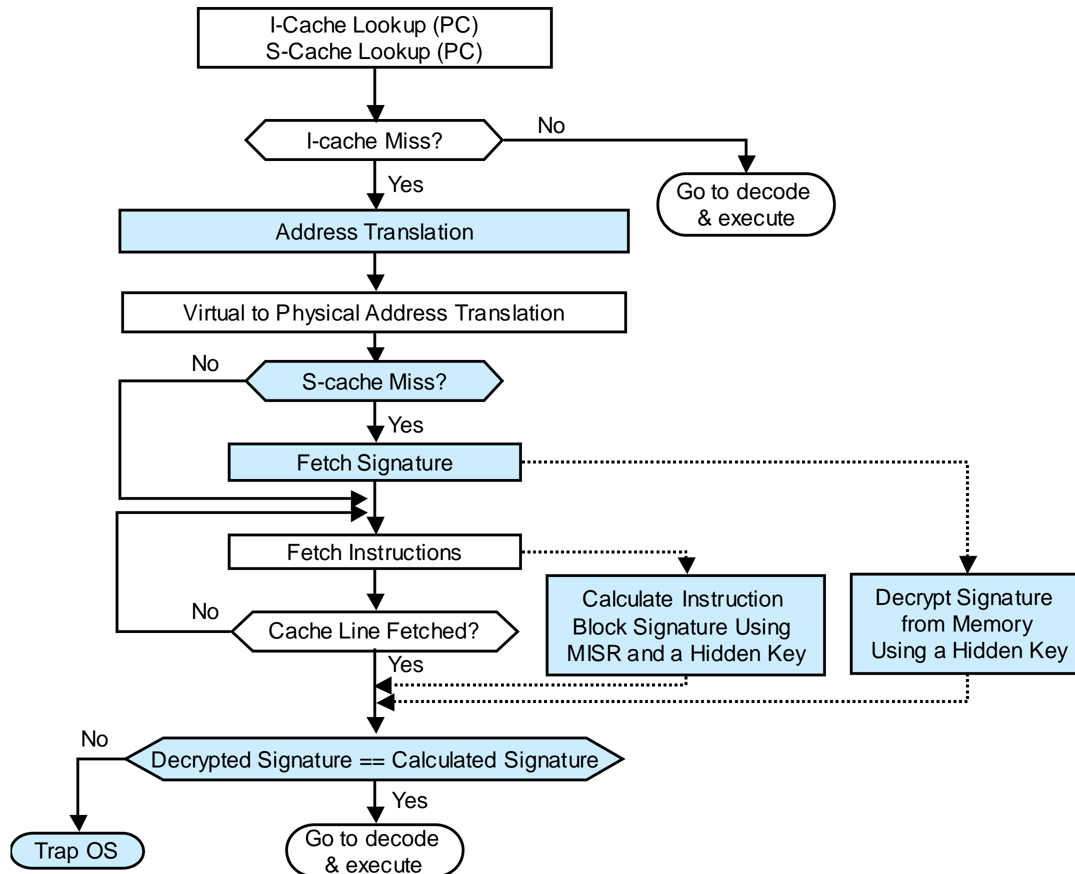

Figure 4.9 SIGCEK: Signature verification control flow

Legend: Dotted lines indicate parallel tasks. Shaded blocks indicate steps needed to support instruction block verification.

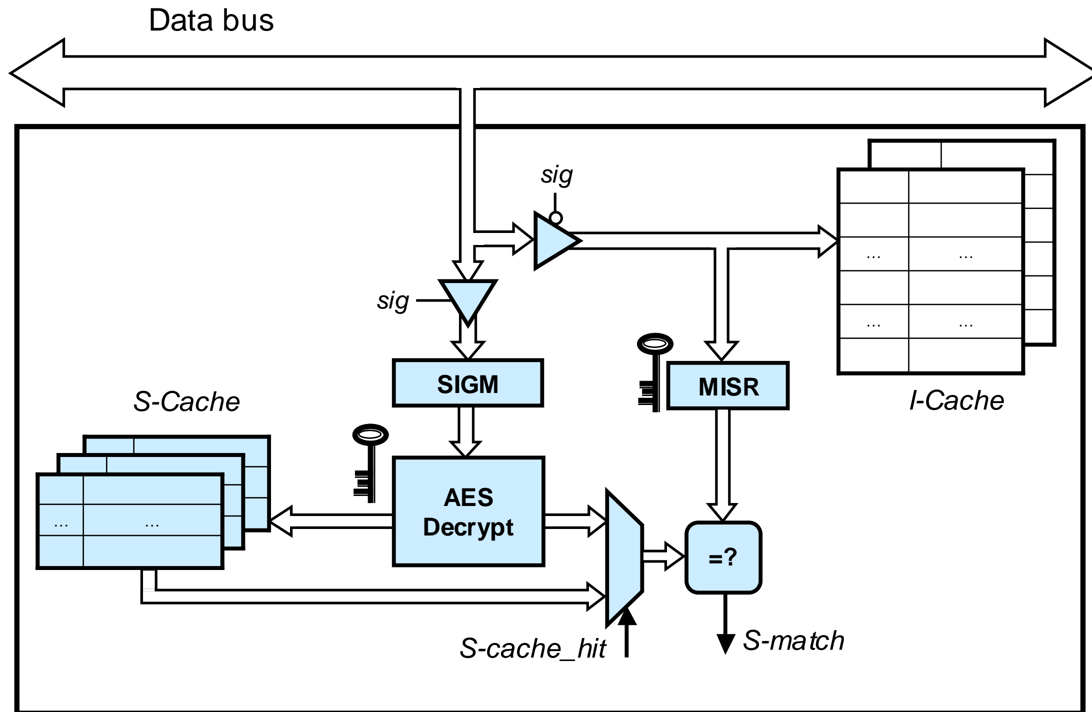

Figure 4.10 SIGCEK: Instruction Block Signature Verification Unit

4.3.3 SIGCEV

In all three SIGCE techniques, the virtual addresses of instructions as seen by the processor are the same for the original code and for the protected code with embedded signatures; that is, a virtual address of an instruction in the protected code after address translation is equal to the virtual address of that instruction in the original code. Two previously described techniques, the SIGCED and SIGCEK, use translated virtual addresses in the cache, so both processor and cache see only the original virtual addresses. Another option is the SIGCEV technique: translated addresses are used only in the processor, and the instruction cache sees the non-translated virtual address space that includes the signatures and padding. Hence, in the SIGCEV, the address translation must be done before each I-cache lookup. The advantage of this technique is that the translation in most cases can be done in advance, together with the prediction of the next instruction address. The only case when the performance overhead due to the translation cannot be hidden is when a branch is mispredicted. Figure 4.11 shows the SIGCEV control flow.

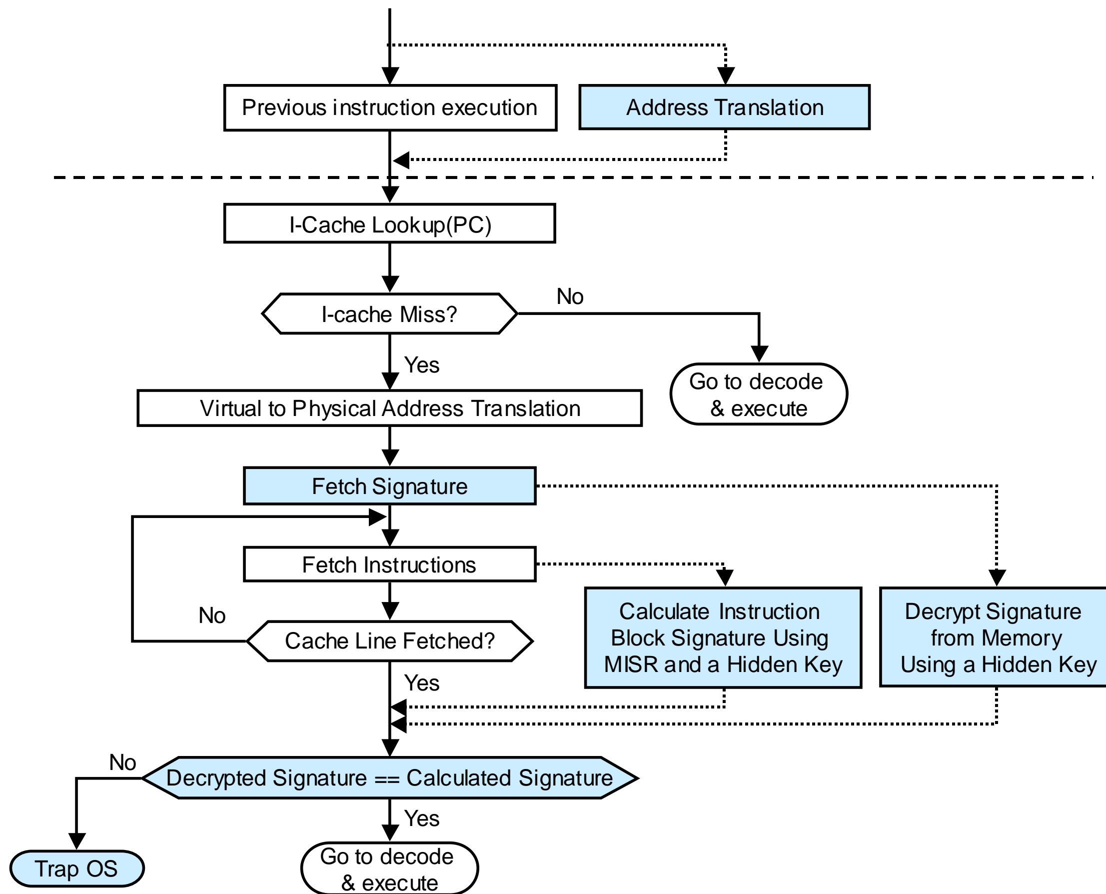

Figure 4.11 SIGCEV: Signature verification control flow

Legend: Dotted lines indicate parallel tasks. Shaded blocks indicate steps needed to support instruction block verification.

A simple and fast cache access mechanism requires both the cache line size and a cache line address to be a power of two. Hence, in the SIGCED and SIGCEK techniques, the size of a protected block is the power of two. Since instructions in the SIGCEV technique are stored in the I-cache using non-translated virtual addresses, the sum of sizes of a protected block and its signature are a power of two. For example, if an I-cache line size is 64 bytes, we can store 64 instruction bytes in one I-cache line in the SIGCED and SIGCEK, and 64 – *SigSize* bytes in the SIGCEV cache. Figure 4.12 shows the content of an I-cache line with the SIGCEV technique: The length of one instruction word *W* is 4 bytes, and the signature length is 16 bytes, i.e., 4 words. Although the signatures are visible to the SIGCEV I-cache, they are never stored in the I-cache, which is indicated by an *X* in the signature field *Sig*. Therefore, the cache

line size and consequently the cache size are effectively smaller than the corresponding SIGCED/SIGCEK values with the same cache line alignment (48 bytes instead of 64 in the example in Figure 4.12). If we know in advance that a system will execute only protected code, the SIGCEV I-cache line may be implemented as physically smaller, too. However, if we want to allow the unprotected mode, the I-cache line must have the "regular" length. The *Sig* field is unused in the protected mode and treated as a part of the *Instruction Words* field in the unprotected mode.

Another consequence of the requirement that the sum of the protected block size and signature size is a power of two is that the SIGCEV technique does not require page padding, thus simplifying address translation.

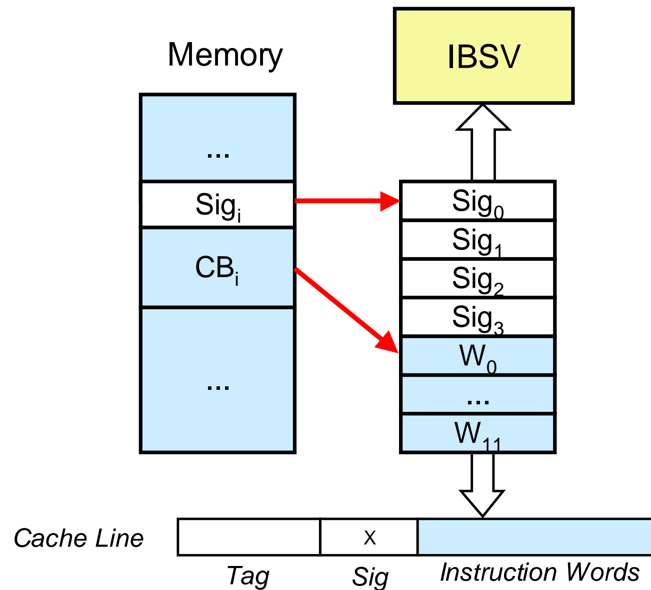

Figure 4.12 The content of an I-cache line with the SIGCEV technique

4.4 Details of SIGCT Techniques

In this section we will explain details of two SIGCT techniques, where signatures are stored in a separate code section. These techniques are

- SIGCTD – signatures are discarded after verification;
- SIGCTK – signatures are kept in the S-cache.

The SIGCT techniques have both advantages and disadvantages over the SIGCE techniques. Since signatures are stored separately from instructions, there is no need for hardware address translation and padding. On the other hand, signature fetch from memory requires a completely separate memory access. If the application code is relatively large, instructions and signatures may even be located on separate pages, so accesses to signatures may cause page faults.

4.4.1 SIGCTD

The flow of the instruction fetch process for the SIGCTD technique is shown in Figure 4.13. As with the SIGCE techniques, execution continues without signature verification in the case of an I-cache hit. The address of the corresponding signature is calculated in parallel, so the signature can be fetched first in the case of an I-cache miss. Since all protected blocks are of the same size, signature address *SigAddress* can be easily calculated as in Equation (4.5). As in previous equations, the value *SigSize* is the size of the signature, *BlockSize* is the size of the protected block, *TextBase* is the starting address of the text segment, and *PC* is the program counter. The value *SigTableStart* is the starting address of the table segment SigTable. The calculated *SigAddress* must not be greater than the *SigTableEnd*, the address of the last signature in the SigTable (block *Address in Sigtable* in Figure 4.13).

$$SigAddress = SigTableStart + SigSize \cdot \left(\frac{PC\text{-}TextBase}{BlockSize} \right). \quad (4.5)$$

The instruction fetch starts when the signature fetch is finished. The signature is decrypted in parallel with the instruction fetch, so the decryption latency is partially or completely hidden by the instruction fetch latency, as in the SIGCE techniques. The decryption latency hiding is the reason for the fetch order (signature first, then instructions).

If the decryption latency is completely hidden, performance overhead is of the SIGCTD is due only to the signature fetch latency, t_{SigLat}(SIGCTD) in Equation (4.6). However, since the signature fetch requires a separate memory access, signature fetch latency is longer than for SIGCE techniques. As before, *MemBusWidth* is the width of the data bus between memory and the I-cache in bytes, and t_{Dbus} is time needed for one data bus transfer. The value of $t_{MemAccess}$ encompasses all components of memory access

latency, such as address decoding latency, word line activation latency, word line activation latency, and output driving latency.

$$t_{SigLat}(SIGCTD) = t_{MemAccess} + \left\lceil \frac{SigSize}{MemBusWidth} \right\rceil * t_{Dbus} . \quad (4.6)$$

The Instruction Block Verification Unit for the SIGCTD is very similar to the IBSVU for the SIGCED (Figure 4.8), except differences in control logic.

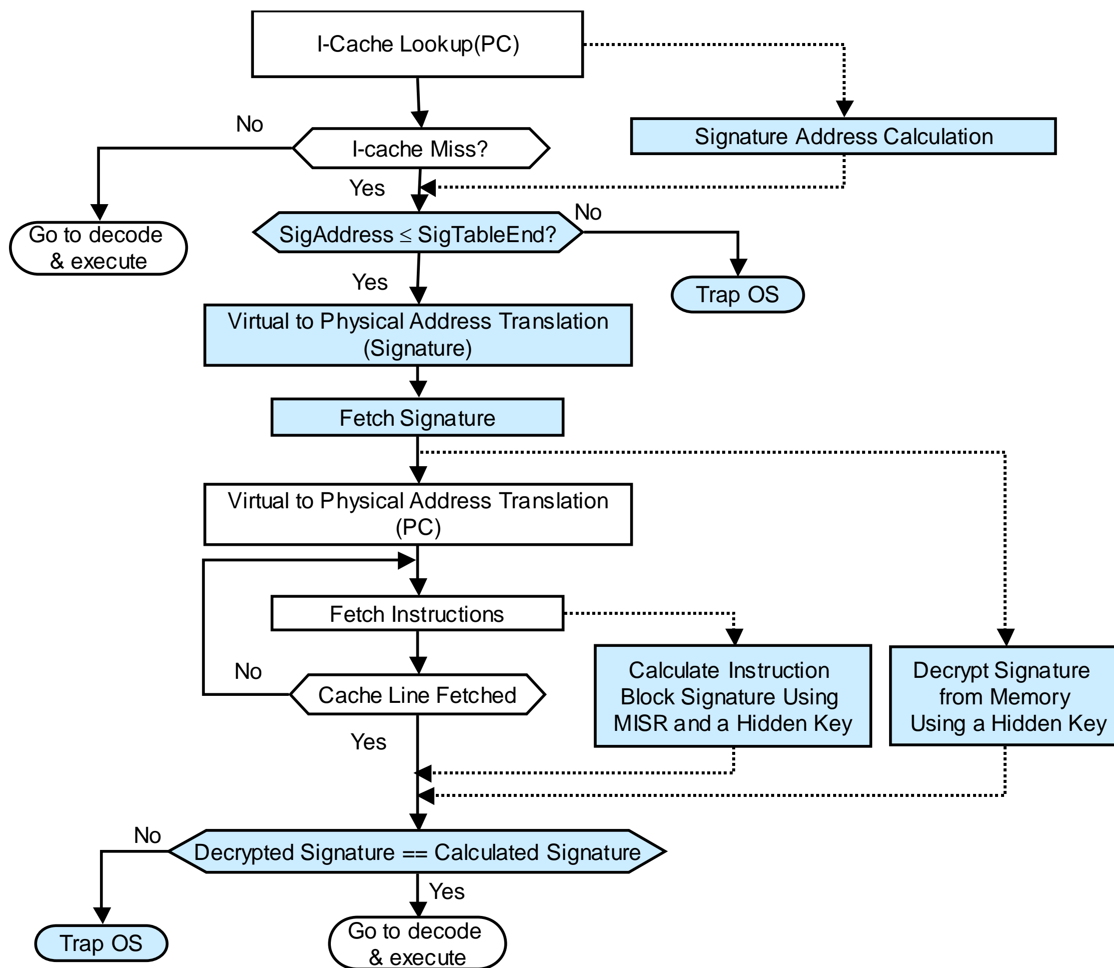

Figure 4.13 SIGCTD: Signature verification control flow

Legend: Dotted lines indicate parallel tasks. Shaded blocks indicate steps needed to support instruction block verification.

4.4.2 SIGCTK

Figure 4.14 shows the control flow for signature verification with the SIGCTK technique. If a signature of a block that is not in the I-cache is not found in the S-cache, signature latency is the same as for the SIGCTD (4.6). However, if the signature is in the S-cache and the decryption latency is hidden as explained before, the SIGCTK technique has virtually no overhead. The SIGCTK IBSVU is very similar to the SIGCEK IBSVU (Figure 4.10).

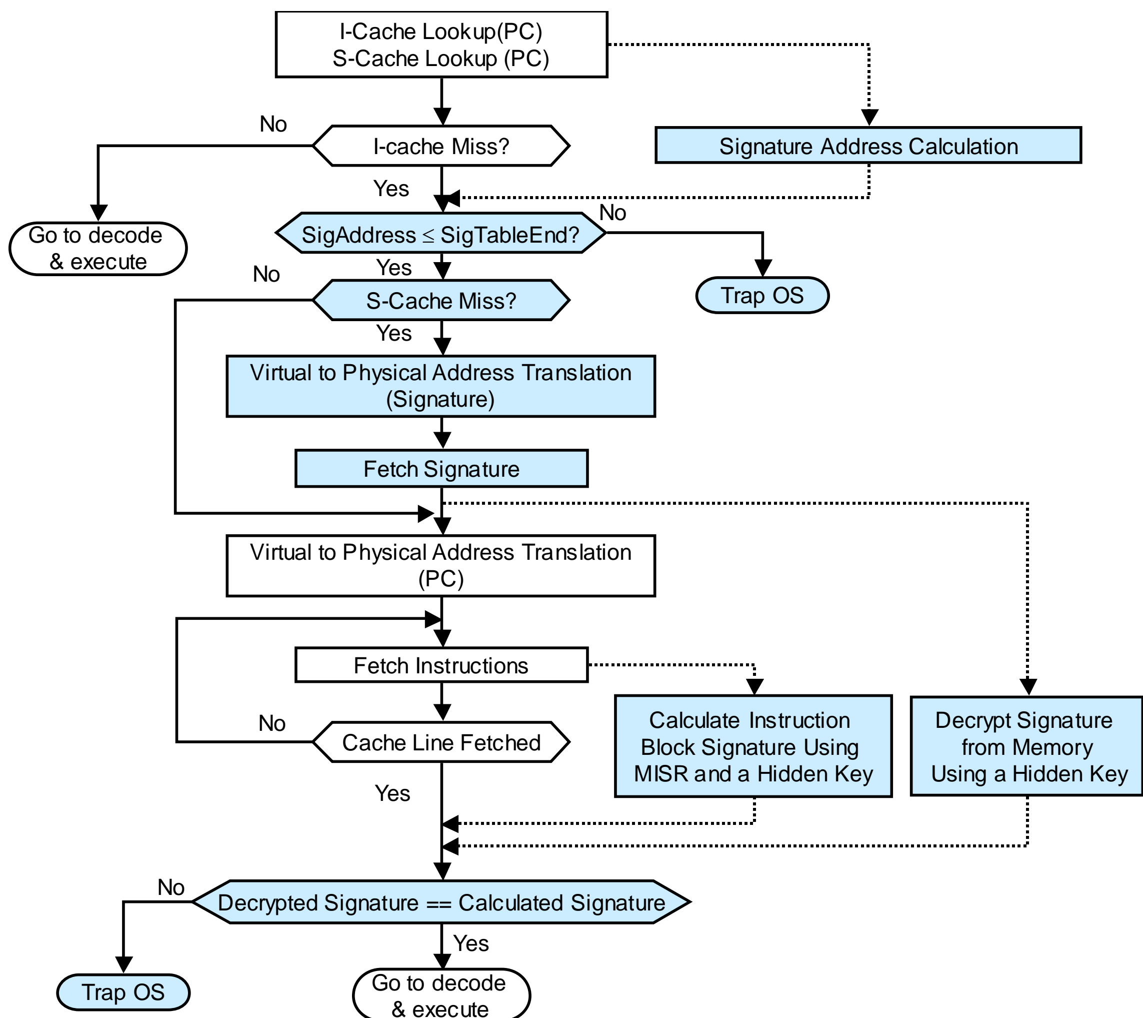

Figure 4.14 SIGCTK: Signature verification control flow

Legend: Dotted lines indicate parallel tasks. Shaded blocks indicate steps needed to support instruction block verification.

4.5 Details of SIGB Techniques

In this section we explain details of three SIGB techniques, where protected instruction blocks correspond to basic blocks. These techniques are

- SIGBEV – signatures are embedded in the code;
- SIGBTD – signatures are stored in a separate code section and discarded after verification;
- SIGCTK – signatures are stored in a separate code section and kept in the S-cache.

Techniques protecting basic blocks have a significant advantage of being able to avoid some verifications. One approach is to verify only the last basic blocks in instruction streams. In the rest of this dissertation, we focus on this approach, as it significantly reduces the number of verifications with no negative effects to the security (any injected code will change the original control flow). For instance, a consecutive sequence of basic blocks BB1, BB2, and BB3 in Figure 4.15 makes one instruction stream if the branches in BB1 and BB2 are not taken and the branch in BB3 is taken. In this case, only the BB3 signature will be verified in the stream BB1–BB2–BB3. Moreover, the signature needs to be verified only if at least one instruction in the BB3 basic block caused an I-cache miss.

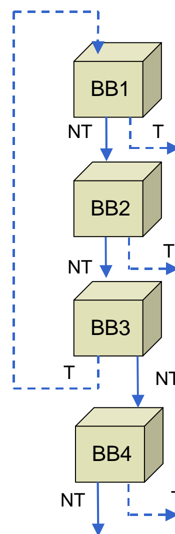

Figure 4.15 Instruction streams.
Legend: BB – Basic Block, T – Taken branch, NT – Not Taken.

Another advantage of the SIGB techniques is that they are applicable to systems without the I-cache memory, since they do not depend on the cache mechanism. In such a system, all last basic blocks in instruction streams need to be verified. A SIGC technique in a system without the I-cache would require fetching of instructions that might not be executed in order to verify the signature of the whole block. Moreover, signatures of all protected blocks would need to be verified, which would add a significant performance overhead.

Whereas the considered SIGC techniques do not allow execution of unverified instructions, the SIGB techniques overlap signature verification of one basic block with execution of the following one, so the detection of code injection attack is delayed. However, since memory writes are usually buffered, the actual verification can be safely delayed until the memory buffer is full. Hence, the main component of performance overhead will be due to the signature fetch.

However, the SIGB techniques also have several disadvantages when compared to the corresponding SIGC techniques. Unlike the SIGC, they require compiler support, although a modest one. In addition, the number of basic blocks in the code is likely to be larger than the number of instruction blocks of the size of the cache line, especially for predominantly integer applications. In that case, the code size increase with the SIGB is larger than with the SIGC techniques.

4.5.1 SIGBEV

In the SIGBEV, the signatures are embedded in the code, with each signature placed before the first instruction in the corresponding basic block. Although a control flow-changing instruction can be the last instruction in more than one basic block, each basic block has a unique first instruction, so the basic block start is a better choice for the signature placement. Moreover, if the signature is placed before instructions, than signature decryption may be partially overlapped with instruction execution. The signature instruction words are converted to no-ops in the decode stage of the pipeline.

An example of protected code and the corresponding original code is shown in Figure 4.16. This is an excerpt of assembly code of a simple loop written in C, on a Sun SPARC system. According to the definition of a basic block that we use, the original code in the example has five basic blocks, so the protected code has five embedded signatures. Note that the second, third, and fourth basic block have the

common end, the instruction `9. ble .LL9`. Let us consider the case when the program has just jumped to LL3, i.e., instruction 1. Hence, this instruction is the beginning of an instruction stream. If we assume that the branches in the instructions 3 and 9 are both not taken, the last basic block in this instruction stream consists of the instruction 10, and only its signature needs to be verified.

```
.LL3:
1.      ld      [%fp-24], %o0
2.      cmp     %o0, 0
3.      ble     .LL4
4.      ld      [%fp-24], %o0
5.      st      %o0, [%fp-28]
.LL5:
.LL4:
6.      st      %g0, [%fp-36]
.LL6:
7.      ld      [%fp-36], %o0
8.      cmp     %o0, 9
9.      ble     .LL9
10.     b       .LL7
```

Original code

```
.LL3:
     signature(1.-3.)
1.   ld   [%fp-24], %o0
2.   cmp  %o0, 0
3.   ble  .LL4
     signature(4.-9.)
4.   ld   [%fp-24], %o0
5.   st   %o0, [%fp-28]
.LL5:
.LL4:
     signature(6.-9.)
•    st   %g0, [%fp-36]
.LL6:
     signature(7.-9.)
7.   ld   [%fp-36], %o0
8.   cmp  %o0, 9
9.   ble  .LL9
     signature(10.)
10.  b    .LL7
```

Protected code

Figure 4.16 SIGBEV: An example of the original and the protected code

Figure 4.17 shows the IBSVU for the SIGBEV technique, and Figure 4.18 shows a high-level description of the corresponding SIGBEV procedures in pseudo-code. The IBSVU has two input control signals, *NewBB* and *IBSVU_Start*, and one output signal, *IBSVU_Busy*. The SIGM register holds the signature stored in memory, i.e., embedded in the code; the Last Block Signature register (LB.S) holds the calculated signature for the last basic block.

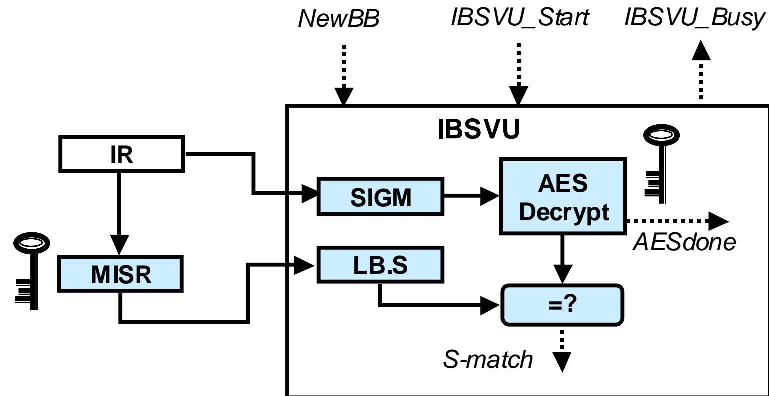

Figure 4.17 SIGBEV: Instruction Block Signature Verification Unit

Legend: IR – Instruction Register, SIGM – Signature stored in memory, LB.S –Last Block Signature register. Dotted lines indicate signals.

```
Instruction Decode Stage:
if (signature instruction) {
      if (IBSVU_Busy) wait;
      // IBSVU is free, start new basic block
      CacheMissFlag <= 0;
      NewBB <= 1;
}
else
      NewBB <= 0;


Branch Instruction Execution Stage:
if (BranchTaken && CacheMissFlag) {
      if (IBSVU_Busy) wait;
      IBSVU_Start <= 1;
}
else
      IBSVU_Start <= 0;

Instruction Block Verification Unit:
if (NewBB)
      SIGM <= IR;

if (IBSVU_Start){
      LB.S <= MISR; // signature of the last basic block in a stream
      StartAES;     // start decryption
}

if (AES_done) {
      IBSVU_Busy <= 0;
      if (AES(SIGM) == LB.S) SuccessfulVerification;
      else TrapOS;
}
```

Figure 4.18 SIGBEV Procedures

For the sake of simplicity, we assume that the signature length is one instruction word. The signature instruction is detected in the instruction decode stage. The control logic in the decoder then waits until the IBSVU clears the signal *IBSVU_Busy*. When the IBSVU indicates it is free, the decoder asserts the signal *NewBB*. The internal *CacheMissFlag* is set to 0; this flag is set to 1 on any I-cache miss.

The end of an instruction stream is detected in the execution stage of the pipeline, when a branch instruction is resolved as taken. If a branch is taken, and any of the instructions in the corresponding basic block caused an I-cache miss, the signature of that basic block should be verified. The control logic in the execution stage waits until the IBSVU is free and then asserts the *IBSVU_Start* signal.

The IBSVU loads the encrypted embedded signature from the IR to the SIGM register, when the signal *NewBB* is asserted. This signal also resets the MISR. When the signal *IBSVU_Start* is asserted, the IBSVU loads the calculated signature from the MISR to the LB.S register and starts AES decryption of the signature in the SIGM. When the AES block indicates that it is done with decryption (*AES_Done*), the IBSVU clears the *IBSVU_Busy* signal and compares the decrypted and calculated signatures. As in the SIGC techniques, if the two values do not match, the trap to operating system is asserted, and the operating system aborts the process.

Note that decryption of a signature does not start immediately after that signature is fetched, but when it is determined that the verification is needed. This implementation choice is motivated by two reasons. First, we want to avoid a more complex algorithm, where decryption of a basic block signature is aborted after fetch of a signature of the following basic block from the same instruction stream. In addition, unnecessary decryptions would increase the power consumption.

In the SIGBEV implementation explained above, the IBSVU cannot accept new basic block signatures while it is still decrypting the previous one. This condition can be relaxed if the SIGM is implemented as a First In First Out buffer (FIFO). In addition, the calculated signatures may also be buffered. We may allow further instruction execution until the memory write buffer is full, or, in even more relaxed model, until the SIGM or the LB.S is full.

Assuming that the decryption latency can be completely hidden, one component of performance overhead due to signature mechanism overhead is the signature fetch latency, $t_{SigFetch}$ as in Equation (4.7). However, since the signatures are stored in the I-cache together with instructions, they are likely to increase

the number of I-cache misses. More I-cache misses mean more fetch overhead for both instructions and signatures. For some applications, the additional instruction fetch overhead may be a dominant component.

$$t_{SigFetch} = \left\lceil \frac{SigSize}{MemBusWidth} \right\rceil * t_{Dbus} \, . \tag{4.7}$$

4.5.2 SIGBT

With both SIGBT techniques, the signatures are kept in a separate code section, SigTable. As explained in the Section 4.4.1, for SIGCT techniques the address of a signature can be easily calculated. Due to the variable size of basic blocks, mapping between basic block start addresses and the corresponding signatures is not a simple function, so a SigTable record in SIGBT techniques must include both a signature and a unique tag. The tag does not have to be encrypted. The simplest possible tag is the starting address of the basic block, relative to the beginning of the code. The offset of the new basic block from the beginning of the code is calculated by deducting the value in the PC register from the value stored in the program Starting Address register (SA), and stored in the Current Block Starting Address register (CB.SA). The SIGBTK technique stores decrypted basic block signatures and the corresponding starting address offsets in the S-cache; the starting address offset is used to determine the S-cache index.

Figure 4.19 shows the IBSVU for the SIGBTK technique, and Figure 4.20 shows a high-level description of the corresponding SIGBTK procedures in pseudo-code. The control logic in the decoder calculates the CB.SA value for each new basic block. As with the SIGBEV technique, the control logic in the execution stage asserts the signal *IBSVU_Start* when there was an I-cache miss in the last basic block in an instruction stream.

When the signal *IBSVU_Start* is asserted, the IBSVU loads the current value of the CB.SA into the LB.SA register, and loads the calculated signature from the MISR to the LB.S register. The S-cache is searched for an entry with the LB.SA tag. In the case of an S-cache hit, the signature from the S-cache is compared to the calculated signature in LB.S.

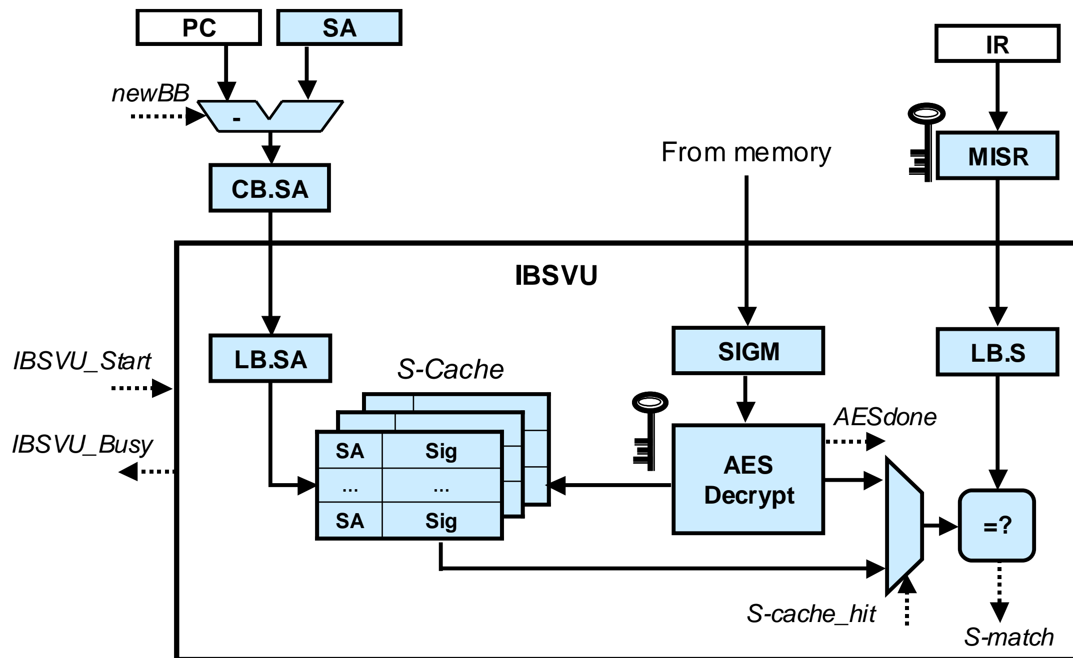

Figure 4.19 SIGBTK: Instruction Block Signature Verification Unit

Legend: PC – Program Counter, IR – Instruction Register, SA- Starting Address, LB.S – Current Block Signature Register, CB.SA/LB.SA Current Block/Last Block Starting Address Offset, SIGM – Signature from memory. Dotted lines indicate signals.

An S-cache miss indicates either an infrequently executed basic block or injected code, so the SigTable section in memory must be searched for the record with the starting address offset equal to LB.SA. If such record is found, the signature is fetched and the AES decryption started. If there is no record in the SigTable with the LB.SA tag, the last executed basic block is not signed, so the IBSVU asserts the trap to the operating system.

The result of the AES decryption is compared to the calculated signature in LB.S, as in the SIGBEV. In addition, the IBSVU updates the S-cache with the decrypted signature and the LB.SA.

Sequential search of the SigTable is inadmissibly slow, so we must use other search methods. One implementation of an improved binary search is illustrated in Figure 4.21. The SigTable is divided into segments and sorted in a monotonic order by starting address offset. The SigTable start and end addresses are stored in the IBSVU, as well as the values of starting address offset fields for each beginning of the segment. Hence, we know exactly which segment should be searched with a given starting address offset. That segment than can be searched using binary search.

```
Instruction Decode Stage:
if (last instruction == control flow instruction)
      NewBB <= 1;
else
      NewBB <= 0;
if (NewBB)  CB.SA <= PC - SA;


Branch Instruction Execution Stage:
if (BranchTaken && CacheMissFlag) {
      if (IBSVU_Busy) wait;
      IBSVU_Start <= 1;
}
else
      IBSVU_Start <= 0;

Instruction Block Verification Unit:
if (IBSVU_Start){
      LB.S <= MISR; // signature of the last basic block in a stream
      LB.SA <= CB.SA;  // start address offset
      if (S-cache_hit(LB.SA)){ // S-cache hit
            IBSVU_Busy <= 0;
            if (S-cache.S == LB.S) SuccessfulVerification;
            else TrapOS;
      }
      else {  // fetch encrypted signature from memory and decrypt
            if (SearchSig(LB.SA)){  // a signature found in memory
                  SIGM <= FetchedSig;
                  StartAES;     // start decryption
            }
            else  TrapOS;
      }
}

if (AES_done) {
      IBSVU_Busy <= 0;
      if (AES(SIGM) == LB.S){
            SuccessfulVerification;
            UpdateS-cache(LB.SA, AES(SIGM));
      }
      else TrapOS;
}
```

Figure 4.20 SIGBTK Procedures

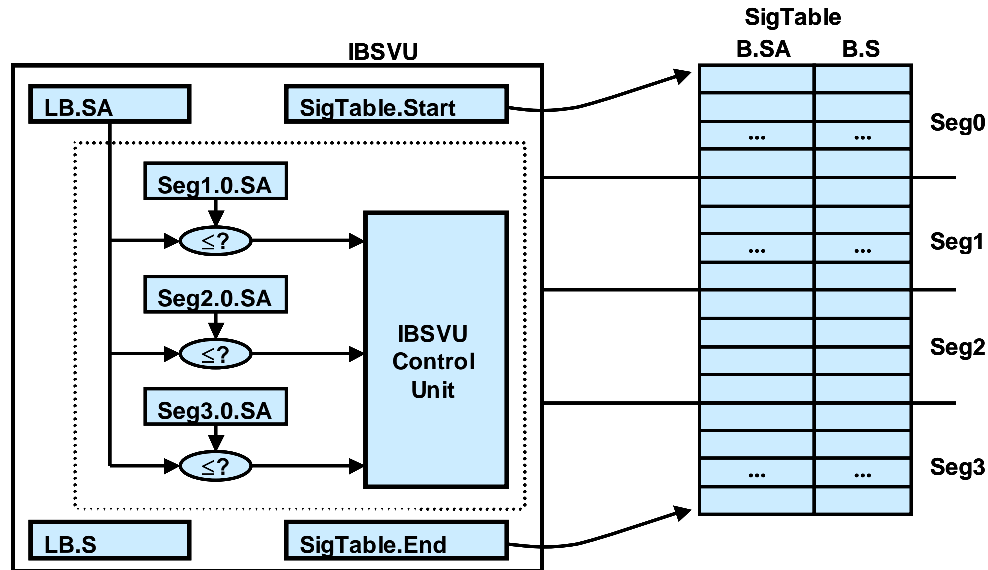

Figure 4.21 SigTable access using segment approach

*Legend: LB.S – Last Block Signature Register, LB.SA – Last Block Starting Address Offset, SigTable.Start and SigTable.End – the addresses of the first and last records in the SigTable, Seg*i*.0.SA – starting address offset value in the first record in the* i*-th segment in the SigTable.*

The search mechanism can be even faster. Since the SigTable does not change for a given program, the secure installation process may find a near-perfect hash function for a particular application [129, 130], or choose the most suitable hash function from a predefined set of functions. A perfect hash function is a function that maps the domain of records to the hash table one-to-one, that is, without collision. The information about the chosen hash function may be kept in the program header in an encrypted form.

Since the S-cache stores decrypted signatures, in the case of an S-cache hit there is no additional latency due to verification. On the other hand, in the case of an S-cache miss the signature must be found in the SigTable, fetched, and decrypted. Decryption can be done in parallel with execution of the following basic block, so the main performance overhead component is due to the SigTable access latency and the number of times the SigTable must be accessed to find a signature or to decide that such signature does not exist.

The S-cache is the only difference between the SIGBTD and SIGBTD techniques. With the SIGBTD, the SigTable access latency cannot be eliminated.

4.6 Discussion

Efficient implementation of a signature verification mechanism poses quite a few challenging questions. In this section we discuss various complexity, performance and security issues of the proposed mechanism.

4.6.1 Reducing Memory Overhead

An instruction block signature must have at least 128 bits to be cryptographically sound. On the other hand, the code size increase is directly proportional to the average length of protected blocks and the signature length. For example, if the protected block size in a SIGC technique is 128B and the signature is 16B, the increase of code section of a binary due to signatures is 12.5%, plus the increase due to padding. Though this increase will not put a lot of additional strain on the memory requirements in high-end systems, it may be significant in embedded processors with smaller caches and consecutively smaller cache lines. To address this problem, one approach would be to protect multiple cache blocks with a single signature. All blocks protected with one signature must be brought to the IBSVU and may or may not be stored in the I-cache, so this mechanism can be combined with instruction prefetching. The evaluation of this approach is out of the scope of this dissertation.

For the SIGB techniques, the average length of protected blocks is application dependent. For example, the average basic block length in SPEC CPU2000 benchmark set is 19.3 – 41.9 bytes on Alpha architecture [131]. One way to reduce the number of signatures would be to sign only selected basic blocks, applying strategies from fault-tolerant computing [113]. However, reducing the number of signatures must not reduce the ability of a system to detect code injection attacks.

4.6.2 Reducing Performance Overhead

For the SIGC techniques, in this dissertation we focus on conservative verification, where instructions of a signed instruction block are not allowed to execute before their signature is fetched, decrypted, and verified. A more relaxed approach allows blocks to execute unverified, but unverified instructions are not allowed to write to memory. An even more relaxed approach may even allow

execution of a certain number of unverified stores, but it then must be able to perform a rollback if a code injection attack is detected. A similar relaxed approach can be applied to the SIGB techniques.

Performance overhead for all signature verification techniques might also be reduced with prefetching of code and signatures as described above, or only with signature prefetching in techniques with the signature cache.

4.6.3 Systems with more than one level of cache memory

Signature verification is always related to the lowest level of cache memory, i.e., the cache that is the closest to memory. For example, in a system with two levels of cache memory, a signature should be verified if the corresponding protected block is not in the L2 I-cache. Performance overhead of the proposed techniques is insignificant in systems with more than one level of cache, since the number of cache misses relative to the number of executed instructions is very low in lower level caches.

If the lowest cache level is unified, the proposed techniques exploit the *dirty* cache line bit. This bit is set to 1 if the contents of the cache line have changed since loading it into the cache, i.e., if there were any stores to the cached locations. As described in previous sections, a signature is verified if the corresponding instruction block is not in the lowest level of cache. It is also verified if it is found in the cache, but with a dirty cache line.

4.6.4 Dynamically Linked Libraries

With the proposed techniques, each dynamically linked library (DLL) has its own signature section or embedded signatures, so all code can be safely verified. The pointers to signature sections or the beginning of the code with embedded signatures can be loaded to the IBSVU when a particular library is dynamically linked. The IBSVU stores a fixed number of such pointers. When an application is dynamically linked with more DLLs than the IBSVU can hold, the overflow is handled by the operating system, and the overflow data is stored in memory.

4.6.5 Context switch

In the case of a context switch or interrupt, the signature information such as the current MISR value is saved together with the other process data, and restored when the process resumes execution. The S-cache may be flushed at each context switch, or it may include a process ID tag.

4.6.6 Dynamically Generated Code

In some cases, the executing code may be dynamically generated and possibly never saved in an executable file. One such example is the code generated by the Java Just-In-Time compiler (JIT). The dynamically generated code can be marked as non-signed and executed in the unprotected mode. Another option is to let the dynamic code generator generate the signatures together with the code. If the generator is trusted, its output should be trusted too. The same argument applies to the interpreted code.

4.6.7 Return-Into-Libc Attacks

So-called *return-into-libc* attacks overwrite a code pointer to point to the regular application code, usually the library code. Signature verification alone cannot detect a return-into-libc attack, since the library code is also signed. However, our approach can be combined with other defense techniques to prevent the success of return-into-libc; these techniques may also benefit from hardware support. One such technique is address layout randomization [76], and another one is denying a system call requests if they do not adhere to predefined rules [89].

4.6.8 High-Level Attacks

We believe that the proposed mechanism is an important step toward more secure computer systems. However, we do not claim that signature verification will solve all the security-related issues we face today. Rather, we believe that no single technique can do that, but that a combination of hardware and software techniques is needed, working in concord at different levels of abstraction. For example, our technique will not prevent execution of malware elevated to a trusted level by users, nor will it prevent

high-level attacks such as SQL injection, where malicious SQL keywords are added to an SQL database query. One example of SQL injection vulnerability is a login form that requires username and password from the database by using SQL command WHERE, but allows any characters in the input. Hence, an attacker might use the SQL keyword OR to set a condition which is always true and bypass the login condition [132].

CHAPTER 5

EXPERIMENTAL METHODOLOGY

"If you don't know where you are going, you will wind up somewhere else."
Yogi Berra

This chapter describes experimental methodology used to evaluate techniques for instruction block verification. We first describe the experiments and metrics used to evaluate performance overhead. To mimic the secure installation process, we modify files in the Executable and Linkable Format (ELF) [133]. To simulate secure execution with the SIGC techniques, we modify the execution-driven simulator SimpleScalar [134] to execute modified ELF files and to measure additional latencies due to signature verification. For evaluation of the SIGB techniques, we use a custom-made trace-driven simulator. We measure sensitivity of performance overhead to various architectural parameters, and with benchmarks selected from both the embedded and high-end processor domain.

5.1 Evaluation of Proposed Techniques

All proposed techniques successfully detect code injection attacks, since an injected code sequence cannot have a valid signature. Whereas the security is the main criteria when assessing the quality of a defense technique, it is far from being the only one. A hardware-supported defense technique should not add significant overhead in hardware complexity, execution time, and memory requirements. Hardware complexity of all proposed techniques without the S-cache is very low. The complexity of

techniques with the S-cache depends on the S-cache size. We evaluate only implementations where the S-cache is smaller than the I-cache size, so the complexity of these techniques is also moderate.

Memory overhead is simply determined by comparing the sizes of the original code and the code with protected instruction blocks. As explained in the next section, an executable file consists of different segments, and only segments containing instructions are protected with signatures. To better understand memory overhead, we measure the ratio between the sizes of original and signed instruction sections, and the ratio between the sizes of original and signed executable files.

Performance overhead for an application may be determined by comparing the execution time of the original code in the base system without signature verification, and the signed code in a system with signature verification. Performance overhead can also be assessed by measuring the number of events that are a major contributor to total overhead, such as the increase in the number of I-cache misses. The SIGC techniques are more readily applicable than the SIGB, so we evaluate the SIGC using a detailed execution-driven architectural simulator. We modified the most detailed simulator in the SimpleScalar tool set [134] to support the SIGCED, SIGCEK, SIGCEV, SIGCTD, and SIGCTK techniques. The performance metric is the CPI, the number of processor clock cycles per instruction. As an indirect indication of performance overhead, we can also use the number of S-cache misses for the SIGCEK and SIGCTK, and the increase of I-cache misses for the SIGCEV technique. The metric for cache misses is the number of misses per one thousand instructions, or per one million instructions (1M).

For the SIGB techniques, originally developed trace-driven simulators are used to assess performance overhead. In this case, we measure the number of signature verifications per one million executed instructions. In addition, we measure the increase in the I-cache misses for the SIGBEV, and the number of S-cache misses for the SIGBTK technique.

5.2 ELF Format

When instruction block signatures are added to an executable file, the resulting file must still conform to the executable format. To offer a proof-of-concept of the proposed techniques, we modified executable files in the ELF format. This section explains the main characteristics of this format.

ELF is the executable format widely used in Linux, Unix, Solaris, and other similar operating systems. An ELF file may be a relocatable, executable, or shared object file. As defined in the Tool Interface Standard (TIS) ELF Specification, version 1.2, a relocatable file is a file that can be linked with other object files to create an executable or a shared object file, and an executable file holds an executable program [133]. Shared object files are essentially libraries, which can be linked with other object files either statically or dynamically.

Figure 5.1 illustrates the view of an ELF file as seen by the linker (linking view) or program loader (execution view). Any ELF file begins with the ELF header, which describes the organization of the file. For example, an ELF header specifies the file type, required architecture, the virtual address of the start of the program, and the position of program header table and section header table in the file.

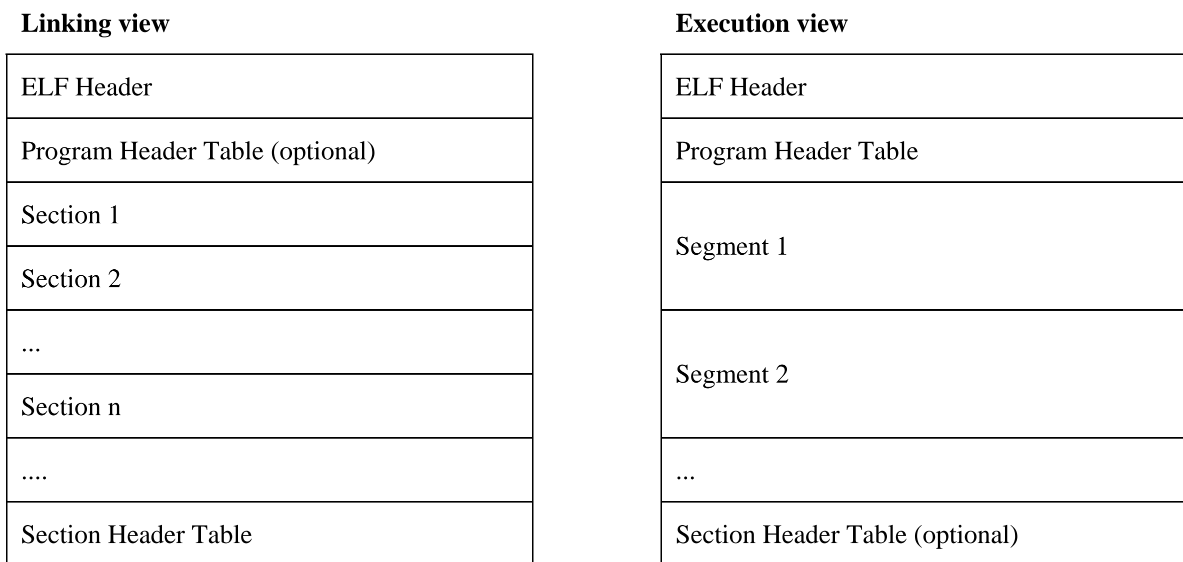

Figure 5.1 Linking and execution view of an ELF file

From the linker's point of view, an ELF file is divided into sections. Section Header Table is an array of structures that specify file sections. One such structure specifies the name, type, start address in the memory image of a process, position in the file, size, and the attribute flag for one section, as well as some other information. For example, the attribute flag specifies whether a section contains instructions or data. It is interesting to note that the section name is an index into a special section, called header string

table, which holds actual names in ASCII. The ELF specification defines a number of special sections. Table 5.1 lists some common ELF file sections and the corresponding content.

Table 5.1 Some common ELF file sections

Section	Content
.bss	Uninitialized program data
.data	Initialized program data
.debug	Symbolic debugging information
.dynamic	Dynamic link information
.rodata	Read-only program data
.shstrtab	Header string table
.strtab	Symbol table names
.symtab	Symbol table
.text	Program instructions
.init	Program initialization code
.fini	Program terminating code

When the loader loads a program in the memory, it considers the ELF file as a group of segments, which in turn have one or more sections. Segments are specified in program header table. For each segment, this table specifies the type, the position in the file, the virtual and sometimes physical address of the first segment byte in memory, the size in the file and the size in memory, and some other information. For example, the segment type specifies whether a segment is loadable, i.e., mapped to memory.

5.3 Secure Installation of Files in ELF Format

Since the SimpleScalar simulators can execute only statically linked code, we wrote a program that emulates the secure installation process for the executable ELF files. An executable ELF file with embedded signatures is modified in the following way:

- Encrypted signatures and padding are added to instruction blocks in sections containing instructions, i.e., *init*, *text*, and *fini*. By default, these sections are mapped in memory one after another, so they can be treated as a single entity for the signing purpose.
- The section length and position information are adjusted. We change only the length of the *fini* section. If a section is located in the file after *fini* in the original code, it is shifted for the total length of embedded signatures and padding, and its position is adjusted in the section table header.
- Program table header should also be updated. However, the *gcc* cross-compiler that comes with the SimpleScalar creates a section header table with information about virtual memory address for loadable sections, and the SimpleScalar loader considers this header only when loading a program into simulated memory. Hence, we only update the section header table.
- By default, the data sections are mapped to virtual memory after code sections, which may cause problems to instructions that read or write to memory. Instead of disassembling the code and recalculating all load and store addresses, we chose to modify the default options of the GNU linker, so that data sections are mapped before the code.

An executable ELF file with instruction block signatures in the signature table section requires the following modifications:

- The signature table section, *.sigt*, is added as the last section in the file.
- The entry for the signature table section is added to the section header table, and the entry for the corresponding segment is added to the program header file (signature table section makes one segment).
- The signature virtual address is set to be larger than the last address of loadable sections, so there is no need to change header information for other sections.

ELF dynamically linked library files can be modified in a similar way. Since only already linked files and libraries are securely installed, signatures are never added to relocatable ELF files.

5.4 SimpleScalar Simulator

The SimpleScalar [134] encompasses architectural simulators that are the most widely used tools in computer architecture research. The SimpleScalar website, www.simplescalar.com, lists its use in papers published in top computer architecture conferences. For example, in the year 2002 more than one third of all top conference papers used the SimpleScalar. Such popularity is due to several reasons. First, the SimpleScalar is an open-source free tool developed in academia (University of Wisconsin). It includes simulators for Alpha, PISA (a MIPS-like portable instruction set architecture), ARM, and PowerPC architectures. In addition, it can run on various host platforms, including Linux, FreeBSD, Alpha OSF1, SPARC SunOS, AIX Unix, and Windows NT under CygWin.

SimpleScalar simulators are primarily execution-driven. The input of an execution-driven simulator is a "regular" executable file, whereas the input of a trace-driven simulator is an instruction execution trace. Trace-driven simulators do not have to decode and perform instructions, so they are usually less complex than the execution-driven simulators with the same functionality. Another advantage is that trace-driven simulations are completely repeatable experiments, while the results of some execution-driven simulators may differ between several runs. However, for each application we want to simulate we must first collect and store its trace, while executable files are readily available and do not require a lot of storage space. Moreover, traces collected with certain architecture may not be suitable for simulation of another system. SimpleScalar simulators handle system calls by passing them on to the host system, so simulations are not absolutely repeatable. However, some simulators in the set include an option to use EIO (external input output) traces, i.e., traces of interaction with the host system, thus providing complete repeatability.

Any system simulator is either fast but simulating the system at a high level of abstraction, or more detailed, but slower. The SimpleScalar balances these opposite requirements by including several simulators at various level of detail, such as sim-safe, sim-profile, sim-cache, sim-bpred, and sim-outorder. The sim-safe and sim-profile are functional simulators executing instructions serially, without accounting for execution time. The difference between the two is that the sim-profile gives various statistics. The sim-cache and sim-bpred simulate details of memory hierarchy and branch prediction, respectively. The sim-cache can simulate up to two levels of cache memory, with unified or split data and instruction caches,

various cache sizes, degree of associativity, and replacement policies. The sim-bpred simulator can simulate five types of predictors: always taken, always not taken, bimodal, two-level, and a combination of a bimodal and a two-level predictors. It also simulates a branch target buffer with various number of entries and sets, and the return address predictor. Finally, the sim-outorder simulator provides a detailed microarchitectural timing model of a system with possibly out-of-order execution and various available resources. It reuses functionality of sim-bpred and sim-cache.

The SimpleScalar toolset also includes precompiled libraries and applications, as well as the complete GNU tool chain for PISA and ARM architectures. The source code is written in C, very well documented and well organized, so the simulators are relatively easy to modify. Table 5.2 lists *.c* files that are used by various SimpleScalar simulators, and their short description. Most of these files have corresponding header files with function prototypes. Figure 5.2 shows the main simulator loop body in the sim-outorder simulator (some comments and "sanity checks" are removed).

```
/* commit entries from RUU/LSQ to architected register file */
ruu_commit();

/* service function unit release events */
ruu_release_fu();

/* service result completions, also readies dependent operations */
ruu_writeback();

/* try to locate memory operations that are ready to execute */
lsq_refresh();

/* issue operations ready to execute from a previous cycle */
ruu_issue();

/* decode and dispatch new operations */
ruu_dispatch();

/* call instruction fetch unit if it is not blocked */
if (!ruu_fetch_issue_delay)
     ruu_fetch();
else
     ruu_fetch_issue_delay--;
```

Figure 5.2 The main simulator loop body in the sim-outorder simulator

Table 5.2 Descriptions of .c files used by SimpleScalar simulators

File name	**Functionality**
bpred.c	Handles branch predictors
cache.c	Handles cache memory
eio.c	Interface to EIP traces
eventq.c	Handles ordered event queues
loader.c	Target program loading in simulated memory
machine.c	ISA definition routines
main.c	Initialization, launch
memory.c	Access to simulated memory space (large flat space)
misc.c	Support function such as *fatal()* and *warning ()*
options.c	Processing options from configuration files or command line
ptrace.c	Pipeline traces
regs.c	Manages register files
resource.c	Manages functional units
stats.c	Handles statistics
syscall.c	Interface between system calls in simulator and on the host machine

In order to eliminate synchronization problems, the pipeline stages are traversed in the reverse order. The fetch stage is implemented in the function *ruu_fetch()*, which models fetch bandwidth. The dispatch stage is implemented in *ruu_dispatch()*, which models instruction decoding and register renaming. The instructions are actually executed in this function, to facilitate data-dependent optimizations and early detection of branch mispredictions. The scheduler, which issues instructions to different functional units, is implemented in *ruu_issue()*; the memory scheduler, which issues memory operations, is implemented in *lsq_refresh()*. The instruction execution stage of the pipeline is also implemented in *ruu_issue()*. Whereas the results of instruction execution are already known in the *ruu_dispatch()*, the *ruu_issue()* function models various latencies related to execution. The writeback stage is implemented in *ruu-writeback*, which

models the writeback bandwidth and in the case of branch misprediction, initiates misprediction recovery. Finally, *ruu_commit()* implements the commit pipeline stage, and models in-order retirement of instructions.

5.5 SimpleScalar Modifications

To evaluate the proposed SIGC techniques, we modify the SimpleScalar ARM architecture. The sim-outorder for ARM has the same configuration parameters as the sim-outorder for Alpha target. Hence, we can simulate both embedded and high-end processor systems using the same simulator, by changing only configuration data. In addition, the SimpleScalar/ARM compiler kit enables us to choose benchmarks at will, without being restricted to a precompiled set of applications.

We have modified simoutorder.c, memory.c, memory.h, cache.c, and cache.h SimpleScalar files. The implementation of the signature verification mechanism required the following modifications:

- Support for execution of signed ELF files. The simulators of the SIGCE techniques require support for the instruction address translation, and data address translation for data located in the text segment.
- Additional latencies due to signature verifications in *ruu_fetch()*: the translation latency, decryption latency, and signature fetch latency.
- Support for the S-cache. All SimpleScalar cache objects are created and accessed using common functions, with different functions handling cache misses. We use the same approach for the S-cache.
- Support for additional simulator options to specify latencies and S-cache parameters. In the SimpleScalar simulators, all options are registered by type into an options database, so we add the code for declaring and registering options required by the modified simulators.

5.6 Custom-Made Trace-Driven Simulator

A trace-driven simulator may be an ideal choice for modeling particular aspects of processor architecture, especially when a large trace database is already available. During prior research of efficient traces compression, we proposed Stream-Based Compression (SBC), a novel technique for single-pass compression of address traces and various extended trace formats [135, 136]. We also collected address

traces for SPEC CPU2000 benchmark set [137], and stored it using the SBC compression. Traces are generated using a modified SimpleScalar environment, precompiled Alpha binaries, and SPEC CPU2000 reference inputs. We traced two segments for each benchmark: the first two billion instructions (F2B), and two billion instructions after skipping 50 billion (M2B).

The SBC algorithm relies on extracting instruction streams. A stream table created during compression encompasses all relevant information about streams: the starting address, stream length, instruction words in the stream, and their types. All instructions from a stream are replaced by its index in the stream table, creating a trace of instruction streams. SBC also features an efficient on-line algorithm for compression of data address references. Unlike instruction addresses, data addresses for a memory referencing instruction rarely stay constant during program execution, but they can have a regular stride. The SBC-compressed data address trace encompasses a data address stride and the number of repetitions for each memory-referencing instruction in a stream. A change of the data address stride results in another record in the compressed trace. The records are ordered by the corresponding stream appearances in the original trace. The SBC algorithm achieves very good compression ratio and decompression time for instruction and data address traces, and can be successfully combined with general compression algorithms, such as Ziv-Lempel algorithm used in *gzip* utility. An instruction and data address in the SBC format encompasses three files: Stream Table File (STF), Stream-Based Instruction Trace (SBIT), and Stream-Based Data Trace (SBDT).

The SIGB techniques relate signature verification to the end of an instruction stream, so traces in the SBC format are very suitable for the SIGB simulators. Since the SIGB techniques protect basic blocks, we have written a program named *stream2bb* to generate an additional table with information about basic blocks.

The SBC Stream Table File has information only about executed instruction streams, which is sufficient to reconstruct the original address trace. However, the simulation of the secure installation process requires information about all basic blocks in the code, in the order to generate the signed code. We have modified a disassembling feature of SimpleScalar to generate the table of all basic blocks.

The secure installation process is simulated as follows. For the SIGBEV, basic block lengths are increased by the signature length, and the basic block starting addresses are recalculated to take into

account the embedded signatures. For the SIGBT techniques, we already have a table for all basic blocks in the code.

Figure 5.3 illustrates the implementation of the SIGB simulators. The simulator first reads input parameters from the command line: name of the trace, number of I-cache sets, and number of I-cache ways for both SIGBEV and SIGBTK, and also the number of S-cache sets and ways for the SIGBTK. The simulator reads information about the number of basic blocks in each instruction stream, and the starting addresses and lengths for each executed basic block.

```
// main program routine
main {
  get_input_arguments();
  read_sbc_table(); // read basic block and stream info
  read_trace();
  print_stats();
}

// read SBC traces and simulate I-cache accesses
read_trace() {

   while (more data in stream trace) {
      get Stream ID from stream trace;

      // for all basic blocks but the last, update cache
      for (i=1 to NumBB[StreamID] -1)
         instruction_address = BB[StreamID][i].start_address;

         for (j=1 to BB[StreamID][i]).length {
            // check if in cache, update cache if not
            is_cache_miss(instruction_address);
            instruction_address += instruction_length;
         }
      }

      // for the last basic block, call verify_signature on cache miss
      any_BB_miss = 0;
      instruction_address = BB[StreamID][last].start_address;
      for (j=1 to BB[StreamID][i]) {
         if ( is_cache_miss(instruction_address))
            any_BB_miss = 1;
         instruction_address += instruction_length;
      }
      if (any_BB_miss) verify_signature();
   } // end while
}
```

Figure 5.3 Pseudo-code for the trace-driven SIGB simulator

The SIGB simulator then reads stream identificators, StreamID, from the SBC trace file. Note than we need only the Stream-Based Instruction Trace, and not the Stream-Based Data Trace. For each basic block but the last, we just update the I-cache if necessary and count the cache misses. The function *is_cache_miss()* is used both to verify the I-cache hit/miss and to update the cache. The I-cache is implemented as a simple two-dimensional array, Icache[number of ways][number of sets], with the true LRU replacement policy. For the last basic block in an instruction stream, the program calls the procedure *verify_signature()* if any of the instructions in that cache block caused a cache miss. Finally, *print_stats()* prints the number of cache misses and signature verifications.

The main difference between the SIGBEV and the SIGBTK simulators is the function *verify_signature()* (Figure 5.4). In the SIGBEV simulator, this function just increments the number of signature verifications; the total number of verifications is printed in *print_stats()*. In the SIGBTK simulator, this function also verifies whether the signature of the currently executing basic block is in the S-cache, by calling the function *is_Scache_miss()*. The S-cache is implemented in the same way as the I-cache. In the case of an S-cache miss, the simulator counts the number of memory accesses needed to find the signature in the signature table section in memory, by calling the function *count_memory_accesses()*. In the case of a perfect hash mapping, this function always returns 1. We also experiment with the segmented binary search as described in Section 4.5.

```
// SIGBEV verify_signature procedure
verify_signature() {
   // count signature verifications
   ++signature_check;
}

// SIGBTK verify_signature procedure
verify_signature() {
   ++signature_check;

   // check if the signature is in the S-cache
   if (is_Scache_miss()) {
      ++signature_fetch;
      count_memory_accesses();
   }
}
```

Figure 5.4 Pseudo-code for the function verify_signature()

5.7 Simulator Parameters

In order to evaluate sensitivity of the proposed techniques to different system configurations, we vary several simulator parameters. In this section we specify the values of fixed simulator parameters and describe evaluated configurations.

5.7.1 SIGC Simulator Parameters

We have two sets of experiments for the SIGC techniques, one set with the simulator configured as an in-order embedded processor system such as XScale [127], and another set with a high-end, super-scalar, out-of-order processor configuration. The signature size in all experiments is 128 bits, i.e., 16 bytes. The D-cache (data cache) and I-cache have the same size and organization.

For the embedded processor configuration, we vary the following simulation parameters:

- The I-cache size (1, 2, 4, and 8KB);
- The I-cache line size (64 and 128 bytes);
- The width of a bus between memory and the I-cache (32 and 64 bits);
- The speed of processor core relative to memory (fast and slow).

The values of other simulator parameters for the embedded system configuration are shown in Table 5.3. We assume that the AES decryption latency with a 128-bit key is 12 processor cycles for slow, and 22 cycles for fast processor core, which are the speeds that can be attainable with current optimized ASIC solutions [138]. Since a signature is always fetched first, signature decryption is finished before the protected block is fetched, so the decryption latency is completely hidden in all evaluated system configurations. Translation latency is one cycle for the SIGCED and SIGCEK techniques, and one cycle on a mispredicted branch for the SIGCEV technique.

For the SIGC techniques with the S-cache, the S-cache has 8 ways, random cache replacement policy, and twice as many entries as the corresponding I-cache. Note that an S-cache line contains only one signature of 16 bytes, whereas an I-cache line contains 64 or 128 bytes. Hence, the size of an I-cache with *n* entries is approximately two or four times larger than the size of an S-cache with *2n* entries

(approximately because we do not take tag fields into account). We also experiment with the S-cache with the same number of entries as the I-cache, full associativity, and LRU replacement policy.

Table 5.3 Simulator parameters for the embedded processor configuration

Simulator parameter	Value
Branch predictor type	Bimodal
Branch predictor table size	128 entries, direct-mapped
Return address stack size	8 entries
Instruction decode bandwidth	1 instruction/cycle
Instruction issue bandwidth	1 instruction/cycle
Instruction commit bandwidth	1 instruction/cycle
Pipeline with in-order issue	True
I-cache/D-cache	4-way, FIFO replacement, first level only
I-TLB/D-TLB	32 entries, fully associative, FIFO replacement
Execution units	1 floating point, 1 integer
Memory fetch latency (first chunk/other chunks)	12/3 cycles for slow core, 24/6 cycles for fast core
Branch misprediction latency	2 cycles for slow core, 3 cycles for fast core
TLB latency	30 cycles for slow core, 60 cycles for fast core
Register update unit size	8
Load/store queue (LSQ) size	4

For the high-end processor configuration, we vary the following simulation parameters:

- The I-cache size (8, 16, and 32KB);
- The I-cache line size (64 and 128 bytes).

The values of other simulator parameters for the high-end configuration are shown in Table 5.4. In this configuration we also assume that the decryption latency is completely hidden, and translation address latency of the SIGCE is one processor cycle.

Table 5.4 Simulator parameters for the high-end processor configuration

Simulator parameter	**Value**
Branch predictor type	Bimodal
Branch predictor table size	512 sets, 4 way
Return address stack size	8 entries
Instruction decode bandwidth	4 instruction/cycle
Instruction issue bandwidth	4 instruction/cycle
Instruction commit bandwidth	4 instruction/cycle
Pipeline with in-order issue	False
I-cache/D-cache	4-way, LRU replacement, first level only
I-TLB/D-TLB	16 ways, 4 sets, LRU replacement
D-TLB	32 ways, 4 sets, LRU replacement
Execution units	4 floating point and 4 integer ALU's, 1 floating point and 1 integer multiplier
Memory fetch latency (first chunk/other chunks)	18/2 cycles
The width of a bus between memory and the I-cache	64 bits
Branch misprediction latency	3 cycles
TLB latency	30 cycles
Register update unit size	16
Load/store queue (LSQ) size	8

5.7.2 SIGB Simulator Parameters

For the SIGB experiments, we vary the I-cache size: 16, 32, and 64KB. The I-cache has 4 ways, 64B line size, and LRU replacement policy. The S-cache in the SIGBTK technique has the same number of ways as the I-cache and 2 ways, i.e., the I-cache has twice as many entries as the S-cache.

5.8 Benchmarks

For the SIGC experiments and embedded processor configurations, we use benchmarks from several benchmark suites for embedded systems: MiBench [139], MediaBench [140], and Basicrypt [141]. Table 5.5 lists the benchmarks and their short descriptions. Since the signature verification is done only at an I-cache miss, the benchmarks are selected so that most of them have a relatively high number of I-cache misses for at least some of the simulated cache sizes.

All benchmarks but *mpeg2encode* use the largest possible provided input. The *Mpeg2encode* benchmark uses the provided test input. Table 5.6 shows the total size of the original binary and the total size of the executable code sections in bytes, and the number of executed instructions. Since only the executable code sections are signed, the memory overhead of signature verification techniques depends on the size of these sections. All benchmarks are written in C language; all are compiled using provided makefiles and ARM gcc cross-compiler included in the SimpleScalar toolset with the *-static* option. This compiler includes all library functions in the code, and not only those invoked by the actual program.

The benchmarks *blowfish_dec* and *blowfish_enc* execute the same program, *blowfish*, for decoding and encoding. Hence, these two benchmarks have the same code size, but different number of executed instructions. Similarly, the benchmarks *rijndael_dec* and *rijndael_enc* execute the program *rijndael* for decoding and encoding.

For the SIGC experiments and high-end processor configurations, we use selected benchmarks from the SPEC CPU2000 benchmark set [137]. This benchmarks set consists of CPU-intensive applications that focus on integer or floating point calculations. Table 5.8 lists the SPEC CPU2000 benchmarks used for the SIGC experiments, along with the corresponding executable file/code section size

when benchmarks are compiled with ARM gcc cross-compiler and makefiles provided with the source code. In simulations, all benchmarks are run for the first one billion instructions.

For the SIGB experiments, we use SPEC CPU2000 traces of the first two billion executed instructions in the SBC format [135, 136]. Table 5.7 lists the traced benchmarks, the language of the source code, short description, and the size of Alpha precompiled binaries and code sections.

Table 5.5 Description of benchmarks from embedded domain

Benchmark	Suite	Description
blowfish_dec	MiBench	Blowfish decryption
blowfish_enc	MiBench	Blowfish encryption
cjpeg	MiBench	JPEG compression
djpeg	MiBench	JPEG decompression
ecdhb	Basicrypt	Diffie-Hellman key exchange
ecdsignb	Basicrypt	Digital signature generation
ecdsverb	Basicrypt	Digital signature verification
ecelgdecb	Basicrypt	El-Gamal encryption
ecelgencb	Basicrypt	El-Gamal decryption
ispell	MiBench	Spell checker
mpeg2_enc	MediaBench	MPEG2 compression
qsort	MiBench	Quicksort
rijndael_dec	MiBench	Rijndael decryption
rijndael_enc	MiBench	Rijndael encryption
stringsearch	MiBench	String search

Table 5.6 Benchmark code size and executed instructions for embedded systems

Benchmark	Executable file size in bytes total (code section)	Executed instructions in millions
blowfish_dec	1,032,731 (190,900)	544.0
blowfish_enc	1,032,731 (190,900)	544.0
cjpeg	1,261,485 (298,916)	104.6
djpeg	1,274,670 (311,108)	23.4
ecdhb	1,102,298 (258,188)	122.5
ecdsignb	1,254,373 (310,068)	131.3
ecdsverb	1,254,519 (310,212)	171.9
ecelgdecb	1,102,207 (258,092)	92.4
ecelgencb	1,102,271 (258,156)	180.2
ispell	1,238,144 (240,972)	817.7
mpeg2_enc	1,318,326 (317,504)	127.5
qsort	1,180,697 (252,284)	737.9
rijndael_dec	1,045,273 (199,364)	307.9
rijndael_enc	1,045,273 (199,364)	320.0
stringsearch	1,025,446 (188,484)	3.7

Table 5.7 Description of SPEC CPU2000 benchmarks and the size of precompiled Alpha binaries

Benchmark		Language	Description	File size total (code section)
Integer	164.gzip	C	Compression	376832 (212992)
	176.gcc	C	C Programming Language Compiler	3792896 (1990656)
	181.mcf	C	Combinatorial Optimization	303104 (163840)
	186.crafty	C	Game Playing: Chess	942080 (442368)
	197.parser	C	Word Processing	598016 (319488)
	252.eon	C++	Computer Visualization	1187840 (794624)
	253.perlbmk	C	PERL Programming Language	2154496 (876544)
	254.gap	C	Group Theory, Interpreter	1458176 (933888)
	255.vortex	C	Object-oriented Database	2310144 (819200)
	300.twolf	C	Place and Route Simulator	917504 (450560)
Floating point	168.wupwise	Fortran 77	Physics / Quantum Chromodynamics	1114112 (819200)
	171.swim	Fortran 77	Shallow Water Modeling	1105920 (819200)
	172.mgrid	Fortran 77	Multi-grid Solver: 3D Potential Field	1089536 (802816)
	177.mesa	C	3-D Graphics Library	2875392 (917504)
	178.galgel	Fortran 90	Computational Fluid Dynamics	1458176 (1048576)
	179.art	C	Image Recognition / Neural Networks	368640 (237568)
	183.equake	C	Seismic Wave Propagation Simulation	385024 (253952)
	188.ammp	C	Computational Chemistry	655360 (385024)
	189.lucas	Fortran 90	Number Theory / Primality Testing	1146880 (851968)
	191.fma3d	Fortran 90	Finite-element Crash Simulation	4505600 (1867776)
	200.sixtrack	Fortran 77	Nuclear Physics Accelerator Design	6348800 (2596864)
	301.appsi	Fortran 77	Meteorology: Pollutant Distribution	1531904 (1114112)

Table 5.8 The size of SPEC CPU2000 benchmarks when compiled with the ARM gcc compiler

Benchmark	**Executable file size total (code section)**
164.gzip	1285947 (294924)
176.gcc	2825380 (1562516)
181.mcf	1160438 (248848)
s197.parser	1323097 (349888)
177.mesa	1799804 (765748)
179.art	1223820 (264688)
183.equake	1238847 (269568)
188.ammp	1470400 (390656)

CHAPTER 6

EVALUATION RESULTS

"You can observe a lot just by watching."
Yogi Berra

In this chapter we present and discuss results of evaluation of the proposed techniques for instruction block verification. We are primarily interested in performance overhead relative to the Base system without the signature verification mechanism. We also evaluate memory overhead, i.e., the length of a signed executable file versus the length of the corresponding original executable.

6.1 SIGC Evaluation

We evaluate performance overhead of the SIGC techniques by comparing the number of processor clock cycles per instruction (CPI) with signature verification to the corresponding number with the Base system. To gain insight into reasons of performance overhead, we also measure the number of S-cache misses for the SIGCEK and SIGCTK, and the increase of I-cache misses for the SIGCEV technique.

6.1.1 Base System

Table 6.1 shows the number of I-cache misses per 1000 executed instructions for the Base system, for benchmarks from the embedded domain. Most of these benchmarks benefit from having larger 128B cache lines, especially with small I-caches. However, for the largest considered 8K I-cache, more than half of benchmarks have less cache misses with 64B I-cache lines. The *ispell*, *qsort*, *rijndael_enc*, *rijndael_dec*,

and *stringsearch* are the only benchmarks having more than 10 I-cache misses per 1000 instructions with caches larger than 1K. In addition, other benchmarks have less than 1 I-cache miss per 1000 instructions with the 8K I-cache.

The values of CPI for the base system are shown in Table 6.2, Table 6.3, Table 6.4, and Table 6.5, for slow processor core/32-bit memory bus, slow core/64-bit bus, fast core/32-bit bus, and fast core/64-bit bus, respectively. A configuration with 64B I-cache lines always outperforms the corresponding 128B line configuration. Note that it takes more processor cycles to fetch a 128B cache line than a 64B one. Hence, even a smaller number of I-cache misses with 128B cache lines does not necessarily mean faster execution, i.e., a lower CPI. The wider 64-bit memory reduces the cache miss penalty of the 32-bit bus; therefore, it reduces the CPI. On the other hand, a fast processor core has to wait longer for completion of a fetch from memory than a slow one, which results in a higher CPI. If both fast and slow processor systems have the same memory speed, the cycle time of the fast processor is shorter, so the higher CPI does not have to indicate a longer execution time. Detailed analysis of relationships between common architectural parameters and CPI is out of the scope of this dissertation; we need evaluation results for the base system primarily as a reference point for signature verification techniques.

Table 6.6 shows the number of I-cache misses per 1000 instructions for selected SPEC CPU 2000 benchmarks and high-end processor configurations, and Table 6.7 shows the corresponding CPI.

Table 6.1 Base: I-cache misses per 1000 instructions in embedded processor configurations

Benchmarks	**I-cache misses per 1000 instructions**							
	64B cache line				**128B cache line**			
	1K	2K	4K	8K	1K	2K	4K	8K
blowfish_dec	22.19	5.59	0.08	0.00	13.65	3.78	0.80	0.01
blowfish_enc	22.19	4.56	0.09	0.00	12.90	3.78	0.80	0.01
cjpeg	6.16	1.60	0.31	0.09	6.60	1.65	0.27	0.08
djpeg	8.43	3.99	1.10	0.24	6.17	2.94	0.97	0.24
ecdhb	20.31	5.97	2.26	0.13	14.57	6.20	1.63	0.16
ecdsignb	15.92	4.61	1.74	0.07	17.33	4.82	1.25	0.11
ecdsverb	21.31	5.21	2.03	0.29	16.88	5.35	1.46	0.29
ecelgdecb	26.16	0.34	0.03	0.01	22.44	2.50	0.04	0.01
ecelgencb	23.41	3.21	1.15	0.06	18.71	4.37	0.84	0.10
ispell	61.67	51.07	21.66	2.86	40.35	35.75	20.94	3.50
mpeg2_enc	1.83	0.79	0.33	0.16	2.12	0.59	0.27	0.12
qsort	44.23	29.44	22.19	5.45	32.76	21.09	15.27	7.41
rijndael_dec	70.62	68.64	67.96	6.63	41.59	40.26	37.61	9.92
rijndael_enc	73.70	70.52	67.96	8.12	42.58	39.40	38.10	11.19
stringsearch	55.32	35.42	12.89	3.70	37.95	24.34	10.63	1.92

Table 6.2 Base: CPI in embedded processor configurations, slow core, memory bus 32 bits

Benchmarks	**CPI**							
	64B cache line				**128B cache line**			
	1K	2K	4K	8K	1K	2K	4K	8K
blowfish_dec	4.26	3.56	2.64	2.30	5.49	4.55	3.03	2.30
blowfish_enc	4.26	3.53	2.64	2.30	5.42	4.55	3.03	2.30
cjpeg	3.02	2.48	2.23	1.58	5.21	3.64	2.99	1.81
djpeg	3.83	2.81	2.24	1.71	8.39	5.17	3.41	1.86
ecdhb	2.75	1.93	1.72	1.61	3.28	2.28	1.77	1.62
ecdsignb	2.58	1.94	1.77	1.68	3.36	2.21	1.81	1.70
ecdsverb	2.76	1.96	1.78	1.69	3.37	2.26	1.83	1.71
ecelgdecb	3.13	1.80	1.78	1.78	4.20	1.99	1.78	1.78
ecelgencb	2.95	1.87	1.75	1.69	3.80	2.14	1.77	1.70
ispell	6.31	5.48	3.34	2.16	9.57	7.93	5.48	2.65
mpeg2_enc	2.38	1.93	1.60	1.49	3.39	2.42	1.75	1.52
qsort	4.02	3.14	2.71	1.77	5.59	3.97	3.16	2.31
rijndael_dec	8.87	7.97	6.83	2.71	15.65	12.88	9.06	4.47
rijndael_enc	8.87	7.88	6.64	2.70	15.51	12.51	8.89	4.41
stringsearch	5.15	3.85	2.52	1.92	7.95	5.57	3.18	2.00

Table 6.3 Base: CPI in embedded processor configurations, slow core, memory bus 64 bits

Benchmarks	**CPI**							
	64B cache line				**128B cache line**			
	1K	2K	4K	8K	1K	2K	4K	8K
blowfish_dec	3.37	2.98	2.47	2.30	3.98	3.47	2.67	2.30
blowfish_enc	3.37	2.96	2.48	2.30	3.94	3.47	2.67	2.30
cjpeg	2.34	2.03	1.89	1.52	3.46	2.62	2.28	1.64
djpeg	2.82	2.25	1.93	1.64	5.17	3.47	2.54	1.71
ecdhb	2.23	1.78	1.67	1.60	2.48	1.96	1.69	1.61
ecdsignb	2.17	1.82	1.73	1.68	2.56	1.96	1.75	1.69
ecdsverb	2.27	1.83	1.73	1.68	2.56	1.98	1.75	1.69
ecelgdecb	2.52	1.79	1.78	1.78	3.05	1.89	1.78	1.78
ecelgencb	2.38	1.79	1.72	1.69	2.80	1.92	1.73	1.69
ispell	4.36	3.90	2.71	2.05	5.97	5.10	3.81	2.31
mpeg2_enc	1.96	1.71	1.53	1.48	2.48	1.97	1.61	1.49
qsort	2.89	2.41	2.17	1.63	3.66	2.80	2.37	1.92
rijndael_dec	5.63	5.12	4.50	2.25	9.04	7.57	5.56	3.15
rijndael_enc	5.66	5.10	4.41	2.26	9.00	7.40	5.49	3.14
stringsearch	3.62	2.89	2.15	1.82	5.01	3.74	2.48	1.85

Table 6.4 Base: CPI in embedded processor configurations, fast core, memory bus 32 bits

Benchmarks	**CPI**							
	64B cache line				**128B cache line**			
	1K	**2K**	**4K**	**8K**	**1K**	**2K**	**4K**	**8K**
blowfish_dec	6.36	4.94	3.02	2.31	8.81	6.93	3.81	2.32
blowfish_enc	6.36	4.88	3.03	2.31	8.66	6.93	3.81	2.32
cjpeg	4.66	3.55	3.03	1.72	9.04	5.88	4.56	2.18
djpeg	6.23	4.14	2.97	1.91	15.43	8.89	5.34	2.20
ecdhb	4.00	2.30	1.87	1.63	5.06	3.00	1.97	1.66
ecdsignb	3.56	2.23	1.88	1.70	4.89	2.76	1.96	1.72
ecdsverb	3.95	2.29	1.91	1.72	4.99	2.88	2.01	1.76
ecelgdecb	4.63	1.84	1.80	1.80	6.75	2.25	1.81	1.80
ecelgencb	4.33	2.08	1.83	1.71	6.02	2.63	1.89	1.73
ispell	11.00	9.29	4.87	2.46	17.54	14.19	9.23	3.46
mpeg2_enc	3.36	2.44	1.75	1.53	5.39	3.43	2.06	1.59
qsort	6.73	4.93	4.04	2.13	9.86	6.58	4.93	3.22
rijndael_dec	16.58	14.74	12.39	3.84	30.13	24.52	16.74	7.39
rijndael_enc	16.52	14.50	11.96	3.76	29.78	23.69	16.34	7.21
stringsearch	8.82	6.17	3.44	2.21	14.46	9.59	4.76	2.35

Table 6.5 Base: CPI in embedded processor configurations, fast core, memory bus 64 bits

Benchmarks	CPI							
	64B cache line				128B cache line			
	1K	2K	4K	8K	1K	2K	4K	8K
blowfish_dec	4.59	3.78	2.70	2.30	5.78	4.76	3.10	2.31
blowfish_enc	4.59	3.75	2.70	2.30	5.70	4.76	3.10	2.31
cjpeg	3.29	2.66	2.36	1.61	5.55	3.84	3.13	1.85
djpeg	4.22	3.02	2.36	1.75	9.00	5.49	3.58	1.89
ecdhb	2.96	2.00	1.75	1.62	3.45	2.35	1.80	1.64
ecdsignb	2.75	2.00	1.80	1.69	3.52	2.26	1.83	1.71
ecdsverb	2.96	2.03	1.81	1.70	3.54	2.32	1.86	1.73
ecelgdecb	3.39	1.82	1.80	1.80	4.45	2.04	1.81	1.80
ecelgencb	3.18	1.92	1.78	1.71	4.01	2.20	1.80	1.72
ispell	7.09	6.12	3.61	2.24	10.30	8.50	5.84	2.76
mpeg2_enc	2.54	2.01	1.63	1.51	3.57	2.52	1.78	1.53
qsort	4.48	3.46	2.96	1.86	5.99	4.23	3.35	2.43
rijndael_dec	10.10	9.05	7.72	2.90	16.91	13.89	9.73	4.74
rijndael_enc	10.09	8.94	7.49	2.88	16.75	13.48	9.54	4.67
stringsearch	5.76	4.24	2.69	1.99	8.54	5.94	3.35	2.06

Table 6.6 Base: I-cache misses per 1M executed instructions in high-end processor configurations

Benchmarks	I-cache misses per 1M instructions					
	64B cache line			128B cache line		
	8K	16K	32K	8K	16K	32K
164.gzip	6.60	0.95	0.43	5.15	0.82	0.26
176.gcc	36437.27	21440.29	5981.83	27171.32	18723.99	7074.12
177.mesa	23742.87	3148.07	9.62	19754.18	4822.73	668.02
179.art	0.26	0.25	0.25	0.17	0.15	0.15
181.mcf	15086.39	94.25	0.48	15387.58	859.36	778.61
183.equake	15813.20	3085.84	534.49	26766.70	3982.24	949.73
188.ammp	7674.31	1133.98	3.35	6429.59	1144.26	4.45
197.parser	707.16	321.44	7.34	656.77	232.69	6.13

Table 6.7 Base: CPI in high-end processor configurations

Benchmarks	CPI					
	64B cache line			128B cache line		
	8K	16K	32K	8K	16K	32K
164.gzip	1.00	0.97	0.92	1.20	1.13	1.06
176.gcc	2.01	1.50	1.00	2.36	1.82	1.18
177.mesa	1.37	0.76	0.67	1.60	0.92	0.70
179.art	1.12	1.12	1.12	1.03	1.03	1.03
181.mcf	1.10	0.66	0.65	1.38	0.74	0.69
183.equake	1.12	0.73	0.66	1.99	0.84	0.69
188.ammp	2.21	2.01	1.98	3.05	2.81	2.76
197.parser	0.73	0.72	0.71	0.75	0.72	0.71

6.1.2 Performance Overhead

Table 6.8 and Table 6.9 show the number of I-cache misses with the SIGCEV technique. Since the SIGCEV I-cache is smaller, for most benchmarks it has more cache misses than the Base I-cache. Table 6.10 and Table 6.11 show the number of S-cache misses with SIGCEK and SIGCTK techniques. Let us say again that the considered S-cache has the same number of sets and twice as many ways (eight) as the corresponding I-cache, and one S-cache line can store one instruction block signature. The total size of the S-cache is approximately ¼ of the I-cache size for I-caches with 128B lines, and approximately ½ of the I-cache size with 64B lines. The S-cache is accessed on an I-cache miss to retrieve the signature. Hence, the number of S-cache misses decreases faster than the number of I-cache misses, with the I-cache size increase.

Let us first consider the performance of the SIGC techniques with 32-bit bus, slow processor core, and 128B I-cache lines, with benchmarks from the embedded domain (Figure 6.1). The results indicate a low performance overhead of the SIGCED technique. Even with the very small 1K I-cache, this technique increases CPI in the range 0.8-7.4%, with 8 out of 15 benchmarks having more than 5% increase. With the 4K I-cache, CPI increases for more than 5% for only 3 benchmarks, since the influence of signature verification overhead is reduced with I-cache miss reduction. With the largest considered I-cache (8K), the maximum SIGCED CPI increase is 3.8%, and only 5 benchmarks have more than 1% increase.

The absolute CPI increase for the SIGCED technique depends on the number of I-cache misses given in Table 6.1: more cache misses means more signature verifications, that is, increased performance overhead. However, the ratio of CPI for SIGCED and Base does not absolutely follow the trend of the number of I-cache misses, since for an application with a relatively large number of I-cache misses a relative CPI increase may be smaller than for an application with fewer cache misses. For example, with the 1K I-cache *rijndael_enc* has a 3% CPI increase and *ecdhb* has a 5.8% increase, whereas *rijndae_enc* has 42.58 I-cache misses per 1000 instructions, and *ecdhb* only 14.57. This can be easily explained by the fact that the Base CPI for this system configuration is 15.51 for *rijndael_enc* and 3.28 for *ecdhb* (Table 6.2), and the absolute CPI increase with the SIGCED technique is 0.19 for *ecdhb* and 0.47 for *rijndael_enc*.

As explained in Chapter 4, the SIGCED overhead can be reduced if signatures are kept in the S-cache, i.e., with the SIGCEK technique. The SIGCEK CPI increase is in the range 0.3-5% with the 1K, 0.1-4.4% with the 2K, 0.01-2% with 4K, and 0-0.4% with the 8K I-cache. The SIGCEK reduces the performance overhead of the SIGCED for 6.5-83.5%, 24.7-91.2%, and 58.3-90.8%, with the 1K, 2K, and 4K I-cache, respectively. With the 8K I-cache, the low number of I-cache misses enables the SIGCEK to virtually remove performance overhead of signature verification.

The SIGCEV protected block size in these experiments is 112B, so the actual I-cache size is 0.875 of the Base I-cache size. The large SIGCEV performance overhead of 14% for *mpeg2_enc* with 1K I-cache, 26% for *ecelgdecb* with the 2K I-cache, 33% for *stringsearch* with the 4K I-cache, and 24% for *rijndael_enc* is due to the significant relative increase in the number of cache misses (Table 6.9).

However, the SIGCEV may have even a lower CPI than the Base case. The SIGCEV I-cache has a different mapping function, so the number of I-cache misses may be lower. If such a benchmark also has a relatively low branch misprediction rate, such that performance overhead due to the SIGCEV address translation is negligible, the SIGCEV might marginally outperform the Base case. This is the case with the *ecdsignb* and *ecdsverb* benchmarks with the 1K I-cache, *rijndael_dec* with the 2K and 8K, and *blowfish* with the 4K I-cache.

Somewhat surprisingly, the SIGCEV technique outperforms the SIGCED for 11 out of 15 benchmarks with the 1K I-cache. This is due to the difference in the instruction block address translation. With the SIGCEV, the address translation overhead is added to the branch misprediction penalty, and with the SIGCED, it is added to the I-cache miss penalty. Hence, the SIGCEV total overhead might be smaller, especially with small caches with more capacity misses.

The SIGCTD technique always introduces more performance overhead than the SIGCED, since signatures stored in the separate code section require an additional memory access. However, this difference is more significant with small caches: the ratio between the CPI for SIGCTD and Base is 1.013-1.119 with the 1K I-cache, and 1.0001-1.066 with the 8K I-cache. As with the SIGCED, the S-cache is able to significantly reduce performance overhead. For example, with the 1K I-cache, the SIGCTK reduces the overhead of the SIGCTD 7.2-90.6%. Note that the S-cache is more beneficial to the SIGCTD than to the SIGCED technique. This effect is also due to the longer signature fetch latency with the SIGCTD.

Overall, the SIGCTK technique is a good alternative to SIGCEK, especially since the SIGCTK does not require the translation address mechanism on each I-cache miss.

Let us now consider the sensitivity of SIGC techniques to the processor core speed, memory bus width, and size of protected instruction block, i.e., the I-cache line size (Figure 6.2, Figure 6.3, Figure 6.4, Figure 6.5, Figure 6.6, Figure 6.7, and Figure 6.8). The number of processor clock cycles needed for signature fetch will decrease with the wider data memory bus, and increase with the faster processor core, so we may expect similar behavior from total signature verification overhead. Another interesting architectural parameter is the cache line size. Without simulation, it is hard to predict the sensitivity of the SIGCE techniques to this parameter. For example, with 64B I-cache lines, 32-bit bus and slow core, the signature fetch for the SIGCED technique increases the number of cycles required for an instruction block fetch by 21.05%; the corresponding increase with 128B cache lines is 11.43%. On the other hand, the Base configuration with 64B cache has lower CPI (Table 6.2, Table 6.3, Table 6.4, and Table 6.5).

We may group the benchmarks in two groups, according to the number of cache misses with all considered cache sizes. The influence of the bus width, the core speed, and the cache line size will be discussed for one benchmark from each group: *ecdhb* with a relatively low number of cache misses, and *rijndael_enc* which is one of the two benchmarks with the largest number of cache misses per 1000 instructions for each cache size and line size (Table 6.1).

The SIGCED technique has the largest impact on performance with the 64B cache line size, the 32-bit bus, and a fast processor core. However, even with this system configuration the SIGCED performance overhead is never more than 13% for both benchmarks, with relatively small variations between configurations with a fixed I-cache size. For *ecdhb*, the largest variation is for the 1K I-cache with 64B lines, from 6% overhead with the 64-bit bus and slow core, to 12% overhead with the 32-bit bus and fast core. For *rijndael_dec*, the largest variation is also with 64B I-cache lines, but with the 4K I-cache: from 10% with the 64-bit bus and slow core, to 13% overhead with the 32-bit bus and fast core. Overall, the SIGCED technique has more performance overhead with 64B I-cache lines than with 128B, as well as more sensitivity to the memory bus width and processor speed. With 128B I-cache lines, the largest overhead variation range is 4-7% for *ecdhb*, and 2.3-3.5% for *rijndael_enc*. Clearly, if the number of I-

cache misses is very low, as it is for *ecdhb* in the 8K I-cache, the SIGCED overhead does not depend on system parameters, since it is always close to zero.

It is interesting to note that the ratio of SIGCED CPI and the Base CPI decreases with larger caches for *ecdhb* and not for *rijndael_enc*. The *rijndael_enc* benchmark has a very large number of I-cache capacity misses in 1, 2, and 4K caches, such that the number of I-cache misses is only slightly reduced with the cache size increase before the 8K size. Hence, the absolute overhead of the SIGCED technique does not considerably decrease with the cache size increase. However, for *rijndael_enc* even a relatively small reduction in the number of cache misses significantly improves the Base CPI, so the CPI ratio for the SIGCED actually grows, up to the 4K cache size.

Since the S-cache eliminates a lot of signature verification overhead, the SIGCEK technique is much less sensitive to configuration parameters. For *ecdhb* and a fixed I-cache size, the SIGCEK performance overhead is almost constant. For *rijndael_enc*, the largest overhead variation range is 7-9%.

The SIGCEV has more I-cache misses than the Base case for both *ecdhb* and *rijndael_enc* (Table 6.1, Table 6.8), so it always has lower performance. For both benchmarks the SIGCEV is more sensitive to configuration change than the SIGCED, since a narrower bus and a faster core increase both the cache miss latency and the latency due to signature fetch. For *ecdhb*, the largest variation is for the 2K I-cache with 64B lines, from 12% overhead with the 64-bit bus and slow core, to 36% overhead with the 32-bit bus and fast core. For *rijndael_dec*, the largest variation is with 64B I-cache lines and the 8K I-cache: from 43% with the 64-bit bus and slow core, to 98% overhead with the 32-bit bus and fast core. The SIGCEV with shorter cache lines has unacceptable performance overhead, so it is not suitable for systems with 64B cache lines. We can explain this overhead by looking again at I-cache misses and CPI for the Base system: For both *ecdhb* and *rijndael_enc*, the number of I-cache misses is significantly larger with 64B cache lines, whereas the CPI is lower. Hence, the increase in the number of I-cache misses due to the SIGCEV is more detrimental to performance with 64B cache lines.

The *rijndael_enc* benchmark has a very large SIGCEV performance overhead with the 8K I-cache. This happens because the number of I-cache misses has doubled compared to the Base case. Moreover, there is a sharp drop in the Base CPI for 8K I-cache, so the relative performance overhead increase is even more noticeable.

The SIGCTD technique in configurations with 128B I-cache lines is only moderately sensitive to the memory bus width and core speed variation: The largest overhead variation range is 5-9% for *ecdhb*, and 8-11% for *rijndael_enc*. However, in configurations with 64B I-cache lines, the SIGCTD performance overhead increases significantly, up to 21% for *ecdhb*, and 25% for *rijndael_enc.* This is also due to the large number of I-cache misses, and consequently, large overhead of additional memory accesses.

The SIGCTK technique successfully eliminates a large portion of SIGCTD overhead. However, due to larger penalty for signature verification, this technique is more sensitive to changes of architectural parameters than the SIGCEK, especially with 64B I-cache lines. The largest overhead variation range is with 64B I-cache lines and 1K I-cache: 4-7% for *ecdhb*, and 13-17% for *rijndael_enc*.

For embedded systems with 128B cache lines, we may conclude the following. If such system has a low hardware budget, and all programs executing in protected mode, the SIGCEV technique has the best price-performance tradeoff, since in small caches it outperforms the SIGCED for most benchmarks and employs less hardware resources. However, the SIGCED is better for systems with larger caches. With 25% larger hardware budget invested in the S-cache, the SIGCEK technique has a very low performance overhead across all considered system configurations. With larger caches we may also use the techniques with signature stored in the separate code section.

For systems with 64B cache lines, we recommend the use of the SIGCED or SIGCEK only, depending on the available hardware resources. The increase in the number of I-cache misses becomes a prohibiting factor in the use of the SIGCEV and SIGCTD techniques; even the SIGCTK may have relatively significant performance overhead.

We also evaluate the SIGCE techniques in high-end processor configurations with out-of-order execution (Figure 6.9 and Figure 6.10). The SIGCED has very low performance overhead, from nearly 0 to 9%. With the SIGCEK, the worst-case overhead is reduced to 5%. With 128B I-cache lines, the SIGCEV overhead is up to 14%; as in the embedded domain, some applications benefit from the SIGCEV cache mapping function and have lower CPI than with the Base system. However, the significant increase in the number of I-cache misses with 64B lines (Table 6.6) results in SIGCEV overhead up to more than 70%. The results indicate that the SIGCTD technique is also not suitable for 64B high-end configurations, with the overhead of up to 35%. With the SIGCTK, the worst-case overhead is almost halved to 18%.

Table 6.8 SIGCEV: I-cache misses per 1000 executed instructions in embedded processor configurations

Benchmarks	I-cache misses per 1000 instructions							
	64B cache line				128B cache line			
	1K	2K	4K	8K	1K	2K	4K	8K
blowfish_dec	29.23	16.66	0.15	0.02	14.85	9.76	0.81	0.01
blowfish_enc	28.49	14.41	3.09	0.05	15.00	6.92	0.82	0.01
cjpeg	10.76	3.29	0.42	0.14	8.76	2.14	0.29	0.07
djpeg	21.52	6.95	2.59	0.32	7.02	3.53	1.48	0.24
ecdhb	30.73	13.70	4.93	0.51	17.21	8.65	2.21	0.21
ecdsignb	24.25	10.75	3.79	0.38	13.53	6.66	1.70	0.15
ecdsverb	25.95	11.49	4.20	0.65	14.55	7.17	1.91	0.33
ecelgdecb	40.53	9.13	0.23	0.02	23.71	6.95	0.07	0.01
ecelgencb	35.76	11.50	2.59	0.26	20.48	7.82	1.15	0.11
ispell	76.15	65.79	32.80	6.88	48.09	42.89	23.29	5.91
mpeg2_enc	8.66	1.31	0.57	0.25	6.86	0.79	0.32	0.14
qsort	52.37	38.86	28.97	13.51	31.67	25.09	18.01	9.72
rijndael_dec	89.74	85.78	85.77	33.78	44.23	41.59	40.26	9.97
rijndael_enc	88.94	87.03	86.37	45.58	47.03	43.85	41.30	22.94
stringsearch	69.04	43.63	19.51	0.27	44.44	32.86	20.81	5.46

Table 6.9 SIGCEV: I-cache misses per 1M executed instructions in high-end processor configurations

Benchmarks	**I-cache misses per 1M instructions**					
	64B cache line			**128B cache line**		
	8K	**16K**	**32K**	**8K**	**16K**	**32K**
164.gzip	13.86	1.37	0.53	5.87	0.60	0.28
176.gcc	47021.24	30973.39	11149.76	30133.89	21066.56	7835.23
177.mesa	34994.13	7566.03	416.88	22750.82	5407.08	1219.94
179.art	0.35	0.32	0.31	0.18	0.16	0.16
181.mcf	27247.82	1073.50	4.02	13878.65	2993.24	0.32
183.equake	44478.80	10602.37	848.80	27442.49	3978.82	408.98
188.ammp	13671.61	3982.53	139.25	7878.14	3861.29	1409.66
197.parser	1141.27	596.88	65.03	943.63	274.62	58.13

Table 6.10 S-cache misses per 1000 executed instructions in high-end processor configurations

Benchmarks	**S-cache misses per 1000 instructions**					
	64B cache line			**128B cache line**		
	8K	**16K**	**32K**	**8K**	**16K**	**32K**
164.gzip	0.51	0.43	0.43	0.33	0.27	0.26
176.gcc	18692.02	5478.01	891.82	16307.49	6019.75	961.11
177.mesa	850.19	2.93	1.75	964.94	3.21	1.07
179.art	0.25	0.25	0.25	0.15	0.15	0.15
181.mcf	0.69	0.50	0.47	9.57	0.30	0.27
183.equake	1517.90	0.49	0.41	2398.67	0.37	0.26
188.ammp	631.48	0.62	0.54	947.07	2.12	0.33
197.parser	166.46	5.99	2.82	152.18	4.06	1.75

Table 6.11 S-cache misses per 1000 executed instructions in embedded processor configurations

Benchmarks	S-cache misses per 1000 instructions							
	64B cache line				128B cache line			
	1K	2K	4K	8K	1K	2K	4K	8K
blowfish_dec	4.56	0.05	0.00	0.00	4.79	0.07	0.00	0.00
blowfish_enc	4.56	0.05	0.00	0.00	4.23	0.07	0.00	0.00
cjpeg	1.06	0.21	0.05	0.01	1.24	0.19	0.06	0.01
djpeg	3.70	0.81	0.16	0.05	2.90	0.78	0.16	0.03
ecdhb	6.55	1.70	0.09	0.02	4.76	1.45	0.09	0.01
ecdsignb	5.02	1.30	0.05	0.01	3.67	1.10	0.06	0.01
ecdsverb	5.52	1.62	0.19	0.01	4.01	1.30	0.20	0.01
ecelgdecb	0.24	0.02	0.01	0.01	2.21	0.02	0.01	0.01
ecelgencb	3.43	0.87	0.04	0.01	3.56	0.74	0.05	0.01
ispell	39.65	7.88	2.33	0.31	30.61	9.42	2.51	0.44
mpeg2_enc	0.65	0.28	0.12	0.05	0.50	0.23	0.09	0.04
qsort	29.65	16.06	1.43	0.00	21.48	13.26	4.03	0.00
rijndael_dec	66.49	45.14	0.05	0.00	38.82	28.67	1.40	0.00
rijndael_enc	68.48	47.82	2.73	0.00	39.34	29.27	3.97	0.00
stringsearch	31.59	5.89	0.09	0.07	24.24	6.51	0.06	0.05

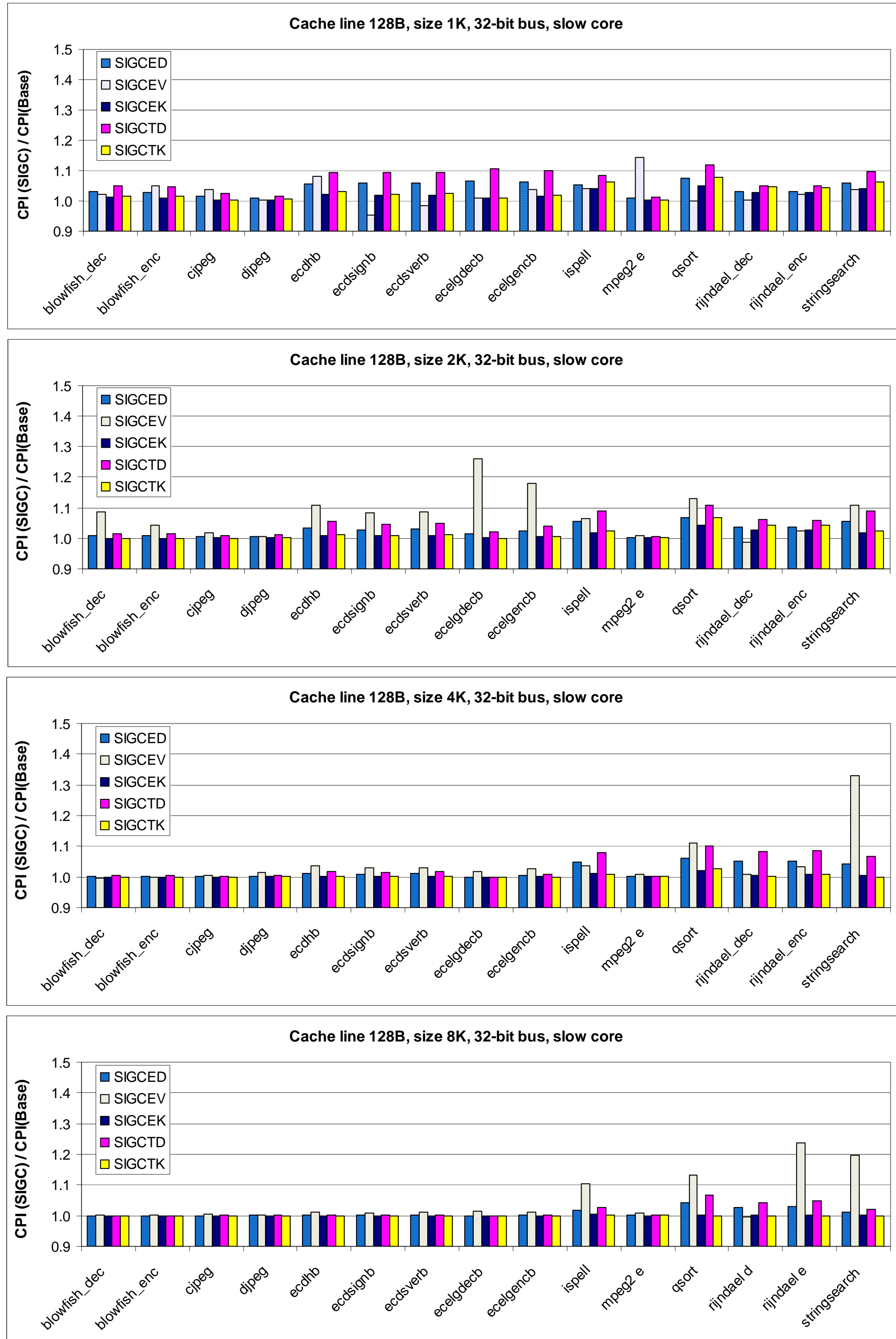

Figure 6.1 SIGC: embedded processor configuration, I-cache line 128B, 32-bit bus, slow core

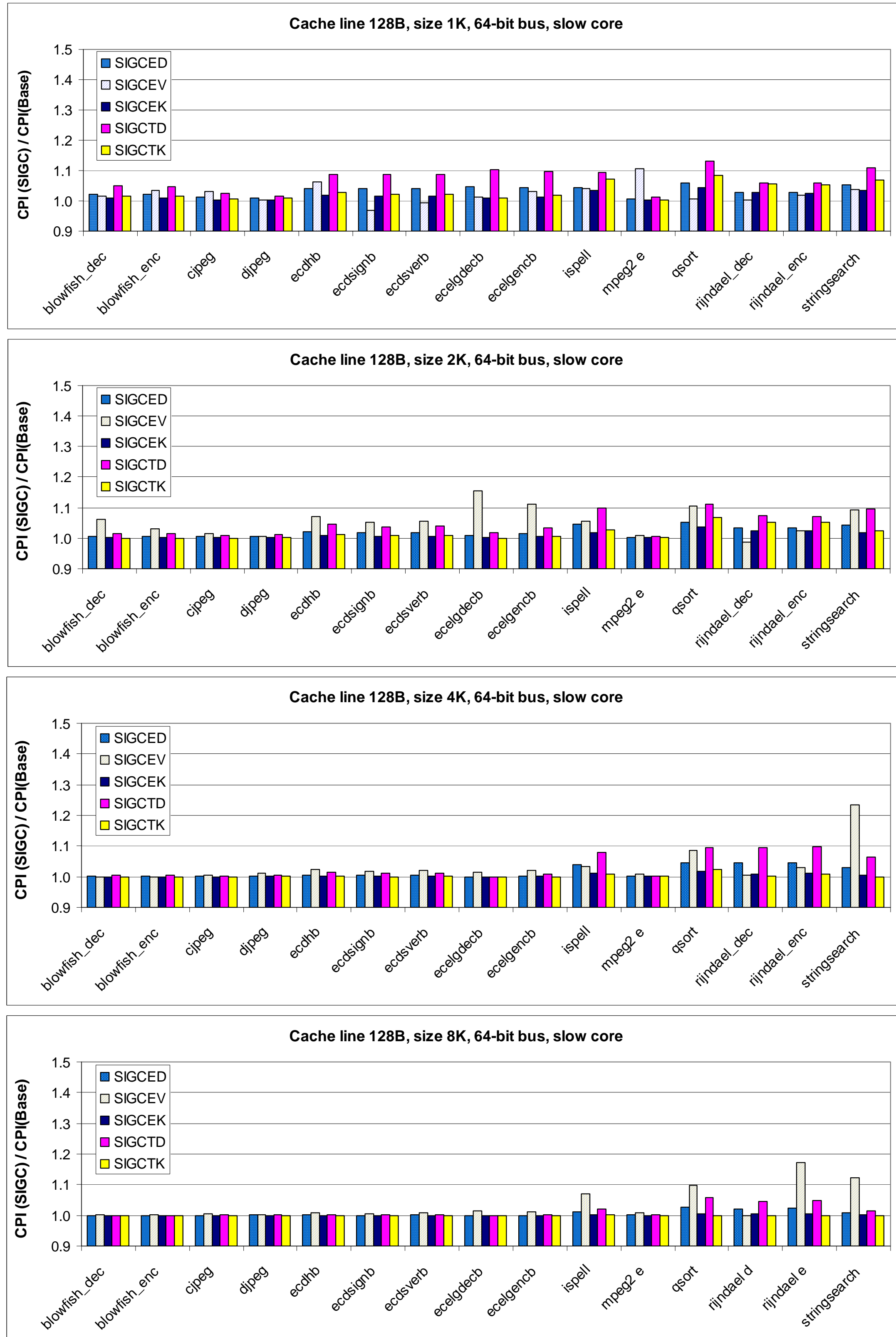

Figure 6.2 SIGC: embedded processor configuration, I-cache line 128B, 64-bit bus, slow core

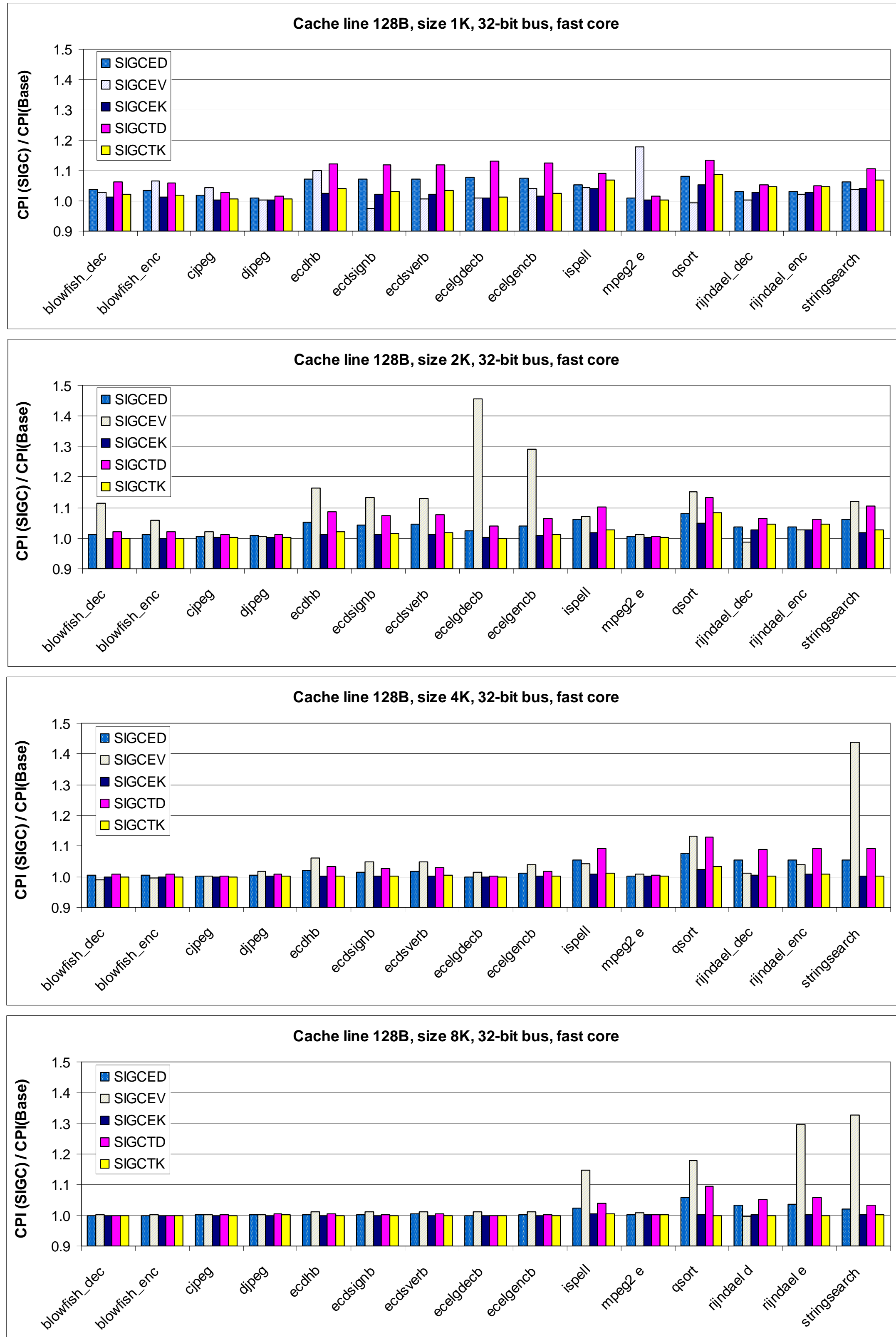

Figure 6.3 SIGC: embedded processor configuration, I-cache line 128B, 32-bit bus, fast core

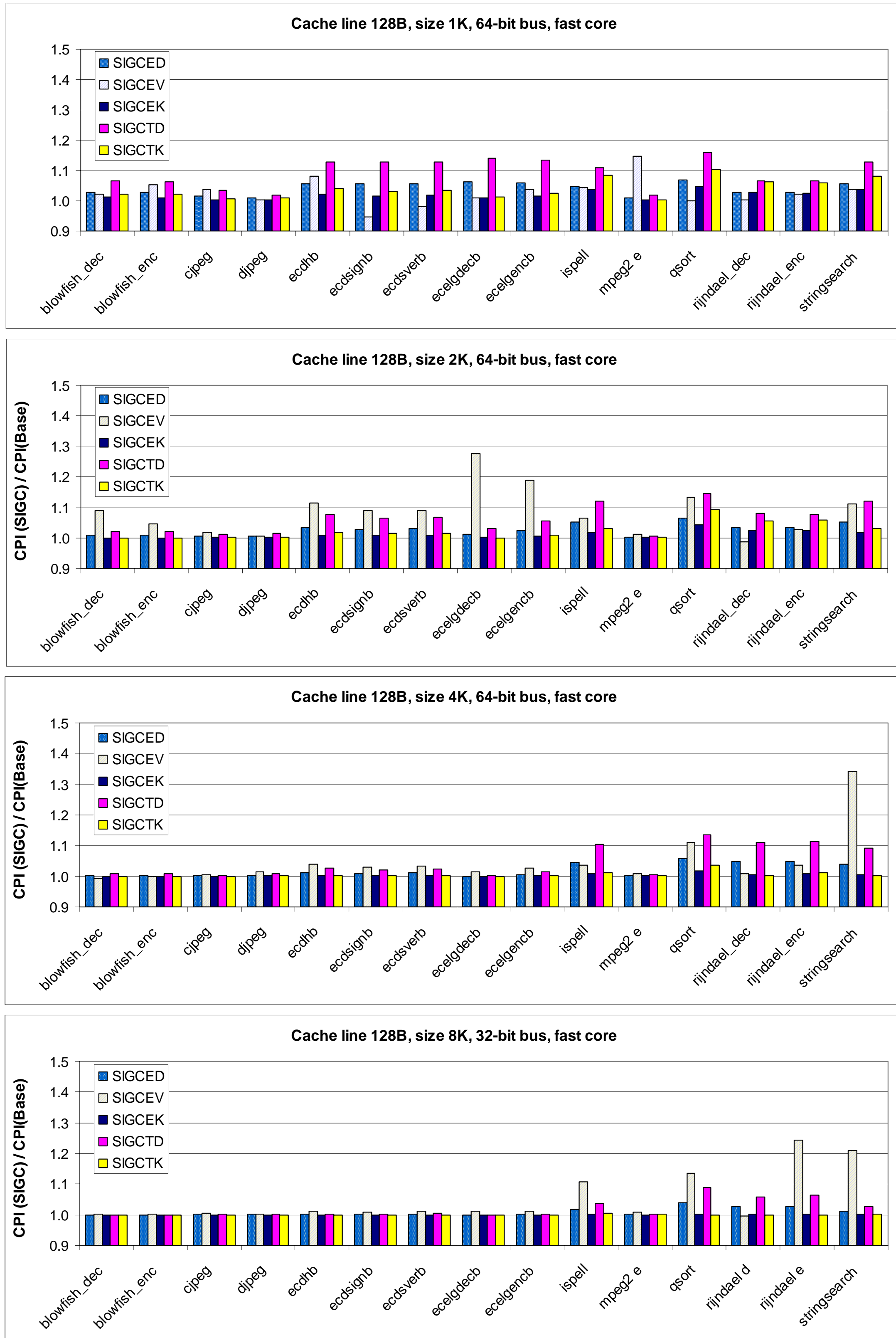

Figure 6.4 SIGC: embedded processor configuration, I-cache line 128B, 64-bit bus, fast core

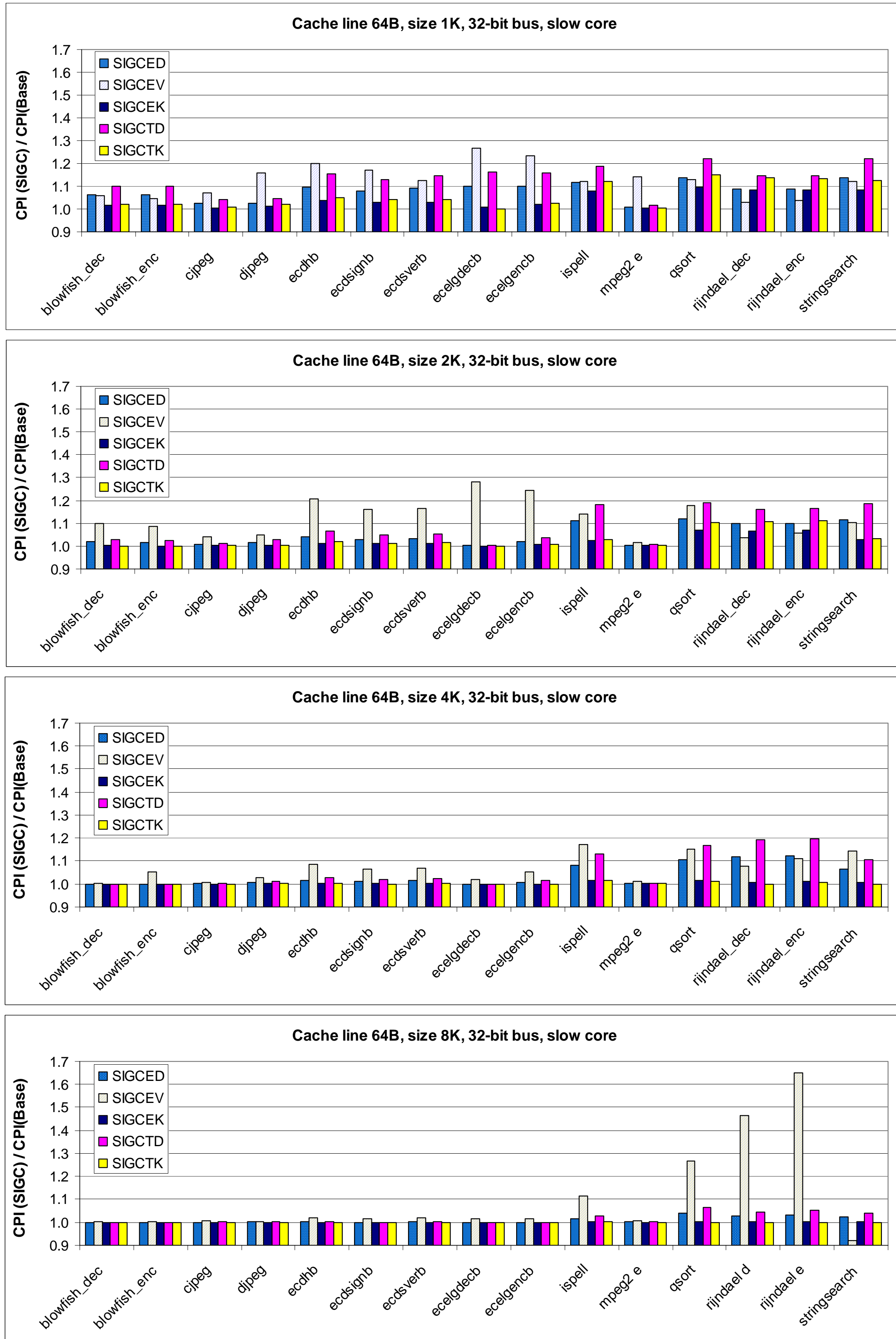

Figure 6.5 SIGC: embedded processor configuration, I-cache line 64B, 32-bit bus, slow core

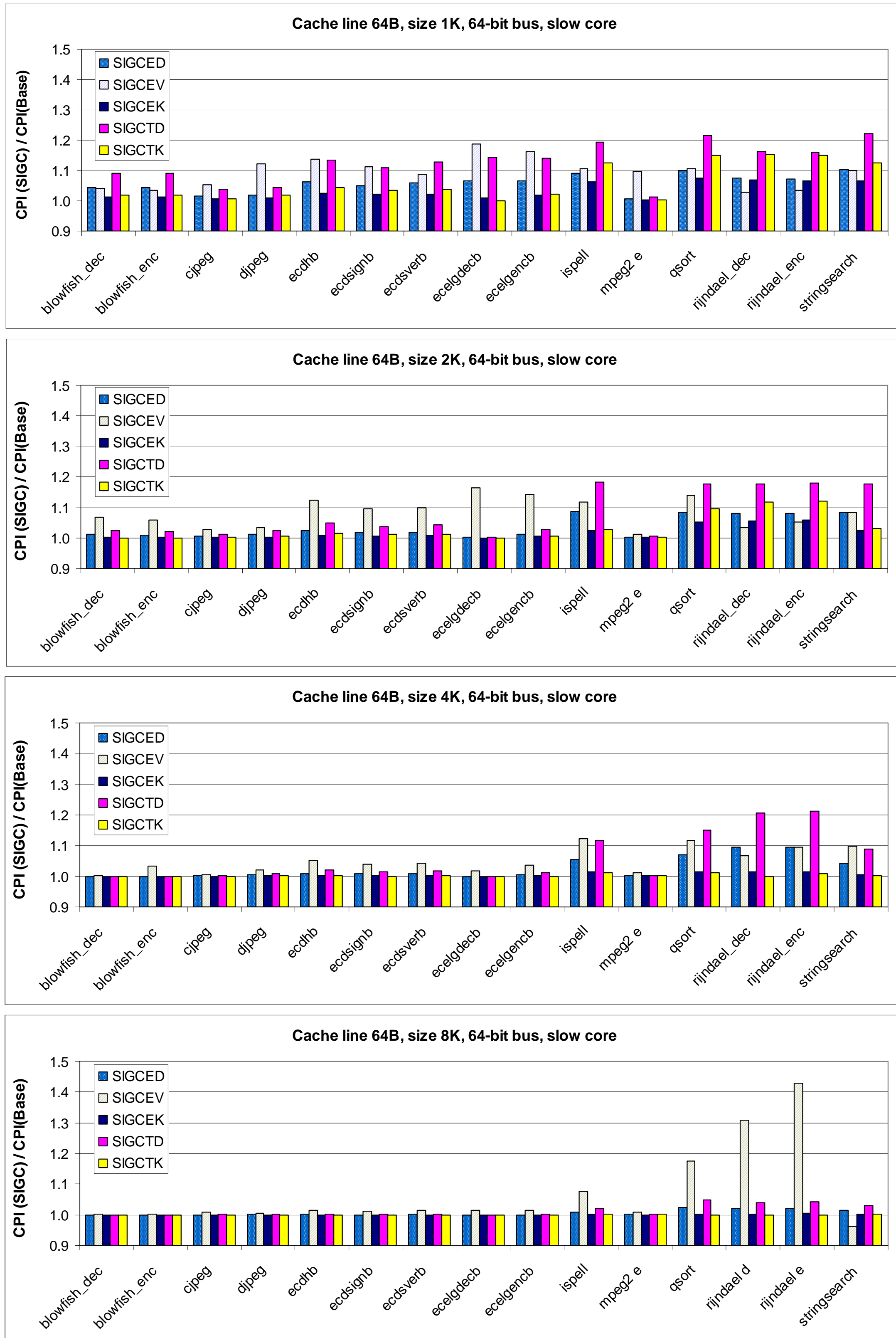

Figure 6.6   SIGC: embedded processor configuration, I-cache line 64B, 64-bit bus, slow core

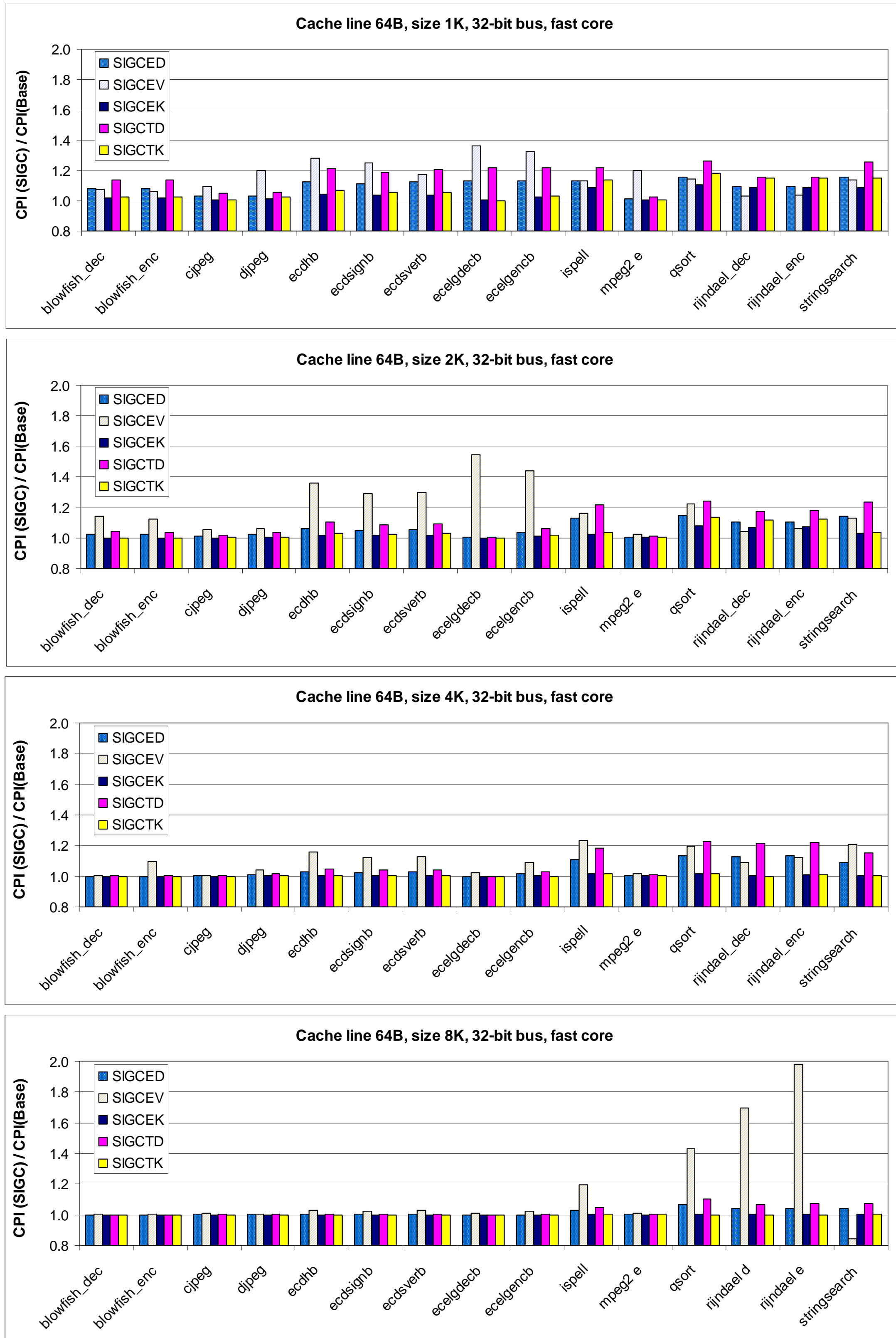

Figure 6.7 SIGC: embedded processor configuration, I-cache line 64B, 32-bit bus, fast core

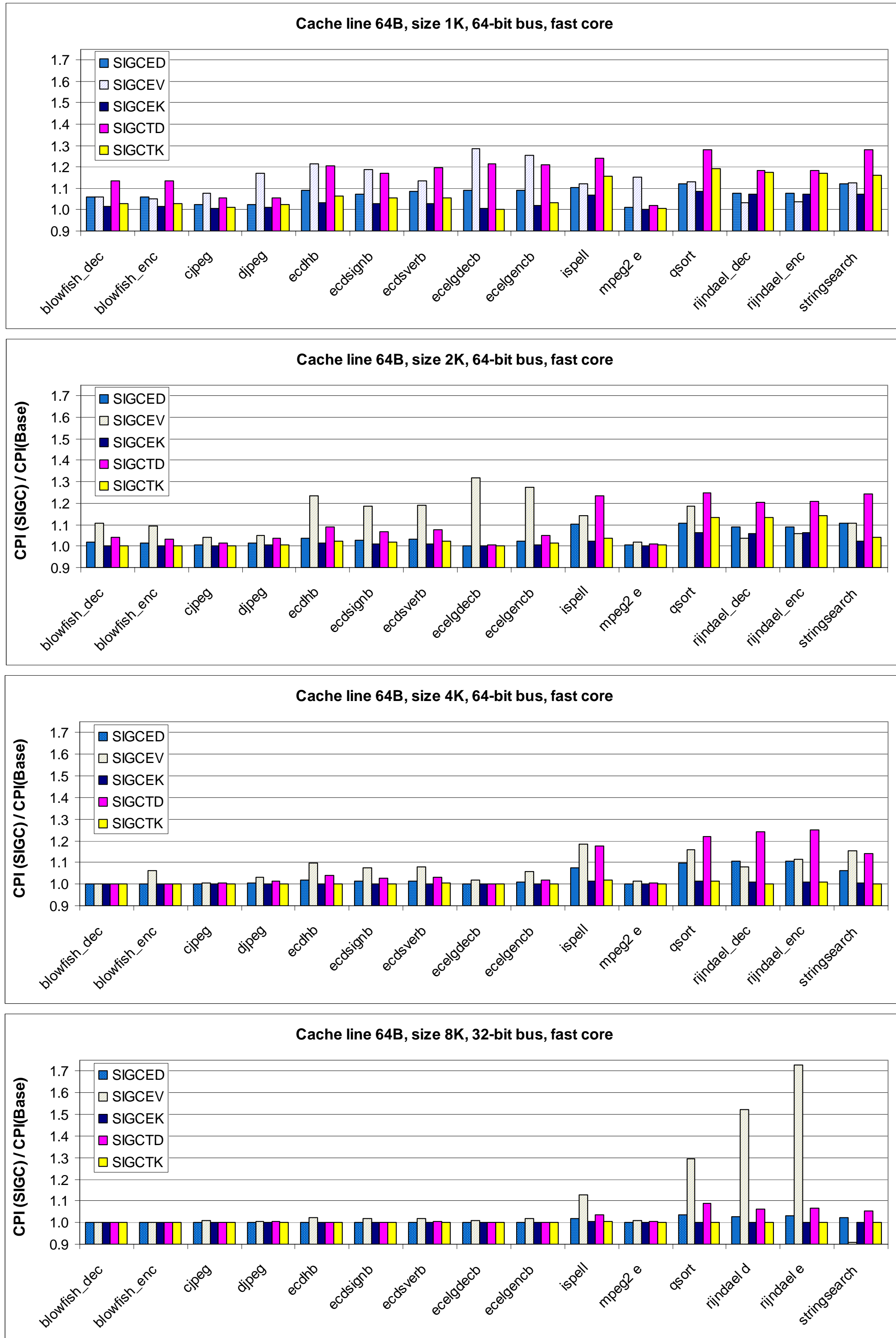

Figure 6.8 SIGC: embedded processor configuration, I-cache line 64B, 64-bit bus, fast core

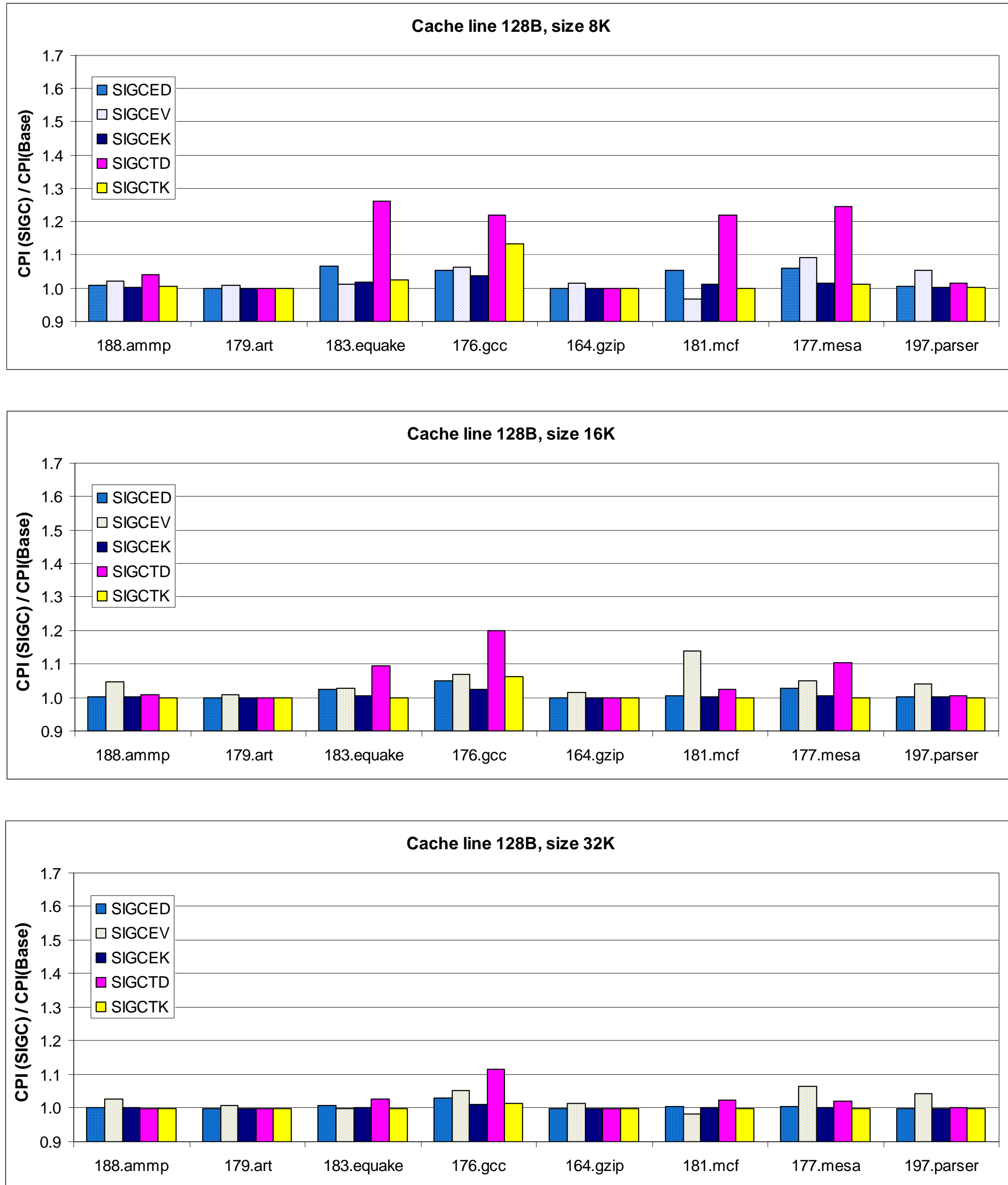

Figure 6.9 SIGC: high-end processor configuration, I-cache line 128B

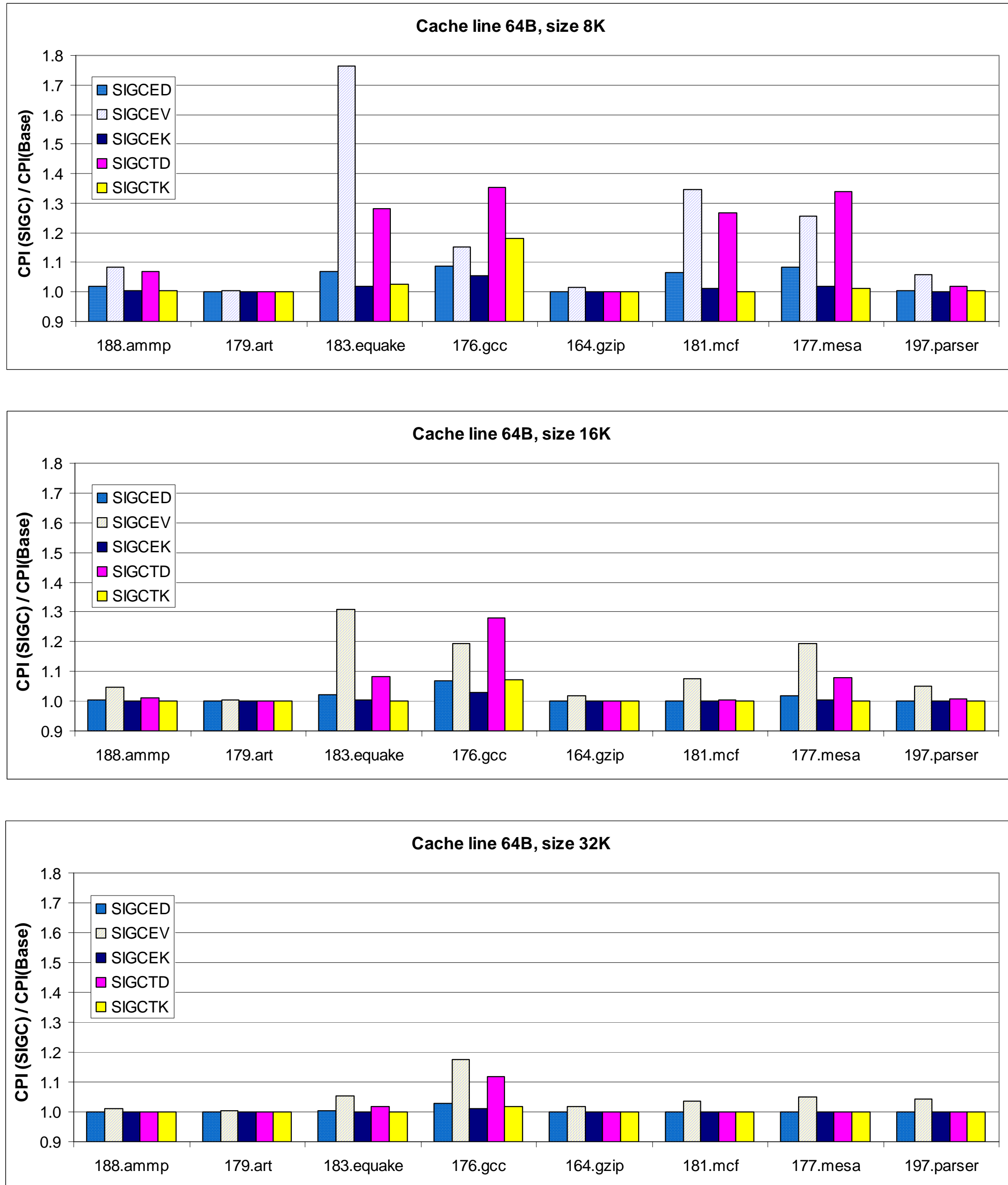

Figure 6.10 SIGC: high-end processor configuration n, I-cache line 64B

6.1.3 Memory Overhead

Memory overhead is an inherent characteristic of all proposed techniques, since instruction block signatures are added to the executable code. The increase of a code section depends mostly on the size of signatures and the size of protected blocks; techniques with signatures embedded in the code also may add some padding.

Let the signature size be 16 bytes. On average, the SIGCED technique then increases the size of the code section by 25.5% with 64B-protected blocks, and by 14.3% with 128B-protected blocks. The SIGCEV technique has shorter protected blocks than the SIGCTK; on the other hand, it might have less padding. With 64B I-cache lines, i.e., 48B-protected blocks, the SIGCEV increases the code section by 33.3%, and by 14.3% with 112B-protected blocks. The SIGCTD technique does not require padding, so the code section increase is 25% with 64B cache lines, and 12.5% with 128B cache lines. The S-cache does not influence the code size, so memory overhead is the same for SIGCED and SIGCEK, and for SIGCTD and SIGCTK.

An executable file typically encompasses more than only code section, so the SIGC techniques add even less memory overhead to executable files. Table 6.13 and Table 6.14 show the percentage of executable file size increase, for selected benchmarks from the embedded domain and the SPEC CPU2000 benchmark set. For all considered benchmarks except *176.gcc* and *177.mesa*, the SIGC techniques have less than 4% memory overhead with 128B I-cache lines, and less than 9% overhead with 64B cache lines.

Table 6.12 Percentage of file size increase for SPEC CPU2000 benchmarks

Benchmark	**I-cache line 64B**			**I-cache line 128B**		
	SIGCEV	**SIGCED**	**SIGCTD**	**SIGCEV**	**SIGCED**	**SIGCTD**
164.gzip	7.65	5.85	5.73	3.28	3.29	2.87
176.gcc	18.44	14.10	13.83	7.90	7.90	6.91
181.mcf	7.15	5.47	5.36	3.07	3.07	2.68
197.parser	8.82	6.74	6.61	3.78	3.78	3.31
177.mesa	14.19	10.85	10.64	6.08	6.08	5.32
179.art	7.21	5.51	5.41	3.10	3.09	2.70
183.equake	7.25	5.55	5.44	3.11	3.11	2.72
188.ammp	8.86	6.77	6.64	3.80	3.80	3.32

Table 6.13 Percentage of file size increase for benchmarks from the embedded domain

Benchmark	I-cache line 64B			I-cache line 128B		
	SIGCEV	**SIGCED**	**SIGCTD**	**SIGCEV**	**SIGCED**	**SIGCTD**
blowfish	6.17	4.71	4.62	2.65	2.65	2.31
cjpeg	7.90	6.04	5.92	3.39	3.39	2.96
djpeg	8.14	6.23	6.10	3.49	3.49	3.05
ecdhb	7.81	5.98	5.86	3.35	3.36	2.93
ecdsignb	8.24	6.30	6.18	3.54	3.54	3.09
ecdsverb	8.24	6.31	6.18	3.54	3.54	3.09
ecelgdecb	7.81	5.97	5.85	3.35	3.35	2.93
ecelgencb	7.81	5.97	5.86	3.35	3.35	2.93
ispell	6.49	4.97	4.87	2.79	2.78	2.43
mpeg2_enc	8.03	6.14	6.02	3.44	3.44	3.01
qsort	7.12	5.45	5.34	3.06	3.05	2.67
rijndael	6.36	4.87	4.77	2.74	2.73	2.38
stringsearch	6.13	4.69	4.60	2.63	2.63	2.30

Table 6.14 Percentage of file size increase for SPEC CPU2000 benchmarks

Benchmark	I-cache line 64B			I-cache line 128B		
	SIGCEV	SIGCED	SIGCTD	SIGCEV	SIGCED	SIGCTD
164.gzip	7.65	5.85	5.73	3.28	3.29	2.87
176.gcc	18.44	14.10	13.83	7.90	7.90	6.91
181.mcf	7.15	5.47	5.36	3.07	3.07	2.68
197.parser	8.82	6.74	6.61	3.78	3.78	3.31
177.mesa	14.19	10.85	10.64	6.08	6.08	5.32
179.art	7.21	5.51	5.41	3.10	3.09	2.70
183.equake	7.25	5.55	5.44	3.11	3.11	2.72
188.ammp	8.86	6.77	6.64	3.80	3.80	3.32

6.2 SIGB Evaluation

Due to the ever-increasing processor-memory speed gap, the memory access overhead will be the predominant overhead component of the SIGB techniques. To assess this overhead, we measure the number of S-cache misses for SIGBTK technique, and the number of additional I-cache misses for the SIGBEV. We also measure memory overhead due to basic block signatures. All SIGB techniques have the same memory overhead, since none of them requires padding.

6.2.1 Performance Overhead

Table 6.15 shows the number of I-cache misses and signature verifications per one million (1M) instructions for the SIGBTK technique and 8K, 16K, and 32K I-cache sizes. The considered I-caches all have 4 ways and LRU replacement policy. Unlike the SIGC techniques where each I-cache miss triggers signature verification, the SIGB techniques verify basic block signatures only for the last basic block in an instruction stream that caused at least one I-cache miss. Hence, the number of signature verifications may be less than the number of I-cache misses. For SPEC CPU2000 benchmarks, the number of signature verifications ranges from less than the quarter of the number of I-cache misses for *301.appsi* and the 16K I-cache, to only slightly less than the number of I-cache misses for *300.twolf* and the 32K I-cache. On average, the ratio of the number of I-cache misses to the number of verifications is 1.95.

With the SIGBTK technique, each signature verification corresponds to an S-cache lookup. Table 6.16 shows the number of S-cache misses for 8K, 16K, and 32K I-cache sizes. The S-cache can store 256 signatures, organized in 128 sets and 2 ways. With the 32K I-cache, which is a reasonable I-cache size for high-end processors, the number of S-cache misses is less than one thousand per 1M instructions for 17 out of 22 benchmarks, or less than once in 1000 executed instructions. Even with the relatively small 8K I-cache, only two benchmarks, *253.perlbmk* and *186.crafty*, have more than 10000 S-cache misses in 1M instructions.

We also evaluate the sensitivity of the number of S-cache misses to the S-cache size and associativity. Figure 6.11 shows the number of S-cache misses per 1M instructions, with the 32K I-cache

and various S-cache sizes. We simulate the S-cache with two ways and 16, 32, 64, 128, and 256 sets. Only three SPEC CPU2000 integer applications have over 1000 misses per 1M instructions for all simulated S-cache sizes -- *255.vortex*, *176.gcc*, and *253.perlbmk* -- and of the floating point applications only *191.fma3d* has over 1000 misses, and only for the smallest simulated sizes. The results indicate that a very small S-cache size is enough for most considered applications. We also evaluate the influence of S-cache associativity to the number of misses for an S-cache with 128 entries, and direct mapped organization, 2, 4, and 8 ways (Figure 6.12). Most applications do not significantly benefit from more than two ways.

Each S-cache miss causes additional memory accesses for the SigTable search. Figure 6.13 shows the number of required memory access per 1M instructions for *176.gcc*, *172.mgrid*, and *164.gzip*, with the segmented binary search. However, a simple hash function may significantly reduce the number of memory accesses, and a perfect hash function may reduce it to only one access.

Let us compare the SIGBTK technique with the SIGCTK, with the same S-cache size. With the perfect hash function, the SigTable search adds approximately the same overhead as with the SIGCTK technique. On the other hand, since the SIGBTK technique has a smaller number of signature verifications, it will have less S-cache misses than the SIGCTK technique. Hence, with the appropriate search function the SIGBTK can outperform the SIGCTK.

With the SIGBEV technique, the signatures are fetched from memory into the I-cache together with the regular instructions, so there are no extra memory accesses for signature verification, but the overall number of I-cache misses increases. To assess the SIGBEV potential, we compare the number of I-cache misses per one million instructions for the original code and protected code. Table 6.17 shows the number of I-cache misses and signature verifications per 1M instructions for the SIGBEV. The SIGBEV should not significantly influence the overall program performance for applications with relatively few I-cache misses. For one application, *183.equake*, the number of I-cache misses in the 32K I-cache is even reduced, due to the better alignment of some portions of the code. However, for some applications the increase in the number of I-cache misses can be considerable: for example, for *252.eon* and 32K I-cache, this number increases from about 300 to about 7000 I-cache misses per one million instructions.

Table 6.15 SIGBTK: Number of I-cache misses and signature verifications per 1M instructions

Benchmark		I-cache misses per 1M instructions			Signature verifications per 1M instructions		
		8K	16K	32K	8K	16K	32K
Integer	164.gzip	7.67	0.25	0.24	4.79	0.13	0.12
	176.gcc	27952.10	17077.95	5596.00	14578.33	9319.48	3243.21
	181.mcf	10584.81	533.72	0.28	6292.26	215.58	0.16
	186.crafty	55757.21	20795.13	2898.48	28537.25	10964.53	1624.79
	197.parser	834.11	522.10	101.60	489.47	307.59	68.02
	252.eon	27417.56	8415.70	321.16	14483.61	5383.62	308.14
	253.perlbmk	42082.99	28221.11	11886.71	22446.80	15577.89	6606.16
	254.gap	10213.49	3009.15	367.86	6229.38	1865.26	164.88
	255.vortex	32974.83	19878.95	10427.85	16879.17	10155.27	5444.23
	300.twolf	17235.97	3762.88	44.35	9100.11	2339.60	43.25
Floating point	168.wupwise	414.50	1.01	0.75	269.88	0.56	0.40
	171.swim	32.00	13.41	2.76	15.77	5.61	1.26
	172.mgrid	24.00	14.18	3.96	10.53	5.88	1.61
	177.mesa	18721.89	1193.49	12.64	6923.03	754.49	7.35
	178.galgel	1.60	1.17	0.91	0.81	0.60	0.45
	179.art	2.57	0.18	0.18	1.08	0.08	0.08
	183.equake	22179.91	2842.17	1326.53	10965.51	2176.89	888.22
	188.ammp	2198.80	76.59	0.50	1334.89	62.78	0.31
	189.lucas	0.76	0.60	0.51	0.41	0.30	0.25
	191.fma3d	16946.39	10266.68	4126.92	8897.55	5905.82	2352.48
	200.sixtrack	7051.75	2686.58	871.97	3246.70	1324.99	474.79
	301.appsi	19248.16	12368.17	2217.44	5315.02	2944.37	741.99

Table 6.16 SIGBTK: Number of S-cache misses per 1M instructions

Benchmark		S-cache misses per 1M instructions (128 sets, 2 ways)		
		8K	16K	32K
Integer	164.gzip	1.54	0.12	0.12
	176.gcc	8231.29	5611.80	1951.31
	181.mcf	320.08	0.17	0.16
	186.crafty	11660.36	4082.43	480.86
	197.parser	260.34	205.41	20.61
	252.eon	5284.62	681.44	1.87
	253.perlbmk	12203.76	8150.48	2281.50
	254.gap	1792.36	479.88	5.15
	255.vortex	8926.09	6655.88	3592.60
	300.twolf	1668.19	64.94	0.67
Floating point	168.wupwise	0.53	0.45	0.39
	171.swim	2.95	0.91	0.59
	172.mgrid	3.12	2.06	0.53
	177.mesa	573.47	8.92	0.73
	178.galgel	0.55	0.48	0.42
	179.art	0.09	0.08	0.08
	183.equake	494.30	0.13	0.11
	188.ammp	135.34	0.47	0.24
	189.lucas	0.29	0.26	0.23
	191.fma3d	4417.81	2092.49	50.87
	200.sixtrack	1063.22	612.81	158.06
	301.appsi	608.34	182.37	1.31

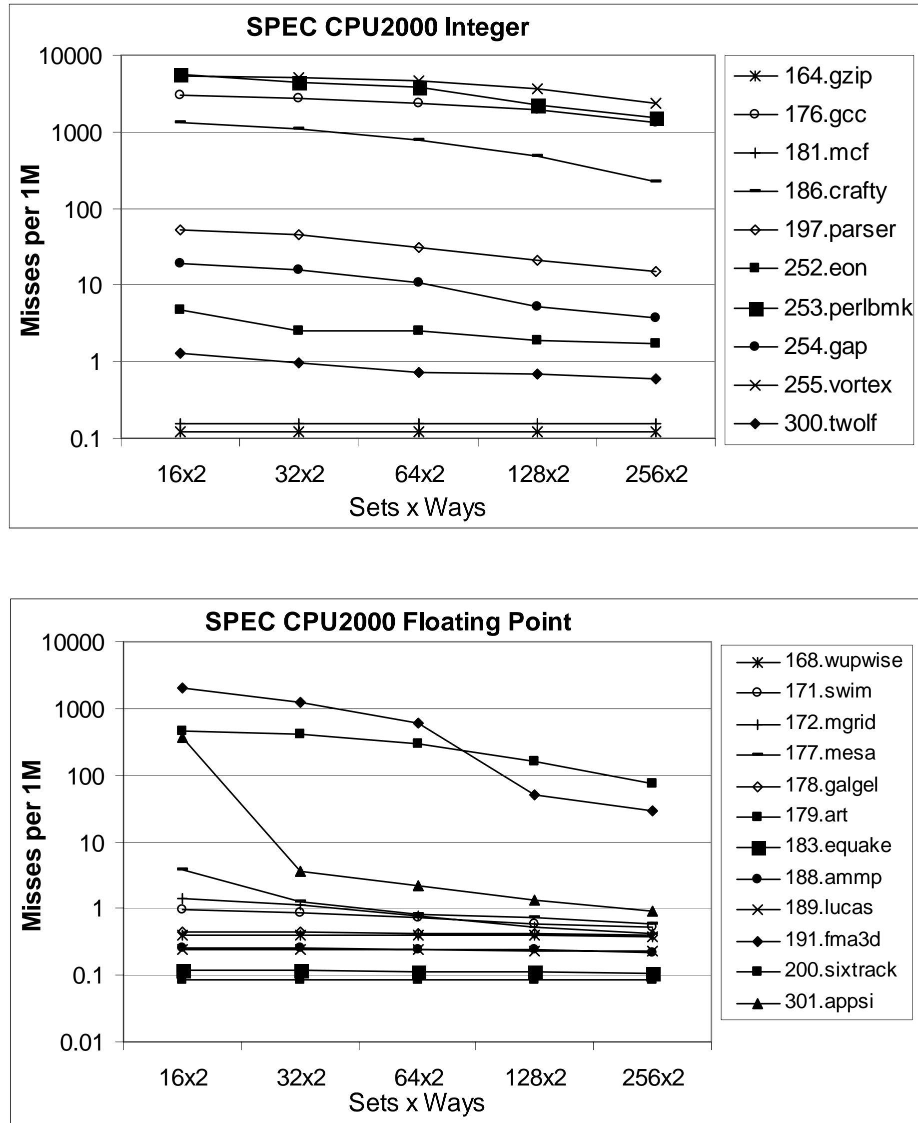

Figure 6.11 SIGBTK: Number of S-cache misses as a function of S-cache size
I-cache size: 32K

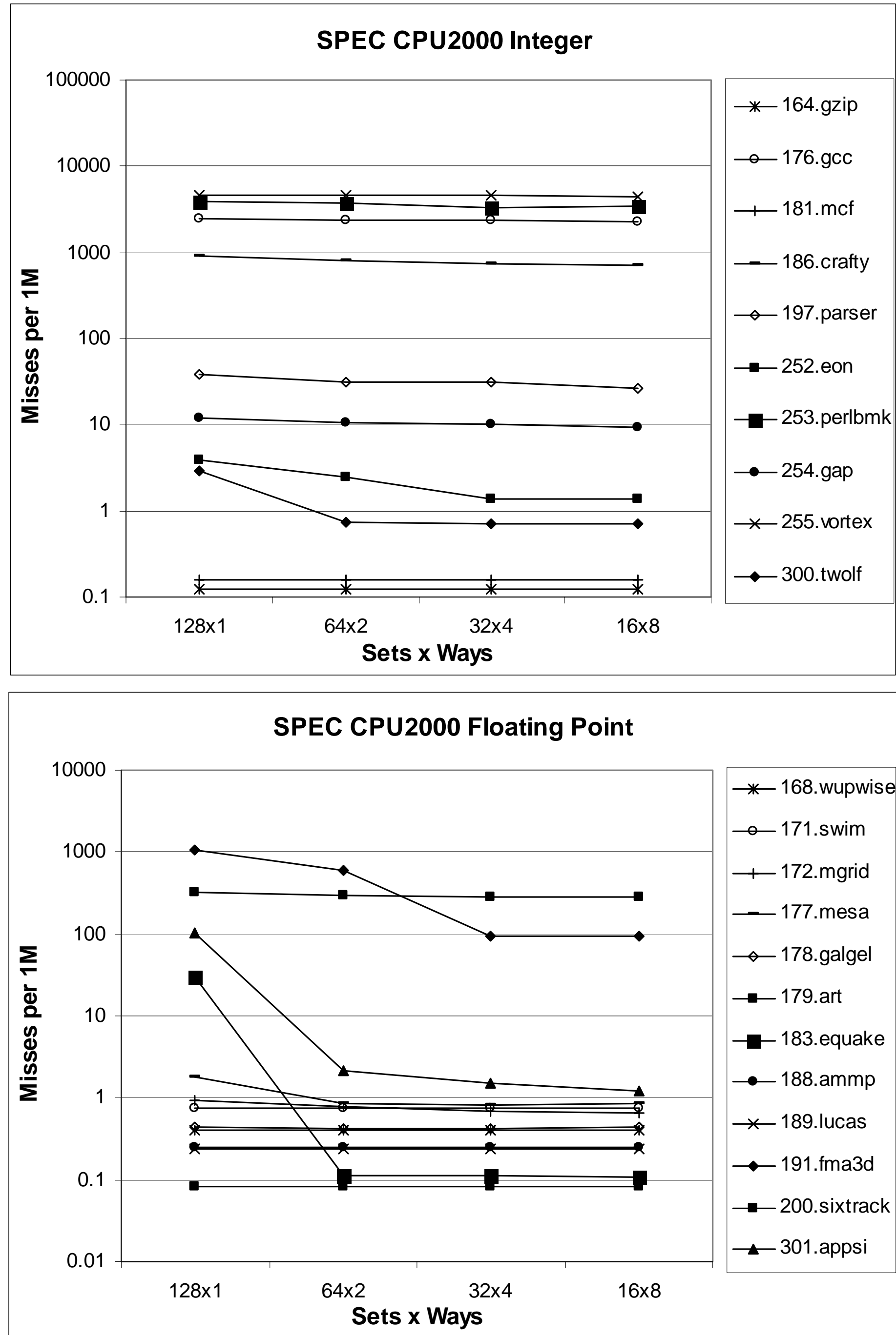

Figure 6.12 SIGBTK: Number of S-cache misses as a function of S-cache associativity I-cache size: 32K

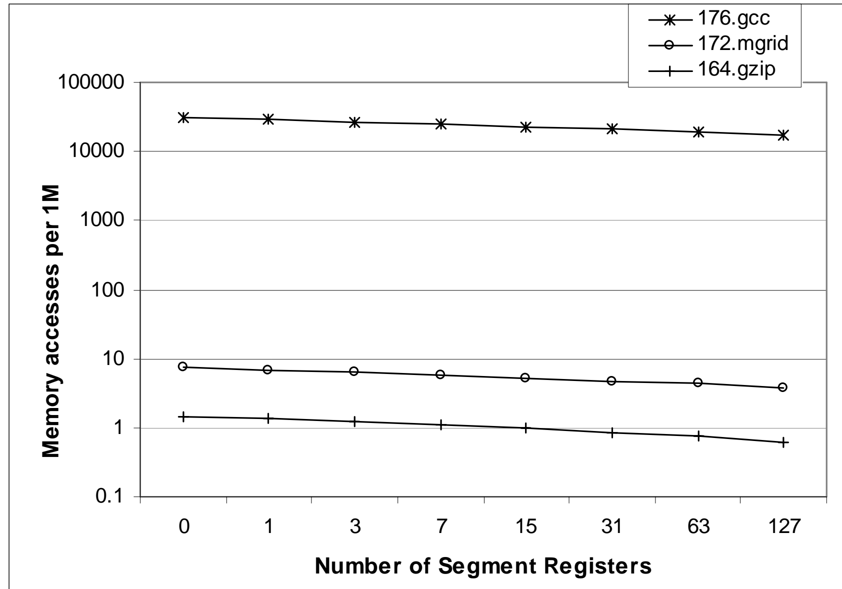

Figure 6.13 SIGBTK: Number of memory accesses per 1M instructions due to S-cache misses with the segmented binary search,(128-set, 2-way) S-cache, and 32K I-cache

Table 6.17 SIGBEV: Number of I-cache misses and signature verifications per 1M instructions

Benchmark		I-cache misses per 1M instructions			Signature verifications per 1M instructions		
		8K	16K	32K	8K	16K	32K
Integer	164.gzip	24.60	8.15	0.40	13.33	4.66	0.19
	176.gcc	49391.24	37419.57	19094.73	23046.80	17903.47	9611.68
	181.mcf	31315.41	3502.30	0.48	16963.19	2329.56	0.24
	186.crafty	95729.56	54278.95	14933.42	44114.61	25748.65	7904.31
	197.parser	5409.83	1246.97	716.58	2819.35	667.98	407.28
	252.eon	48810.07	28504.23	7114.48	23514.00	14684.02	4506.36
	253.perlbmk	73594.06	57448.06	31947.90	36188.91	28473.97	16707.06
	254.gap	22038.24	11897.55	1557.94	12343.94	6458.76	838.91
	255.vortex	67081.58	44479.63	25254.76	30196.85	20144.33	11539.29
	300.twolf	41304.43	11997.24	238.49	18854.01	6567.28	125.87
Floating point	168.wupwise	2620.05	195.90	1.35	1305.26	158.00	0.68
	171.swim	68.80	32.76	4.85	29.74	15.18	2.26
	172.mgrid	38.79	26.47	10.76	17.17	11.39	4.52
	177.mesa	43888.48	6978.36	103.67	18914.99	2956.38	64.20
	178.galgel	3.61	2.19	1.55	1.61	1.01	0.71
	179.art	18.10	1.04	0.27	7.74	0.39	0.12
	183.equake	63206.61	27839.49	676.70	30144.21	11949.41	373.13
	188.ammp	6238.59	1240.54	156.60	3221.81	649.25	96.63
	189.lucas	1.47	1.05	0.83	0.73	0.51	0.39
	191.fma3d	28979.05	23596.61	12348.08	13951.06	11924.89	7009.36
	200.sixtrack	13924.67	7399.13	2954.09	6628.74	3526.30	1496.33
	301.appsi	26646.36	17234.43	4331.44	8069.01	4847.86	1295.26

6.2.2 Memory Overhead

Table 6.18 shows the increase of the code section and of the complete executable file, for precompiled SPEC CPU2000 Alpha binaries and 16-byte signatures. The SIGB signatures increase the size of code section from 38.2% for *200.sixtrack*, to 82.3% for *188.ammp*, much more than with any of the SIGC techniques. The increase of the executable file is also significant, from 15.6 to 52%.

Table 6.18 Number of basic blocks and percentage of file size increase

Benchmark		Number of basic blocks	Code section increase [%]	Executable file increase [%]
Integer	164.gzip	8660	65.1	36.8
	176.gcc	98478	79.2	41.5
	181.mcf	7401	72.3	39.1
	186.crafty	17761	64.2	30.2
	197.parser	14663	73.4	39.2
	252.eon	24285	48.9	32.7
	253.perlbmk	43294	79.0	32.2
	254.gap	47365	81.1	52.0
	255.vortex	33336	65.1	23.1
	300.twolf	17931	63.7	31.3
Floating point	168.wupwise	32989	64.4	47.4
	171.swim	32759	64.0	47.4
	172.mgrid	32312	64.4	47.5
	177.mesa	33757	58.9	18.8
	178.galgel	41805	63.8	45.9
	179.art	9600	64.7	41.7
	183.equake	9436	59.5	39.2
	188.ammp	19917	82.8	48.6
	189.lucas	33246	62.4	46.4
	191.fma3d	59790	51.2	21.2
	200.sixtrack	61938	38.2	15.6
	301.appsi	35393	50.8	37.0

CHAPTER 7

CONCLUSION

"It's tough to make predictions, especially about the future."
"It's déjà vu all over again."
Yogi Berra

Failing to resist attacks on computer systems can incur significant direct costs as well as costs in lost revenues and opportunities, and can even jeopardize national security. Consequently, computer security is becoming a critical issue, and current trends in hardware and software will bring it even more into focus due to the following reasons. First, increased complexity of high-end systems and the large-scale deployment and diversity of low-end systems make it difficult to uncover system vulnerabilities. In addition, exhaustive testing is virtually impossible as software grows in size and complexity, and time-to-market decreases.

One of the major security problems is execution of unauthorized and potentially malicious code. The existing defense techniques often fail to counter attacks, lack generality, induce significant overhead in performance and cost, or generate a significant number of false alarms.

This dissertation proposes new architectural extensions to ensure trusted program execution, i.e., run-time code integrity, in both high-end and embedded computing platforms. All proposed techniques share a common mechanism, which encompasses two phases: secure program installation and secure program execution. The secure installation process determines a signature for each instruction block in a program, using secret keys stored in hardware. Signatures are encrypted and stored with the code. During secure execution, signatures are recalculated from fetched instructions and compared to decrypted stored

signatures. If a signature calculated in run-time does not match the signature calculated during installation, the program cannot be trusted and it receives an abort signal by the operating system.

We propose eight techniques: SIGCED, SIGCEK, SIGCEV, SIGCTD, SIGCTK, SIGBEV, SIGBTD, and SIGBTK. For each technique, we provide a detailed architectural design, a proof-of-concept using functional simulation, and performance and memory overhead analysis. The performance analysis, based on execution- and trace-driven simulation utilizing state-of-the-art benchmarks, proves that the proposed techniques incur very low performance overhead for a broad spectrum of computer platforms.

The main contributions of this dissertation are as follows:

- Proposed novel hardware-based mechanism for trusted program execution based on runtime verification of instruction block signatures.
- Proposed a set of eight techniques that employ the common mechanism and differ in the type of protected instruction blocks, signature placement in the address space, signature placement in the physical memory, and signature handling after the verification.
- Developed simulation environment, including architectural execution-driven and trace-driven simulators, and a program for modification of ELF binaries to emulate the secure installation process.
- Evaluated performance overhead of the proposed techniques and analyzed their sensitivity to a common set of architectural parameters.
- Surveyed the existing software and hardware techniques, developed to counter code injection and similar malicious attacks.

The main findings are as follows:

- The SIGCED and SIGCEK techniques are main candidates for implementation in future processors.
- The SIGCED technique performs consistently well across various system configurations, with the worst-case overhead of 15.6% with 64B I-cache lines, and 8% with 128B I-cache lines.
- If the hardware budget allows the S-cache, the SIGCEK successfully reduces the overhead of signature fetching.

- The evaluation of the SIGCEV technique shows the importance of not keeping the signatures in the cache, since for most applications it increases the number of I-cache misses and consequently the performance overhead. However, the SIGCEV performs well with small caches and 128B cache lines.
- The SIGCTD technique has more performance overhead than the SIGCED; this overhead can be significantly reduced with the S-cache i.e., with the SIGCTK technique. The SIGCTK overhead is less than 10% for 128B cache lines.
- The memory overhead of the SIGC techniques is less than 9% for all but two of the considered benchmarks.
- With a fast search function for signatures stored in memory, the SIGBTK technique promises to have low performance overhead, whereas the SIGBEV technique is another example of how signatures embedded in the code can increase the number of I-cache misses.

Although the main goal of the proposed mechanism is to prevent code injection attacks, it can be applied to other purposes, such as fault-tolerant execution, virus protection, and protection from software tampering.

The proposed techniques open a number of challenging questions for future research, including but not limited to the following:

- Evaluation of power overhead.
- Techniques for reduction of memory overhead, e.g., protecting multiple instruction blocks with the same signature.
- Techniques for reduction of power overhead, e.g., signature and/or instruction prefetching.
- Extension of the basic mechanism to cover other classes of attacks, such as return-into-libc.

REFERENCES

[1] A. One, "Smashing The Stack For Fun And Profit," *Phrack Magazine*, Vol. 7, November 1996.

[2] M. Conover, "w00w00 on Heap Overflows," <http://www.w00w00.org/files/articles/heaptut.txt> (Available January 2005).

[3] T. Newsham, "Format String Attacks," September 2000, <http://www.securityfocus.com/guest/3342> (Available January 2004).

[4] D. Ahmad, "The Rising Threat of Vulnerabilities Due to Integer Errors," *IEEE Security & Privacy*, Vol. 1, July-August 2003, pp. 77-82.

[5] I. Dobrovitski, "Exploit for CVS Double free() for Linux pserver," <http://seclists.org/lists/bugtraq/2003/Feb/0042.html> (Available January 2005).

[6] US-CERT/CC, "CERT/CC Statistics," <http://www.cert.org/stats/> (Available December 2004).

[7] US-CERT, "Cyber Security Bulletin SB04-231," <http://www.us-cert.gov/cas/bulletins/SB04-231.html> (Available November 2004).

[8] US-CERT, "Cyber Security Bulletin SB04-175," <http://www.us-cert.gov/cas/body/bulletins/SB04-175_H.html> (Available November 2004).

[9] S. Smith, "Magic Boxes and Boots: Security in Hardware," *IEEE Computer*, Vol. 37, October 2004, pp. 106-109.

[10] L. Garber, "New Chips Stop Buffer Overflow Attacks," *IEEE Computer*, Vol. 37, October 2004, pp. 28.

[11] O. Gay, "Exploitation Avancée de Buffer Overflows," Security and Cryptography Laboratory (LASEC), Département d'Informatique de L'Ecole Polytechnique Fédérale de Lausanne, Switzerland, June 2002.

[12] J. Pincus and B. Baker, "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns," *IEEE Security and Privacy*, Vol. 2, July-August 2004, pp. 20-27.

[13] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier, "FormatGuard: Automatic Protection From printf Format String Vulnerabilities," in *10th USENIX Security Symposium*, Washington, DC, USA, 2001, pp. 191-200.

[14] Anonymous, "Once Upon a free()," *Phrack Magazine*, Vol. 11, 2001.

[15] Y. Younan, W. Joosen, and F. Piessens, "Code Injection in C and C++: A Survey of Vulnerabilities and Countermeasures," Katholieke Universiteit Leuven, Belgium, Technical Report CW386, July 2004.

[16] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," in *DARPA Information Survivability Conference and Exposition*, Hilton Head, SC, USA, 2000, pp. 119-129.

[17] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard™: Protecting Pointers From Buffer Overflow Vulnerabilities," in *12th USENIX Security Symposium*, Washington, DC, USA, 2003, pp. 91-104.

[18] W. Landi, "Undecidability of Static Analysis," *ACM Letters on Programming Languages and Systems (LOPLAS)*, Vol. 1, December 1992, pp. 323-337.

[19] B. Chess and G. McGraw, "Static Analysis for Security," *IEEE Security & Privacy Magazine*, Vol. 2, November-December 2004, pp. 76-79.

[20] "ITS4: Software Security Tool," <http://www.cigital.com/its4/> (Available January 2005).

[21] J. Viega, J. T. Bloch, T. Kohno, and G. McGraw, "Token-Based Scanning of Source Code for Security Problems," *ACM Transactions on Information and System Security (TISSEC)*, Vol. 5, August 2002, pp. 238-261.

[22] J. Viega, J. T. Bloch, T. Kohno, and G. McGraw, "ITS4: A Static Vulnerability Scanner for C and C++ Code," in *Annual Computer Security Applications Conference*, New Orleans, LA, USA, 2000.

[23] D. Wheeler, "Flawfinder," <http://www.dwheeler.com/flawfinder/> (Available January 2005).

[24] "Free Software Security Tools," <http://www.securesw.com/resources/tools.html> (Available January 2005).

[25] A. DeKok, "PScan: A Limited Problem Scanner for C Source Files," July 2000, <http://www.striker.ottawa.on.ca/~aland/pscan/> (Available January 2005).

[26] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," in *Network and Distributed System Security Symposium (NDCS)*, San Diego, CA, USA, 2000.

[27] D. Wagner, "BOON: Buffer Overrun Detection," <http://www.cs.berkeley.edu/~daw/boon/> (Available January 2005).

[28] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek, "Buffer Overrun Detection Using Linear Programming and Static Analysis," in *10th ACM Conference on Computer and Communications Security*, Washington, DC, 2003, pp. 345-354.

[29] Y. Xie, A. Chou, and D. Engler, "ARCHER: Using Symbolic, Path-Sensitive Analysis to Detect Memory Access Errors," in *9th European Software Engineering Conference (ESEC) held jointly with 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Helsinki, Finland, 2003, pp. 327-336.

[30] "Splint," <http://splint.org/> (Available January 2005).

[31] D. Larochelle and D. Evans, "Statically Detecting Likely Buffer Overflow Vulnerabilities," in *10th USENIX Security Symposium*, Washington, DC, USA, 2001, pp. 177-189.

[32] D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis," *IEEE Software*, Vol. 19, January/February 2002, pp. 42-51.

[33] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Automated Detection of Format-String Vulnerabilities Using Type Qualifiers," in *10th USENIX Security Symposium*, Washington, DC, USA, 2001.

[34] N. Dor, M. Rodeh, and M. Sagiv, "CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C," in *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, CA, USA, 2003, pp. 155-167.

[35] K. Ashcraft and D. Engler, "Using Programmer-Written Compiler Extensions to Catch Security Holes," in *IEEE Symposium on Security and Privacy*, Berkeley, CA, USA, 2002, pp. 131–147.

[36] B. Chess, "Improving Computer Security Using Extended Static Checking," in *IEEE Symposium on Security and Privacy*, Berkeley, CA, USA, 2002, pp. 160-173.

[37] G. J. Holzmann, "Static Source Code Checking For User-Defined Properties," in *6th World Conference on Integrated Design and Process Technology*, Pasadena, CA, USA, 2002.

[38] D. Evans, J. Guttag, J. Horning, and Y. M. Tan, "LCLint: A Tool for Using Specifications to Check Code," in *2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, New Orleans, LA, USA, 1994, pp. 87-96.

[39] J. S. Foster, M. Fähndrich, and A. Aiken, "A Theory of Type Qualifiers," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, Atlanta, GA, USA, 1999, pp. 192-203.

[40] N. Dor, M. Rodeh, and M. Sagiv, "Cleanness Checking of String Manipulations in C Programs via Integer Analysis," in *8th International Symposium on Static Analysis*, Paris, France, 2001, pp. 194-212.

[41] J. Wilander and M. Kamkar, "A Comparison of Publicly Available Tools for Static Intrusion Prevention," in *7th Nordic Workshop on Secure IT Systems*, Karlstad, Sweden, 2002.

[42] M. Zitser, R. Lippmann, and T. Leek, "Testing Static Analysis Tools Using Exploitable Buffer Overflows From Open Source Code," in *12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Newport Beach, CA, USA, 2004, pp. 97-106.

[43] S. C. Kendall, "Bcc: Runtime Checking for C Programs," in *USENIX Summer 1983 Conference*, Toronto, ON, Canada, 1983, pp. 5-16.

[44] J. L. Steffen, "Adding Run-Time Checking to the Portable C Compiler," *Software—Practice & Experience*, Vol. 22, April 1992, pp. 305-316.

[45] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient Detection of All Pointer and Array Access Errors," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, USA, 1994, pp. 290-301.

[46] H. Patil and C. Fischer, "Low-Cost, Concurrent Checking of Pointer and Array Accesses in C programs," *Software - Practice and Experience*, Vol. 27, January 1997, pp. 87-110.

[47] R. W. M. Jones and P. H. J. Kelly, "Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs," in *AADEBUG'97 - Proceedings of the 3rd International Workshop on Automated and Algorithmic Debugging*, Linköping, Sweden, 1997, pp. 13-26.

[48] A. Loginov, S. H. Yong, S. Horwitz, and T. W. Reps, "Debugging via Run-Time Type Checking," in *4th International Conference on Fundamental Approaches to Software Engineering*, Genova, Italy, 2001, pp. 217-232.

[49] K. Lhee and S. J. Chapin, "Type-Assisted Dynamic Buffer Overflow Detection," in *11th USENIX Security Symposium*, San Francisco, CA, USA, 2002, pp. 81-88.

[50] K. Avijit, P. Gupta, and D. Gupta, "TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection," in *13th Usenix Security Symposium*, San Diego, CA, USA, 2004.

[51] Y. Oiwa, T. Sekiguchi, E. Sumii, and A. Yonezawa, "Fail-Safe ANSI-C Compiler: An Approach to Making C Programs Secure," in *International Symposium on Software Security 2002*, Tokyo, Japan, 2002, pp. 133-153.

[52] O. Ruwase and M. S. Lam, "A Practical Dynamic Buffer Overflow Detector," in *11th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, 2004, pp. 159-169.

[53] W. Xu, D. C. DuVarney, and R. Sekar, "An Efficient and Backwards Compatible Transformation to Ensure Memory Safety of C Programs," in *12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Newport Beach, CA, USA, 2004, pp. 117-126.

[54] M. Rinard, C. Cadar, D. Roy, D. Dumitran, and T. Leu, "A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors)," in *20th Annual Computer Security Applications Conference (ACSAC 2004)*, Tucson, AZ, USA, 2004.

[55] S. H. Yong and S. Horwitz, "Protecting C Programs from Attacks via Invalid Pointer Dereferences," in *9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Helsinki, Finland, 2003, pp. 307-316.

[56] R. Hastings and B. Joyce, "Purify: Fast Detection of Memory Leaks and Access Errors," in *Proceedings of the Winter '92 USENIX conference*, San Francisco, CA, USA, 1992, pp. 125-136.

[57] E. Haugh and M. Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities," in *10th Network and Distributed System Security Symposium (NDSS'03)*, San Diego, CA, USA, 2003.

[58] E. Larson and T. Austin, "High Coverage Detection of Input-Related Security Faults," in *12th USENIX Security Symposium*, Washington, D.C., USA, 2003, pp. 121-136.

[59] A. K. Ghosh, T. O'Connor, and G. McGraw, "An Automated Approach for Identifying Potential Vulnerabilities in Software," in *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 1998, pp. 104-114.

[60] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer Overflow Attacks," in *7th USENIX Security Conference*, San Antonio, TX, USA, 1998, pp. 63-78.

[61] Vendicator, "Stack Shield," 08 January 2000, <http://www.angelfire.com/sk/stackshield/> (Available January 2005).

[62] T. Chiueh and F. Hsu, "RAD: A Compile-Time Solution to Buffer Overflow Attacks," in *21st International Conference on Distributed Computing Systems*, Phoenix, AZ, USA, 2001, pp. 409-420.

[63] M. Prasad and T. Chiueh, "A Binary Rewriting Defense Against Stack-based Buffer Overflow Attacks," in *Usenix Annual Technical Conference*, San Antonio, TX, USA, 2003, pp. 211-224.

[64] H. Etoh and K. Yoda, "Protecting from Stack-smashing Attacks," June 2000, <http://www.trl.ibm.com/projects/security/ssp/main.html> (Available January 2005).

[65] A. Baratloo, N. Singh, and T. Tsai, "Transparent Run-Time Defense Against Stack Smashing Attacks," in *USENIX 2000 Annual Technical Conference Proceedings*, San Diego, CA, USA, 2000, pp. 251-262.

[66] C. Fetzer and Z. Xiao, "Detecting Heap Smashing Attacks Through Fault Containment Wrappers," in *20th IEEE Symposium on Reliable Distributed Systems*, New Orleans, LA, USA, 2001, pp. 80-89.

[67] C. Fetzer and Z. Xiao, "HEALERS: A Toolkit for Enhancing the Robustness and Security of Existing Applications," in *International Conference on Dependable Systems and Networks (Practical Experience and Demonstrations)*, San Francisco, CA, USA, 2003, pp. 317-322.

[68] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur, "Run-time Detection of Heap-based Overflows," in *17th Large Installation Systems Administrators Conference*, San Diego, CA, USA, 2003, pp. 51-60.

[69] R. DeLine and M. Fähndrich, "Enforcing High-Level Protocols in Low-Level Software," in *2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, UT, USA, 2001, pp. 59-69.

[70] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fähndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy, "Righting Software," *IEEE Software*, Vol. 21, May-June 2004, pp. 92-100.

[71] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A Safe Dialect of C," in *USENIX Annual Technical Conference*, Monterey, CA, USA, 2002, pp. 275-288.

[72] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, "Region-based Memory Management in Cyclone," in *ACM Conference on Programming Language Design and Implementation*, Berlin, Germany, 2002, pp. 282-293.

[73] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe Retrofitting of Legacy Code," in *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, OR, USA, 2002, pp. 128-139.

[74] S. Kowshik, D. Dhurjati, and V. Adve, "Ensuring Code Safety Without Runtime Checks for Real-time Control Systems," in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, Grenoble, France, 2002, pp. 288-297.

[75] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner, "Memory Safety Without Runtime Checks or Garbage Collection," in *2003 ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, San Diego, CA, USA, 2003, pp. 69-80.

[76] P. Busser, "Memory Protection with PaX and the Stack Smashing Protector: Breaking out Peace," *Linux Magazine* March 2004, pp. 36-39.

[77] "Homepage of The PaX Team," <http://pax.grsecurity.net/> (Available December 2004).

[78] J. Xu, Z. Kalbarczyk, and R. K. Iyer, "Transparent Runtime Randomization for Security," in *22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, Florence, Italy, 2003, pp. 260-269.

[79] M. Chew and D. Song, "Mitigating Buffer Overflows by Operating System Randomization," Carnegie Mellon University, Technical Report CMU-CS-02-197, December 2002.

[80] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address Obfuscation: An Approach to Combat Buffer Overflows, Format-String Attacks, and More," in *12th USENIX Security Symposium*, Washington, DC, USA, 2003, pp. 105-120.

[81] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," in *10th ACM Conference on Computer and Communication Security*, Washington, DC, USA, 2003, pp. 281-289.

[82] R. Sekar, T. Bowen, and M. Segal, "On Preventing Intrusions by Process Behavior Monitoring," in *8th USENIX Security Symposium*, Washington, DC, USA, 1999, pp. 29-40.

[83] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," in *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 1999, pp. 133-145.

[84] I. Sato, Y. Okazaki, and S. Goto, "An Improved Intrusion Detection Method Based on Process Profiling," *IPSJ Journal*, Vol. 43, November 2002, pp. 3316-3326.

[85] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion Detection Using Sequences of System Calls," *Journal of Computer Security*, Vol. 6, 1998, pp. 151-180.

[86] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors," in *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2001, pp. 144-155.

[87] D. L. Oppenheimer and M. R. Martonosi, "Performance Signatures: A Mechanism for Intrusion Detection," in *1997 IEEE Information Survivability Workshop*, San Diego, CA, USA, 1997.

[88] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A Secure Environment for Untrusted Helper Applications," in *6th USENIX Security Symposium*, San Jose, CA, USA, 1996, pp. 1-13.

[89] M. Bernaschi, E. Gabrielli, and L. V. Mancini, "Operating System Enhancements to Prevent the Misuse of System Calls," in *Conference on Computer and Communications Security*, Athens, Greece, 2000, pp. 174-183.

[90] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure Execution Via Program Shepherding," in *11th Annual Usenix Security Symposium*, San Francisco, CA, USA, 2002, pp. 191-206.

[91] G. McGary, "Bounds Checking Project," <http://gnu.mirror.widexs.nl/software/gcc/projects/bp/main.html> (Available January 2005).

[92] J. Wilander and M. Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," in *10th Network and Distributed System Security Symposium*, San Diego, CA, 2003, pp. 149-162.

[93] T. Toth and C. Kruegel, "Accurate Buffer Overflow Detection via Abstract Payload Execution," in *5th Symposium on Recent Advances in Intrusion Detection (RAID)*, Zurich, Switzerland, 2002, pp. 274-291.

[94] D. C. DuVarney, V. N. Venkatakrishnan, and S. Bhatkar, "SELF: A Transparent Security Extension for ELF Binaries," in *2003 Workshop on New Security Paradigms*, Ascona, Switzerland, 2003, pp. 29-38.

[95] "Hardened Gentoo," <http://www.gentoo.org/proj/en/hardened/> (Available February 2004).

[96] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer, "Architecture Support for Defending Against Buffer Overflow Attacks," in *Workshop on Evaluating and Architecting System dependability (EASY)*, San Jose, CA, USA, 2002.

[97] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi, "Enlisting Hardware Architecture to Thwart Malicious Code Injection," in *Security in Pervasive Computing*, Boppard, Germany, 2003, pp. 237-252.

[98] H. Ozdoganoglu, C. E. Brodley, T. N. Vijaykumar, B. A. Kuperman, and A. Jalote, "SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address," Purdue University, Technical Report TR-ECE 03-13, November 22, 2003.

[99] D. Ye and D. Kaeli, "A Reliable Return Address Stack: Microarchitectural Features to Defeat Stack Smashing," in *Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, Boston, MA, USA, 2004, pp. 69-76.

[100] M. Corliss, E. C. Lewis, and A. Roth, "Using DISE to Protect Return Addresses from Attack," in *Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, Boston, MA, USA, 2004, pp. 61-68.

[101] K. Inoue, "Energy-Security Tradeoff in a Secure Cache Architecture Against Buffer Overflow Attacks," in *Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, Boston, MA, USA, 2004, pp. 77-85.

[102] Z. Shao, Q. Zhuge, Y. He, and E. H.-M. Sha, "Defending Embedded Systems Against Buffer Overflow via Hardware/Software," in *19th Annual Computer Security Applications Conference (ACSAC 2003)*, Las Vegas, NV, USA, 2003, pp. 352-363.

[103] N. Tuck, B. Calder, and G. Varghese, "Hardware and Binary Modification Support for Code Pointer Protection from Buffer Overflow," in *37th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Portland, OR, USA, 2004, pp. 209-220.

[104] D. Keen, F. Lim, F. T. Chong, P. Devanbu, Matthew Farrens, P. Sultana, C. Zhuang, and R. Rao, "Hardware Support for Pointer Safety in Commodity Microprocessors," UC Davis, Technical Report CSE-2002-1, 2002.

[105] D. Kirovski, M. Drinic, and M. Potkonjak, "Enabling Trusted Software Integrity," in *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, San Jose, CA, USA, 2002, pp. 108-120.

[106] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks with Instruction-set Randomization," in *10th ACM Conference on Computer and Communication Security*, Washington, DC, USA, 2003, pp. 272-280.

[107] G. E. Suh, J. W. Lee, and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking," in *11th Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, USA, 2004, pp. 85-96.

[108] J. R. Crandall and F. T. Chong, "Minos: Control Data Attack Prevention Orthogonal to Memory Model," in *37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Portlant, OR, USA, 2004, pp. 221-232.

[109] M. Milenkovic, A. Milenkovic, and E. Jovanov, "A Framework For Trusted Instruction Execution Via Basic Block Signature Verification," in *42nd Annual ACM Southeast Conference*, Huntsville, AL, USA, 2004, pp. 191-196.

[110] M. Milenkovic, A. Milenkovic, and E. Jovanov, "Using Instruction Block Signatures to Counter Code Injection Attacks," in *Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, Boston, MA, USA, 2004, pp. 104-113.

[111] M. Drinic and D. Kirovski, "A Hardware-Software Platform for Intrusion Prevention," in *37th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Portland, OR, USA, 2004, pp. 233-242.

[112] A. Mahmood and E. J. McCluskey, "Concurrent Error Detection Using Watchdog Processors - A Survey," *IEEE Transactions on Computers*, Vol. 37, February 1988, pp. 160-174.

[113] K. Wilken and J. P. Shen, "Continuous Signature Monitoring: Low-cost Concurrent Detection of Processor Control Errors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 9, June 1990, pp. 629-641.

[114] J. Ohlsson and M. Rimen, "Implicit Signature Checking," in *25th International Symposium on Fault-Tolerant Computing*, Pasadena, CA, USA, 1995, pp. 218-227.

[115] S. Kim and A. K. Somani, "On-Line Integrity Monitoring of Microprocessor Control Logic," *Microelectronics Journal*, Vol. 32, December 2001, pp. 999-1007.

[116] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control Flow Checking by Software Signatures," *IEEE Transactions on Reliability*, Vol. 51, March 2002, pp. 111-122.

[117] M. K. Joseph and A. Avizienis, "A Fault Tolerance Approach to Computer Viruses," in *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 1988, pp. 52-58.

[118] G. I. Davida, Y. G. Desmedt, and B. J. Matt, "Defending Systems Against Viruses Through Cryptographic Authentication," in *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 1989, pp. 312 -318.

[119] D. Lie, C. Thekkath, M. Mitchell, P. Lincolny, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," in *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, USA, 2000, pp. 168-177.

[120] D. Lie, J. Mitchell, C. A. Thekkath, and M. Horowitz, "Specifying and Verifying Hardware for Tamper-Resistant Software," in *IEEE Conference on Security and Privacy*, Berkeley, CA, USA, 2003, pp. 166-177.

[121] C. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation-Tools for Software Protection," *IEEE Transactions on Software Engineering*, Vol. 28, August 2002, pp. 735-746.

[122] B. Horne, L. R. Matheson, C. Sheehan, and R. E. Tarjan, "Dynamic Self-checking Techniques for Improved Tamper Resistance," in *Digital Rights Management Workshop*, Philadelphia, PA, USA, 2001, pp. 141-159.

[123] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski, "Oblivious Hashing: A Stealthy Software Integrity Verification Primitive," in *Information Hiding: 5th International Workshop*, Noordwijkerhout, Netherlands, 2002, pp. 400 - 414.

[124] M. Jochen, L. Marvel, and L. L. Pollock, "A Framework for Tamper Detection Marking of Mobile Applications," in *14th International Symposium on Software Reliability Engineering (ISSRE)*, Denver, CO, USA, 2003, pp. 143-153.

[125] TCG, "TCG Specification: Architecture Overview," <https://www.trustedcomputinggroup.org/> (Available January 2005).

[126] NIST, "FIPS PUB 197: Advanced Encryption Standard (AES)," 2001.

[127] "Intel XScale® Core Developer's Manual," <http://www.intel.com/design/intelxscale/> (Available December 2004).

[128] L. Gwennap, "Digital 21264 Sets New Standard," *Microprocessor Report* October 1996, pp. 11-16.

[129] R. Sprugnoli, "Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets," *Communications of the ACM*, Vol. 20, November 1977, pp. 841-850.

[130] E. A. Fox, Q. F. Chen, and L. S. Heath, "A Faster Algorithm for Constructing Minimal Perfect Hash Functions," in *15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Copenhagen, Denmark, 1992, pp. 266-273.

[131] A. Milenkovic, M. Milenkovic, and J. Kulick, "A Compression of Branch Instruction Traces and Basic Block Length Analysis for SPEC CPU2000," LaCASA Lab, University of Alabama in Huntsville, Technical Report, 2002.

[132] "SQL Injection," <http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf> (Available January 2005).

[133] TIS, "Executable and Linking Format (ELF) Specification," <http://x86.ddj.com/ftp/manuals/tools/elf.pdf> (Available January 2005).

[134] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, Vol. 35, February 2002, pp. 59-67.

[135] A. Milenkovic and M. Milenkovic, "Exploiting Streams in Instruction and Data Address Trace Compression," in *IEEE 6th Annual Workshop on Workload Characterization*, Austin, TX, 2003, pp. 99-107.

[136] A. Milenkovic and M. Milenkovic, "Stream-Based Trace Compression," *Computer Architecture Letters*, Vol. 2, September 2003.

[137] SPEC, "SPEC 2000 CPU Benchmark Suite," February 2004, <http://www.spec.org> (Available February 2004).

[138] "Enhanced AES (Rijndael) IP Core," <http://www.asics.ws> (Available December 2004).

[139] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, USA, 2001.

[140] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *IEEE Micro*, Vol. 30, December 1997, pp. 330-335.

[141] I. Branovic, R. Giorgi, and E. Martinelli, "A Workload Characterization of Elliptic Curve Cryptography Methods in Embedded Environments," *ACM SIGARCH Computer Architecture News*, Vol. 32, June 2004, pp. 27-34.