DESIGNING COST-EFFECTIVE SECURE PROCESSORS FOR EMBEDDED SYSTEMS: PRINCIPLES, CHALLENGES, AND ARCHITECTURAL SOLUTIONS

by

AUSTIN ROGERS

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

2010

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/her 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 Austin Rogers 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, 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 Austin Rogers

Title Designing Cost-Effective Secure Processors for Embedded Systems: Principles, Challenges, and Architectural Solutions

Computer security in embedded systems is becoming more and more important as these systems diversify and proliferate, with the cost of security violations ranging from loss of revenue to loss of life. This dissertation addresses the problem of computer security at the hardware level, proposing a sign-and-verify secure processor architecture to ensure integrity (preventing the execution or use of unauthorized instructions or data) and confidentiality (preventing the unauthorized copying of instructions or data). Integrity is ensured by signing blocks of instructions or data when they are created and then verifying them when they are used. Confidentiality is ensured by encryption. We thoroughly explore the design challenges of the secure processor architecture, including signature generation, signature placement, code and data encryption, verification latency reduction, and memory overhead reduction. We propose a number of architectural solutions to address these challenges. A cycle-accurate simulator is used to explore the secure processor design space and evaluate the proposed solutions. We also develop a prototype secure processor in actual hardware, implemented on an FPGA-based platform. Our simulation results show that the proposed solutions can ensure security without incurring prohibitive performance overhead, and our hardware implementation demonstrates that our architecture is feasible and practical.

Abstract Approval: Committee Chair _____

Department Chair _____

Graduate Dean _____

ACKNOWLEDGMENTS

"The law of the LORD is perfect, converting the soul: the testimony of the LORD is sure, making wise the simple. The statutes of the LORD are right, rejoicing the heart: the commandment of the LORD is pure, enlightening the eyes. The fear of the LORD is clean, enduring for ever: the judgments of the LORD are true and righteous altogether."

Psalms 19:7-9

Before beginning this dissertation, I must acknowledge all the LaCASA researchers, past and present, who have made this work possible: Dr. Milena Milenković, Dr. Emil Jovanov, and Chris Otto. But most of all, I wish to thank Dr. Aleksandar Milenković, my friend and advisor, whose knowledge, experience, assistance, and patience have been absolutely invaluable.

I must also thank Dynetics, my employer, who has generously paid for my graduate work and even funded trips to conferences so that I could present papers on this research. I would also like to thank Dr. Derek Bruening for providing a Perl script for plotting data in stacked columns.

Finally, I wish to dedicate this dissertation to the two wonderful ladies in my life: Huichen Venus Hung Rogers, my wife, and Brenda Lee Nixon Rogers, my mother. Without their steadfast love and support, and the rich blessings of God, I would never have been able to accomplish this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS AND UNITS

ADPCM	Adaptive Differential Pulse Code Modulation
AES	Advanced Encryption Standard
ARM	Advanced RISC Machine
BTB	Branch Target Buffer
CBC-MAC	Cipher Block Chaining Message Authentication Code
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DCOM	Data Confidentiality Only Mode
DICM	Data Integrity and Confidentiality Mode
DIOM	Data Integrity Only Mode
DLL	Dynamically Linked Library
DMA	Direct Memory Access
DRM	Digital Rights Management
ELF	Executable and Linkable Format
EVU	Encryption and Verification Unit
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GCM	Galois/Counter Mode
GHz	Gigahertz
GSM	Global Standard for Mobile

IP	Intellectual Property
IVB	Instruction Verification Buffer
JPEG	Joint Photographic Experts Group
KB	Kilobyte
LaCASA	Laboratory for Advanced Computer Architectures and Systems
MP3	MPEG-1 Audio Layer 3
MPEG	Moving Pictures Experts Group
OTP	One-Time Pad
PMAC	Parallelizable Message Authentication Code
RbV	Run Before Verified
RGB	Red, Green, and Blue
RISC	Reduced Instruction Set Computer
RSA	Rivest, Shamir, and Adleman
SCOM	Software Confidentiality Only Mode
SDRAM	Synchronous Dynamic Random Access Memory
SICM	Software Integrity and Confidentiality Mode
SIOM	Software Integrity Only Mode
SOPC	System-on-a-Programmable Chip
TIFF	Tagged Image File Format
TLB	Translation Lookaside Buffer
VHDL	Very-high-speed integrated circuit Hardware Description Language
WtV	Wait ‘Till Verified
XOR	Exclusive OR

CHAPTER 1

INTRODUCTION

Embedded computer systems have become ubiquitous in modern society. A wide range of applications rely on embedded systems, from consumer electronics, communications, transportation, medicine, to national security. Security breaches in embedded systems could thus have wide ranging impacts, from loss of revenue to loss of life. As these systems continue to proliferate, the potential damage that can be caused by security compromises increases. Methods for improving computer security are therefore highly desirable.

The problem of computer security is further compounded by the fact that methods for improving security tend to degrade performance or consume precious computational resources. As embedded systems often have stringent design constraints, security extensions for embedded systems must incur as little overhead as possible. This dissertation addresses these problems by exploring the subject of secure processors, which ensure security at the hardware level. We lay out the principles of secure processor design and asses the various challenges faced when designing such a processor. We explore the design space by evaluating different approaches to these challenges, and offer architectural solutions to alleviate the performance and complexity overhead incurred by adding security to the processor design. Finally, we prove the feasibility of

our security architectures by implementing security extensions for a soft-core processor on a low-cost field programmable gate array (FPGA).

1.1 Secure Processors: Motivation and Background

Computer security is a broad and dynamic field. Computer systems are subject to a broad range of attacks, and suffer from many vulnerabilities. According to the National Institute of Standards and Technology, 5,632 software vulnerabilities were identified in 2008 alone [1]; the number of attacks was much greater. Furthermore, the unauthorized copying of software, also known as piracy, is a major economic threat. The Business Software Alliance, in their annual piracy study [2], estimates that 41% of software in use worldwide during the year 2008 was unlicensed, with an economic impact of 53 billion dollars. These figures are increasing every year.

The vast majority of microprocessors are manufactured and sold for use in embedded systems. Indeed, only 2% of processors are manufactured as general-purpose computer processors [3]. That other 98% may be found in such diverse embedded system applications as coffeemakers, automobiles, cellular telephones, and intercontinental ballistic missiles. As the embedded market has evolved, many modern embedded systems now have some form of network connectivity, often to the internet itself. This exposes these systems to many of the same attacks that general-purpose systems suffer. Many embedded systems may also operate in hostile environments where they are subjected to physical attacks aimed at subverting system operation, extracting key secrets, or intellectual property theft. Similarly, a system may operate in harsh conditions such as outer space, where natural phenomena may compromise the integrity of data.

This dissertation focuses on the development of secure processors for ensuring the integrity and confidentiality of instructions and data in embedded systems. Integrity is violated whenever any unauthorized code is executed or unauthorized data used by a microprocessor. Confidentiality is violated whenever some entity, human or computer, is able to view, copy, or reverse-engineer the system. Integrity and confidentiality, along with a third concept, availability, comprise the computer security triad, which will be discussed in greater detail in Chapter 2. Our work only indirectly addresses availability, insofar as integrity and confidentiality concerns influence availability.

1.2 Principles and Challenges in Ensuring Software/Data Integrity and Confidentiality

This dissertation is intended to lay out the fundamental principles in designing a secure processor, and to explore the various challenges that must be addressed when designing such a processor. We use a cycle-accurate simulator to evaluate the performance overhead incurred by a secure processor, which is defined as the additional time required to execute a program on the secure processor as compared to a similar processor without security extensions. Our simulations explore many of the design choices that must be made when addressing design challenges. Furthermore, we have implemented a research prototype secure processor in actual hardware, demonstrating the practicality of our proposed security extensions.

The basic principle for ensuring integrity is the use of cryptographically sound signatures, that is, signatures that cannot be easily duplicated without the knowledge of certain secrets. Data must be signed when they are first stored, whether during installation (instructions and static data) or runtime (dynamic data). When data are used

at runtime, their signature must be fetched. The signature is also calculated independently based on the fetched data, and the fetched and calculated signatures are compared. If the signatures match, then the data can be trusted. If they do not match, then the data have been subjected to tampering and should not be used. A system implementing such a scheme is called a sign-and-verify system, as it signs data upon their creation and then verifies data upon their use.

The basic principle for ensuring confidentiality is the use of strong encryption. The goal of encryption is to render sensitive data illegible to any party lacking certain secrets. Thus, for full protection of both integrity and confidentiality, data must be both signed and encrypted before being stored, and both decrypted and verified before use. Signature generation and encryption may or may not be intertwined; some cryptographic schemes perform both at the same time while others perform them independently.

Although these basic principles may seem simple, many challenges arise when implementing them in an actual system. For instance, how should one go about performing encryption and decryption? How should one calculate signatures? Where should signatures be stored? How will performance be affected, and what can be done to improve the performance of secure systems? Furthermore, in addition to integrity and confidentiality, how can we be sure that a chunk of dynamic data is up-to-date? How much memory will signatures and other additional data require, and can it be minimized? What are the trade-offs between performance and on-chip complexity? All of these issues and more are addressed in this dissertation.

1.3 Main Contributions and Findings

The main contributions of this dissertation are as follows:

- We establish a framework for the protection of instructions, static data, and dynamic data using a sign-and-verify architecture with optional encryption.
- We explore the design space in several areas, including issues such as where to store signatures, how to perform encryption, and what cryptographic mode to use for calculating signatures.
- We present several enhancements to reduce performance and memory overhead, including signature victim caches, the instruction verification buffer, and protecting multiple blocks of data with one signature.
- We propose a tree-like structure to ensure that protected dynamic data are up-to-date without adversely impacting performance.
- We develop a cycle-accurate simulator to evaluate the performance overhead of the secure architecture.
- We use the simulator to investigate the effects of the various design choices and enhancements on the performance overhead of the secure architecture and establish an analytical model for performance overhead.
- We implement a secure processor in actual hardware using existing system-on-a-programmable chip (SOPC) technologies.

Our main finding is that integrity and confidentiality can be implemented with low performance overhead by using established cryptographic modes and latency-hiding architectural enhancements. These enhancements, however, add complexity, so system designers must make trade-offs between complexity and performance. We also

demonstrate the practicality and feasibility of our security enhancements by augmenting an existing soft-core processor with a subset of our proposed mechanisms and implementing them on an FPGA platform.

1.4 Outline

The remainder of this dissertation is organized as follows. Chapters 2 and 3 present basic background material to aid in understanding the remainder of the dissertation, describing several types of threats to computer security and various cryptographic concepts that we use to counter those threats. Chapter 4 introduces the general principles of secure processor design, but does not concern itself with particulars or optimizations. The subsequent two chapters address the various choices and challenges that a computer architect will face when implementing a secure processor and introduce architectural enhancements to help overcome these challenges, including trade-offs between performance, security, and complexity. Chapter 5 is concerned with general issues that apply to protecting both instructions and data, while Chapter 6 focuses on the special issues that arise from protecting dynamic data. Chapter 7 describes our simulation methodology for evaluating secure processor designs, and includes simulation results for many of the choices presented in the preceding chapters, as well as an analytical model for secure processor performance. Chapter 8 details our implementation of a prototype secure processor using an FPGA-based platform. Related work is presented in Chapter 9, and Chapter 10 concludes the dissertation.

CHAPTER 2

BACKGROUND: COMPUTER SECURITY

Notions of computer security have evolved greatly over the years. At one time, most systems were standalone; computer security meant locking the computer room. As computer networking (and especially the Internet) became more prominent, computer systems could come under attack from the other side of the building, or from the other side of the world. These attacks are mainly what we call software attacks, where the attacker has access to a system, either directly or over a network. Once embedded systems began to proliferate, they became subject to further vulnerabilities such as physical attacks, where the attacker has physical access to the system but not necessarily software access. More sophisticated computer hardware and software enables side-channel attacks, in which the attacker attempts to gain knowledge about the system by indirect analysis.

This chapter provides the relevant background information on computer security that is necessary for a clear understanding of the remainder of the dissertation. We first examine the security triad of integrity, confidentiality, and availability. We then examine the types of attacks mentioned above, including software-based attacks, physical attacks, and side-channel attacks.

2.1 The Computer Security Triad

Computer security broadly consists of three concepts: integrity, confidentiality, and availability, which together constitute the well-known computer security triad. Our research applies these concepts at the microprocessor level, with the goal of designing truly secure processors. The first two concepts, integrity and confidentiality, are most relevant to our research.

Protecting integrity means that the processor will not execute any unauthorized code or use any unauthorized data; any tampering should be detected. Attacks against integrity may be deliberate, such as those performed directly by a human intruder or an automated attack set up by a human. Integrity may also be compromised by harsh environmental factors, such as bit flipping caused by radiation in systems operating in outer space, or due to aggressive semiconductor technologies whose low swing voltages make them vulnerable to noise.

Protecting confidentiality means that instructions and data must be illegible to all unauthorized entities, be they human or machine. Attacks against confidentiality include piracy and identity theft, both of which have huge economic impacts [2]. Ensuring confidentiality is a concern in a wide variety of areas. Digital rights management (DRM) is primarily concerned with ensuring confidentiality in consumer markets. Corporations and governments may also be concerned with protecting confidentiality to prevent espionage.

Availability requires that a system be available to legitimate users when needed. Attacks against availability try to render a system inaccessible. A classic example of such an attack is the denial of service attack, which attempts to consume a system's

resources so that it is unavailable to other users. Ensuring availability can include techniques such as error correction and fault recovery, as well as algorithms and heuristics for detecting malicious access patterns. Attacks on integrity and confidentiality may also be part of an attack on availability. Our research is primarily focused on ensuring integrity and confidentiality; we only address availability insofar as it is influenced by integrity and confidentiality concerns.

2.2 Software Attacks

Software attacks require the attacker to have some form of access to the target computer system. This could be direct access, with a lower permission level than the attacker desires. The access could also be across a network, which would require the attacker to sniff the system's open ports, looking for services with known vulnerabilities. The goal of software attacks is to modify a running program by injecting and executing code. The foreign instructions must be injected into memory, and then the return address of the currently executing function must be overwritten to force the processor to execute the injected instructions. These attacks are only briefly documented here; a more detailed treatment can be found in [4].

2.2.1 Buffer Overflow Attacks

A common class of attacks is buffer overflow. These attacks take advantage of I/O instructions that simply store incoming data to a buffer, without bothering to check to see if the amount of incoming data will exceed the buffer size. After the buffer fills, memory locations beyond the buffer are overwritten. Most systems have stacks that grow counter to memory address growth. If the buffer is on the stack, then this attack can

overwrite the data at any address on the stack beyond the buffer with malicious instructions. This overwrite includes the return address, allowing the attacker to divert the program to the newly injected instructions. If the buffer is on the heap near a function pointer, then the attacker's goal is to inject code and overwrite that function pointer.

2.2.2 Format String Attacks

Format string attacks take advantage of *printf*-family of functions that take a format string as an input. These functions will accept any pointer and interpret the contents of memory at that address as a format string. By skillfully manipulating the inputs passed to the *printf* function, the attacker can read from any address in memory. The *%n* format character presents an additional vulnerability. This character causes a *printf* function to write the number of characters output by the function before it reached *%n* to a specified address. A skillful attacker could use this to write an arbitrary integer to any address.

2.2.3 Integer Error Attacks

Errors arising from integer operations cannot be used as a direct attack. However, integer errors can facilitate other forms of attacks. For instance, an unsigned integer overflow can result in a smaller number than expected. If this is used to allocate a buffer, then the buffer will also be smaller than expected. This exposes the system to a buffer overflow attack, even if subsequent input operations using that buffer check input length. A more thorough treatment of integer error attacks may be found in [5].

2.2.4 Dangling Pointer Attacks

Dangling pointers become an issue if the *free* function is called twice for the same pointer. The vulnerability arises from the way that the GNU C library handles memory allocation [6]. When a chunk of memory is freed, it is inserted into a doubly linked list of free chunks. If *free* is called twice, the pointers to the next and previous entries may wind up pointing back to the same chunk. An attacker may write malicious code to the chunk's data area and put a pointer to that code in place of the pointer to the previous list entry. If that chunk is allocated again, the memory manager will try to unlink the chunk from the list, and will write the attacker's pointer to an address calculated from the pointer to the next entry. If that address happens to contain a function's return address, then a successful attack has been accomplished.

2.2.5 Arc-Injection Attacks

An arc-injection or "return-into-libc" involves overwriting a return address such that control flow is disrupted. Oftentimes the address of a library function is used. Library system calls can be used to spawn other processes on the system with the same permissions as the compromised program. If the operating system (OS) itself is compromised, then the attacker can run a malicious program that will have the ability to access any and every memory location.

2.3 Physical Attacks

In contrast to software attacks, physical attacks involve tampering with the actual computer hardware [7]. Probes are often inserted on the address and data bus, allowing the attacker to monitor all transactions and override data coming from memory with

his/her own data. This is a tool often used in industrial and military espionage. This section describes three such attacks: spoofing, splicing, and replay.

2.3.1 Spoofing Attacks

A spoofing attack occurs when an attacker intercepts a request for a block of memory, and then manually supplies a block of his/her choice. This block may contain either data or instructions of a malicious nature. In an unsecured system, the processor naïvely conducts a bus cycle, and is unaware that the data it received came from an attacker rather than from main memory. Figure 2.1 illustrates a spoofing attack. The processor initiates a bus read cycle for a block at memory location DB_j. The attacker intercepts the request and supplies a potentially malicious block M_j instead of the correct block DB_j.

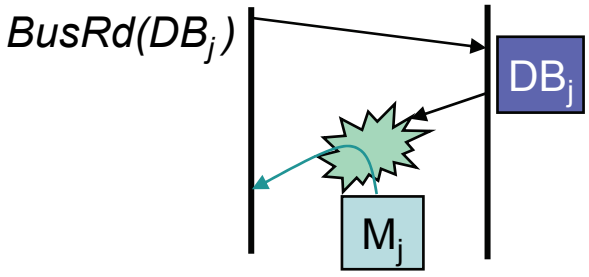


Figure 2.1 Spoofing Attack

2.3.2 *Splicing Attacks*

Splicing attacks involve intercepting a request for a block of memory and then supplying the data from a different block. The supplied block is a valid block from somewhere in the address space, but it is not the actual block that the processor requested. This attack may be performed with either data or instruction blocks. Once again, the unsecured processor is unaware that it has received the incorrect memory block. Figure 2.2 depicts a splicing attack. The processor initiates a bus read cycle for a block at memory location DB_j. The attacker intercepts the request and supplies a valid block from memory, but from address DB_i rather than the desired address, DB_j.

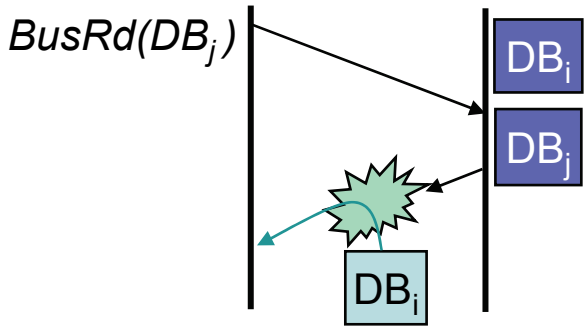


Figure 2.2 Splicing Attack

2.3.3 *Replay Attacks*

In a replay attack, the attacker intercepts a request for a block of memory, and then supplies an older copy of that block. This is a concern for dynamic data blocks and instructions generated at runtime, such as self-modifying code and just-in-time

compilation. The supplied block was valid at some point in the past, but now it may be obsolete. Figure 2.3 illustrates a replay attack. The processor initiates a bus read cycle for the data block at address DB_j. The attacker intercepts the request and returns an older version of that block, DB_j^*, which may be different from the current version in memory.

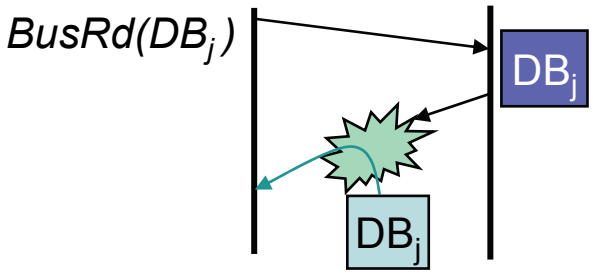


Figure 2.3 Replay Attack

2.4 Side-Channel Attacks

Side-channel attacks attempt to gather information about a system or program via indirect analysis. These attacks involve first collecting information about the system and then analyzing that information in an attempt to deduce the system's secrets [8]. The information gathering stage requires some form of access to the system. The attacker may have direct physical access to the system and its components, or have some level of privileges to run programs on the target system. In this section, we briefly describe a few examples of the myriad possible side-channel attacks, including timing analysis, differential power analysis, fault exploitation, and architectural exploitation.

2.4.1 Timing Analysis

Timing attacks are, perhaps, the simplest type of side-channel attacks, taking advantage of the fact that different operations require different amounts of time to execute. Kocher [9] illustrates how this can be used to break cryptographic algorithms, given a known algorithm and either known plaintext or known ciphertext. He uses timing analysis to determine the secret exponent in the Diffie-Hellman algorithm, factor RSA private keys, and determine the private key used by the Digital Signature Standard algorithm.

2.4.2 Differential Power Analysis

A microprocessor's power consumption at any given moment can indicate what operations it is performing. A differential power analysis can be used to determine what instructions are executed and when. Kocher *et al.* [10] discuss how to break a known, data-driven encryption algorithm using such an attack. Instantaneous CPU power consumption is measured at intervals during a cryptographic operation, forming a trace. Multiple traces can be compiled and compared, revealing patterns produced by the execution of certain instructions. Since the encryption algorithm is both known and data-driven, the data being processed can be revealed solely from the power traces.

2.4.3 Fault Exploitation

A fault exploitation attack takes advantage of hardware faults to discover secrets. These hardware faults may be transiently occurring within the processor, or induced externally. Boneh *et al.* [11] describe a simple fault exploitation attack, whereby the modulus used by an RSA algorithm may be calculated. A signature must be calculated

from the same data two times. One signature is calculated without a hardware fault. The second is calculated in the presence of a hardware fault, either transient or induced. The modulus of the RSA system can then be factored by analyzing the difference between the two signatures. Boneh *et al.* go on to break even more sophisticated cryptographic schemes using similar techniques.

2.4.4 Architectural Exploitation

Due to the well-known effect of Moore's Law, microprocessor designers have been able to introduce more and more advanced features. Sometimes these advanced features may be exploited to reveal information about the processor. A prime example of an architectural exploitation attack is the Simple Branch Prediction Analysis attack devised by Aciiçmez *et al.* [12]. This attack expands on the classical timing attack by taking advantage of the branch prediction unit and multi-threading capabilities of the Pentium 4 processor. A spy process is executed in parallel with a process performing a known cryptographic algorithm. The spy process executes branch instructions, flooding the processor's branch target buffer (BTB), while measuring the execution time required for those branch instructions. When the cryptographic process executes a branch instruction that results in the branch not being taken, no BTB eviction is needed. Thus, the next time the spy process executes a corresponding branch, it will execute quickly, thereby revealing that the cryptographic process had a branch not taken. Conversely, a taken branch in the cryptographic process results in a BTB eviction, which in turn causes a spy process branch to take longer to execute, revealing that the cryptographic process had a taken branch. The recorded trace of branches that were taken and not taken can then be used to deduce the cryptographic secret key. This attack relies on detailed

information about the underlying hardware and software, but such information is often available and can be obtained using microbenchmarks [13].

Cache misses provide another avenue for hardware exploitation. Percival [14] demonstrates that a thread running concurrently with a cryptographic thread can pollute the cache and time its own cache accesses, thus detecting which cache lines were evicted by the cryptographic thread. With knowledge of the implementation of the RSA algorithm, these eviction patterns may be analyzed to determine the encryption key. Bernstein [15] attacks the Advanced Encryption Standard (AES) algorithm simply by observing algorithm run time. The attack involves first running the algorithm on a large set of inputs with a known key, and then running it for another large set of inputs with an unknown key. Cache misses caused by table lookups in the AES algorithm cause discrepancies in run time, which may then be analyzed to determine the unknown key.

CHAPTER 3

BACKGROUND: CRYPTOGRAPHIC CONCEPTS

This chapter provides background information on various cryptographic concepts. As with the previous chapter, familiarity with these concepts will greatly help in understanding the remainder of this dissertation. We begin with a discussion on encryption methods for ensuring confidentiality. We then discuss methods for ensuring integrity, focusing on two methods for generating signatures. Finally, we discuss strategies for integrating the protection of both integrity and confidentiality.

3.1 Ensuring Confidentiality

Confidentiality is traditionally ensured by using some form of encryption. The straightforward method is to use symmetric key cryptography, where the same key is used for both encryption and decryption. A commonly used symmetric key cryptography scheme is the AES cipher [16]. The AES operation is what is known as a block cipher; it operates on a 128-bit block of data using either a 128-bit, 192-bit, or 256-bit key. AES can operate in either encryption mode (Equation (3.1) and Figure 3.1(a)) to produce a chunk of unintelligible ciphertext C_i from a chunk of plaintext I_i, or in decryption mode (Equation (3.2) and Figure 3.1(b)) to produce a chunk of plaintext I_i from a chunk of ciphertext C_i. Note that the sizes of I_i, P_i, and K are determined by the width of the AES

unit being used; throughout this dissertation, a 128-bit AES unit is assumed unless otherwise stated.

$$C_i = AES_{e,K}(I_i) \tag{3.1}$$

$$I_i = AES_{d,K}(C_i) \tag{3.2}$$

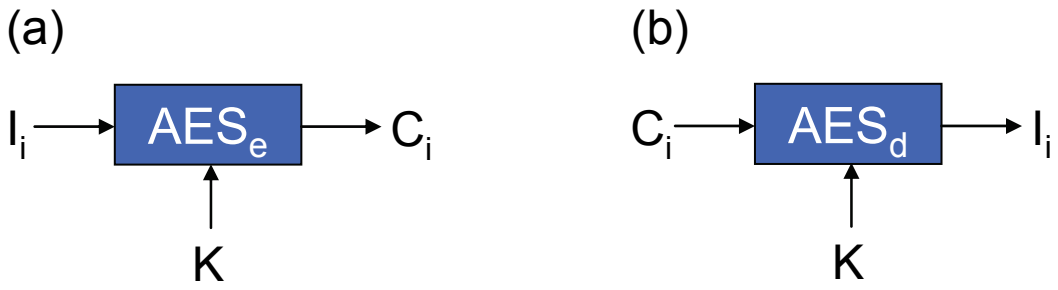


Figure 3.1 Data Flow of Symmetric Key (a) Encryption and (b) Decryption

Symmetric key cryptography is conceptually simple, but can introduce significant cryptographic latency in some applications. If the application is such that data must be fetched or are being received in a stream, performance could be improved by somehow overlapping cryptographic latency with data retrieval. This can be accomplished by using a one-time-pad (OTP) cryptographic scheme. An OTP scheme using AES encryption is shown in Equations (3.3) and (3.4) and Figure 3.2. For either encryption or decryption, a secure pad is calculated by encrypting an initial vector P_i using a key K. One requirement of OTP is that the secure pad should always be unique (i.e., each pad is only used "one time"). Therefore, for a fixed key K, a unique initial vector P_i should be used for each encryption. The ciphertext C_i is then produced by performing an exclusive

or (XOR) of the secure pad and the plaintext I_i. Conversely, taking the XOR of the pad with the ciphertext C_i will produce the plaintext I_i. The circles with crosses inside them in the figures below, as well as in figures throughout the remainder of this dissertation, represent a 128-bit array of XOR gates.

$$C_i = I_i \; xor \; AES_K(P_i) \tag{3.3}$$

$$I_i = C_i \; xor \; AES_K(P_i) \tag{3.4}$$

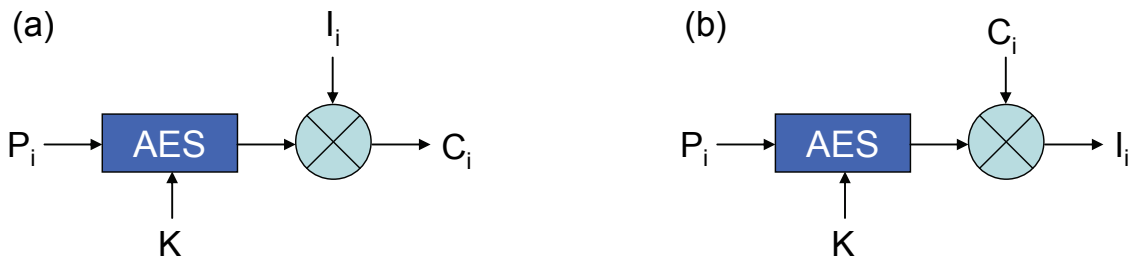


Figure 3.2 Data Flow of One Time Pad (a) Encryption and (b) Decryption

Using OTP cryptography may hide cryptographic latency in some applications. It also only requires the use of the AES cipher in encryption mode, allowing for simpler hardware and/or software AES implementations. Therefore, in Figure 3.2 and throughout the remainder of this dissertation, any instance of AES without the subscript *e* or *d* will indicate an AES encryption operation.

3.2 Ensuring Integrity

Message authentication codes, also known as signatures, are the traditional method for ensuring data integrity. The basic concept is to sign a chunk of data, broadly known as a "message," in such a way that its signature attests to its integrity. The message text and other parameters may be used as inputs into the signature generation process, as appropriate. When the message's integrity needs to be verified, its signature is recalculated and compared to the original signature. The signature generation methodology must be such that any tampering with the message will result in a mismatch between the new and original signatures.

One well-known method for calculating signatures is the cipher block chaining message authentication code (CBC-MAC) [17]. As its name implies, CBC-MAC signatures are calculated via a chain of cryptographic operations. A CBC-MAC implementation using the AES block cipher is depicted in Equation (3.5) and Figure 3.3. In the equation and figure, a signature S is calculated for an arbitrary number of data chunks, I_1 - I_N. An initial vector P is first encrypted using a key K_1. The result is then XORed with the first data chunk, I_1, and encrypted using another key K_2. The result of that encryption is then XORed with the second data chunk, I_2, and encrypted using K_2. These operations continue until the final data chunk, I_N, has been XORed with the final intermediate value and encrypted using K_2.

$$S = AES_{K2}(I_N \; xor \ldots AES_{K2}(I_2 \; xor \; AES_{K2}(I_1 \; xor \; AES_{K1}(P)))\ldots) \tag{3.5}$$

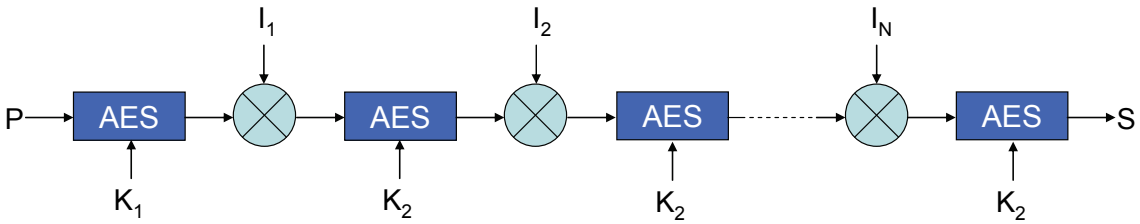


Figure 3.3 Data Flow of Signature Generation Using Cipher Block Chaining

CBC-MAC can produce cryptographically sound signatures [18], but due to its chaining nature, all operations must be performed in series. If the data chunks become available at intervals equal to or greater than the time required to perform an AES operation, then CBC-MAC may be a good cipher choice. However, if the data chunks become available more quickly, CBC-MAC may potentially introduce long signature generation latencies.

Black and Rogaway [19] developed the parallelizable message authentication code (PMAC) algorithm to address the latency issue. The PMAC cipher, which is proven to be secure [19], calculates the signature for each chunk of data in parallel and XORs these together to form the final signature, as demonstrated in Equations (3.6) and (3.7) and Figure 3.4. Each data chunk I_i has an associated initial vector P_i, which is encrypted using a key K_1. The result is XORed with the data chunk I_i and encrypted using K_2 to produce that chunk's signature $Sig(I_i)$. Each of these signatures may be calculated in parallel, and are XORed together to produce the message's overall signature S. The PMAC approach is ideal if multiple AES operations can be performed at one time, or if the AES operations can be pipelined.

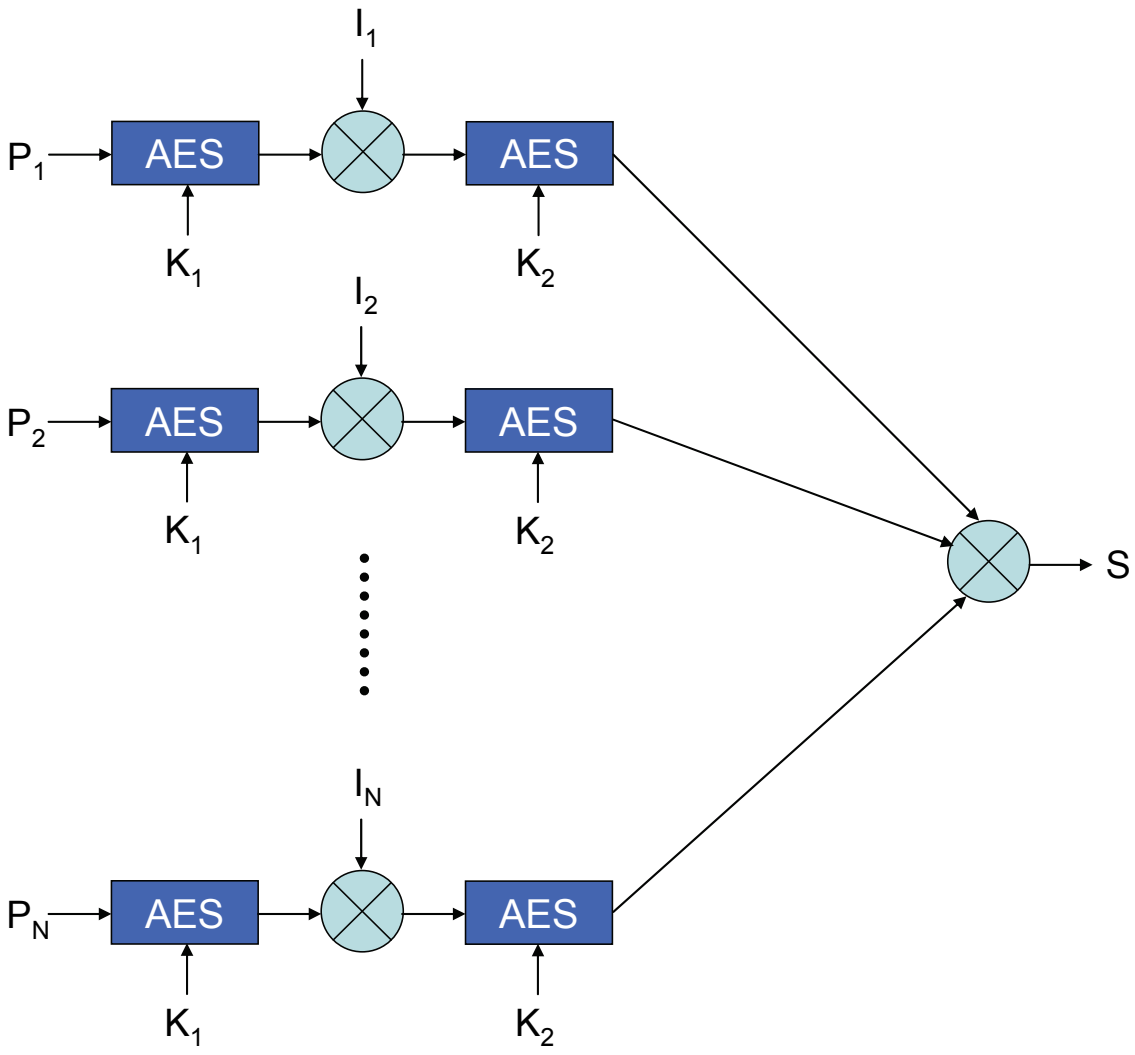


Figure 3.4 Data Flow of Signature Generation Using

Parallelizable Message Authentication Code

$$Sig(I_i) = AES_{K2}(I_i \ xor \ AES_{K1}(P_i)) \quad for \ i = 1..N \tag{3.6}$$

$$S = Sig(I_1) \ xor \ Sig(I_2) \ xor \ ... \ Sig(I_N) \tag{3.7}$$

3.3 Integrating Integrity and Confidentiality

When both integrity and confidentiality are to be protected, one must choose the order in which to calculate signatures and/or perform encryption. Variations in the order in which signing and encryption are performed give rise to three known approaches: encrypt&sign (ES), encrypt, then sign (EtS), and sign, then encrypt (StE) [20]. The high-level data flow for each of these approaches is shown in Figure 3.5, along with the degenerate case of signature generation without any encryption. The ES scheme involves encryption and signature generation performed in parallel; the signature is calculated on the plaintext, which is also encrypted. The EtS scheme requires that the plaintext be encrypted first; the signature is then calculated on ciphertext. The StE scheme calculates the signature on plaintext, and then encrypts both the plaintext and the signature.

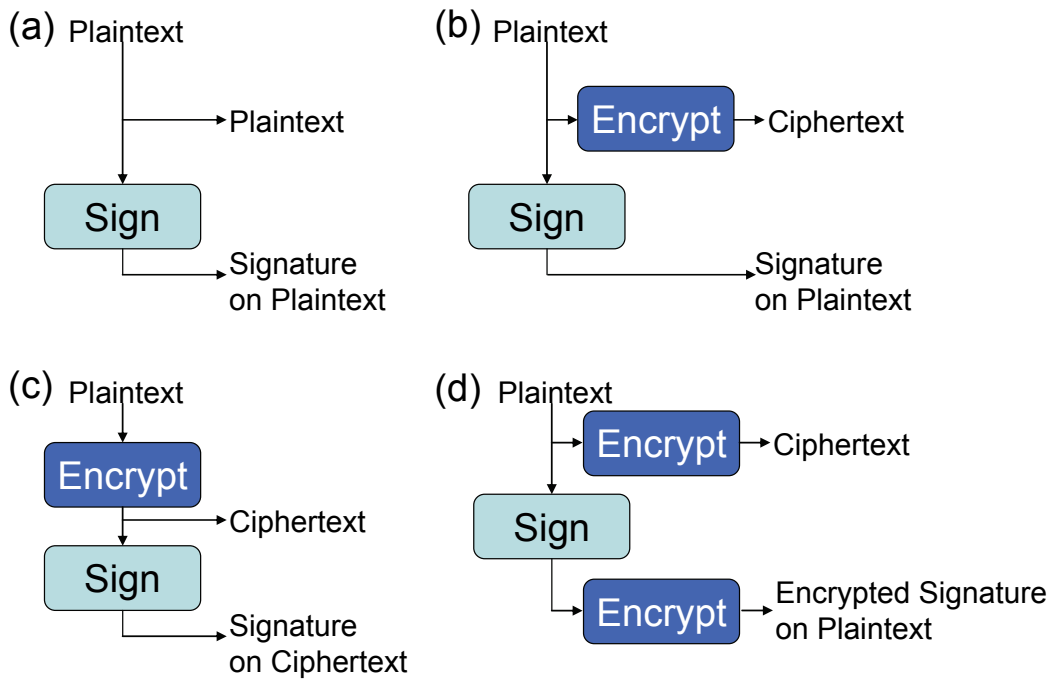


Figure 3.5 Approaches to Encryption and Signing: (a) Signed Plaintext, (b) ES, (c) EtS, and (d) StE

These schemes are further illustrated using actual data in Figure 3.6. The plaintext in this case is a 64 byte binary instruction block, encoded for the ARM architecture and expressed in hexadecimal. This block is assumed to reside in memory beginning at address 3000a80, and its 128-bit signature is stored immediately following the block at address 3000ac0. This block's confidentiality is ensured using OTP encryption, and its integrity is ensured using the PMAC method. The initial vectors P_i for each 128-bit sub-block consist of the sub-block's address left padded with zeros; these are used for both encryption and signature generation. *Key1* and *Key2* are used for signature generation, while *Key3* is used for encryption.

Key1: 0123456789abcdef012345678abcdef0			
Key2: fedcba9876543210fedcba9876543210			
Key3: 02132435465768798a9bacbdcedfe0f1			
P_1: 00000000000000000000000003000a80			
P_2: 00000000000000000000000003000a90			
P_3: 00000000000000000000000003000aa0			
P_4: 00000000000000000000000003000ab0			
P_{sig}: 00000000000000000000000003000ac0			
3000a80: e3a02000	3000a80: 09389787	3000a80: 09389787	3000a80: 09389787
3000a84: e50b2030	3000a84: ec965efc	3000a84: ec965efc	3000a84: ec965efc
3000a88: e59f122c	3000a88: 2e33ac4e	3000a88: 2e33ac4e	3000a88: 2e33ac4e
3000a8c: e5812000	3000a8c: 4885154b	3000a8c: 4885154b	3000a8c: 4885154b
3000a90: e50b2034	3000a90: ba26d576	3000a90: ba26d576	3000a90: ba26d576
3000a94: e1a06000	3000a94: f15f6ea5	3000a94: f15f6ea5	3000a94: f15f6ea5
3000a98: e59f0220	3000a98: 453cdd9c	3000a98: 453cdd9c	3000a98: 453cdd9c
3000a9c: eb002c5b	3000a9c: 40af6677	3000a9c: 40af6677	3000a9c: 40af6677
3000aa0: e2505000	3000aa0: 105aa547	3000aa0: 105aa547	3000aa0: 105aa547
3000aa4: 0a000033	3000aa4: f1b7f562	3000aa4: f1b7f562	3000aa4: f1b7f562
3000aa8: e1a00005	3000aa8: 689b2016	3000aa8: 689b2016	3000aa8: 689b2016
3000aac: e3a0102f	3000aac: e6a28d0e	3000aac: e6a28d0e	3000aac: e6a28d0e
3000ab0: eb004ad2	3000ab0: 94d4e3f1	3000ab0: 94d4e3f1	3000ab0: 94d4e3f1
3000ab4: e3500000	3000ab4: 10e25c31	3000ab4: 10e25c31	3000ab4: 10e25c31
3000ab8: 0a000004	3000ab8: 7f00577b	3000ab8: 7f00577b	3000ab8: 7f00577b
3000abc: e59f3200	3000abc: 31cd649d	3000abc: 31cd649d	3000abc: 31cd649d
3000ac0: 094a0eb7	**3000ac0: 094a0eb7**	**3000ac0: d2acbdef**	**3000ac0: 5a07eb1d**
3000ac4: be78f193	**3000ac4: be78f193**	**3000ac4: bca24992**	**3000ac4: b6db16db**
3000ac8: e2ee9fc4	**3000ac8: e2ee9fc4**	**3000ac8: c6028b0c**	**3000ac8: 48269248**
3000acc: 11dc5edb	**3000acc: 11dc5edb**	**3000acc: 440b6d3f**	**3000acc: 7f1d8ba2**
(a)	(b)	(c)	(d)

Figure 3.6 Signed Binary Data Block: (a) Signed Plaintext, (b) ES, (c) EtS, and (d) StE

The relative strength of these implementations is still a subject for debate [20, 21]. However, another cryptographic algorithm has been developed that specifically address the need for both encryption/decryption and signature generation. The Galois/Counter Mode (GCM) of cryptographic operation was introduced by McGrew and Viega [22]. It implements an EtS cryptographic scheme, incorporating the block cipher of choice (for this dissertation, we continue to use AES). The goal of GCM was to provide a secure yet flexible solution with low latency. With appropriate inputs, GCM is as secure as the chosen underlying block cipher [22].

The flexibility of GCM is apparent in Figure 3.7, which shows a high-level "black box" view of the inputs and outputs of GCM. The inputs include an arbitrary number of

data blocks ($I_1 - I_N$). An additional data block (*ADD_DATA*) may also be inputted; these data will be used for signature generation but will not be encrypted. GCM requires only two cryptographic keys, K_1 and K_2. Key K_1 is used for encryption, and K_2 is used for signature generation. An initial vector *IV* is also required. The initial vector must be a nonce, that is, it should be unique with a high probability. The size of *IV* is a design parameter, but 96 bits is recommended as the most efficient when working with 128-bit block ciphers [22]. The final input, *ENCRYPT*, specifies whether GCM is operating in encryption or decryption mode. If *ENCRYPT* is 1, then GCM is running in encryption mode and the input blocks I_1 - I_N are interpreted as plaintext. Otherwise, GCM is running in decryption mode and the input blocks are interpreted as ciphertext.

The outputs of GCM include output blocks O_1 - O_N and a signature *S*. If GCM is running in encryption mode, then the output blocks are ciphertext. If GCM is running in decryption mode, the output blocks are plaintext. The signature is internally

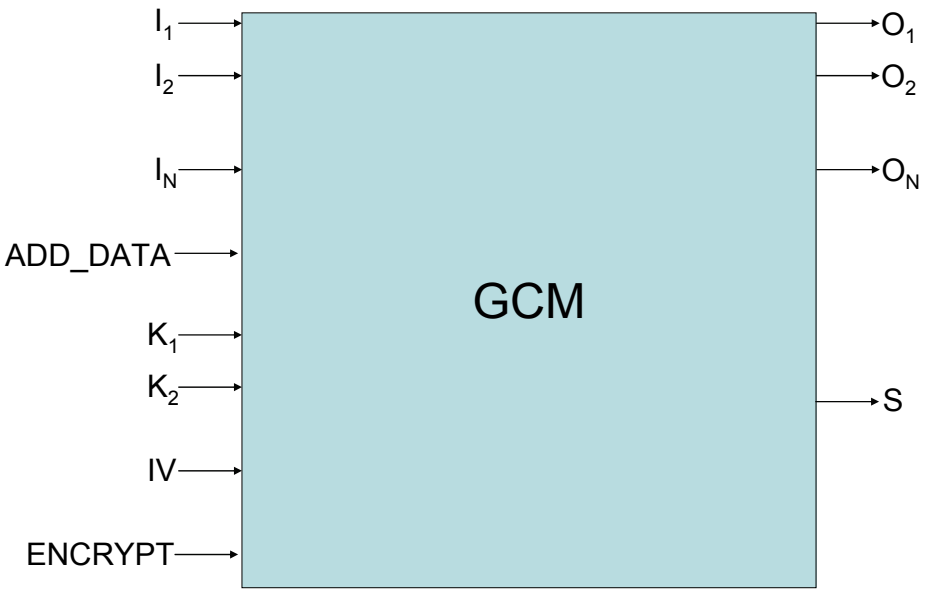


Figure 3.7 High Level View of a Hardware Implementation of Galois/Counter Mode

calculated on the ciphertext blocks (which are internally available in either mode) and the additional data (*ADD_DATA*). Note that the *ADD_DATA* is never encrypted.

The implementation of GCM used in this dissertation is shown in Figure 3.8 and described mathematically in Equations (3.8) - (3.13). As before, we use 128-bit input and output blocks, and 128-bit keys. We further chose a 96-bit initial vector for maximum efficiency. Our chosen block cipher is the Advanced Encryption Standard (AES). Note that the initial vector is concatenated (represented by the || symbol) with the number 1 represented in binary and zero-padded out to 32 bits (denoted as 1_{32}), ensuring a 128-bit input to the AES cores. The initial vector is also incremented (represented by inc blocks) before each subsequent AES operation, ensuring unique results for each operation. The results of the AES encryptions are one-time pads, which are XORed with the input blocks I_i to produce ciphertext or plaintext outputs O_i, as appropriate. The additional data block and ciphertext blocks C_i are then used to calculate the signature S, along with the block lengths (represented by the len() symbol and padded to 64 bits each) and a final one-time pad Y_0. The GMULT operation, which is a Galois field multiplication of two 128-bit values in the 2^{128} domain, is instrumental in signature generation. The values X_i in the equations below are intermediate results of GMULT operations.

$$O_i = I_i \ xor \ AES_{K1}(IV \,||\, (i+1)_{32}) \quad for \ i = 1..N \tag{3.8}$$

$$Y_0 = AES_{K1}(IV \,||\, 1_{32}) \tag{3.9}$$

$$C_i = \begin{cases} I_i \ when \ ENC = 0 \\ O_i \ when \ ENC = 1 \end{cases} \quad for \ i = 1..N \tag{3.10}$$

$$X_0 = GMULT(A, K_2) \tag{3.11}$$

$$X_i = GMULT(C_i \ xor \ X_{i-1}, K_2) \quad for \ i = 1 \,..\, N \tag{3.12}$$

$$S = Y_0 \ xor \ GMULT(X_N \ xor \ len(A)_{64} \ || \ len(C_i)_{64}, K_2) \tag{3.13}$$

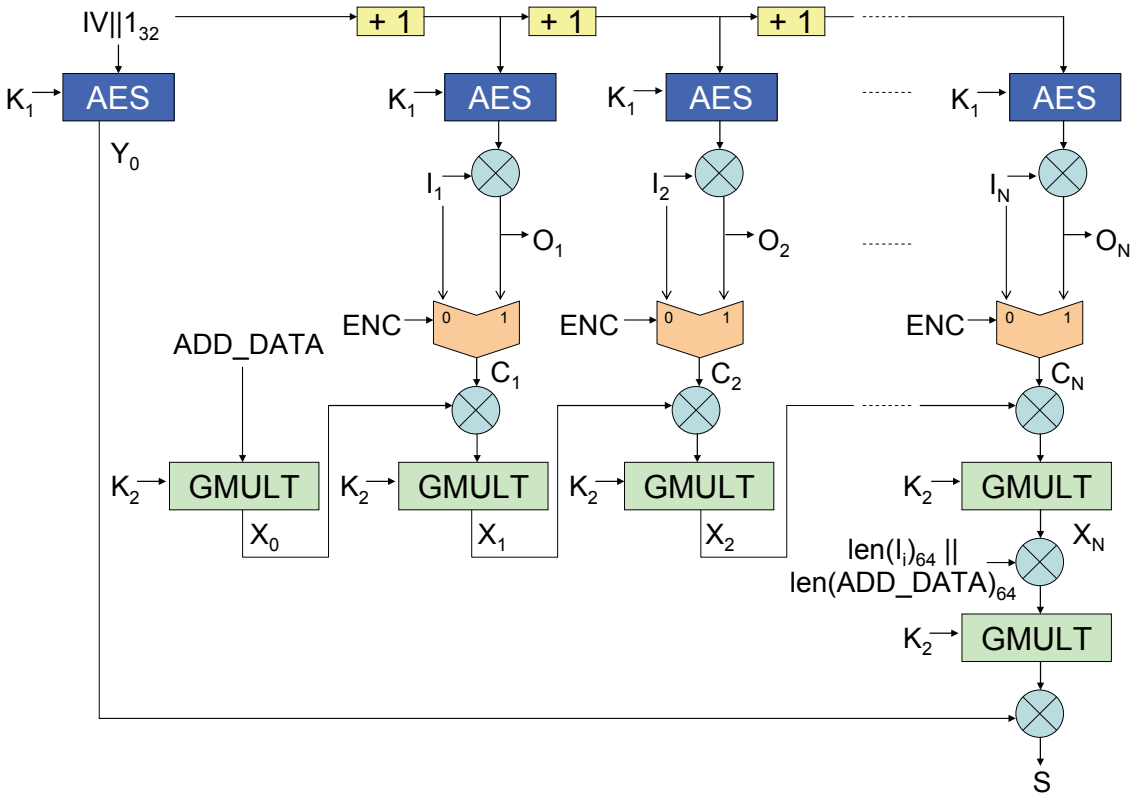


Figure 3.8 Hardware Implementation of Galois/Counter Mode

After the final data block is ready (or has been encrypted, in the case of GCM encryption), two GMULT operations are required to produce the signature. The time required to perform GMULT thus determines whether or not GCM will offer a performance advantage over PMAC. In the PMAC mode, one AES operation is required to produce the signature after the final data block is ready. Thus, GCM achieves better

performance only if the time required to perform two GMULT operations is less than the time required to perform one AES operation.

McGrew and Viega [22] discuss options for implementing GMULT. The fastest option performs the multiplication in only one clock cycle. They state that the hardware complexity for such an implementation is $O(q^2)$ logic gates, where q is the block width in bits. As we are using 128-bit blocks, the fastest GMULT implementation thus has a complexity on the order of 16,384 gates. This is on par with a pipelined AES unit [23], and may be practically implemented in hardware. With this fast implementation, the signature will be available two clock cycles after the final data block is ready, providing a clear performance advantage over PMAC.

A useful abstraction of the GCM hardware is shown in Figure 3.9. This abstraction represents the combinational logic for signature generation as a single function, GHASH. The contents of the GHASH block are shown in Figure 3.10. The inputs to GHASH are a 128 bit key H, an initial pad Y_0, the additional data block A, and the ciphertext blocks C_1 - C_N. The output of GHASH is the signature T, as shown in Equations (3.14) - (3.16). Using this abstraction allows us to write simple equations for signature generation, expressing signatures as a GHASH of various parameters.

$$X_0 = GMULT(A, H) \tag{3.14}$$

$$X_i = GMULT(C_i \; xor \; X_{i-1}, H) \quad for \; i = 1 .. N \tag{3.15}$$

$$\begin{aligned} T &= GHASH(H, Y_0, A, C_1, \ldots, C_N) \\ &= Y_0 \; xor \; GMULT(X_N \; xor \; len(A)_{64} \,||\, len(C_i)_{64}, H) \end{aligned} \tag{3.16}$$

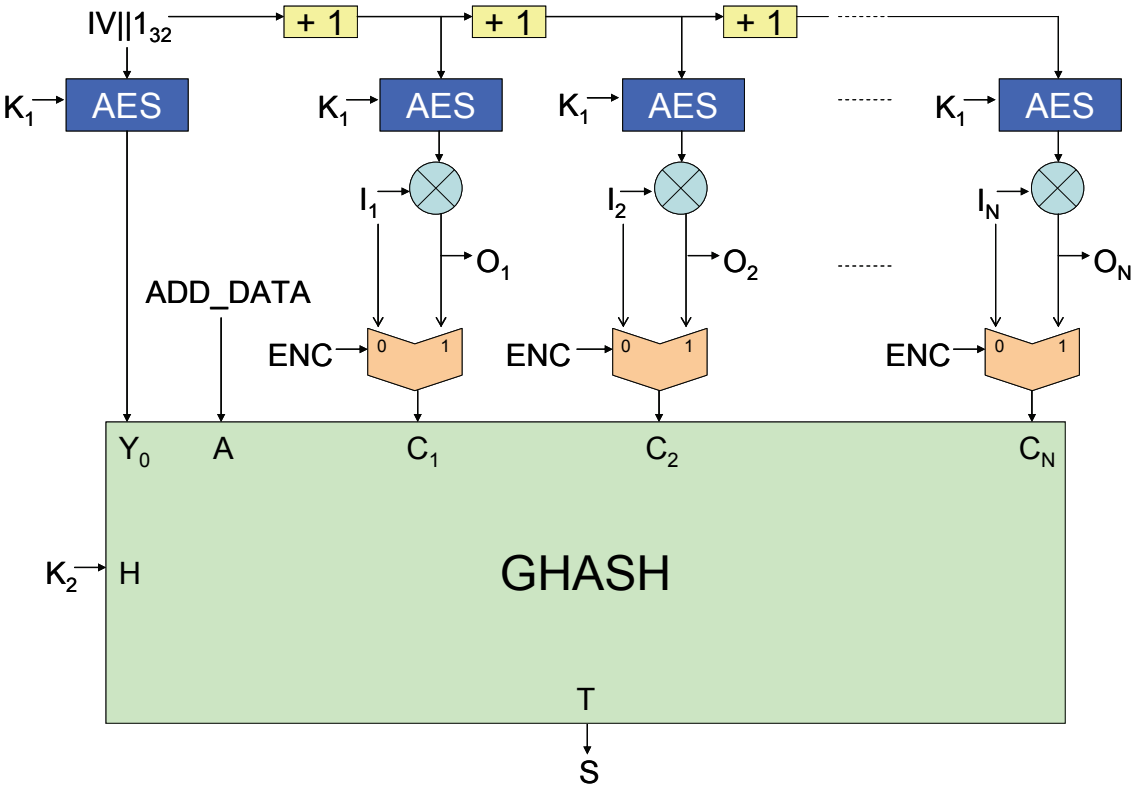


Figure 3.9 Abstraction of GCM showing GHASH

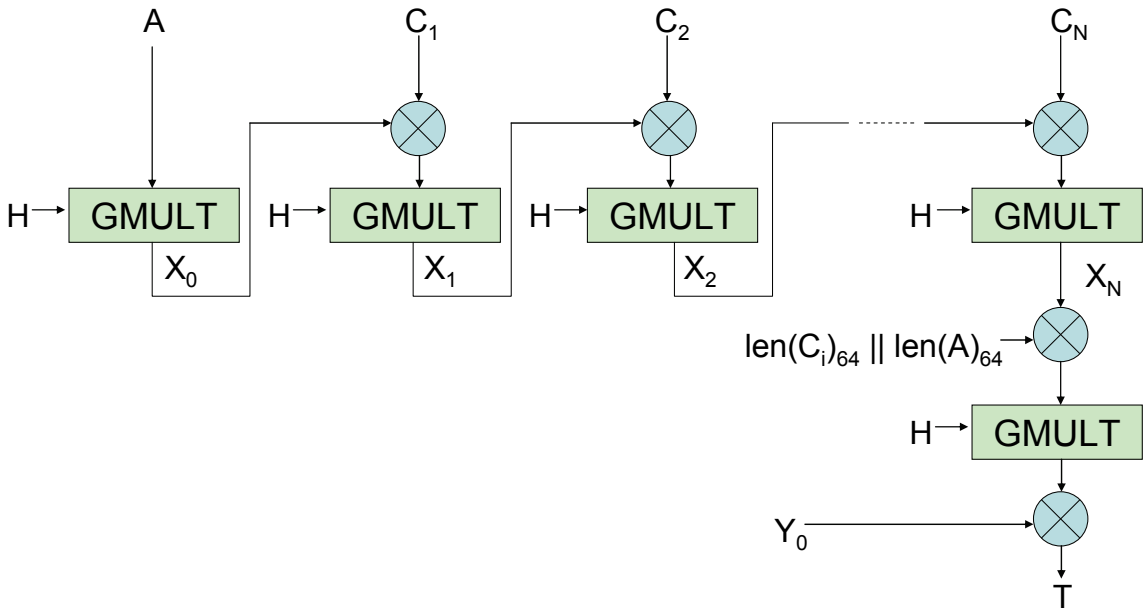


Figure 3.10 GHASH

The results of using GCM on an actual block of binary data are shown in Figure 3.11. The plaintext is the same 64 byte instruction block used above in Figure 3.6. The initial vector is the block's starting address, left padded with zeros to 96 bits. The additional data block is all zeros. Figure 3.11 shows the plaintext block, along with the resulting ciphertext and signature.

Key1: 0123456789abcdef012345678abcdef0	
Key2: fedcba9876543210fedcba9876543210	
ADD_DATA: 00000000000000000000000000000000	
IV: 000000000000000003000a8	
Plaintext:	Ciphertext:
3000a80: e3a02000	3000a80: 3731cfe8
3000a84: e50b2030	3000a84: 92c2b117
3000a88: e59f122c	3000a88: 9982c15d
3000a8c: e5812000	3000a8c: 61935ea6
3000a90: e50b2034	3000a90: d9744f9f
3000a94: e1a06000	3000a94: b501a5e2
3000a98: e59f0220	3000a98: 2aef63da
3000a9c: eb002c5b	3000a9c: d80cfb18
3000aa0: e2505000	3000aa0: 4c439843
3000aa4: 0a000033	3000aa4: 2f96660e
3000aa8: e1a00005	3000aa8: 128ec3ba
3000aac: e3a0102f	3000aac: 745beec3
3000ab0: eb004ad2	3000ab0: 2a2d38a2
3000ab4: e3500000	3000ab4: d3899dd2
3000ab8: 0a000004	3000ab8: 1a2edbbc
3000abc: e59f3200	3000abc: 82349c3c
S: a68c7a304b00e5ef10c99f7678957f38	

Figure 3.11 Binary Data Block Encrypted, then Signed in Galois/Counter Mode

CHAPTER 4

PRINCIPLES OF SECURE PROCESSOR DESIGN

The previous two chapters have provided background information regarding threats to computer security and cryptographic concepts that can be used to help counter these threats. We now delve into the basic principles of secure processor design. We start with the relatively simple case of protecting instructions and other static data. Then we proceed to the more complicated case of protecting dynamic data.

4.1 Protecting Instructions and Static Data

Before going into architectural details, we must define a few concepts. The first concept is that of security mode, which specifies the protection levels for software (instructions and static data) and dynamic data. A secure processor should provide options to protect both integrity and confidentiality. Our proposed architecture is flexible, allowing the system designer to choose the appropriate security level for the target system. The protection levels for compiled binary software code (including static data) and dynamic data may be chosen separately. For software protection, the designer may choose software integrity only mode (SIOM), software confidentiality only mode (SCOM), or software integrity and confidentiality mode (SICM). Similarly, data integrity only mode (DIOM), data confidentiality only mode (DCOM), or data integrity

and confidentiality mode (DICM) are available for protecting dynamic data. As their names imply, the SIOM and DIOM modes are limited to protecting the integrity of software or data. Instruction or data blocks are stored in binary plaintext that could be read by an adversary. The SCOM and DCOM modes encrypt data to ensure confidentiality, but do not ensure their integrity. The SICM and DICM modes ensure both integrity and confidentiality. The system designer may choose the modes for software and data independently, possibly with different levels of protection. The SICM/DICM combination is recommended for maximum security. This section focuses on the relatively simple case of protecting static data and instructions; the architectural principles explained here will be expanded in Section 4.2 to cover the more complex case of protecting dynamic data.

Another concept that must be defined is the security boundary. This is simply the boundary beyond which data are potentially vulnerable to tampering by the threats delineated in Chapter 2. Data inside this boundary are said to be in the secure domain, and are thus trusted. In this research, we assume that the processor chip itself is the secure domain; data stored on-chip are assumed to be invulnerable, while data stored off-chip are unsecure. Thus the security boundary is, conveniently, the processor chip's physical boundary.

The standard approach to protecting integrity is known as a sign-and-verify architecture [4, 24, 25]. As the name implies, a sign-and-verify architecture requires that data be signed with a secure signature when they leave the secure domain, and verified when they come into the secure domain. The standard approach to protecting confidentiality is encryption, which may be added if desired. In our architecture, data are

encrypted and/or signed when they leave the chip, and then decrypted and/or verified when they come back onto the chip.

The use of cryptography requires that the data be divided up into discrete units. This basic unit of security is called a protected block. Each protected block will be independently encrypted, and will have a signature associated with it. Protected block size is a design parameter; however, three factors should be kept in mind when choosing a protected block size. The first is cipher width, which is 128 bits (16 bytes) for the standard AES cipher. Protected block sizes that are some multiple of the selected block cipher width would be the most efficient; otherwise, cryptographic operations will require some padding. The second factor is cache line size. Since a cache line miss results in a block of data being brought on-chip and thus crossing the security boundary, the cache line size or some multiple thereof is a convenient protected block size. These first two factors are, thankfully, easily harmonized, as common cache block sizes such as 32 bytes and 64 bytes are both multiples of the standard AES cipher width. The third factor in choosing protected block size is memory constraints. Since every protected block has its own signature, and those signatures must be stored somewhere in memory, smaller protected blocks incur greater memory overhead than larger protected blocks. Throughout all the theoretical portions of this dissertation we assume, without loss of generality, a system with separate level-1 instruction and data caches with identical cache line sizes. We will mostly confine our discussion to protected block sizes equaling cache line sizes, exploring protected blocks of twice the cache line size in Section 5.5.

Protecting integrity requires that signatures be calculated in such a way that all possible attacks on integrity will be detected. Spoofing attacks can be prevented by using

the block text in signature generation. Splicing attacks can be prevented by incorporating the block's address in the signature generation. One possible variation on the splicing attack would be to splice a block from another executable but at the same address into the present executable; this can be prevented by using keys unique to each executable. Static program code and data are not vulnerable to replay attacks, which will be considered below in Section 4.2.

Our proposed sign-and-verify mechanism involves three stages: secure installation, secure loading, and secure execution [26]. These stages are depicted in Figure 4.1. The first stage is a secure installation procedure, in which binary executables are signed and optionally encrypted for a particular processor. The second stage is secure loading, in which the computer system prepares to run the program. The final stage is secure execution, where the program is run such that its integrity and/or confidentiality is maintained.

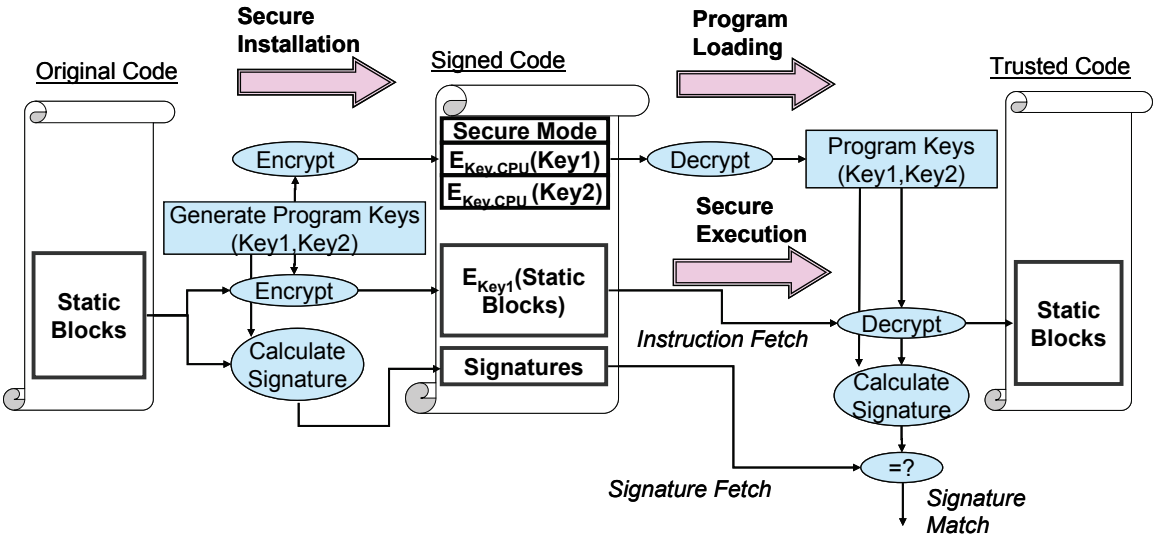


Figure 4.1 Overview of Architecture for Trusted Execution

The process by which an unprotected program is installed on the system to take advantage of hardware support for integrity and/or confidentiality is called secure installation. An unsecured executable file is processed by the CPU to create a secure executable file. The secure installation procedure presented here is similar to that proposed by Kirovski *et al.* [27]. The CPU must perform secure installations in an atomic manner, and must not reveal any secret information during or after the installation. The level of protection must be chosen before beginning secure installation as flag specifying the selected mode is stored in the header of the secure executable.

The first step during secure execution is to generate the keys necessary for cryptographic operations. Depending on the cryptographic mode, either one, two, or three keys will be required. These keys may be randomly generated from thermal noise in the processor chip [28] or by using physical unclonable functions [29]. The keys must be encrypted on-chip using a secret key built into the processor, *Key.CPU*. The encrypted keys are stored in the header of the secure executable. These keys must never leave the secure domain in their unencrypted form.

If the secure executable is to run in a software integrity mode (SIOM or SICM), then every instruction block and static data block must be signed. Signature generation must use the block text, address, and program keys as discussed above. Signatures must be stored somewhere in the secure executable, either embedded immediately preceding or following protected blocks, or in a signature table. The ramifications of this choice are explored in Section 5.1.

Encryption is required for modes protecting confidentiality (SCOM and SICM). The protected blocks are encrypted via the method of choice and stored in the secure executable.

The secure loading process prepares a secure executable to run on the secure architecture. During this process, the encrypted program keys are read from the secure executable header. These are decrypted using the processor's secret key (*Key.CPU*) and loaded into special-purpose registers on the CPU. The plaintext keys may only be accessed by dedicated on-chip hardware resources, and, as mentioned above, they should never leave the CPU as plaintext. If a context switch occurs, these keys must be re-encrypted before leaving the processor to be stored in the process control block. When the context switches back to the secure program, they must be re-loaded into the processor and decrypted once again before secure execution may resume.

The secure execution stage is when the secured program actually runs. The sign-and-verify architecture comes into play when protected data are brought into the secure domain. In our design, that occurs when a cache miss causes a protected block to be fetched from external memory. Since the cache is on-chip, and thus a trusted resource, all data in the cache are assumed to be trusted; thus, only trusted or unprotected data should be placed in the cache.

On a cache miss, the appropriate protected block must be fetched from memory. The block's signature must also be fetched from memory if the system is operating in an integrity mode (SIOM or SICM). The signature must also be recalculated based on the block that was fetched and other parameters as described above. The calculated signature is compared with the fetched signature. If the fetched and calculated signatures match,

then the block is valid and may be inserted in the cache. If the signatures do not match, the block is invalid and an exception is raised. The operating system may then appropriately handle the exception. If the system is operating in a confidentiality mode (SCOM or SICM), the block must be decrypted before being placed in the cache, or perhaps even before calculating its signature, depending on the choice of cryptographic mode.

4.2 Protecting Dynamic Data

Dynamic data is anything that is created and/or changed at runtime. Dynamically generated code, such as that produced by just-in-time compilers, also fits this description, and thus may be considered as dynamic data for discussion purposes. Protecting this data produces additional challenges. Due to its changeable nature, it is subject to replay attacks in addition to spoofing and splicing. A versioning scheme is therefore required to ensure that a fetched protected block is fresh. To that end, we introduce a sequence number for each protected block of dynamic data. These sequence numbers must be stored in a table in memory during runtime.

If encryption is being employed, sequence numbers must be considered during the encryption/decryption process. They are particularly important if OTP encryption is being used. A protected block may need to be encrypted multiple times as it is updated throughout the course of a run. The pads used for encryption must be unique each time. Therefore, the sequence number must be incremented at least as often as the block is encrypted, and must also be included in the initial vector that is used to calculate the pads for encryption to ensure pad uniqueness.

A block's sequence number must also be a parameter in the signatures of dynamic data blocks so that signature verification will detect replay attacks. Furthermore, replay attacks may include a replay of the protected block, its signature, and/or its sequence number. Consider attacks where one of either the protected block or its signature is replayed, but its sequence number is not. These attacks will be detected as if it were a spoofing attack. An attack where both the protected block and its signature are replayed, but the sequence number is not, will be detected because the sequence number is included in the signature generation. Finally, consider a sophisticated attack where the protected block, its signature, and its sequence number are all replayed. Simply verifying the signature will not detect this attack; further protection is needed.

The traditional method for providing this additional protection was to use a Merkle tree over all dynamic data [30, 31]. A Merkle tree involves calculating signatures or hashes over all data, and then calculating signatures over these signatures, for as many levels as desired. The amount of data protected by a single signature at each level is chosen to set the depth of the tree. Eventually, some level of the tree will be protected by a single signature, known as the root. The root will contain information from everything protected by the tree, and must be stored in a secure location as it will provide the root of security. When one node on a Merkle tree needs to be verified, nodes above it must also be verified until a signature can be compared to something that is known to be secure. This can lead to large amounts of performance overhead.

Our previous research has shown that a Merkle tree over all dynamic data is unnecessary [32]; only the sequence numbers need to be protected by a tree-like structure. This was independently proposed in [33]. To that end, our architecture uses a

tree-like structure such as that depicted in Figure 4.2. Signatures are calculated for blocks of sequence number data. The signatures for sequence number blocks associated with one page of data are XORed together to produce a page root signature. A page root signature is thus the overall signature of the sequence numbers associated with a single page of dynamic data. All the program's page root signatures are XORed to produce the program root signature. The program root signature is the root of trust, and must be stored in a special register on the processor. Like the program keys, it must be stored in the process control block in encrypted form during a context switch. Managing this tree efficiently poses a significant design challenge, which shall be discussed below in Section 6.2. Note that sequence number blocks need not be encrypted [34].

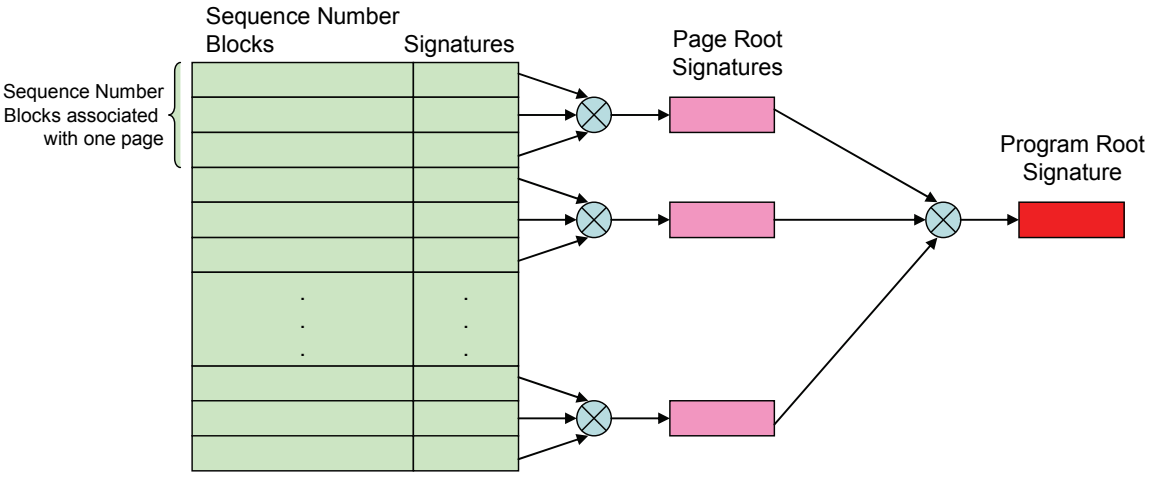


Figure 4.2 Tree Structure for Protecting Sequence Numbers

As in static data protection, protected blocks of dynamic data are verified when they enter the secure domain. In our architecture, this occurs on a cache miss. When

protecting dynamic data, the first operation required on a cache miss is to fetch the sequence number, as the sequence number is required for decrypting the block (when using OTP) and/or calculating its signature. Once the sequence number is available, the protected block and its signature can be fetched, and will be decrypted and/or verified as in the static case.

Unlike static data, dynamic data must be encrypted and/or signed when it leaves the secure domain. In our architecture, this occurs on a cache writeback. As before, the first required operation is to fetch the sequence number. The sequence number must then be incremented and the updated sequence number used to encrypt and/or sign the protected block. The encrypted protected block and its signature may then be written to memory.

When sequence numbers are fetched, they must be verified against higher nodes in the tree. When sequence numbers are incremented, the tree must be updated. As mentioned before, managing the tree efficiently poses a serious challenge, which will be discussed below in Section 6.2.

4.3 Comments

This chapter has presented the basic framework of our sign-and-verify solution. We have not yet delved into any particulars, such as how encryption should be performed, how signatures should be generated, etc. These issues must still be addressed, and we do so in the following two chapters.

CHAPTER 5

GENERAL CHALLENGES IN SECURE PROCESSOR DESIGN

This chapter examines several challenges that the computer architect will encounter when designing a secure processor. The challenges discussed here are general in nature; challenges specifically related to protecting dynamic data are discussed in the next chapter. We address where to store signatures, how to encrypt protected blocks, how to sign protected blocks, speculative execution to hide latency, and methods for reducing memory overhead. We also discuss a way to enable secure input and output, and some other related topics.

5.1 Choosing Where to Store Signatures

One of the first challenges that must be dealt with in designing a secure processor is deciding where to store signatures. This decision influences all three stages: secure installation, loading, and execution, and also has implications for system performance. Signatures may be either stored on-chip in a dedicated resource or off-chip somewhere in memory.

5.1.1 Storing Signatures On-Chip

Storing signatures in a dedicated on-chip memory has great performance advantages. Data stored on-chip can be accessed much more quickly than that stored off-chip, and is assumed to be safe from tampering. Also, depending on the on-chip buses that are used, signatures may be able to be fetched in parallel with fetching the protected block. However, using a dedicated on-chip memory would require a potentially large number of transistors that could be used for other performance-enhancing hardware resources. Also, the amount of data that could be protected would be limited by the size of the on-chip memory reserved for signatures.

A secure installation procedure supporting signatures stored on-chip would create a table of signatures for the static blocks in the secure executable file. The secure loading process would then load the contents of that table into the on-chip resource. Signatures of dynamic blocks would be created at runtime. Context switches would become more time-consuming, as the signatures stored on-chip would need to be dumped into the process control block and later loaded back in.

This signature storage scheme may be desirable for a simple system-on-a-chip that rarely, if ever, performs context switches, but for high-end embedded systems and general purpose systems, it is not likely to be desirable. The remainder of this chapter will therefore assume that signatures will be stored off-chip.

5.1.2 Storing Signatures Off-Chip

The two methods for storing signatures off-chip are to either embed the signatures with the protected blocks or to store them in a signature table. An embedded signature either immediately precedes or immediately follows the protected block it protects. The

major advantage of embedded signatures is that, with appropriate memory controller support, the signature can be fetched in the same pipelined memory operation as the protected block. This is depicted in Figure 5.1 (a). The figure shows the timing of a memory pipeline fetching a 32 byte protected block followed by a 16 byte signature. The first 64-bit chunk is available in 12 clock cycles, with subsequent chunks becoming available every two cycles.

The major disadvantage of embedded signatures is that they clutter and complicate the address space. Of a necessity, they are visible in the physical address space, but the security designer must decide whether they will be visible in the virtual address space, and, if so, whether there will be an additional "secure address space" within which the signatures will not be visible. Either way, additional address translation logic would be required, as blocks of data are no longer contiguous in the physical address space. If the signatures are visible in the virtual address space, cache controllers must also be modified to prevent cache pollution from the signatures.

If this scheme is to be implemented, the signatures for static blocks will be embedded with their protected blocks in the secure executable file during secure installation. Secure loading will not need to specially handle the signatures. During secure execution, dynamic protected blocks and their signatures will be written to memory together to preserve the embedding scheme.

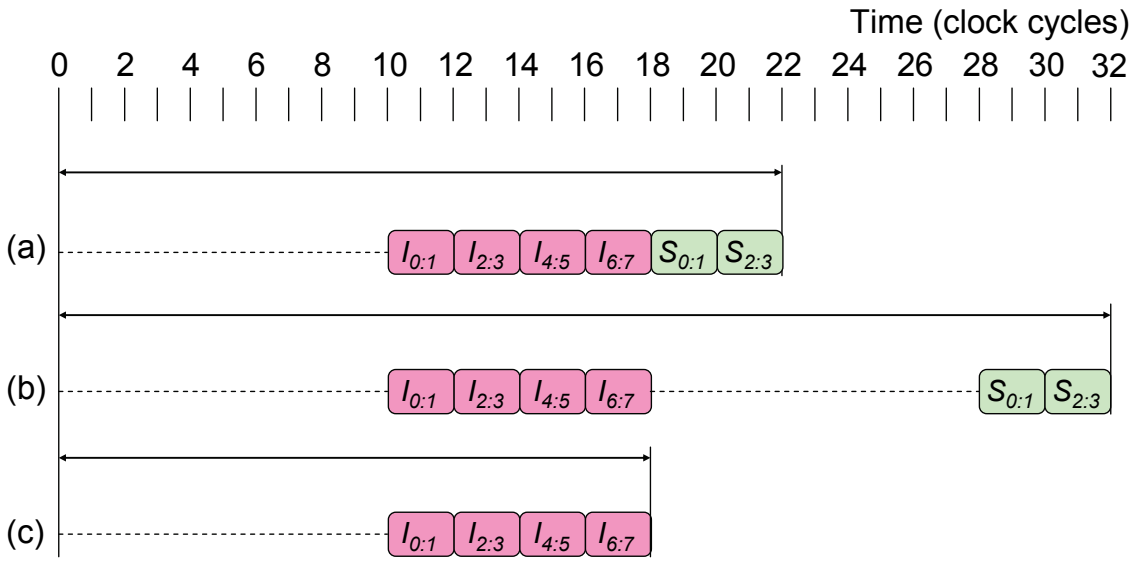


Figure 5.1 Memory Pipeline for (a) Embedded Signatures, (b) Signature Table, and (c) Signature Table with Signature Cache Hit

Storing signatures in a signature table in off-chip memory relieves some of the complications of embedded signatures but introduces others. For instance, it reduces the complexity of address translation hardware, reduces address space confusion, and relieves cache pollution concerns by storing signatures separately from their protected blocks.

The major disadvantage of this signature storage scheme is that a second memory access must be initiated to fetch the signature after the protected block has been fetched. This introduces additional performance overhead, as shown in Figure 5.1 (b). The solution to this is to cache signatures on-chip. The signature cache can be probed in parallel with fetching the protected block. If there is a hit in the signature cache, then the signature need not be fetched from external memory, as in Figure 5.1 (c). This situation obviously minimizes the contribution of memory accesses to performance overhead. If

the signature cache access misses, however, the latency depicted in Figure 5.1 (b) still applies.

One approach to designing a signature cache is to use a dedicated cache structure with cache lines of the same size as the signatures. This approach takes advantage of existing and well-understood cache designs. However, this approach may not be desirable as it requires an additional large, power-hungry on-chip resource. An alternative approach would be to use a much smaller victim cache structure. A victim cache is a relatively small, fully associative structure for storing evicted data in the hope that they will be needed again soon. In this case, the signatures of blocks that are evicted from the instruction and data caches would be placed in the victim cache. This would require that the instruction and data cache lines be widened to include the protected block plus its signature. This approach enlarges existing cache structures, but removes the need for all but a simple victim cache for signatures. An alternative is to regenerate protected blocks' signatures when they are evicted. This would obviate the need to widen instruction and data cache lines, but at the cost of increased performance overhead. The signatures of dynamic and static blocks could use either the same or separate victim caches, as desired. Victim cache depth is a design parameter.

As with storing signatures on-chip, the secure installation stage must create a signature table in the secure executable. The secure loading stage must copy this table into the appropriate location in memory. During context switches, the contents of the signature cache must be invalidated. When dynamic protected blocks are evicted form the data cache, their new signatures should still be written out to memory. Even though

the signatures are no longer contiguous with protected blocks, this should introduce minimal performance overhead given sufficient write buffers.

5.2 Coping with Cryptographic Latency

Chapter 3 above presented two methods for ensuring confidentiality: symmetric key and one-time pad cryptography. Both of these methods have the potential to introduce cryptographic latency, that is, performance overhead caused by encryption/decryption. We here analyze these two methods in the context of a secure processor to determine which is likely to introduce the lowest performance overhead.

As seen in Section 3.1, the straightforward method for encrypting and decrypting a block of data is symmetric cryptography. Equations (5.1) and (5.2) show symmetric cryptography being used to encrypt and decrypt, respectively, a 32 byte plaintext block with a 128-bit AES cipher. In this equation, $I_{0:3}$ and $I_{4:7}$ are the two 16 byte sub-blocks comprising the plaintext block, $C_{0:3}$ and $C_{4:7}$ are the corresponding sub-blocks of ciphertext, AES_e and AES_d allude to the AES cipher operating in encryption and decryption modes, respectively, and *KEY3* is one of the unique program keys discussed above in Section 4.1.

$$C_{4i:4i+3} = AES_{e,KEY3}(I_{4i:4i+3}) \quad for\ i = 0..1 \tag{5.1}$$

$$I_{4i:4i+3} = AES_{d,KEY3}(C_{4i:4i+3}) \quad for\ i = 0..1 \tag{5.2}$$

The major drawback of applying symmetric cryptography in secure processors is the large amount of cryptographic latency it induces. Figure 5.2 (a) shows the cryptographic latency incurred by symmetric cryptography when decrypting a protected block in our example secure processor. A 32 byte protected block is fetched from

memory in 64-bit chunks. Each 16 byte sub-block is decrypted independently using a 128-bit, fully pipelined AES unit that requires 12 clock cycles to complete an AES operation. AES operations in Figure 5.2 are depicted as rows of shaded blocks. As the figure indicates, decryption of a ciphertext sub-block cannot begin until that sub-block is completely available. The end result is that the protected block is not available in its decrypted form until 12 clock cycles after it has been completely fetched from memory.

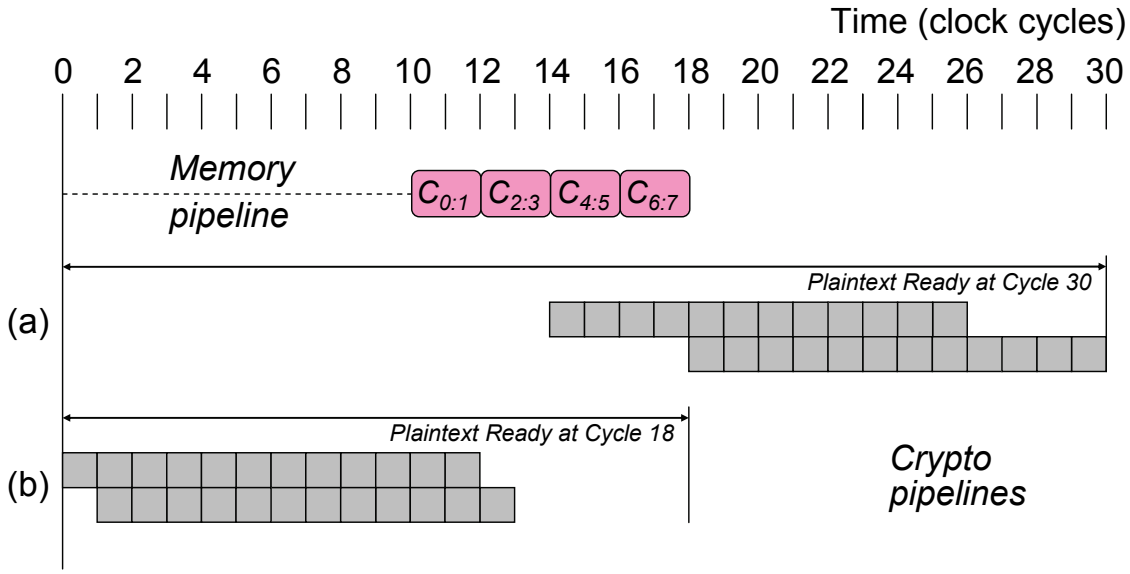


Figure 5.2 Cryptographic Latency for (a) Symmetric Key Decryption and (b) One-Time-Pad Decryption

This cryptographic latency can be alleviated by using OTP cryptography. Equations (5.3) and (5.4) demonstrate how OTP is used for encryption and decryption, respectively. The initial vector for each sub-block is calculated from the sub-block's

address $A(SB_i)$ and the protected block's sequence number *SN*, which are expanded to 128 bits using a secure padding function *SP*. This 128-bit value is then encrypted using the program key *KEY3*. (In OTP, AES need only operate in encryption mode, so the subscript *e* is omitted for clarity.) The result is the actual one-time pad, which is XORed with the plaintext sub-block to produce a ciphertext sub-block when encrypting, or vice-versa when decrypting.

$$C_{4i:4i+3} = (I_{4i:4i+3})\ xor\ AES_{KEY3}(SP(A(SB_i), SN)) \quad for\ i = 0..1 \tag{5.3}$$

$$I_{4i:4i+3} = (C_{4i:4i+3})\ xor\ AES_{KEY3}(SP(A(SB_i), SN)) \quad for\ i = 0..1 \tag{5.4}$$

Galois/Counter Mode cryptography intrinsically utilizes OTP for encryption and decryption, as shown in Equations (5.5) and (5.6), respectively. In GCM, the pads are produced by concatenating a 96-bit initial vector with a 32-bit counter to form a 128-bit value, which is then encrypted. The initial vector consists of the protected block's address *A* and sequence number *SN*, which are extended to 96 bits by a secure padding function SP_{96}. Note that GCM requires only two keys, as opposed to the three keys required for the other modes discussed in this dissertation. When using GCM, we use *KEY1* to perform all AES operations.

$$C_{4i:4i+3} = (I_{4i:4i+3})\ xor\ AES_{KEY1}(SP_{96}(A, SN) \,||\, (i+2)_{32}) \quad for\ i = 0..1 \tag{5.5}$$

$$I_{4i:4i+3} = (C_{4i:4i+3})\ xor\ AES_{KEY1}(SP_{96}(A, SN) \,||\, (i+2)_{32}) \quad for\ i = 0..1 \tag{5.6}$$

The performance advantage of using OTP in a secure processor is evident in Figure 5.2 (b). The sub-block's address is known at the beginning of the memory fetch operation; if we assume that the sequence number is also known at that time, then we may go ahead and perform the cryptographic operations required for calculating the one-time pads in parallel with the memory access. Thus the pads are ready for use even

before the protected block itself is available; a simple XOR operation is all that is required for decryption, leading to no cryptographic latency whatsoever. Furthermore, using OTP cryptography makes more efficient use of the pipelined AES unit.

For blocks of static data, the sequence number is always the same, and so the above assumption regarding the sequence number holds. The confidentiality of static data may thus be ensured without incurring any performance overhead. For dynamic data, however, the sequence number must be fetched and possibly validated before pad calculation can begin. As the sequence number is also required for signature generation, sequence numbers are necessary in both the DICM and DIOM modes. Techniques discussed below in the next chapter may be applied to minimize the overhead caused by sequence numbers.

5.3 Choosing a Cryptographic Mode for Signature Generation

Three cryptographic modes were presented above in Section 3.2: cipher block chaining message authentication code, parallelizable message authentication code, and Galois/counter mode. This section examines the use of these modes in our example architecture, with the memory pipeline, cryptographic pipeline, protected block size, etc. as described above. Each of these modes introduces a different amount of verification latency, that is, the time from when the protected block is available until the time that it is verified. In analyzing the performance of these modes, we will focus on verifying blocks of static data. The performance analyses would also apply to protecting dynamic data if one assumes that the described actions take place after the sequence number is fetched and possibly verified. For the purpose of clarity, this discussion assumes, without loss of generality, that signatures are calculated on plain-text.

5.3.1 CBC-MAC

One well-known cryptographic mode is CBC-MAC, which calculates signatures using a chain of cryptographic operations. An implementation of CBC-MAC in our example system is described in Equation (5.7), with all symbols as defined above. As this mode's name implies, all cryptographic operations must be performed in series; an operation must complete before the next one may begin.

$$S = AES_{KEY2}(I_{4:7}\ xor\ AES_{KEY2}(I_{0:3}\ xor\ AES_{KEY1}(SP(A,SN)))) \tag{5.7}$$

The verification latency that CBC-MAC introduces into our sample system is shown in Figure 5.3. In this figure, *S* denotes the signature fetched from memory, while *cS* denotes the signature being calculated for verification purposes. The serial nature of this cryptographic mode can be plainly seen in this figure. The protected block is completely fetched within 18 clock cycles, but the signature is not ready until 39 clock cycles, leading to a verification latency of 21 cycles, including a cycle to compare the fetched and calculated signatures.

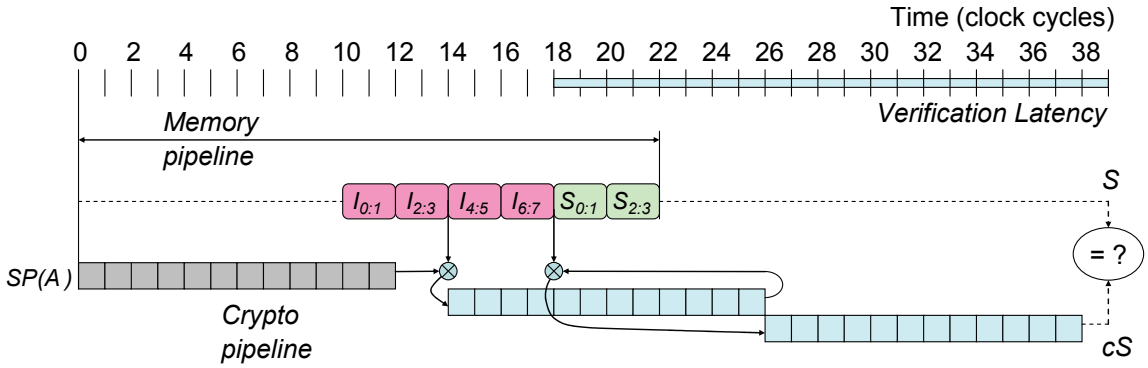


Figure 5.3 Verification Latency for Static Protected Blocks Using CBC-MAC

CBC-MAC is a well-known and established method for calculating signatures. It is also very simple to implement, and can be performed in hardware with a single non-pipelined AES unit. (Note that performing encryption in addition to signature generation would complicate matters, especially if using such simple AES hardware.) The primary drawback is the potentially large verification latencies that it entails, 21 clock cycles in our example architecture. This latency would be less dramatic in systems with longer memory access times, as the signature generation would overlap more of the memory access. However, the minimum lower bound on verification latency is set by the number of cycles required for an AES operation (plus one for signature comparison), as the final operation cannot begin until the protected block is fully available.

5.3.2 PMAC

PMAC is an alternative cryptographic mode that may reduce verification latency in many systems. The PMAC mode is applied to our example architecture in Equations (5.8) and (5.9). Using this mode, a signature $Sig(SB_i)$ is calculated for each 128-bit sub-block (Equation (5.8)) and then XORed together to form the overall signature (Equation (5.9)). Each sub-block's address, $A(SB_i)$, is used in calculating the individual sub-block's signature. Two AES operations are required to calculate each sub-block's signature; these must be performed in sequence. However, the signatures for each sub-block are calculated independently, and thus may be calculated concurrently.

$$Sig(SB_i) = AES_{KEY2}((I_{4i:4i+3}) \; xor \; AES_{KEY1}(SP(A(SB_i), SN))) \quad for \; i = 0..1 \tag{5.8}$$

$$S = Sig(SB_0) \; xor \; Sig(SB_1) \tag{5.9}$$

The verification latency introduced by using the PMAC cipher in our example system is illustrated in Figure 5.4. The figure shows the parallel nature of this mode.

The verification latency of the PMAC mode in our sample system is 13 clock cycles, which is a performance improvement over the CBC-MAC mode.

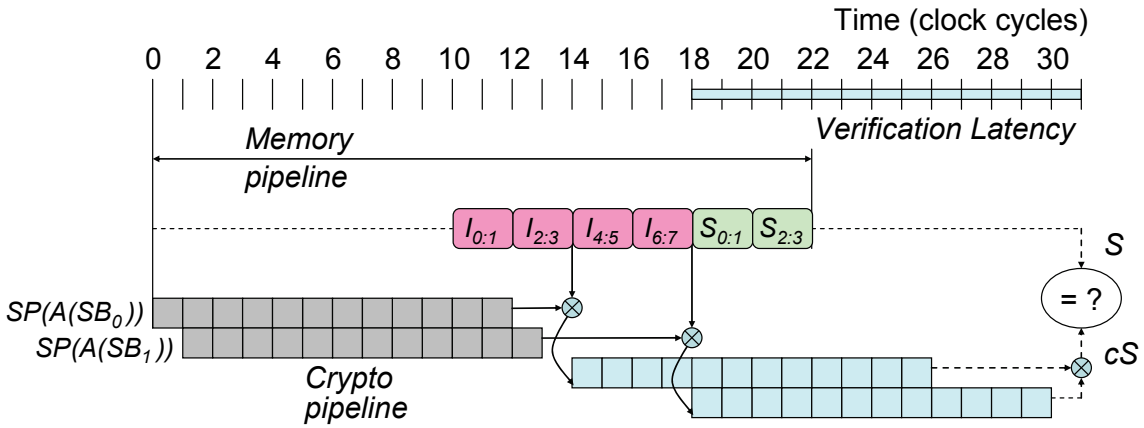


Figure 5.4 Verification Latency for Static Protected Blocks Using PMAC

The glaring advantage of the PMAC mode is its improved performance. Its downside is that it requires either a pipelined AES unit or two simple AES units. If a pipelined AES unit is used, then the PMAC mode makes more efficient use of it than does the CBC-MAC. Furthermore, the PMAC mode has the same minimum lower bound on verification latency as the CBC-MAC, so the simpler mode may be preferable in systems where memory access times dominate. Another disadvantage of PMAC is that it is patent encumbered, but the patent holder has promised to "license these patents under fair, reasonable, and non-discriminatory terms" [35].

5.3.3 GCM

The third and final cryptographic mode discussed in Chapter 3 is the Galois/counter mode. This mode is applied to our example system in Equations (5.10) and (5.11). Recall that 96-bit initial vectors are most efficient for GCM [22]. We therefore use the aforementioned 96-bit secure padding function concatenated with 32-bit counter values as inputs for all AES operations. Equation (5.11) uses the GHASH abstraction for simplicity; see Section 3.3 for the details of this abstraction. Unlike the other two modes discussed above, GCM enforces the encrypt, then sign approach, so the two ciphertext sub-blocks $C_{0:3}$ and $C_{4:7}$ as calculated according to Equations (5.5) and (5.6) are used as parameters to GHASH. The additional data value is 128 bits of zeros (0_{128}), and the program key *KEY2* is used for the Galois field multiplications performed by GHASH.

$$Y_0 = AES_{KEY1}(SP_{96}(A, SN) \,||\, 1_{32}) \tag{5.10}$$

$$S = GHASH(KEY2, Y_0, 0_{128}, C_{0:3}, C_{4:7}) \tag{5.11}$$

Our example system's verification latency when using GCM is shown in Figure 5.5. In actuality, much of the work of GHASH can be done in parallel with the memory operations, with the various GMULT operations being performed as inputs become available, leaving only two GMULTs to be performed once the final ciphertext sub-block is provided. This figure assumes that those two GMULT operations are fully combinational and require only one clock cycle. Such an implementation, while mathematically possible, is likely not feasible in actual hardware. As we have seen in Section 3.3, the practical minimum time required to perform these two GMULTs is two clock cycles. The minimum lower bound on verification latency for a particular

implementation with GCM is either the time it takes to fetch the signature or the time required to perform the last two GMULT operations, whichever is larger, plus one cycle for signature comparison. Our example system can tolerate GHASH latencies of up to four clock cycles without increasing the verification latency, which is shown to be five clock cycles in the figure.

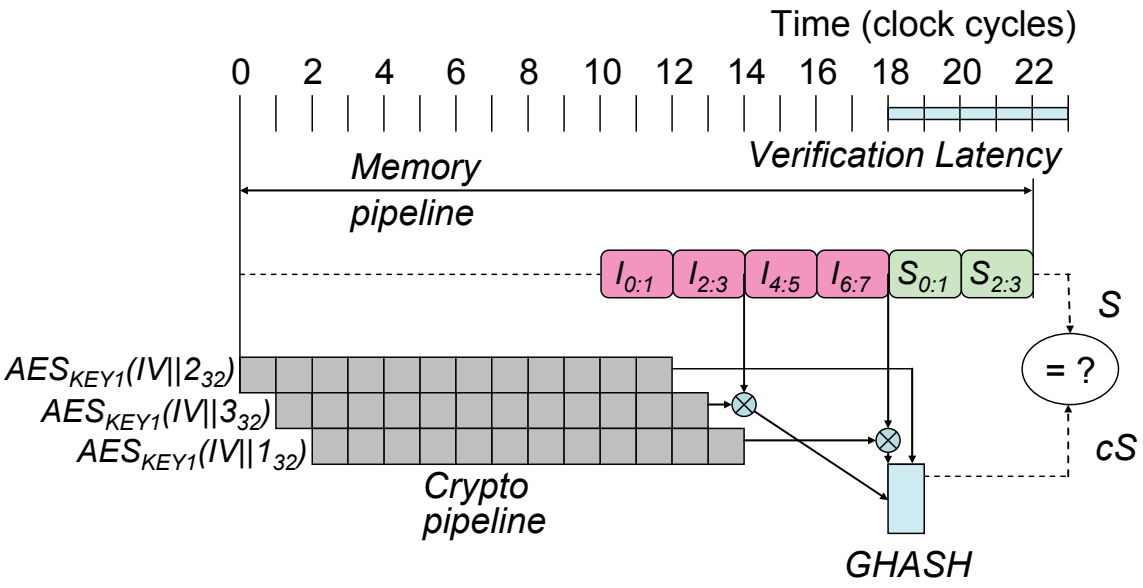


Figure 5.5 Verification Latency for Static Protected Blocks Using GHASH

Section 3.3 stated that GCM will provide better performance than PMAC as long as the time required to perform two GMULTs is less than the time required to perform an AES operation. In our example system, an AES operation requires 12 clock cycles, so GCM yields better performance as long as a GMULT operation requires less than six clock cycles. We have mostly limited our discussion thus far to a fully combinational

GMULT implementation that requires one clock cycle. Although this implementation is practical, the complexity it entails may be undesirable in some applications. Several alternative, sequential implementations of GMULT with lower complexity have been proposed [36-39]. No matter which implementation of GMULT is chosen, as long as it requires less than six cycles, GCM will outperform PMAC. Otherwise, the lower complexity PMAC mode is more attractive. The remainder of this dissertation assumes a GMULT implementation taking no more than two clock cycles, which will yield performance as in Figure 5.5.

5.4 Hiding Verification Latency

A conservative approach to designing a secure processor would wait until a protected block is verified before executing its instructions or using its data. We call such an implementation wait 'till verified (WtV). This exposes the verification latency discussed above, leading to possibly extensive performance overhead.

Ideally, the verification latency should be completely hidden, thus introducing no performance overhead. This would require the processor to resume execution as soon as the whole protected block is available. Such a scheme is called run before verification (RbV). Protected blocks, however, may have been subject to tampering, which will not be evident until signature verification is complete.

The solution to this quandary is to allow the speculative execution of untrusted instructions and the speculative use of untrusted data. Untrusted instructions, and instructions using untrusted data, must not be allowed to commit until their signatures have been verified. This prevents tampered data from propagating into CPU registers or memory. For out-of-order processors, RbV support requires a simple modification to the

reorder buffer, adding two verified flags (one for instruction verification and one for data verification) which will be updated as blocks are verified. Instructions may not be retired until those flags are set. The memory access unit must also be modified to prevent an unverified instruction from writing data to memory. In-order processors require an additional resource: the Instruction Verification Buffer (IVB).

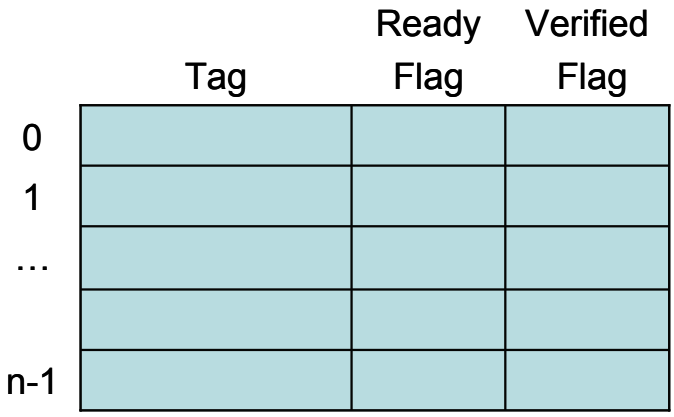


Figure 5.6 Instruction Verification Buffer

The structure of the IVB is shown in Figure 5.6. The IVB's depth (number of instructions whose information it can hold) is a design parameter, represented by *n* in the figure. Two IVBs should be implemented, one for instructions and one for data. As instructions are fetched on I-cache misses, their information is placed in the instruction IVB. When an instruction causes a D-cache miss, its information is placed in the data IVB. Once the processor has completed execution of an instruction, it checks the IVBs to see if that instruction or its data are pending verification. If it is not marked as verified, the instruction may not be retired. In the unlikely event that newly fetched or issued

instructions will not fit in the IVBs, the processor must stall until enough instructions have been removed so that new instructions can be inserted.

Shi and Lee point out that RbV schemes are vulnerable to side-channel attacks if a malicious memory access or jump instruction has been injected into an instruction block [40]. When such instructions execute speculatively during verification, they may reveal confidential data by using it as the target address. This concern may be alleviated by stalling all instructions that would result in a memory access until they have been verified. Alternatively, a bus encryption scheme may be employed, but this would likely increase complexity and performance overhead.

5.5 Coping with Memory Overhead

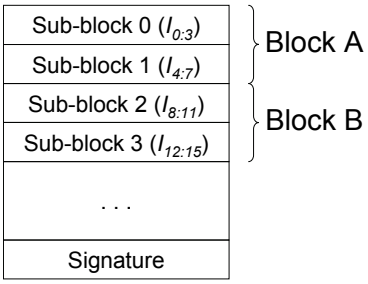
The sample architecture we have discussed so far would introduce a hefty memory overhead. For every 32 byte protected block, a 16 byte signature is required. This overhead could be prohibitive on embedded systems with tight memory constraints. The solution, alluded to in Section 4.1, is to make the protected block size a multiple of the cache line size.

In this section we consider modifying the example architecture to use 64 byte protected blocks, which are twice the size of the 32 byte cache lines. This introduces two additional 128-bit sub-blocks, $I_{8:11}$ and $I_{12:15}$. The equations for the various cryptographic modes presented above need only be extended to take these additional sub-blocks into account. For CBC-MAC, the chain of cryptographic operations may be extended to include the two additional sub-blocks. For PMAC, signatures for the two additional sub-blocks will be independently calculated, just like the first two, and all four signatures

XORed together to produce the total signature. If GCM is used, the GHASH implementation must be expanded to handle four sub-blocks instead of two.

On a cache miss, the cache line that was missed in the cache is needed immediately, while the other cache line contained within the protected block is not. For ease of discussion, we call the immediately needed block the missed block and the other block the unmissed block. A policy is required to handle the unmissed block on a cache miss. Additionally, the amount of data transferred form memory influences both performance and power overhead. The most naïve implementation would always fetch the entire protected block on a cache miss, and discard the unmissed block. A more sophisticated implementation would place the unmissed block into the instruction or data cache, as appropriate, thus exhibiting a prefetching behavior to take advantage of memory access locality. This could prevent future cache misses, but comes at the risk of cache pollution.

The actions required on a cache miss when using double-sized protected blocks can be broken down into four cases. These cases, which are outlined in Figure 5.7, are divided on the basis of which cache line within the protected block was the missed block, and whether or not the other cache line is also in the cache or is dirty. If the unmissed block is available on-chip and is not dirty, then the on-chip version can be used for signature generation. If it is not available on-chip, or is dirty, then the original version of the block must be fetched from memory. Note that if the unmissed block is on-chip and dirty, the unmissed block fetched from memory must be discarded after signature verification. For convenience, we call the first cache line in a protected block Block A, and the second Block B.



Miss On	Condition	Case #
Block A	Block B not on-chip or on-chip and dirty	1
	Block B on-chip and not dirty	2
Block B	Block A not on-chip or on-chip and dirty	3
	Block A on-chip and not dirty	4

Figure 5.7 Memory Layout and Cache Miss Cases

Even if the unmissed block is available on-chip and is not dirty, fetching it from memory may be more economical in some situations. For instance, in Case 2, Block B is available on-chip for signature generation. However, if the system is using embedded signatures, then one continuous memory operation to fetch the entire protected block and its signature is faster than fetching Block A and then starting a new memory operation to fetch the signature. As Figure 5.8 illustrates, if a new memory operation has to be started to fetch an embedded signature, the resulting latency is the same as if the signature were fetched from a signature table.

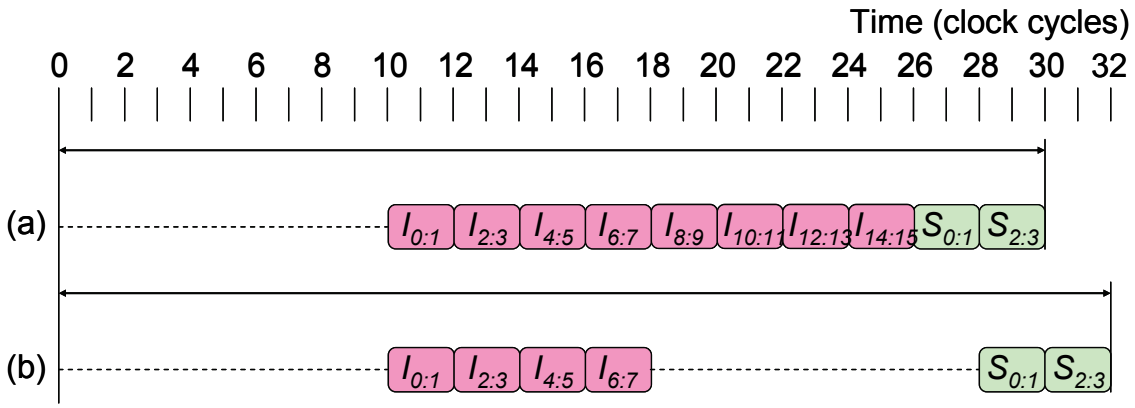


Figure 5.8 Memory Pipeline for Case 2:

(a) Fetching Block B with Embedded Signatures, and

(b) Not Fetching Block B with either Embedded Signatures or Signature Table

We demonstrate the verification latency introduced by doubling the protected block size using our example architecture with embedded signatures and GCM cryptography. Cases 1 and 2 are illustrated in Figure 5.9. We fetch the whole protected block in both cases as it is necessary in Case 1 and fetching the unmissed block reduces memory latency in Case 2, as described above. Our architecture thus automatically fetches the entire protected block when Block A is the missed block. As the figure shows, the verification latency is 13 clock cycles for both of these cases.

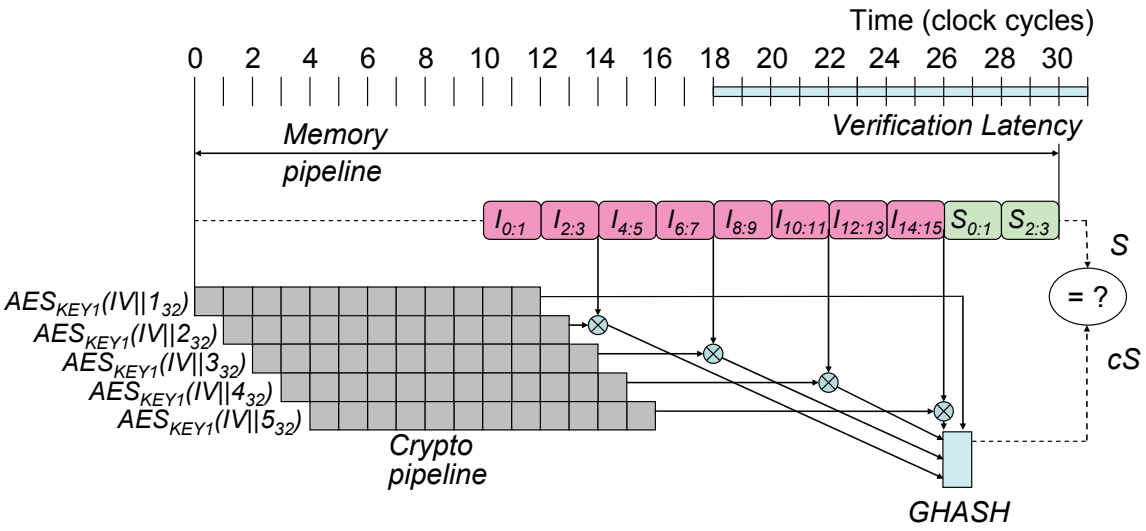


Figure 5.9 Verification Latency for Double Sized Static Protected Blocks Using GHASH, Cases 1 and 2

Cases 3 and 4, however, are slightly more complicated. As these cases involve a miss on Block B, the cache must be probed for Block A before starting a memory access. If Block A is not found in the cache, or is dirty, then the whole protected block must be fetched. This is Case 3 and, as depicted in Figure 5.10, incurs a verification overhead of 14 clock cycles. (Recall that verification overhead is measured from the time that the cache line would have become available if no security extensions were present.) If Block A is found in the cache, then it need not be fetched, significantly reducing the memory overhead. Figure 5.11 shows Case 4, which only has a 6 clock cycle performance overhead. Thus, with some added complexity up front (the cache probe), significant performance gains can be achieved for Case 4 cache misses.

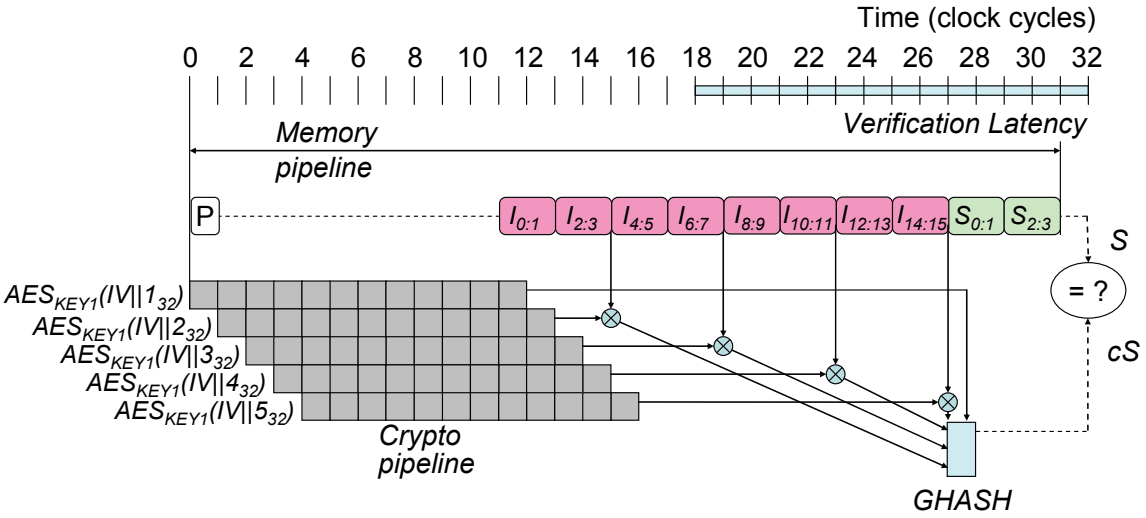


Figure 5.10 Verification Latency for Double Sized Static Protected Blocks Using GHASH, Case 3

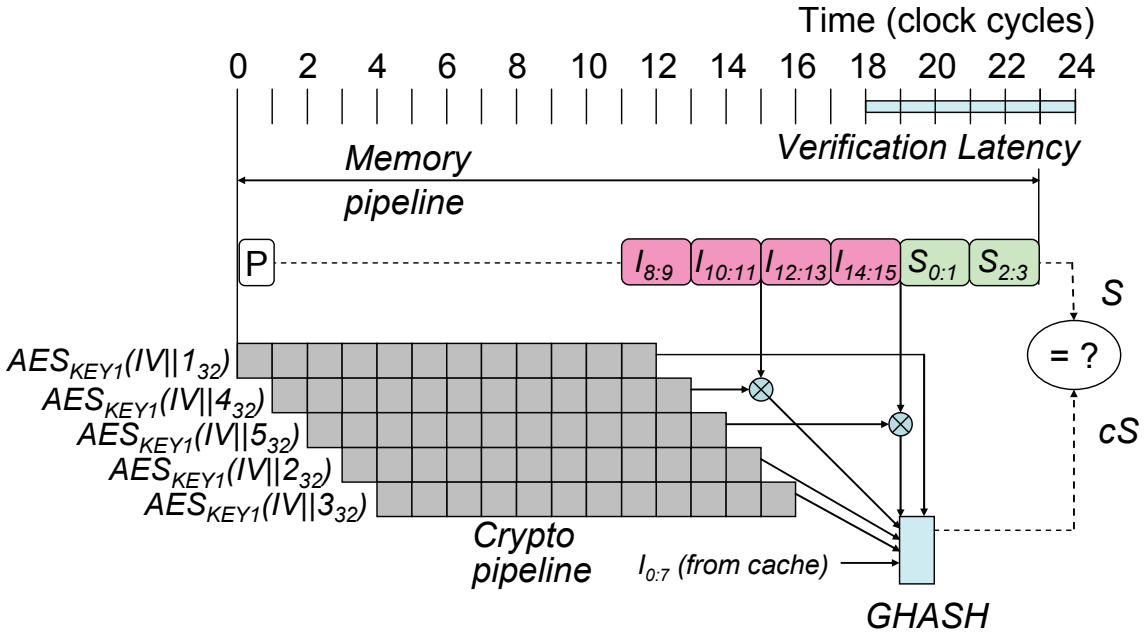


Figure 5.11 Verification Latency for Double Sized Static Protected Blocks Using GHASH, Case 4

5.6 Securing I/O Operations

Providing blanket protection for all input/output devices would be a very tricky, if not impossible, proposition. The first problem is that many devices may exist in separate chips on a board, and the connections between these chips may be vulnerable to physical attacks. Complete protection would require encryption of all communications between these chips, entailing a matched, customized chipset with cryptographic hardware on every chip, and introducing high performance overheads.

Another problem with securing input and output is the wide variety of I/O devices. Some devices transmit or receive a byte at a time; others may stream large amounts of data. Furthermore, other than standardized bus protocols, memory mapped interfaces are not consistent. Supporting such a wide variety of devices could require custom hardware on the CPU to communicate with each device.

Rather than enforce the security of I/O at the hardware level, we propose to provide hardware acceleration for cryptography to be performed in software. We do this by exposing the hardware used for encrypting/decrypting and signing protected blocks on a memory mapped interface. This will allow software developers to use the existing hardware to efficiently and securely perform cryptographic operations. Device drivers for existing peripherals can thus be modified to use the cryptographic accelerator in ways that are appropriate for each individual peripheral. Software development expertise, which is much more prevalent than hardware development expertise, can thus be leveraged to increase security. Furthermore, performing cryptography using an on-chip hardware resource rather than a software implementation of a cryptographic algorithm reduces the vulnerability to side-channel attacks.

The memory mapped interface for accessing the cryptographic software is shown in Figure 5.12. This interface is agnostic with respect to the underlying cryptographic system used (AES encryption vs. OTP, or CBC-MAC vs. PMAC vs. GCM). The interface shown here is for encrypting and/or signing 32 byte blocks; it could easily be modified to support smaller or larger blocks as needed. The right hand column in the figure depicts several 128-bit registers, a 16-bit register, and an 8-bit register. The addresses in the left column are relative to a base address, which will be assigned by the processor designer. We assume a byte-addressable architecture.

The 128-bit registers allow the software developer to specify the inputs for, and read the outputs from cryptographic operations. Inputs include the keys to be used (the number of which will be determined by the underlying cipher) and the initial value *IV* to be used for encryption and/or signature generation (again, dependent on the underlying cipher). If the operation to be performed includes encryption, the 32 byte plaintext block must be loaded into the two plaintext registers, one for each 16 byte sub-block. If the operation includes decryption, then the ciphertext block must be loaded into the two ciphertext registers. Finally, the appropriate 16-bit control word should be set to start the cryptographic operation. The 8-bit status word should be polled until it indicates that the operation is complete. At that point, the results of the operation should be read from the plaintext, ciphertext, and/or signature registers as appropriate.

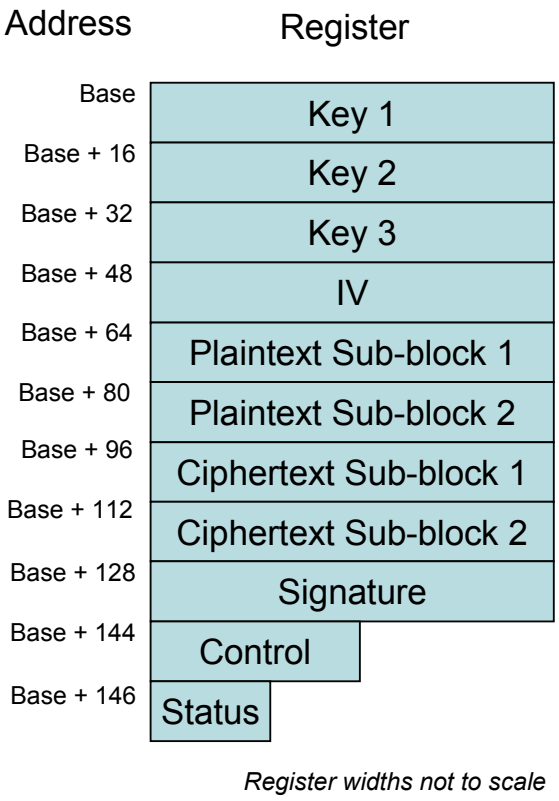


Figure 5.12 Memory Mapped Interface for Cryptographic Acceleration

One possible format for the control word is shown in Figure 5.13, with the most significant bit on the left and the least significant on the right. This format should, of course, be tailored to fit the underlying cipher implementation. The sample format includes three 3-bit fields for selecting each of the up to three keys used in the cryptographic operations. Each key may be user defined (specified in the Key *n*) registers, the CPU key, or any of the up to three program keys for the currently running secure executable. A bit is used to specify whether the desired operation is encryption or decryption; another bit specifies whether or not signature generation is desired.

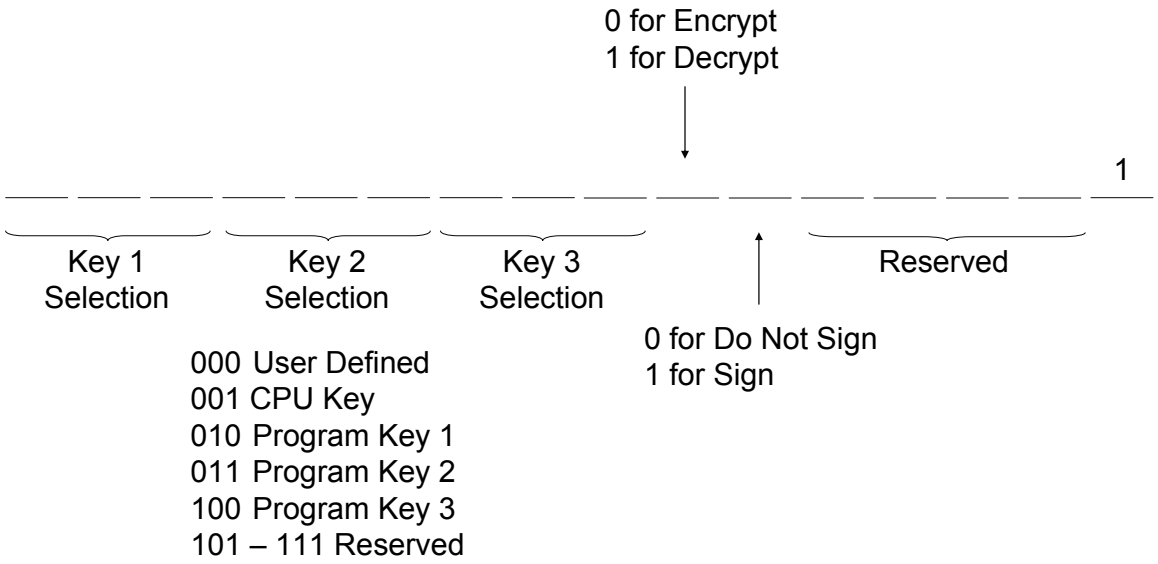


Figure 5.13 Control Word Format

The state machine controlling the cryptographic hardware will look for a 1 in the lowest bit to begin operations. It will then set this bit to 0 and also zero out the status register. The order of encryption and signature generation will be determined by the underlying hardware; greater flexibility may be implemented but at the cost of greater complexity. When the cryptographic operation is complete and all results are in their respective registers, the state machine will set the status register's value to 0x01, which will trigger the polling software. An interrupt system could be implemented if desired to prevent polling, but this would again increase complexity.

5.7 Dynamically Linked Libraries and Dynamic Executable Code

At this point, we must consider two special cases. The first involves dynamically linked libraries (DLLs), which contain binary executable code that is potentially shared

among multiple programs. The simplest option would be to forbid the use of DLLs on the secured system, but that might prove too restrictive. A better option would be to perform a secure installation of any libraries that will be needed, using a single set of program keys for the libraries. Three additional registers would be needed to store the library program keys in addition to the currently running program's keys. During secure loading, the library keys should be read in and decrypted. When a program jumps to a library function, the library keys would then be used to decrypt and authenticate the instructions and static data in the library.

The second case involves instructions that are generated at runtime, including just-in-time compilation and interpreted code. One option is to flag pages containing dynamically generated instructions as unprotected. Another option would be to have the program generating the instructions insert signatures as instruction blocks are created. This requires that the generating program be trusted, and thus the output of the program would also be trusted. Still another option is to treat dynamic instructions like blocks of dynamic data and protect them accordingly [41].

5.8 Comments

A computer architect must address many issues when designing a secure processor. This chapter has addressed issues that apply either to protecting instructions and static data, or to protecting instructions, static, and dynamic data. The architectural solutions we have herein proposed should allow such protection without incurring a large performance overhead. However, the designer must still face challenges peculiar to the protection of dynamic data; these are addressed in the next chapter.

CHAPTER 6

SECURE PROCESSOR DESIGN CHALLENGES FOR PROTECTING DYNAMIC DATA

This chapter continues the theme of addressing challenges that arise when designing secure processors. Here, we focus on challenges related specifically to protecting dynamic data. We first discuss the troubling issue of sequence number overflows. The remainder of the chapter is related to the management of the tree-like structure mentioned in Section 4.2.

6.1 Preventing Sequence Number Overflows

When using OTP cryptography, each encryption operation must use a unique pad. Sequence numbers are used to ensure pad uniqueness for the encryption of dynamic data. The sequence number associated with a protected block must be incremented whenever that block is encrypted (i.e., whenever it is evicted from the cache). A problem arises when the sequence number overflows, that is, when an increment results in it returning to its starting value. Using this overflowed sequence number to calculate a pad for OTP encryption will result in pad reuse, which violates the very principle of OTP.

The obvious solution to prevent sequence number overflows is to use large sequence numbers that are highly unlikely to ever overflow. For instance, if a 32-bit

sequence number is incremented at a rate of one gigahertz (GHz), it will overflow in 4.29 seconds. A 64-bit sequence number incremented at the same rate will overflow in 585 years, which is likely to be far beyond the useful life of the system. However, larger sequence numbers lead to greater memory overhead and longer latency times.

The solution to this quandary is a split sequence number scheme, first introduced by Yan *et al.* [42]. In the split sequence number scheme, each dynamic protected block has its own 8-bit sequence number, called the minor sequence number. Several minor sequence numbers are associated with a 56-bit major sequence number. A protected block's full sequence number consists of the major sequence number concatenated with the minor sequence number, yielding a total of 64 bits. The split sequence number scheme used in our architecture is depicted in Figure 6.1. We assume a sequence number block of 32 bytes, which allows 25 8-bit minor sequence numbers to be associated with a single 56-bit major sequence number. Whenever any of the minor sequence numbers overflow, the major sequence number is incremented and all the data blocks associated with that major sequence number must be re-encrypted and re-signed.

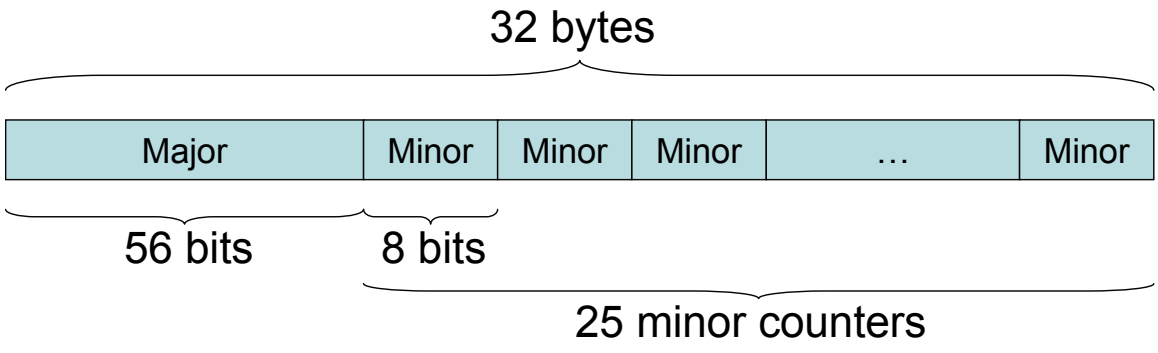


Figure 6.1 Split Sequence Number Block Layout

Let us assume a four kilobyte page size for our example system. Using 32 byte protected blocks, we thus have 128 data blocks per page when storing signatures in a signature table, and 85 data blocks per page (plus some padding) when embedding signatures with protected blocks. As we have seen above, each sequence number block contains 25 sequence numbers, thus requiring six sequence number blocks for the signature table case and four sequence number blocks for the embedded signature case. In both cases, part of the final sequence number block for each page will be unused. The remainder of this chapter will assume that we are using embedded signatures, and thus require four 32 byte sequence number blocks per page of dynamic data.

6.2 Efficiently Managing the Tree

Perhaps the greatest challenge in protecting dynamic data is managing the tree-like structure that was first introduced in Section 4.2. This structure is necessary to prevent sophisticated replay attacks that replay a protected block, its signature, and its sequence number. The tree must be managed in an efficient manner to minimize the performance overhead that it will introduce. Note that if sequence numbers are stored in an on-chip resource, this tree-like structure is not necessary. Storing sequence numbers in an on-chip resource thus reduces overall complexity and performance overhead, but at the cost of limiting the number of protected dynamic blocks. For the remainder of this section, we assume that sequence numbers are stored off-chip, requiring the tree to protect them.

Our approach to managing the tree during secure execution is event driven. When protecting static blocks, the primary event of interest is an instruction cache miss, or a data cache miss on a static block. The complexity required for protecting dynamic data

blocks is much greater due to the need to defend against replay attacks. In addition to high security, we would like to introduce as little overhead as possible. To that end, we have associated operations on each part of the tree with events during secure execution. The goal is to associate high-overhead operations with the rarest events, while ensuring that more common events have lower overhead.

The most frequent of the events discussed in this chapter should be data cache misses. Therefore, the data cache miss should be optimized if at all possible. A block's sequence number is required for both decryption and signature generation, putting sequence number fetching on the critical path of a data cache miss. We introduce an additional on-chip cache resource to reduce the overhead of sequence number retrieval. This cache, called the sequence number cache, will hold the sequence number blocks discussed above in Section 6.1. As one sequence number block contains sequence numbers for multiple data blocks, the sequence number cache exploits both temporal and spatial locality of dynamic data accesses.

6.2.1 Page Allocation

The secure structures required for the dynamic data protection architecture must be prepared for each dynamic data page that is allocated. First, its sequence number blocks must be initialized and used to calculate the initial page root signature (see Figure 4.2). The sequence number blocks and the page root signature must be written to memory in their appropriate reserved areas. The starting address or offset from a known starting address for the page's sequence number blocks must be added to the page's entry in the page table. Secondly, the signatures for the page's data blocks must be calculated and stored in the appropriate location.

One option for implementing these procedures is to assume that the operating system is trusted and allow it to perform the necessary operations on memory allocation. This could potentially introduce high overhead. The other option is to perform the operations in hardware and provide an instruction allowing the OS to trigger them. We choose the latter option for both procedures.

Sequence number blocks must be initialized and used to calculate the page root signature before the allocated page can be used. One approach to calculating the page root signature would be to sign the four sequence number blocks using the mode of choice and then XORing the resulting signatures together. Another method would be to extend the desired mode to sign all four blocks at once. For CBC-MAC, this simply requires extending the chain of AES operations. Doing this for PMAC would effectively be the same as the method previously mentioned. For GCM, the GHASH function must be extended to handle eight 128-bit blocks. This can be implemented with the same hardware used to sign two 128-bit blocks by modifying the state machine that controls the GMULT unit.

The program root signature is calculated by XORing the page root signatures of each dynamic data page. Thus, when a new dynamic data page is allocated, the program root signature must be updated by XORing it with the newly calculated page root signature. All calculations on the program root signature must be performed on-chip. As stated in Section 4.2, it must never leave the CPU in plaintext form.

The other task required for new dynamic data pages is data block signature initialization. This could be done on page allocation, but that could introduce significant overhead. Instead, we propose to create a block's signature on its first write-back. A

block initialization bit vector must be established with a bit for each data block in the new page (85 bits for our example system with embedded signatures). This bit vector specifies which data blocks in the page have been used, with each block initially marked as unused. The block initialization vector is stored in the page table.

The memory structures described above are summarized in Figure 6.2. Part (a) shows the protected dynamic data page with embedded signatures. Part (b) shows the new fields required in the page table, which must be loaded into an expanded TLB. The first field specifies whether this page contains static or dynamic data. The second field is the block initialization vector. The third field is a pointer to the page's root signature in the page root signature table (part (c) in the figure). The final field is a pointer to the first sequence number block for the page (part (e) in the figure).

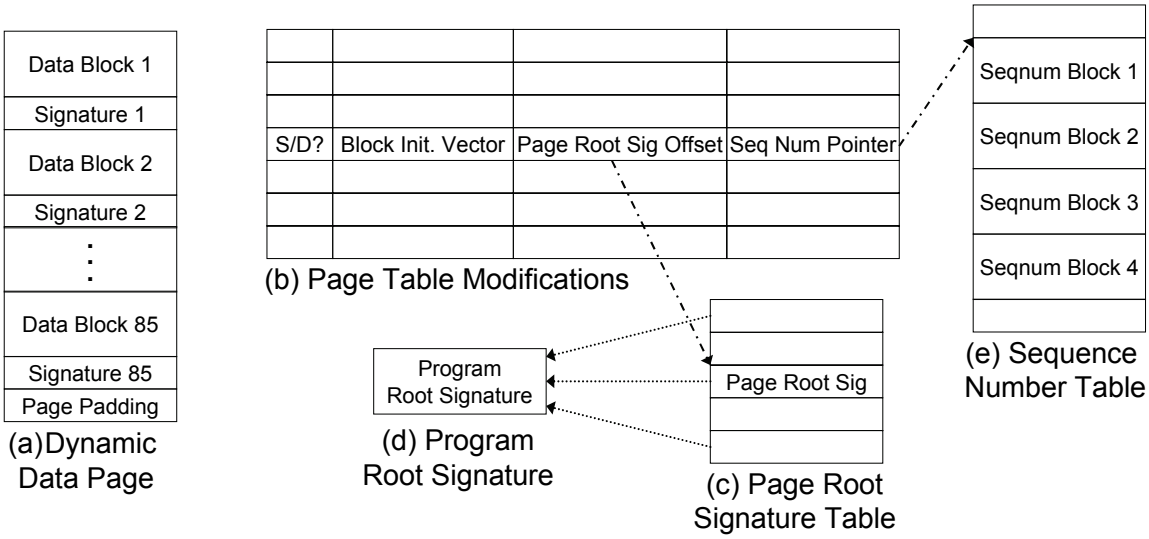


Figure 6.2 Memory Structures for Protecting Dynamic Data

6.2.2 TLB Miss and Write-back

On a TLB miss, information about a data page is brought into the TLB. If the page in question is a dynamic page, the aforementioned extra data required by this architecture must be loaded from the page table and stored in the TLB at this point. The page root signature for the data page should also be stored in the TLB. The integrity of the program root signature is also verified at this time. The signatures from every active dynamic data page are retrieved from the TLB or from memory. These signatures are XORed together to recalculate the current program root signature. If the calculated program root signature does not match that stored on on-chip, then the page root signatures have been subjected to tampering and an exception is raised.

The overhead introduced by the architecture on a TLB miss depends on the number of protected dynamic data pages at the time of the miss. It also depends on design choices made when implementing the architecture. The page root signatures for every protected data page are required. Signatures currently residing in the TLB should be used, as the data in memory might be stale. All signatures not currently in the TLB must be fetched from memory.

This situation leads to a design choice. Consider the case where the TLB contains a noncontiguous subset of the total page root signature table. In some memory architectures, fetching only the signatures not currently in the TLB would introduce greater memory overhead than simply fetching all signatures and ignoring those already in the TLB. This is due to the longer latencies introduced by starting new memory fetches to skip the currently cached signatures. At the cost of additional TLB controller

complexity, control logic could be developed to determine the optimal operation on a signature-by-signature basis.

Our example system has a memory latency of 12 clock cycles for the first eight byte chunk, and two clock cycles for subsequent chunks. Fetching a 16 byte signature by initiating a new fetch operation would cost 14 clock cycles. Fetching the same signature as part of a longer fetch would only cost four clock cycles. Starting new memory fetches to skip signatures currently in the TLB is only advantageous when four signatures must be skipped. Therefore, we choose the simpler implementation of fetching all page root signatures on a TLB miss and simply substituting those found in the TLB.

After each signature becomes available from memory, a simple XOR operation is required for recalculating the program root signature. Once the final signature has been processed, the recalculated program root signature is compared with that stored on the chip. This operation takes less than one clock cycle. Therefore, the added overhead on a TLB miss due to program root signature verification is simply the time required to fetch the page root signatures for all protected data pages. This overhead, $t^{TLBmiss}$, may be calculated for our example architecture according to Equation (6.1), in which *np* represents the number of protected dynamic data pages. The first term in the equation covers fetching the two chunks comprising the first signature while the second term covers fetching the remaining signatures.

$$t^{TLBmiss} = 14 + [(np - 1)\, x\, 4] \tag{6.1}$$

A page root signature will be updated when the sequence number for a data block within that page is incremented. The program root signature will also be updated at that time. See Section 6.2.3 below for discussion on the handling of sequence numbers. Thus

the only action required upon a TLB write-back is to write the page root signature and block initialization bit vector contained in the TLB entry being evicted to memory.

TLB write-backs thus introduce negligible overhead. If the page root signature contained in the entry to be evicted is not dirty, then no operations are required. If it is dirty, the only required operation is to place the appropriate page root signature and bit initialization vector into the write buffer, which will independently write it to memory when the bus is free.

6.2.3 Sequence Number Cache Miss and Write-back

When a block's sequence number is needed, it will first be sought in the sequence number cache. If the requested sequence number is not found in the sequence number cache, it must be fetched from memory. At this point, the integrity of the sequence numbers from the data page in question must be verified. This requires all four sequence number blocks associated with the page. These blocks may be retrieved from the cache or from memory as appropriate. The four sequence number blocks are signed to calculate the page root signature as described in Section 6.2.1. This calculated page root signature is checked against that stored in the TLB. If they do not match, then a trap to the operating system is asserted.

Some of the sequence number blocks needed to calculate the page root signature may already be cached; the rest must be fetched from memory. As with the TLB miss handling scheme, the implementation must balance overhead versus complexity. For our sample implementation, we choose a scheme of moderate complexity. On a sequence number cache miss, the sequence number cache is probed for the page's first sequence number block. If it is found in the cache, the cache is probed for the next block and so

forth until a block is not found in the cache. A memory fetch is initiated for the first sequence number block not found in the cache and all subsequent sequence number blocks associated with that page. Further probing for the rest of the blocks occurs in parallel with the memory fetch. If a sequence number block is found in the cache, the cached version is used and the version from memory is ignored. The blocks that were not previously cached are inserted in the cache.

The performance overhead incurred on a sequence number cache miss depends on the amount of data fetched from memory and the chosen cryptographic mode for calculating the signature. We may set the upper bound for our example system by examining the worst case scenario of having to fetch all four sequence number blocks. The memory latency in this case would be 43 clock cycles, including one cycle required to probe the cache for the first block, 18 cycles to fetch the first block, and eight cycles to fetch each remaining block. The total performance overhead will depend on the cryptographic mode; PMAC would lead to a total overhead of 55 clock cycles, while GCM with a single-cycle implementation of GMULT would require 45 clock cycles.

When sequence number blocks are evicted from the sequence number cache, no cryptographic activity is required. Furthermore, the page root signature is updated during data cache write-back, and will be written to memory during a TLB write-back. Thus the only operation required is to place the evicted sequence number block in the write buffer to be written to memory when the bus is available. This introduces negligible overhead.

6.2.4 Data Cache Miss on a Dynamic Block

Data block verification is performed on data cache read misses for dynamic blocks and write misses on blocks that have already been used. Therefore, on a write

miss the first task is to check the block's entry in the block initialization bit vector in the TLB. If the block has not yet been used then no memory access is required. The cache block is simply loaded with all zeros, preventing malicious data from being injected at this point.

If the miss was a read miss or a write miss on a previously used block, then the data block must be fetched, decrypted, and/or verified using one of the methods described above. Recall that a protected block's sequence number is required for calculating its signature, and also for decrypting the block when using OTP or GCM. Therefore, fetching the sequence number is in the critical path for dynamic data verification. The sequence number is retrieved as described above. Once the sequence number is available, the cryptographic operations may commence in parallel with fetching the data block from memory.

We have seen that the first task that must be performed on a data cache miss is to request the appropriate sequence number form the sequence number cache. In our sample system, this takes only one clock cycle on a sequence number cache hit, and up to 55 clock cycles when using PMAC, or 45 clock cycles when using GCM. Once the sequence number is available, the verification timing for a dynamic block is the same as for a static block.

6.2.5 Data Cache Write-Back

The data cache write-back procedure must be modified to support integrity and confidentiality. When a dirty data block from a dynamic data page is chosen for eviction, it must be encrypted and its signature calculated. Once again, the sequence number fetch is on the critical path, so the current sequence number must be fetched before any other

operations. Once the sequence number is available, the minor sequence number must be incremented. If the increment causes a minor sequence number overflow, then the major sequence number must also be incremented. Such an increment requires special handling as described below.

Regardless of whether or not there was a minor sequence number overflow, the page root signature and program root signature must be updated at this point. The signature of the sequence number block in its existing form is XORed with the page root signature contained in the TLB, effectively subtracting it. The signature of the updated sequence number block then added into the page root signature via another XOR. The same operations are performed on the program root signature to update it. These operations calculate the signature of the sequence number twice, once before and once after the sequence number increment.

The pre-increment signature generation can be eliminated if the sequence number cache lines are widened to include the sequence number block's current signature. This is similar to the technique described in Section 5.1.2. The signature would be populated when it is calculated as part of sequence number verification on a sequence number cache miss, and also after a sequence number increment. As before, this decreases performance overhead at the cost of greater cache complexity.

Once the new sequence number is available, the dynamic data block under eviction may be encrypted and/or signed. If hardware cryptographic resources permit, the cryptographic operations for encryption may be interspersed with those for updating the page root signature. When using PMAC and cached sequence number signatures, this leads to a latency of 27 clock cycles after the sequence number is available; this drops to

18 cycles when using GCM with a single-cycle GMULT implementation. When running with double sized protected blocks, the latencies increase to 31 and 21 clock cycles for PMAC and GCM, respectively.

6.2.5.1 Minor Sequence Number Overflow

If a minor sequence number overflow occurs, the major sequence number must be incremented. This requires that all data blocks associated with that sequence number block and are not currently in the cache must be fetched, decrypted and/or validated, re-encrypted and/or re-signed, and written back to memory. Those that are currently in the data cache may be ignored, but the rest must be fetched from memory, re-encrypted and/or re-signed, and written back to memory.

The best case scenario would occur when all blocks reside in the cache. In this case, 24 cache probes must be performed, which may be overlapped with the cryptographic operations required to update the page root signature and encrypt/sign the data block. The best case overhead in the event of a minor sequence number overflow is thus either 24 clock cycles or the time required to evict a dynamic protected block without an overflow, whichever is longer.

The worst case scenario is when none of the other 24 blocks are available in the cache, and the evicted block is not the first among the 25. In this case, 24 blocks must be fetched from memory, with a new memory access started. (In our example system with embedded signatures, it is faster to start a new memory fetch to skip an unwanted block and its signature than to simply continue fetching and ignore the unwanted data.) The time required to fetch all 25 protected blocks and their signatures is 308 cycles. Assuming a pipelined AES unit and OTP (or GCM) cryptography, cryptographic

operations using the old and new major sequence number can be interleaved; thus the final protected block will be re-encrypted and re-signed at the same time that its previous signature has been verified. Thus, with PMAC, the total worst case overhead from a minor sequence number overflow is 317 cycles. With GCM, it would only be 309 cycles, as GHASH operations would complete while fetching the final signature. These figures assume infinite write buffers; in an actual system there would likely be additional overhead from writing back the re-encrypted and/or re-signed protected blocks due to waiting for write buffers to become free.

One option that should be considered is integrating this functionality with the DMA controller. Such an integration would allow the processor to continue executing while the re-encryption and/or re-signing takes place in the background. The processor would only have to stall if there were a cache miss on one of the 25 affected protected blocks.

6.3 Comments

We have explored the challenges related to protecting dynamic data, including how to efficiently store sequence numbers and prevent sequence number overflows. We have also developed a tree-like structure for defending dynamic data against replay attacks. This tree may be managed efficiently by limiting its scope to protecting sequence numbers and maintaining it on events such as TLB and cache misses. The approaches discussed in this chapter allow the protection of dynamic data without incurring prohibitively large amounts of overhead.

CHAPTER 7

SECURE PROCESSOR DESIGN EVALUATION

Cycle-accurate simulation software was used to evaluate the performance overhead of our proposed secure processor architecture and explore various design choices and tradeoffs as discussed above. This chapter describes the methodology used in evaluating the architecture with the simulator and the observed results. We start with an overview of the experimental flow. We then discuss the simulator we have developed, the simulation parameters used in our evaluation runs, and the benchmarks that are chosen for simulation. We then present the results of our simulations, evaluating the performance of the techniques proposed in Chapters 5 and 6 for each design choice of interest. Finally, we use the simulation results to develop a mathematical model for predicting performance overhead.

7.1 Experimental Flow

The experimental flow for evaluating our proposed architectures is illustrated in Figure 7.1. We start with uncompiled source code for benchmark applications of interest, which are described in Section 7.3 below. These are compiled using a cross-compiler to generate executable binaries in the standard Executable and Linkable Format (ELF) [43].

The cross-compiler encodes the executables for the Advanced Reduced Instruction Set (RISC) Computer Machine (ARM) instruction set. These binaries may then be run in the simulator under a baseline configuration without security enhancements. The simulator, which is described in Section 7.2, mimics an embedded microprocessor based on the ARM architecture, and produces cycle-accurate execution times for benchmarks run thereon. The benchmark binaries are then run with the simulator configured to model a secure processor architecture with the various features as described in Chapters 4-6. Once simulation runs are completed, the relevant results can be mined from the simulator outputs.

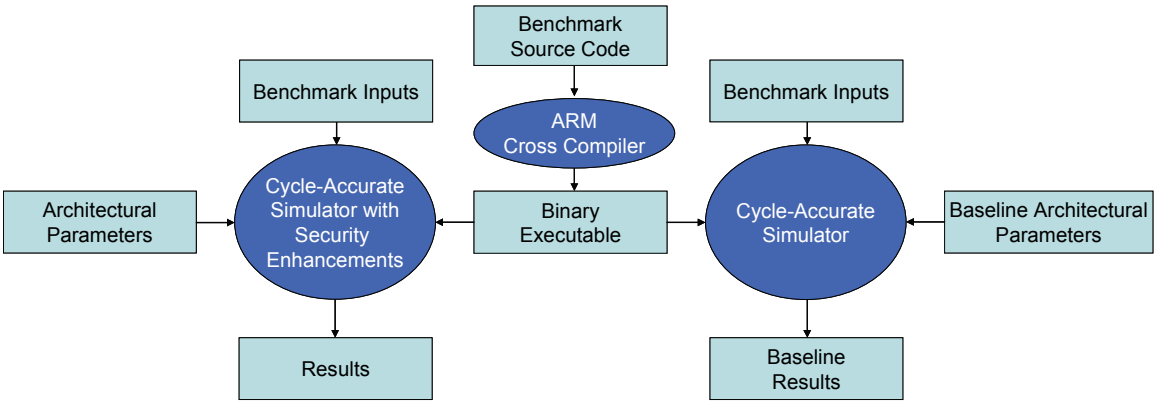


Figure 7.1 Experimental Flow

7.2 Simulator and Parameters

The simulator used to evaluate the performance of the proposed architectures is a derivative of sim-outorder, the most detailed simulator from the SimpleScalar suite [44]. The simulator is updated to provide a cycle-accurate timing analysis for our secure

processor, and supports many of the design choices and tradeoffs discussed in this dissertation. Specifically, the simulator supports the following options:

- Protecting software and/or data integrity and confidentiality
- CBC-MAC, PMAC, and GCM cryptographic modes
- Storing signatures in a signature table or embedded with protected blocks
- Signature victim cache presence and size
- Sequence number cache size
- Instruction verification buffer (IVB) presence and depth

As our proposed architecture is event-driven, so is our simulator: the bulk of the simulator updates are in various event handlers. The instruction and data cache miss and TLB miss handlers have been updated as appropriate. Sequence number cache support was added and its miss handler written. Data structures and routines were written to provide support for victim caches and IVBs; these were written to reuse as much code as possible when using separate instances of these resources to protect static and dynamic data. Using the IVBs required modifications to the instruction issuing and memory fetching code. The source code for our updated simulator, which we call simsec-outorder is available as an electronic appendix to this dissertation.

Performance overhead is analyzed by using the simulator to run the benchmark programs described in Section 7.3. The SimpleScalar metric of interest for performance overhead analysis is `sim_cycle`, the number of simulated clock cycles required for the benchmark to run to completion. After simulation is complete, this value is mined from the simulation outputs. The performance overhead is reported using normalized

execution time - the value of `sim_cycle` for a secure architecture divided by the value of `sim_cycle` from the appropriate baseline simulation run.

The simulator is configured to simulate two architectures based on ARM Cortex cores. The parameters derived from these two architectures are shown in Table 7.1. The Cortex-M3 [45] is a relatively simple, low-cost processor designed for deeply embedded architectures and implementation on FPGAs. It is a single-issue in-order processor with a single integer pipeline. The Cortex-A8 [46] is a faster and much more sophisticated core. The A8 is dual-issue superscalar in-order processor. The A8 exploits instruction-level parallelism, but its faster clock leads to longer memory fetch times and cryptographic latencies. In our simulation runs, the M3 architecture is used to demonstrate how our security extensions operate in a midrange embedded processor, while the A8 represents a more high-end processor.

Table 7.1 Simulation Parameters

Simulator Parameter	ARM Cortex-M3	ARM Cortex-A8
Branch predictor type	None	Two-Level, 4096-entry global branch history buffer indexed by 10-bit branch history register and 4 bits of program counter
Branch Target Buffer (BTB)	N/A	512 entires, 2-way set associative
Instruction decode bandwidth	1 instruction/cycle	2 instructions/cycle
Instruction issue bandwidth	1 instruction/cycle	2 instructions/cycle
Instruction commit bandwidth	1 instruction/cycle	2 instructions/cycle
Pipeline with in-order issue	True	True
I-cache/D-cache	4-way, first level only, 1 KB, 2 KB, 4 KB, or 8 KB	4-way, first level only, 16 KB or 32 KB
Cache Hit Time	1 cycle	1 cycle
I-TLB/D-TLB	32 entries, fully associative	32 entries, fully associative
Execution units	1 floating point, 1 integer	1 floating point, 2 integer
Memory fetch latency (first/other chunks)	12/2 cycles	24/4 cycles
Branch misprediction latency	N/A	13 cycles
TLB latency	30 cycles	60 cycles
AES latency	12 cycles	24 cycles
GHASH latency	1 cycle	2 cycles

7.3 Benchmark Selection

We select a set of benchmarks for evaluating the performance overhead of our proposed security enhancements on the simulated architectures discussed in the previous section. These benchmarks represent typical tasks that an embedded system might perform. They are selected primarily from the MiBench suite [47], with a few from the MediaBench [48] and Basicrypt [49] suites. The primary criteria used in selecting benchmarks are the cache miss rates. In order to properly exercise the proposed security extensions, high miss rates for at least one of the simulated architectures is desired. Thus,

these benchmarks often represent a worst-case scenario with the greatest possible overhead; other benchmarks with very low cache miss rates would only show negligible overhead.

These benchmarks are described in Table 7.2, along with the number of instructions that will be executed when running each benchmark. Their cache miss rates when simulated on the architectures described above are shown in Table 7.3 and Table 7.4.

Table 7.2 Benchmark Descriptions

Benchmark	Description	Executed Instructions [10^6]
adpcm	ADPCM encoder	732.8
blowfish_enc	Blowfish encryption	544.1
cjpeg	JPEG compression	104.6
djpeg	JPEG decompression	23.4
ecdhb	Diffie-Hellman key exchange	122.5
ecelgencb	El-Gamal encryption	180.2
fft	Fast Fourier transform	301.8
ghostcript	Postscript interpreter	708.1
gsm_d	GSM encoder	1299.4
ispell	Spell checker	817.8
lame	MP3 encoder	1151.8
mad	MPEG audio decoder	287.1
mpeg2_enc	MPEG2 compression	127.5
rijndael_enc	Rijndael encryption	259.3
rsynth	Synthesize text to speech	796.1
stringsearch	String search	3.7
sha	Secure hash algorithm	140.9
tiff2bw	Convert color TIFF to black and white	143.4
tiff2rgba	Convert TIFF image to RGB	151.9
tiffdither	Dither a TIFF image	833.0
tiffmedian	Reduce TIFF image color palette	541.5

Table 7.3 Benchmark Instruction Cache Miss Rates

Benchmark	Instruction Cache Misses per 1000 Executed Instructions					
	Cortex M3				Cortex A8	
	1 KB	2 KB	4 KB	8 KB	16 KB	32 KB
adpcm	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
blowfish_enc	41.05	5.98	0.06	< 0.01	< 0.01	< 0.01
cjpeg	8.08	1.40	0.34	0.08	0.03	0.01
djpeg	12.06	5.81	1.38	0.29	0.08	0.06
ecdhb	30.57	9.25	2.97	0.15	0.03	0.01
ecelgencb	28.30	4.97	1.51	0.08	0.01	< 0.01
fft	107.04	90.60	28.35	1.10	< 0.01	< 0.01
ghostcript	153.96	88.35	31.61	1.69	0.66	0.20
gsm_d	4.30	3.49	2.50	2.22	0.23	< 0.01
ispell	94.20	65.19	19.09	2.99	0.73	0.03
lame	22.56	11.90	2.89	2.12	1.56	1.20
mad	43.20	26.06	25.36	1.79	0.57	0.07
mpeg2_enc	2.29	1.16	0.41	0.20	0.07	0.04
rijndael_enc	130.16	127.79	75.51	11.12	< 0.01	< 0.01
rsynth	113.16	13.67	6.41	2.47	0.01	< 0.01
stringsearch	71.24	42.85	5.92	2.84	0.12	0.12
sha	5.56	0.08	< 0.01	< 0.01	< 0.01	< 0.01
tiff2bw	3.10	2.62	1.31	0.12	< 0.01	< 0.01
tiff2rgba	3.60	2.53	0.63	0.01	< 0.01	< 0.01
tiffdither	43.97	10.32	0.82	0.27	0.01	< 0.01
tiffmedian	1.53	1.15	0.47	0.02	< 0.01	< 0.01

Table 7.4 Benchmark Data Cache Miss Rates

Benchmark	Data Cache Misses per 1000 Executed Instructions					
	Cortex M3				Cortex A8	
	1 KB	2 KB	4 KB	8 KB	16 KB	32 KB
adpcm	1.56	1.42	0.15	< 0.01	< 0.01	< 0.01
blowfish_enc	44.93	30.78	6.01	0.09	< 0.01	< 0.01
cjpeg	29.38	22.90	19.15	3.12	0.72	0.47
djpeg	40.93	25.49	16.78	6.59	2.52	0.58
ecdhb	2.56	0.55	0.13	0.07	0.02	0.01
ecelgencb	1.79	0.32	0.05	0.02	0.01	0.01
fft	33.10	9.37	1.31	0.77	0.69	0.65
ghostcript	26.18	9.08	1.64	0.82	0.58	0.47
gsm_d	1.28	0.69	0.23	0.04	< 0.01	< 0.01
ispell	32.61	17.42	2.14	0.68	0.17	0.03
lame	53.91	38.12	25.58	14.82	7.64	4.82
mad	28.01	21.39	12.70	3.83	2.24	0.24
mpeg2_enc	20.96	11.06	2.34	0.55	0.43	0.38
rijndael_enc	134.19	112.97	65.65	7.08	0.02	< 0.01
rsynth	21.35	11.46	2.22	0.82	0.32	0.29
stringsearch	35.43	17.39	2.96	1.33	0.47	0.33
sha	0.94	0.75	0.74	0.27	< 0.01	< 0.01
tiff2bw	26.25	26.22	26.21	25.36	17.63	0.30
tiff2rgba	50.29	50.27	50.26	50.26	37.45	18.24
tiffdither	10.23	3.88	3.73	3.37	1.91	0.04
tiffmedian	29.43	25.45	22.47	21.14	17.10	6.43

Rather than run the entire set of benchmarks for every permutation of architectural parameters that we investigate, we wish to select a meaningful subset of benchmarks. We make the selection using cluster analysis. For each architectural configuration (processor core plus level - caches), we consider the cache miss rates from the above tables in a Cartesian plane with the level-1 instruction cache miss rate as the abscissa and the level-1 data cache miss rate as the ordinate. A minimum spanning tree method is then used to group the benchmarks into four clusters with similar cache miss rate characteristics. The benchmark closest to the centroid of each cluster is selected to represent the cluster. The selected benchmarks for each configuration are shown in Table 7.5. All 12 of these selected benchmarks will be used when evaluating across all the simulated configurations. For evaluations whose scope is limited to one configuration, only the four benchmarks chosen as significant for that configuration will be used.

Table 7.5 Benchmarks Selected by Clustering Analysis

Benchmark	Configurations Relevant to Benchmarks					
	Cortex M3				Cortex A8	
	1 KB	2 KB	4 KB	8 KB	16 KB	32 KB
blowfish_enc			✓			
cjpeg	✓					
fft	✓		✓	✓		
ghostcript	✓	✓				
lame					✓	✓
mad						✓
mpeg2_enc		✓				
rijndael_enc	✓	✓	✓	✓		
stringsearch		✓			✓	
tiff2bw					✓	
tiff2rgba			✓	✓	✓	✓
tiffmedian				✓		✓

7.4 Results

This section presents the results of both qualitative and quantitative analyses of security extensions to the example system discussed in this dissertation. We start with qualitative analyses of the complexity overhead required to implement our architectures on a processor chip, followed by the extra space in memory required to run a secure program on our architecture. We then present quantitative performance overhead results obtained by simulating the execution of secure benchmark programs.

7.4.1 Complexity Overhead

The architecture we have proposed requires state machines for performing various tasks, logic for address translation, buffers and registers, hardware for key generation,

and a pipelined cryptographic unit. All but the last two of these requirements introduce relatively little additional on-chip area. A physical unclonable function (PUF) unit for key generation requires nearly 3,000 gates [29]. The pipelined cryptographic unit, which is shared among both architectures, introduces the greatest amount of overhead. Assuming that this cryptographic unit follows the commercially available Jetstream JetAES Fast high speed 128-bit AES core [23], the on-chip area it requires should be approximately equal to that required for about 31,000 logic gates. If using GCM, a GMULT unit must is also required. As we have seen in Section 3.3, the most complex (but fastest) GMULT implementation has a complexity on the order of 128^2 (16,384) gates; slower implementations require fewer gates, but introduce a small state machine to control the GMULT unit. Entire high-throughput GCM cores are commercially available, which would cover both the AES and GMULT units, with gate counts ranging from 30,000 to 60,000 [50, 51]. An additional source of complexity is the sequence number cache; its complexity is determined by its size and organization, which are design parameters.

The complexity overhead of the optional signature victim caches and instruction verification buffers may be estimated by treating them as fully associative structures. Each register bit in may be modeled as a latch using 2.5 logic gates. Every entry must also have a comparator, also requiring 2.5 gates per bit. Each entry's output must also be protected by tri-state buffers at 0.5 gates per bit [52].

Each entry in the signature victim cache must contain a 128-bit signature and a tag. In the worst case scenario, the tag would be the full 32-bit address, leading to 160 register bits, with a 32-bit comparator for the tag and a 128-bit array of tri-state

buffers. This leads to a total overhead of 544 logic gates per entry. The victim cache could also be implemented as an array of indices into an on-chip memory. In this case, the 128-bit register for the signature and the array of tri-state buffers would not be required. The memory would be indexed by the number of the entry whose tag matched the address. This would reduce the overhead per entry to 160 gates, but on-chip memory resources would also be required. Furthermore, regardless of which victim cache design was chosen, the instruction and data cache lines must be widened by 128 bits to support storing signatures for placement in the victim caches.

Every IVB entry must contain a tag and two single-bit flags. Again assuming the worst case scenario of a 32-bit tag, we have 34 register bits per entry, with a 32-bit comparator and two tri-state buffers. The IVB overhead is thus 166 logic gates per entry.

7.4.2 Memory Overhead

The memory overhead incurred by protecting instructions and static data is a simple function of the protected block size and the number of instruction blocks in the program. Each signature is 16 bytes long. If 32 byte protected blocks are chosen, then the size of the executable segment of the program increases by 50%. This overhead is reduced to 25% for 64 byte protected blocks, and to 12.5% for 128 byte protected blocks.

The memory overhead required for protecting dynamic data is slightly larger. The data signatures lead to the same overhead figures as for static data and instructions. However, each dynamic data page requires sequence number blocks, additional space in the page table, and an entry in the page root signature table. The sample architecture presented in this dissertation requires six sequence number blocks when storing signatures in a signature table, for a total of 192 bytes per protected dynamic page. Only

four sequence number blocks are required when using embedded signatures, lowering the sequence number overhead to 128 bytes per page.

7.4.3 Performance Overhead

We evaluate the performance overhead of our proposed architectural extensions using the experimental flow and simulator described above. Our strategy is to reveal how the various choices and approaches described in Chapters 5 and 6 would affect the performance of modern processor designs using security extensions. For each choice or approach of interest, we set simulation parameters to isolate its influence and then compare simulation results with theoretical projections.

7.4.3.1 Signature Location

We first evaluate how the signature location influences performance overhead. This influence may be isolated by choosing the GCM cipher mode, which will minimize the time required to calculate signatures and ensure that fetching signatures from memory is the greatest contributor to performance overhead. We fix signature cache sizes at 50% of the data cache size (i.e., an architecture with a 4 KB data cache would have a 2 KB signature cache). We evaluate storing signatures in a signature table, a signature table with 32-entry victim caches, and embedding them with protected blocks. We project that using a signature table without victim caches will incur the most performance overhead, while embedded signatures will provide the best performance. When using a signature table with victim caches, the overhead should be somewhere in between.

The simulation results are graphed in Figure 7.2 - Figure 7.7 and presented numerically in Table 7.6 and Table 7.7 for all architectures described above in

Section 7.2. The contributions to overhead from protecting instructions/static data and dynamic data are shown separately. We find that the observed overheads mostly follow the theorized behavior. Embedded signatures generally provide the best performance. However, in a few instances, using a signature table with victim caches outperforms embedded signatures.

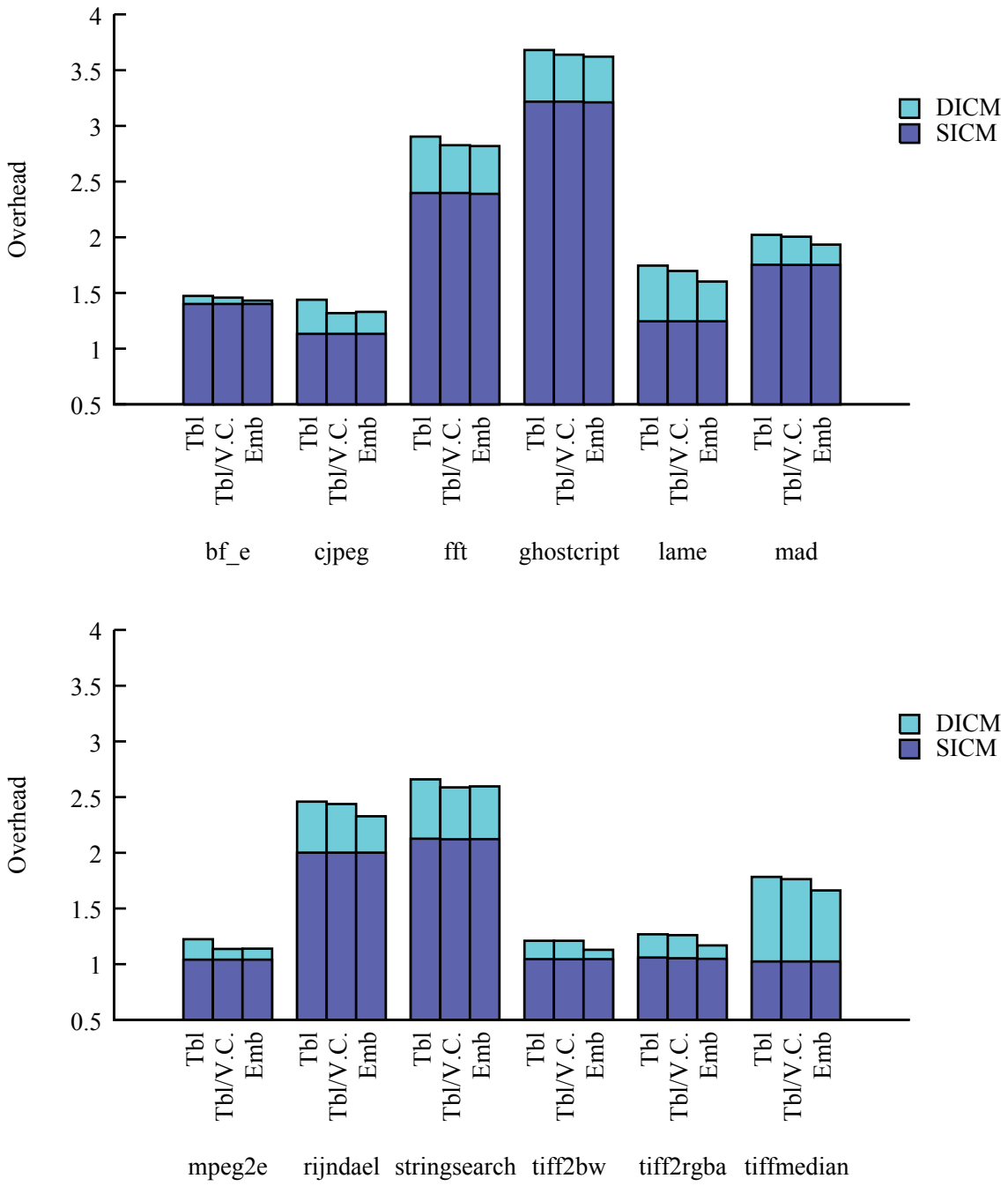


Figure 7.2 Performance Overhead Implications of Signature Location, Cortex M3, 1 KB

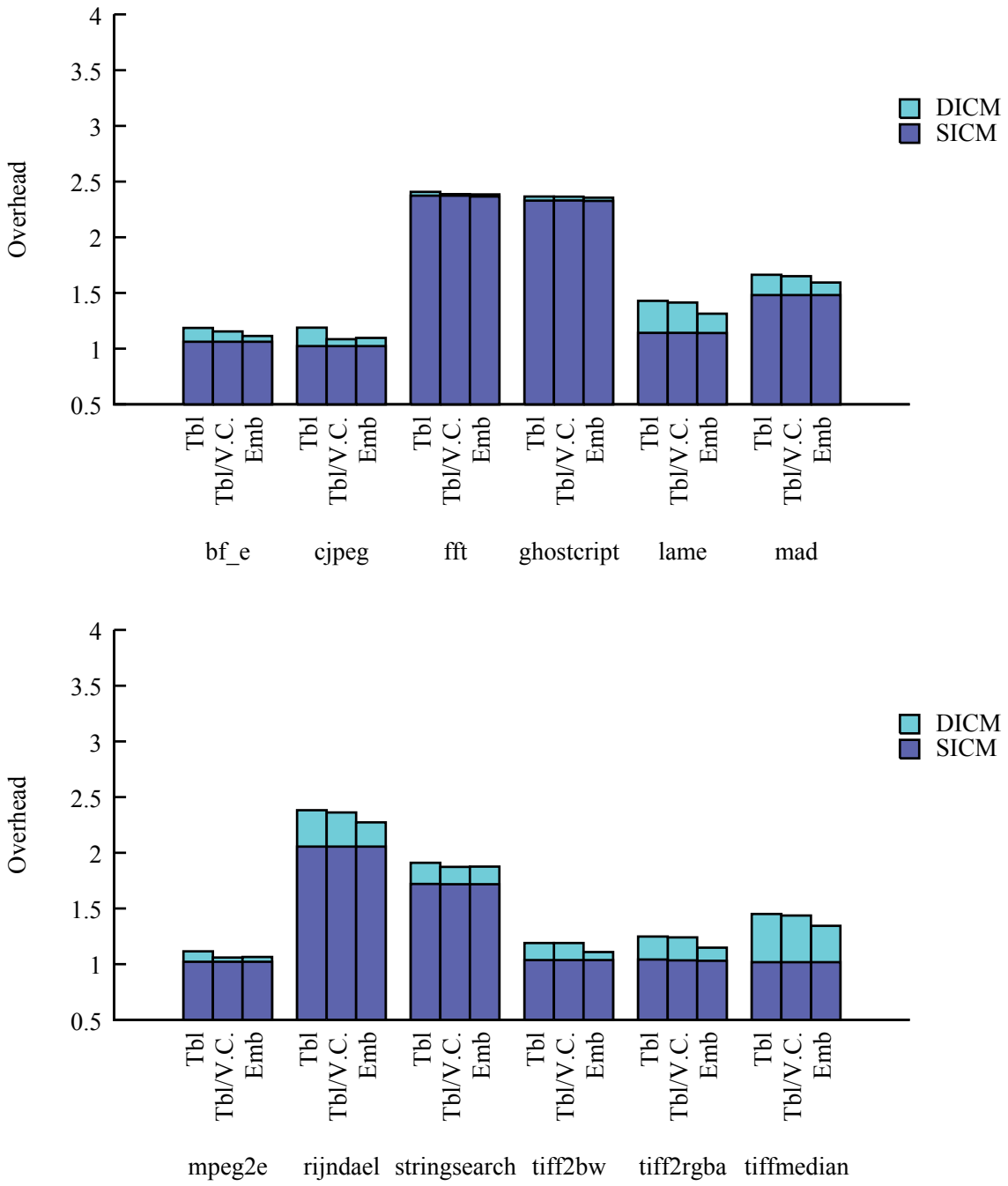


Figure 7.3 Performance Overhead Implications of Signature Location, Cortex M3, 2 KB

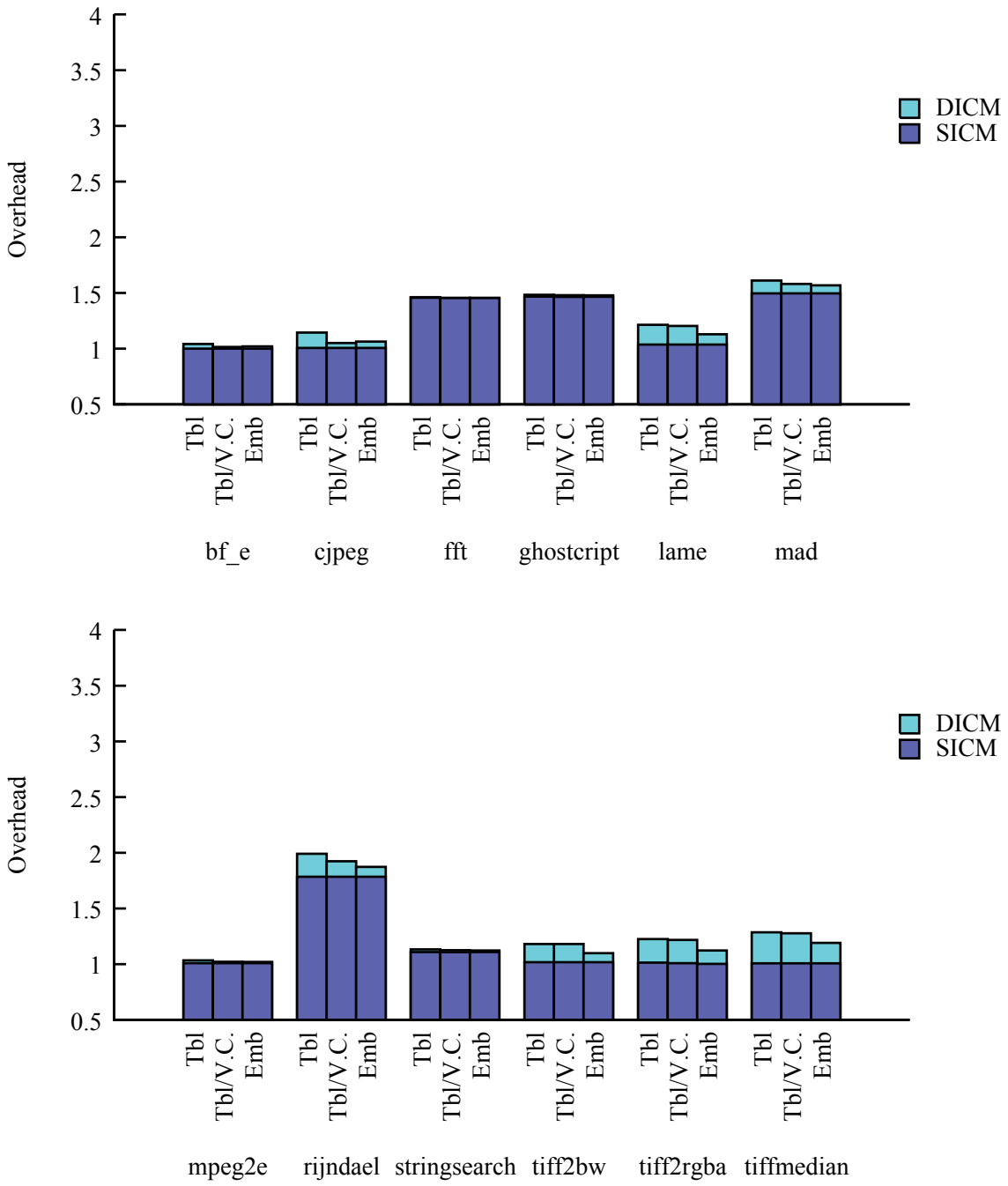


Figure 7.4 Performance Overhead Implications of Signature Location, Cortex M3, 4 KB

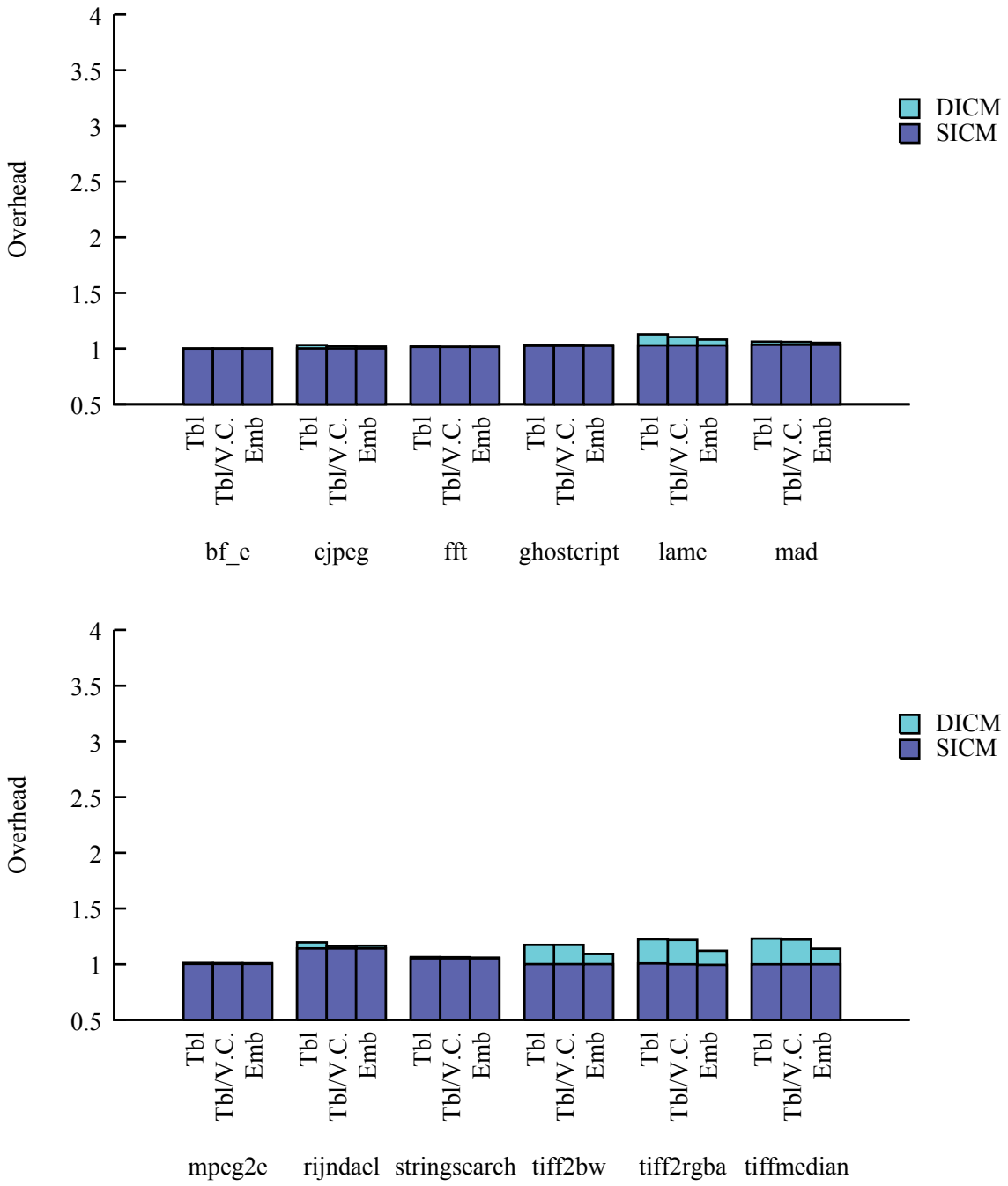


Figure 7.5 Performance Overhead Implications of Signature Location, Cortex M3, 8 KB

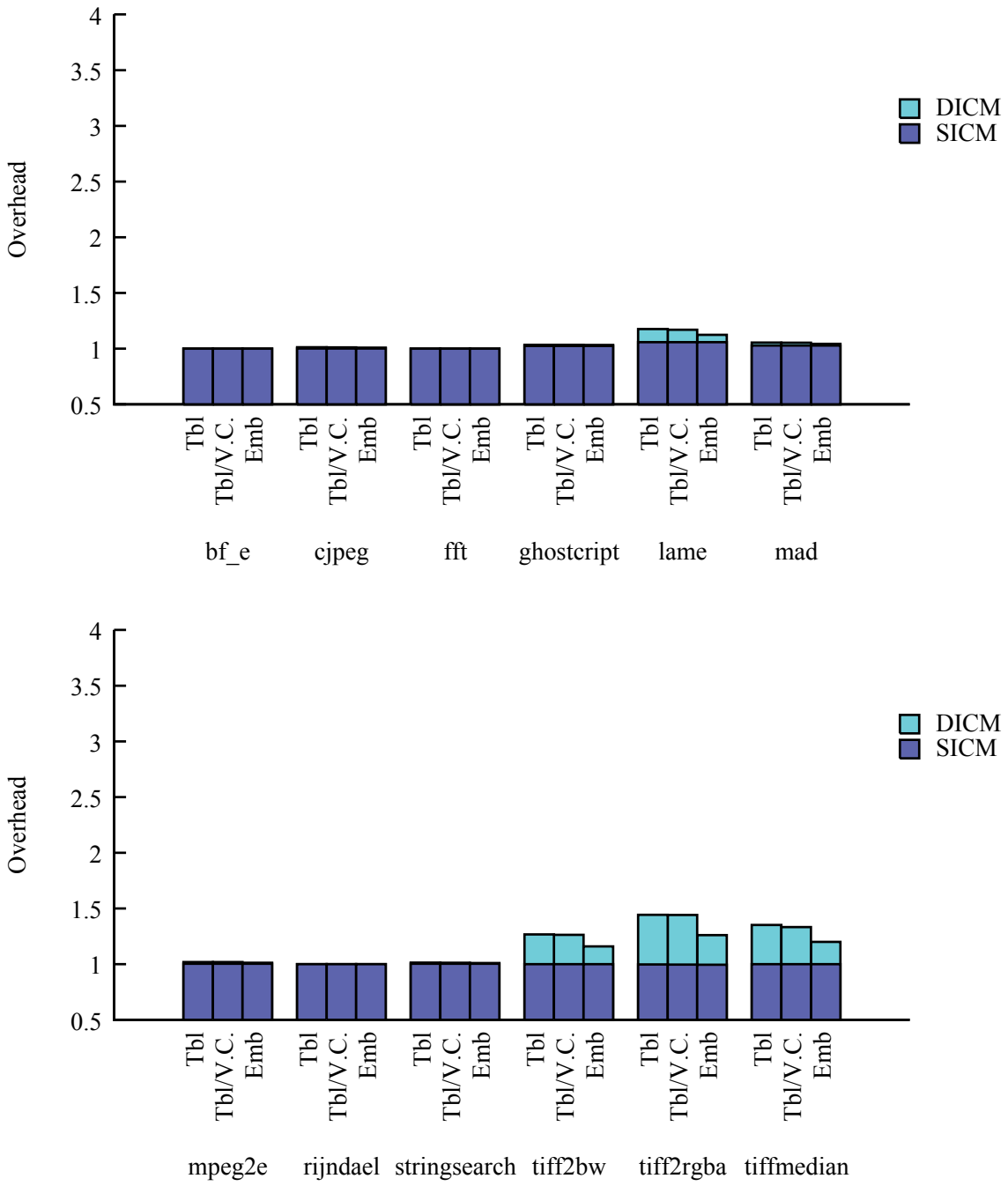


Figure 7.6 Performance Overhead Implications of Signature Location, Cortex A8, 16 KB

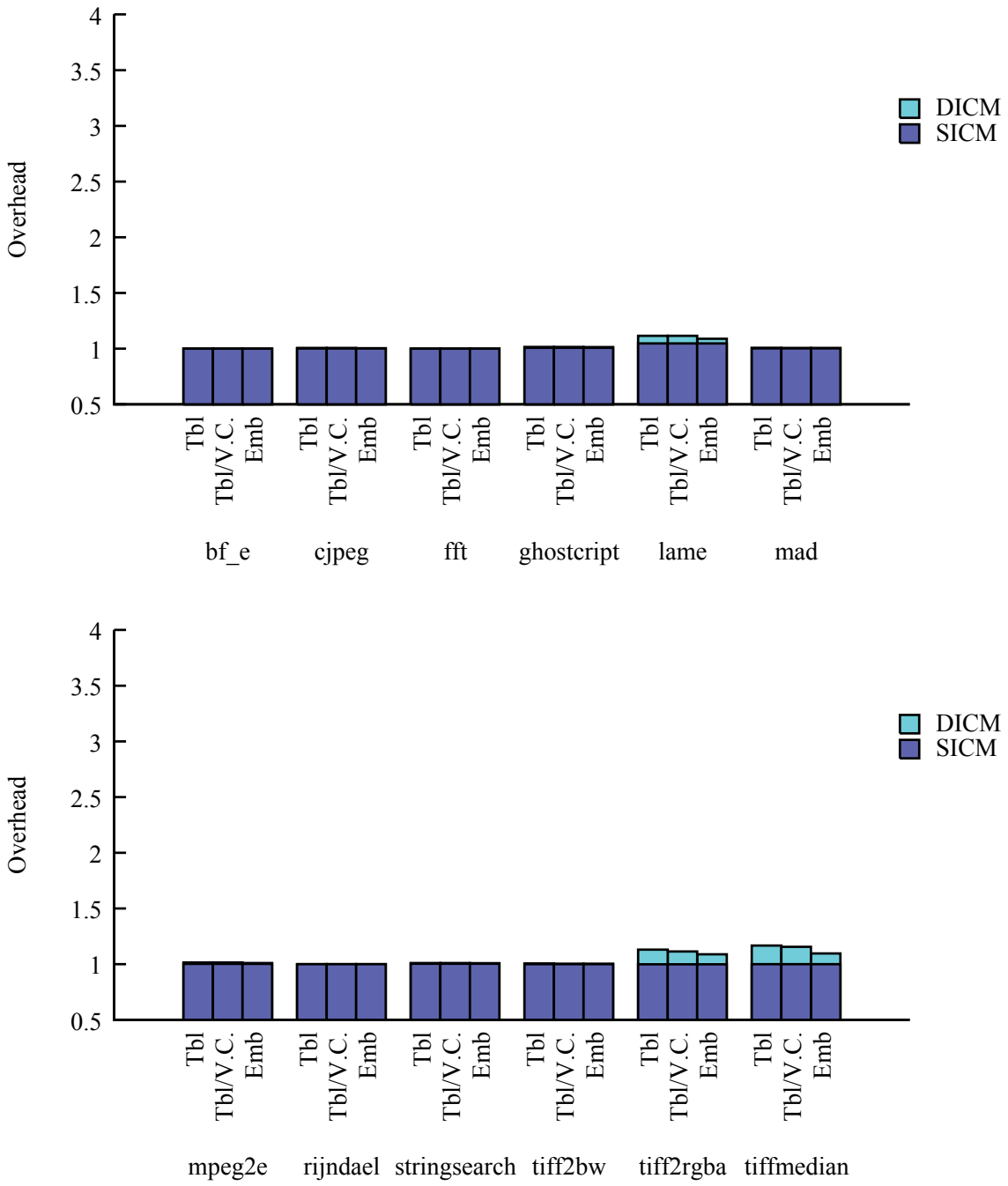


Figure 7.7 Performance Overhead Implications of Signature Location, Cortex A8, 32 KB

Table 7.6 Performance Overhead Implications of Signature Location, Cortex M3

Benchmark	Signature Location	1 KB		2 KB		4 KB		8 KB	
		SICM	Both	SICM	Both	SICM	Both	SICM	Both
bf_e	Tbl	1.40	1.47	1.06	1.19	1.00	1.04	1.00	1.00
	Tbl/V.C.	1.40	1.46	1.06	1.15	1.00	1.02	1.00	1.00
	Emb	1.40	1.43	1.06	1.11	1.00	1.02	1.00	1.00
cjpeg	Tbl	1.13	1.44	1.02	1.19	1.01	1.14	1.00	1.03
	Tbl/V.C.	1.13	1.32	1.02	1.08	1.01	1.05	1.00	1.02
	Emb	1.13	1.33	1.02	1.10	1.01	1.06	1.00	1.02
fft	Tbl	2.40	2.90	2.37	2.41	1.46	1.46	1.02	1.02
	Tbl/V.C.	2.40	2.83	2.37	2.39	1.45	1.46	1.02	1.02
	Emb	2.39	2.82	2.37	2.38	1.45	1.46	1.02	1.02
ghostcript	Tbl	3.22	3.68	2.33	2.37	1.47	1.48	1.03	1.03
	Tbl/V.C.	3.22	3.64	2.33	2.36	1.46	1.48	1.03	1.03
	Emb	3.21	3.62	2.33	2.36	1.47	1.48	1.03	1.03
lame	Tbl	1.25	1.75	1.14	1.43	1.04	1.21	1.03	1.13
	Tbl/V.C.	1.25	1.70	1.14	1.41	1.04	1.20	1.03	1.10
	Emb	1.25	1.60	1.14	1.31	1.04	1.13	1.03	1.08
mad	Tbl	1.75	2.02	1.48	1.66	1.50	1.61	1.03	1.06
	Tbl/V.C.	1.75	2.00	1.48	1.65	1.50	1.58	1.03	1.06
	Emb	1.75	1.93	1.48	1.59	1.50	1.57	1.03	1.05
mpeg2e	Tbl	1.04	1.22	1.02	1.12	1.01	1.03	1.00	1.01
	Tbl/V.C.	1.04	1.14	1.02	1.06	1.01	1.02	1.00	1.01
	Emb	1.04	1.14	1.02	1.06	1.01	1.02	1.00	1.01
rijndael	Tbl	2.00	2.46	2.06	2.38	1.78	1.99	1.14	1.2
	Tbl/V.C.	2.00	2.44	2.06	2.36	1.78	1.92	1.14	1.16
	Emb	2.00	2.33	2.06	2.27	1.78	1.87	1.14	1.17
stringsearch	Tbl	2.13	2.66	1.72	1.91	1.11	1.13	1.05	1.06
	Tbl/V.C.	2.12	2.59	1.72	1.87	1.11	1.13	1.05	1.06
	Emb	2.12	2.60	1.72	1.88	1.11	1.12	1.05	1.06
tiff2bw	Tbl	1.05	1.21	1.04	1.19	1.02	1.18	1.00	1.17
	Tbl/V.C.	1.04	1.21	1.04	1.19	1.02	1.18	1.00	1.17
	Emb	1.05	1.13	1.04	1.11	1.02	1.10	1.00	1.09
tiff2rgba	Tbl	1.06	1.27	1.04	1.25	1.01	1.23	1.01	1.22
	Tbl/V.C.	1.05	1.26	1.03	1.24	1.01	1.22	1.00	1.22
	Emb	1.05	1.17	1.03	1.15	1.00	1.12	0.99	1.12
tiffmedian	Tbl	1.02	1.78	1.02	1.45	1.01	1.29	1.00	1.23
	Tbl/V.C.	1.02	1.76	1.02	1.44	1.01	1.28	1.00	1.22
	Emb	1.02	1.66	1.02	1.34	1.01	1.19	1.00	1.14

Table 7.7 Performance Overhead Implications of Signature Location, Cortex A8

Benchmark	Signature Location	16 KB		32 KB	
		SICM	Both	SICM	Both
bf_e	Tbl	1.00	1.00	1.00	1.00
	Tbl/V.C.	1.00	1.00	1.00	1.00
	Emb	1.00	1.00	1.00	1.00
cjpeg	Tbl	1.00	1.01	1.00	1.01
	Tbl/V.C.	1.00	1.01	1.00	1.01
	Emb	1.00	1.01	1.00	1.00
fft	Tbl	1.00	1.00	1.00	1.00
	Tbl/V.C.	1.00	1.00	1.00	1.00
	Emb	1.00	1.00	1.00	1.00
ghostcript	Tbl	1.02	1.03	1.01	1.02
	Tbl/V.C.	1.02	1.03	1.01	1.02
	Emb	1.02	1.03	1.01	1.01
lame	Tbl	1.06	1.18	1.05	1.12
	Tbl/V.C.	1.06	1.17	1.05	1.12
	Emb	1.06	1.12	1.05	1.09
mad	Tbl	1.03	1.05	1.00	1.01
	Tbl/V.C.	1.03	1.05	1.00	1.01
	Emb	1.03	1.04	1.00	1.01
mpeg2e	Tbl	1.00	1.02	1.00	1.02
	Tbl/V.C.	1.00	1.02	1.00	1.02
	Emb	1.00	1.01	1.00	1.01
rijndael	Tbl	1.00	1.00	1.00	1.00
	Tbl/V.C.	1.00	1.00	1.00	1.00
	Emb	1.00	1.00	1.00	1.00
stringsearch	Tbl	1.00	1.01	1.00	1.01
	Tbl/V.C.	1.00	1.01	1.00	1.01
	Emb	1.00	1.01	1.00	1.01
tiff2bw	Tbl	1.00	1.27	1.00	1.01
	Tbl/V.C.	1.00	1.26	1.00	1.00
	Emb	1.00	1.16	1.00	1.00
tiff2rgba	Tbl	1.00	1.44	1.00	1.13
	Tbl/V.C.	1.00	1.44	1.00	1.11
	Emb	0.99	1.26	1.00	1.09
tiffmedian	Tbl	1.00	1.35	1.00	1.17
	Tbl/V.C.	1.00	1.33	1.00	1.16
	Emb	1.00	1.20	1.00	1.10

7.4.3.1.1 Optimal Signature Victim Cache Size

In addition to evaluating the effects of signature location, we would also like to find the optimal signature victim cache size to balance complexity with performance overhead. We use the same parameters as in evaluating signature location, but fix the architecture on the Cortex M3 with 2 KB caches, as this architecture demonstrates nontrivial performance overhead. We choose the four benchmarks that the clustering analysis found to be significant on this architecture, and simulate them using a signature table. We vary the victim cache sizes among reasonable values, from eight to 64 entries in powers of two, and compare the resulting performance overheads, as well as the overhead from no victim caches. We predict that larger victim caches will provide better performance.

The simulation results, which are shown in Figure 7.8 and Table 7.8, bear out this prediction. They show no clear optimal victim cache size, but that increasing the victim cache size slightly decreases performance overhead. Three of the benchmarks show little sensitivity to victim cache size, but the results for the rijndael benchmark suggest that some workloads would benefit somewhat from larger victim caches. The results also indicate that victim caches are more effective in reducing the overhead of dynamic data protection, and have little to no effect on the overhead from protecting instructions and static data. However, as the overall effect of signature victim caches appears to be negligible, it is probably not worthwhile to employ these caches.

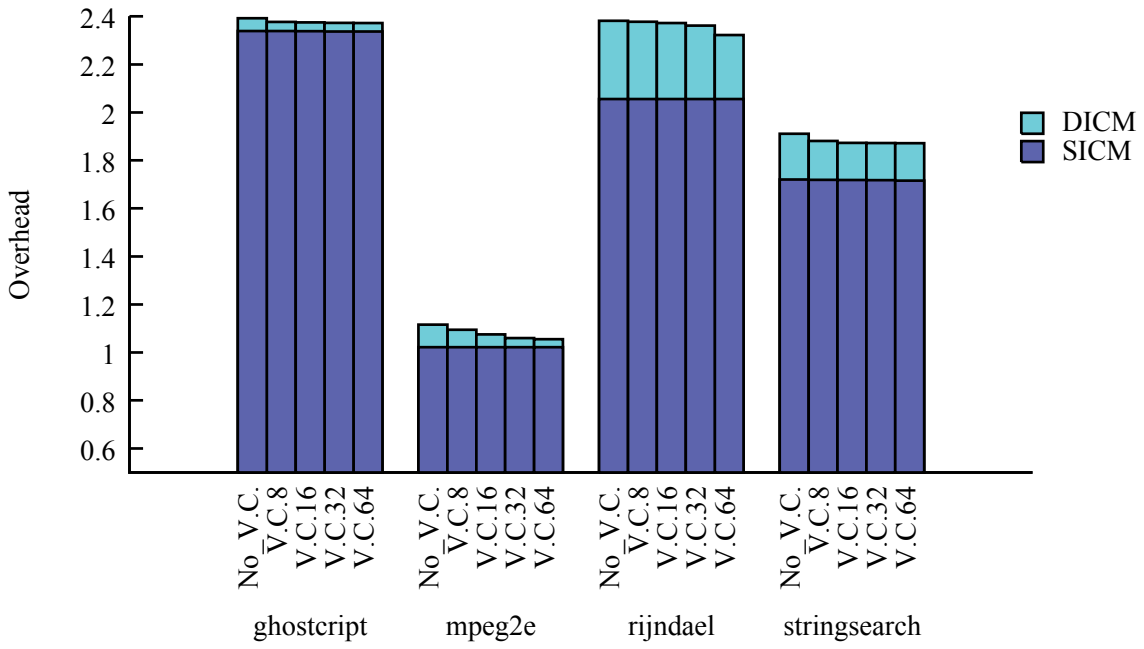


Figure 7.8 Performance Overhead Implications of Signature Victim Cache Size

Table 7.8 Performance Overhead Implications of Signature Victim Cache Size, Cortex M3, 2 KB

Benchmark	Signature Victim Caches	SICM	Both
ghostcript	None	2.33	2.36
	8 Entries	2.34	2.38
	16 Entries	2.34	2.38
	32 Entries	2.34	2.37
	64 Entries	2.34	2.37
	128 Entries	2.34	2.37
mpeg2e	None	1.02	1.06
	8 Entries	1.02	1.09
	16 Entries	1.02	1.07
	32 Entries	1.02	1.06
	64 Entries	1.02	1.06
	128 Entries	1.02	1.05
rijndael	None	2.06	2.36
	8 Entries	2.06	2.38
	16 Entries	2.06	2.37
	32 Entries	2.06	2.36
	64 Entries	2.06	2.32
	128 Entries	2.06	2.25
stringsearch	None	1.72	1.87
	8 Entries	1.72	1.88
	16 Entries	1.72	1.87
	32 Entries	1.72	1.87
	64 Entries	1.72	1.87
	128 Entries	1.72	1.87

7.4.3.2 Cryptographic Modes

We evaluate the influence of cryptographic mode choice by fixing all parameters other than cryptographic mode and simulating all chosen benchmarks for all architectures of interest. For these simulations, signatures are embedded with protected blocks and all sequence number caches are sized at 50% of their respective data caches. The three cipher modes discussed in this dissertation are simulated: CBC-MAC, PMAC, and GCM. Theoretically, CBC-MAC should induce the greatest performance overhead, with PMAC offering a noticeable improvement, and GCM providing the best performance.

The simulation results are graphed in Figure 7.9 - Figure 7.14, and presented numerically in Table 7.9 and Table 7.10. The simulation results closely follow the theoretical projections. They also show that going from CBC-MAC to PMAC exhibits the greatest increase in performance. GCM does provide the best performance, but the difference between PMAC and GCM is not as dramatic for most workloads.

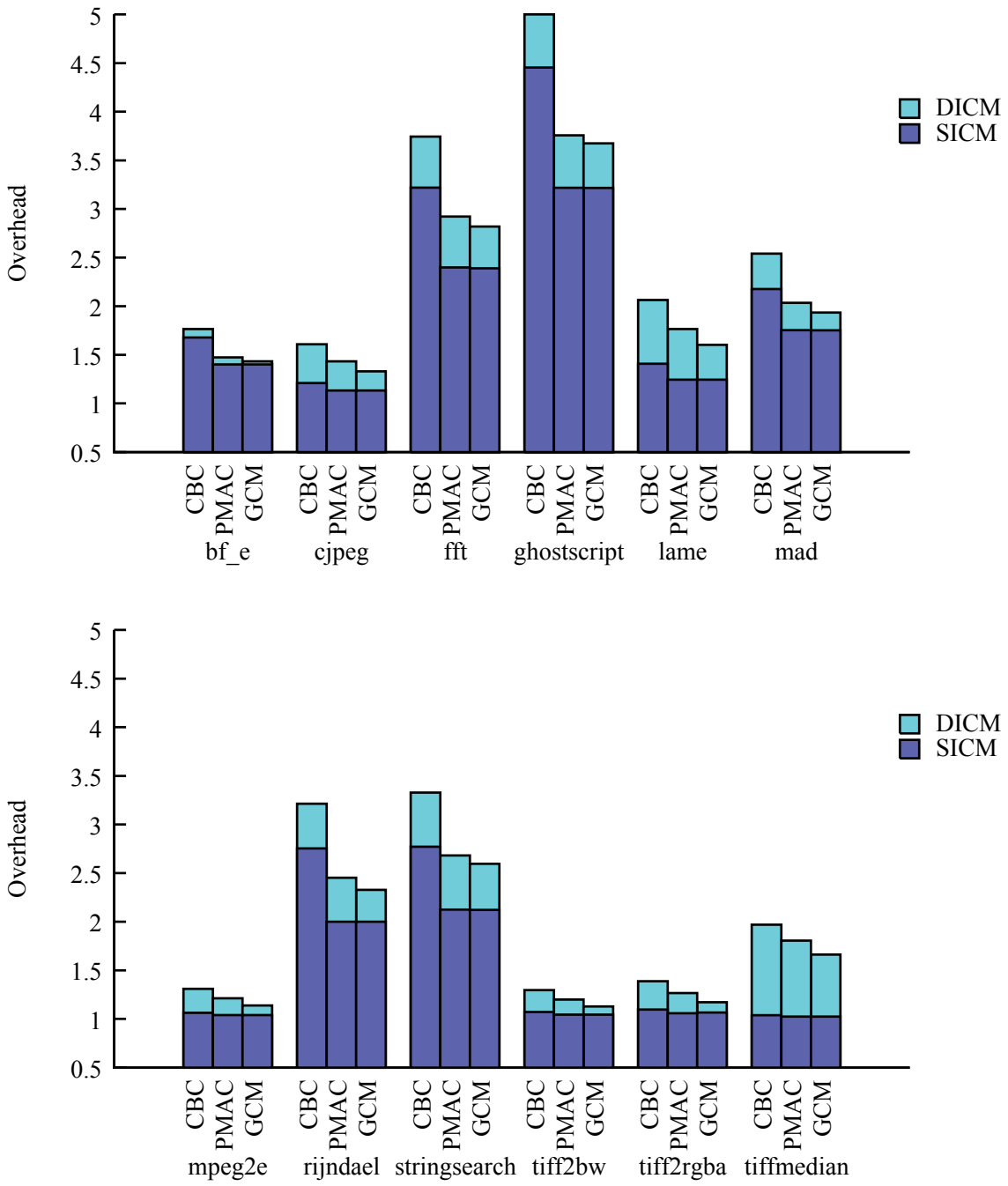


Figure 7.9 Performance Overhead Implications of Cipher Choice, Cortex M3, 1 KB

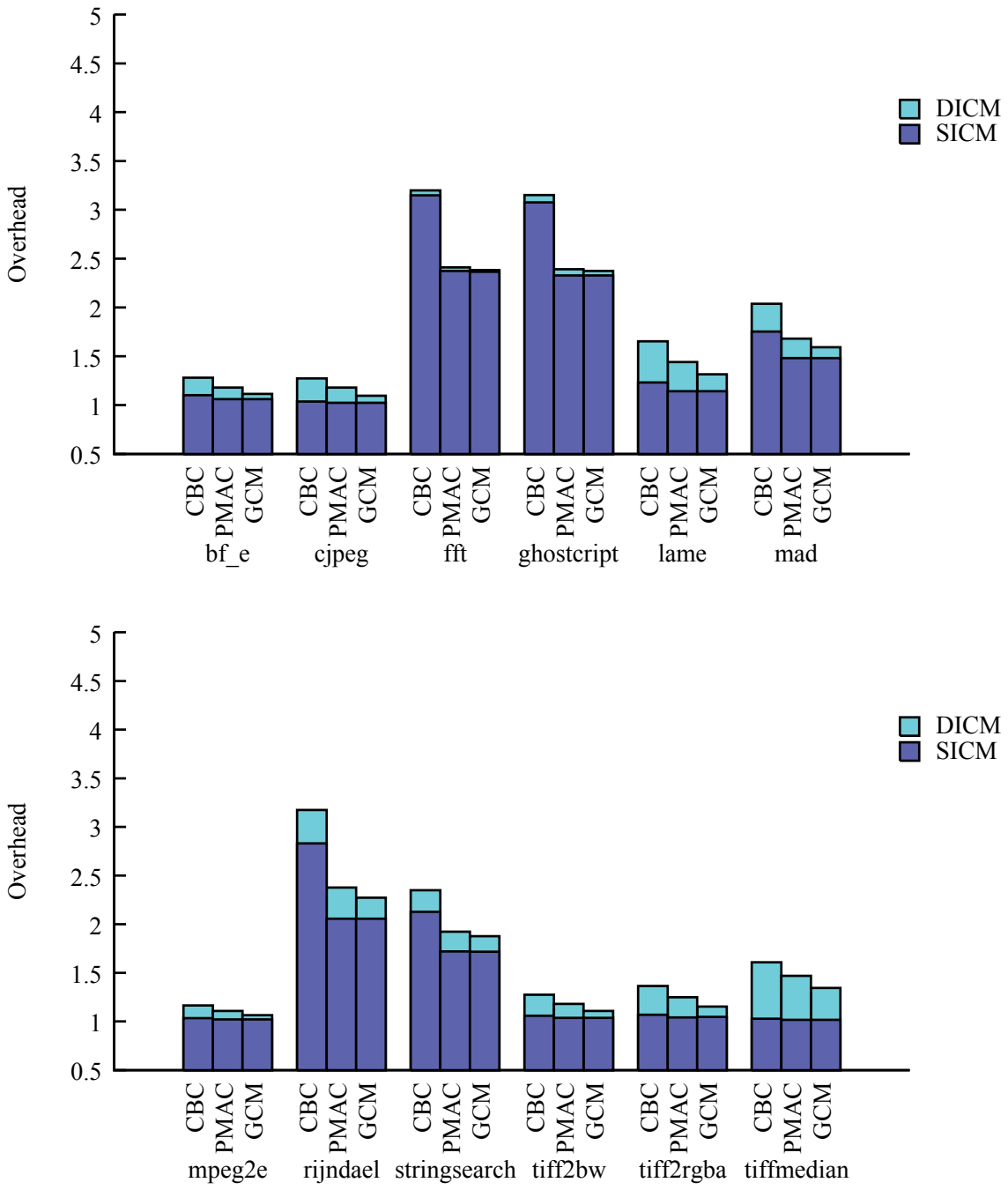


Figure 7.10 Performance Overhead Implications of Cipher Choice, Cortex M3, 2 KB

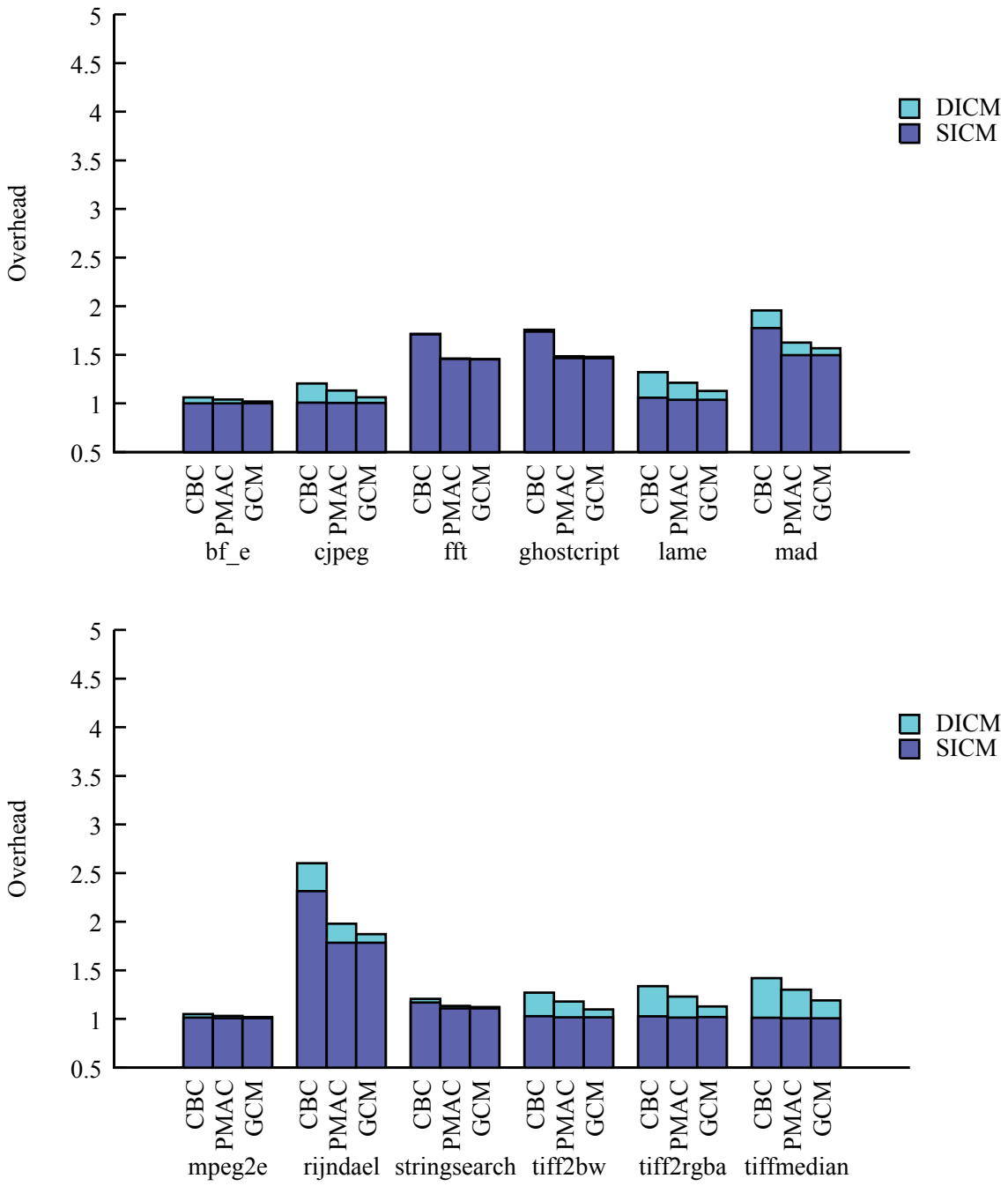


Figure 7.11 Performance Overhead Implications of Cipher Choice, Cortex M3, 4 KB

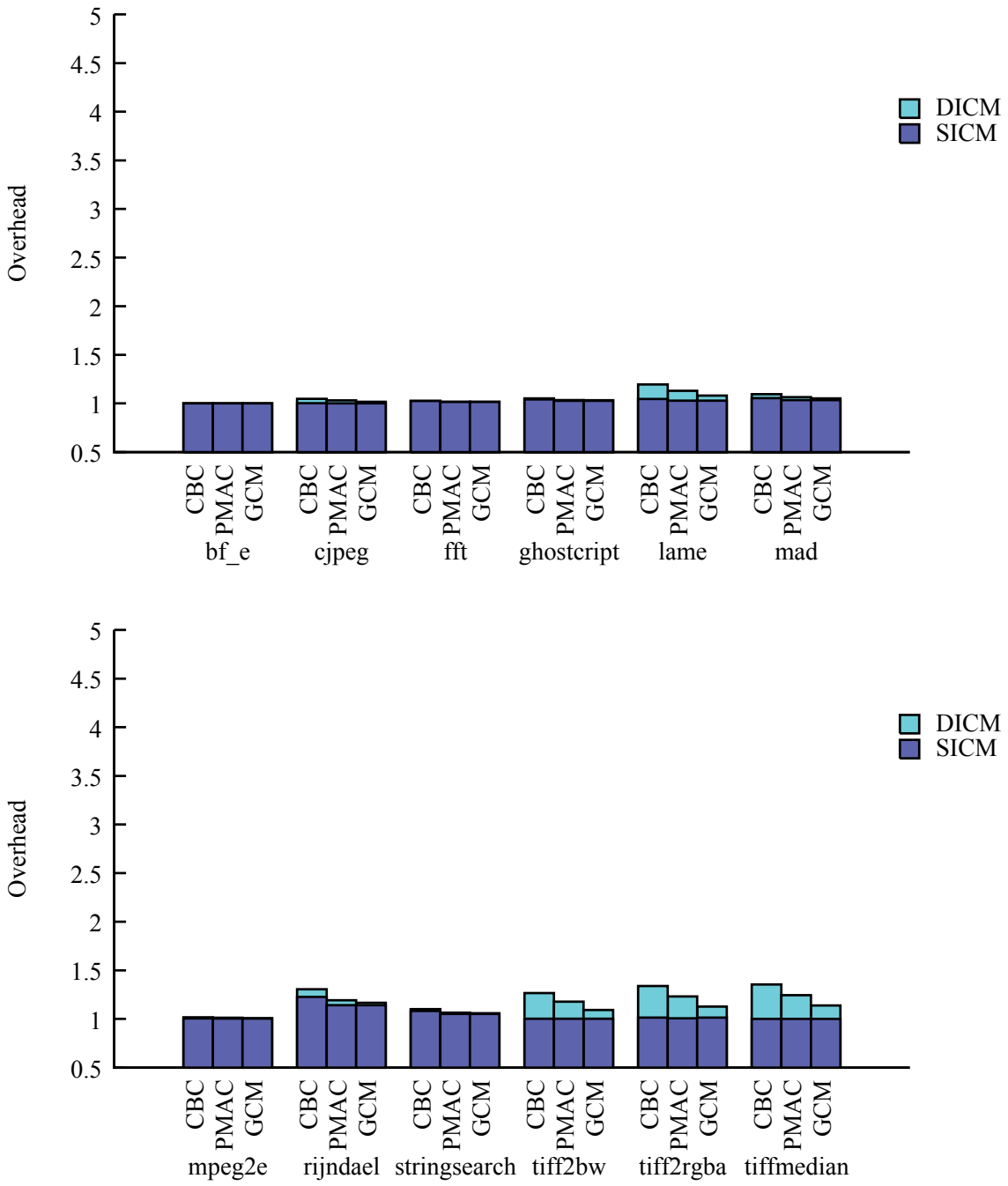


Figure 7.12 Performance Overhead Implications of Cipher Choice, Cortex M3, 8 KB

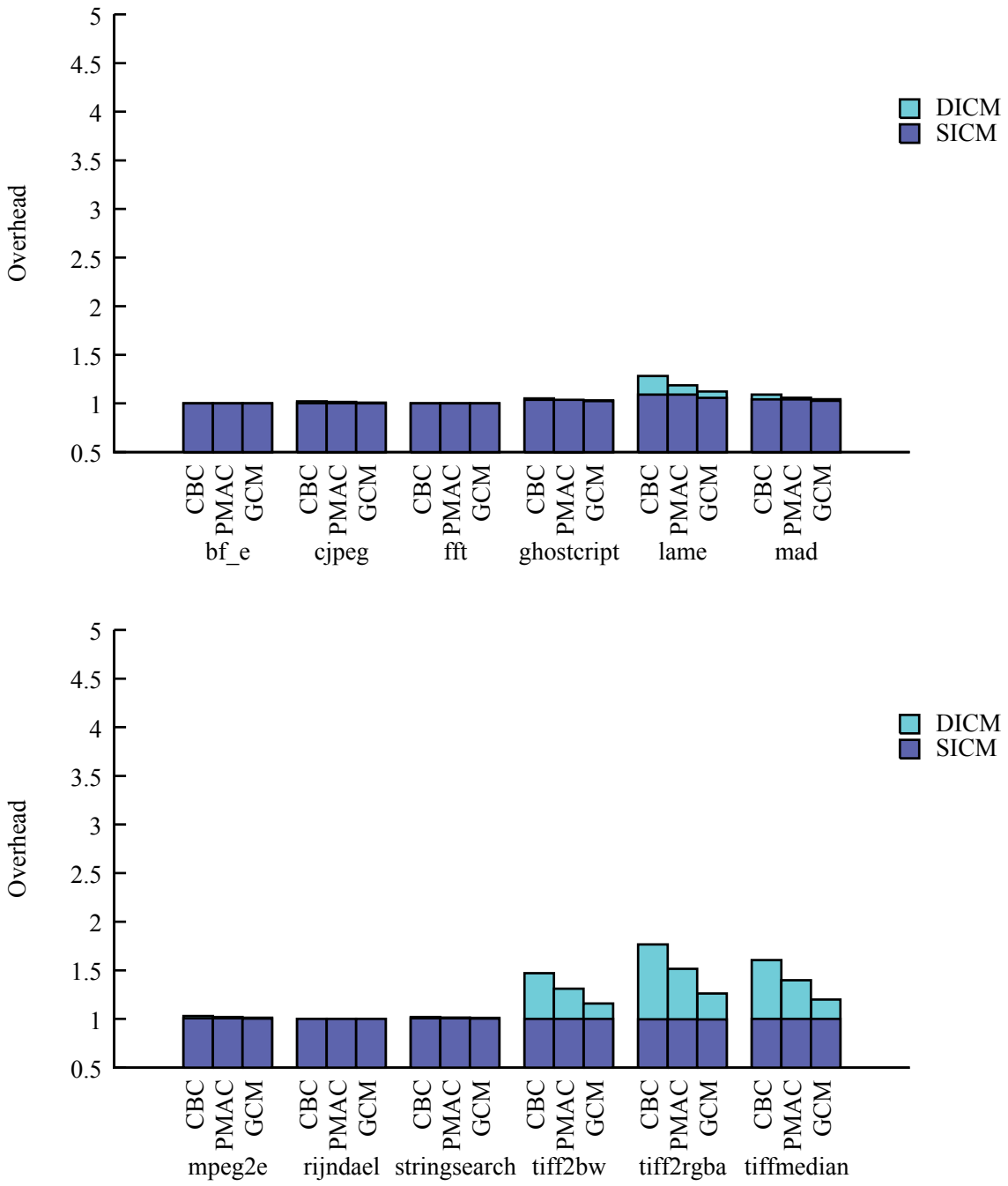


Figure 7.13 Performance Overhead Implications of Cipher Choice, Cortex A8, 16 KB

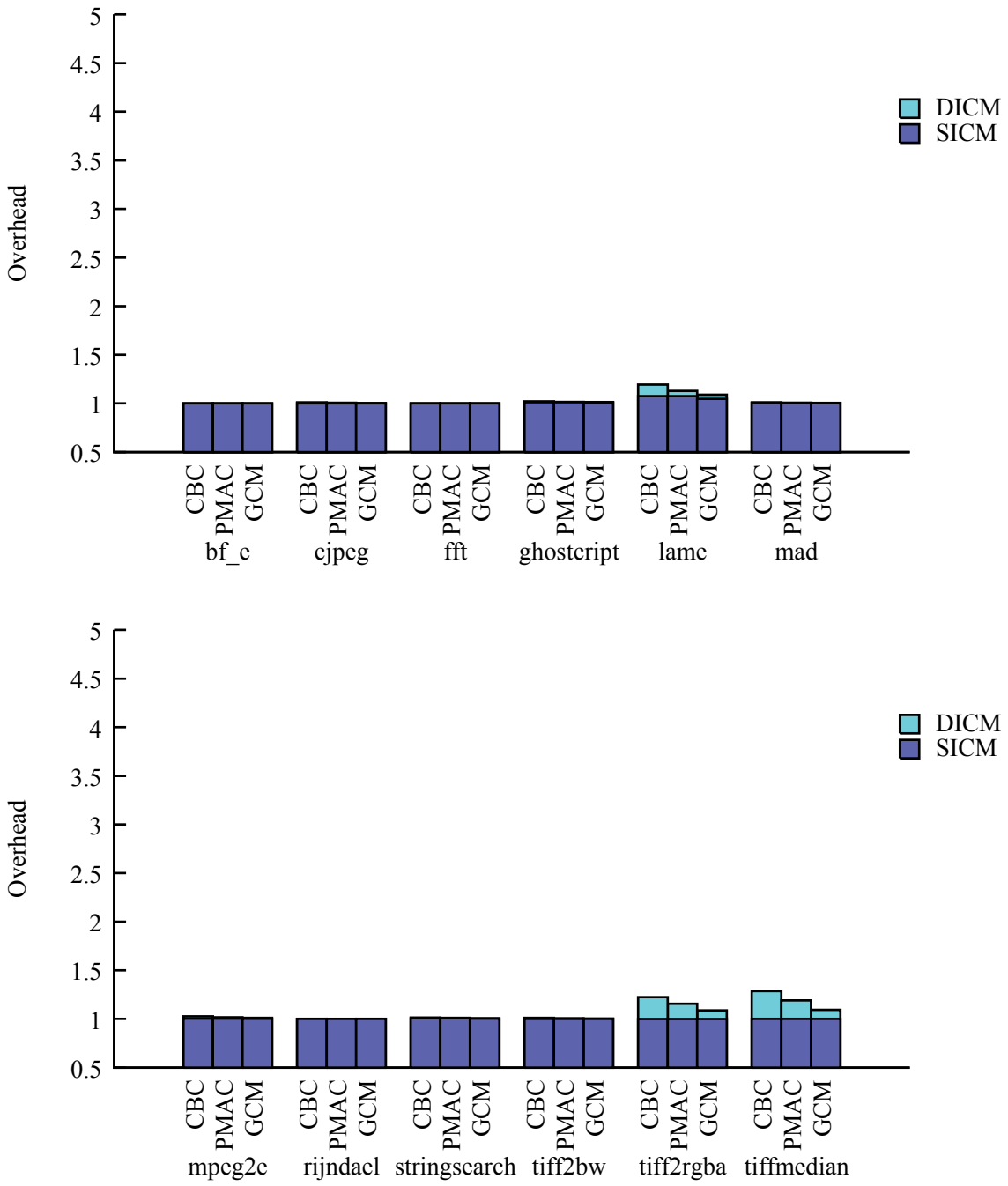


Figure 7.14 Performance Overhead Implications of Cipher Choice, Cortex A8, 32 KB

Table 7.9 Performance Overhead Implications of Cipher Choice, Cortex M3

Benchmark	Cipher	1 KB		2 KB		4 KB		8 KB	
		SICM	Both	SICM	Both	SICM	Both	SICM	Both
bf_e	CBC	1.68	1.76	1.10	1.28	1.00	1.06	1.00	1.00
	PMAC	1.40	1.47	1.06	1.18	1.00	1.04	1.00	1.00
	GCM	1.40	1.43	1.06	1.11	1.00	1.02	1.00	1.00
cjpeg	CBC	1.21	1.61	1.04	1.27	1.01	1.21	1.00	1.05
	PMAC	1.13	1.43	1.02	1.18	1.01	1.13	1.00	1.03
	GCM	1.13	1.33	1.02	1.10	1.01	1.06	1.00	1.02
fft	CBC	3.22	3.74	3.15	3.20	1.71	1.72	1.02	1.03
	PMAC	2.40	2.92	2.37	2.41	1.46	1.46	1.02	1.02
	GCM	2.39	2.82	2.37	2.38	1.45	1.46	1.02	1.02
ghostcript	CBC	4.45	5.00	3.08	3.15	1.74	1.76	1.04	1.05
	PMAC	3.22	3.76	2.33	2.39	1.47	1.48	1.03	1.04
	GCM	3.22	3.67	2.33	2.37	1.47	1.48	1.03	1.03
lame	CBC	1.41	2.06	1.23	1.65	1.06	1.32	1.05	1.19
	PMAC	1.25	1.76	1.14	1.44	1.04	1.21	1.03	1.13
	GCM	1.25	1.60	1.14	1.31	1.04	1.13	1.03	1.08
mad	CBC	2.18	2.54	1.75	2.04	1.77	1.96	1.05	1.10
	PMAC	1.75	2.03	1.48	1.68	1.50	1.63	1.03	1.06
	GCM	1.75	1.93	1.48	1.59	1.50	1.57	1.03	1.05
mpeg2e	CBC	1.06	1.31	1.03	1.16	1.01	1.05	1.01	1.02
	PMAC	1.04	1.21	1.02	1.11	1.01	1.03	1.00	1.01
	GCM	1.04	1.14	1.02	1.06	1.01	1.02	1.00	1.01
rijndael	CBC	2.75	3.21	2.83	3.17	2.31	2.60	1.23	1.31
	PMAC	2.00	2.45	2.06	2.38	1.78	1.98	1.14	1.19
	GCM	2.00	2.33	2.06	2.27	1.78	1.87	1.14	1.17
stringsearch	CBC	2.77	3.33	2.13	2.35	1.17	1.21	1.08	1.10
	PMAC	2.13	2.68	1.72	1.92	1.11	1.13	1.05	1.07
	GCM	2.12	2.60	1.72	1.88	1.11	1.12	1.05	1.06
tiff2bw	CBC	1.07	1.30	1.06	1.28	1.03	1.27	1.00	1.27
	PMAC	1.05	1.20	1.04	1.18	1.02	1.18	1.00	1.18
	GCM	1.05	1.13	1.04	1.11	1.02	1.10	1.00	1.09
tiff2rgba	CBC	1.10	1.39	1.07	1.37	1.03	1.34	1.01	1.34
	PMAC	1.06	1.27	1.04	1.25	1.01	1.23	1.01	1.23
	GCM	1.07	1.17	1.05	1.15	1.02	1.13	1.01	1.13
tiffmedian	CBC	1.04	1.97	1.03	1.61	1.01	1.42	1.00	1.35
	PMAC	1.02	1.81	1.02	1.47	1.01	1.30	1.00	1.24
	GCM	1.02	1.66	1.02	1.34	1.01	1.19	1.00	1.14

Table 7.10 Performance Overhead Implications of Cipher Choice, Cortex A8

Benchmark	Cipher	16 KB		32 KB	
		SICM	Both	SICM	Both
bf_e	CBC	1.00	1.00	1.00	1.00
	PMAC	1.00	1.00	1.00	1.00
	GCM	1.00	1.00	1.00	1.00
cjpeg	CBC	1.00	1.02	1.00	1.01
	PMAC	1.00	1.01	1.00	1.01
	GCM	1.00	1.01	1.00	1.00
fft	CBC	1.00	1.00	1.00	1.00
	PMAC	1.00	1.00	1.00	1.00
	GCM	1.00	1.00	1.00	1.00
ghostcript	CBC	1.04	1.05	1.01	1.02
	PMAC	1.04	1.04	1.01	1.02
	GCM	1.02	1.03	1.01	1.01
lame	CBC	1.09	1.28	1.07	1.19
	PMAC	1.09	1.19	1.07	1.13
	GCM	1.06	1.12	1.05	1.09
mad	CBC	1.04	1.09	1.00	1.01
	PMAC	1.04	1.06	1.00	1.01
	GCM	1.03	1.04	1.00	1.01
mpeg2e	CBC	1.01	1.03	1.00	1.03
	PMAC	1.01	1.02	1.00	1.02
	GCM	1.00	1.01	1.00	1.01
rijndael	CBC	1.00	1.00	1.00	1.00
	PMAC	1.00	1.00	1.00	1.00
	GCM	1.00	1.00	1.00	1.00
stringsearch	CBC	1.01	1.02	1.01	1.01
	PMAC	1.01	1.01	1.01	1.01
	GCM	1.00	1.01	1.00	1.01
tiff2bw	CBC	1.00	1.47	1.00	1.01
	PMAC	1.00	1.31	1.00	1.01
	GCM	1.00	1.16	1.00	1.00
tiff2rgba	CBC	1.00	1.77	1.00	1.22
	PMAC	1.00	1.52	1.00	1.16
	GCM	1.00	1.26	1.00	1.09
tiffmedian	CBC	1.00	1.61	1.00	1.29
	PMAC	1.00	1.40	1.00	1.19
	GCM	1.00	1.20	1.00	1.09

7.4.3.3 Speculative Execution

We would also like to demonstrate the efficacy of speculative execution by comparing the results of runs using the WtV scheme with those using an RbV scheme. We have already simulated all benchmarks on all architectures with all ciphers in WtV mode, so we choose the PMAC cipher and run additional simulations in RbV mode with 16-entry IVBs. Theoretically, the RbV mode should show considerably improved performance over the WtV mode.

The results of the RbV simulations are graphically compared with those of the WtV simulations in Figure 7.15 - Figure 7.20 and numerically in Table 7.11 and Table 7.12. The observed results agree with the theoretical projections. Utilizing speculative execution does provide a dramatic increase in performance, especially in architectures with small caches (and thus higher cache miss rates).

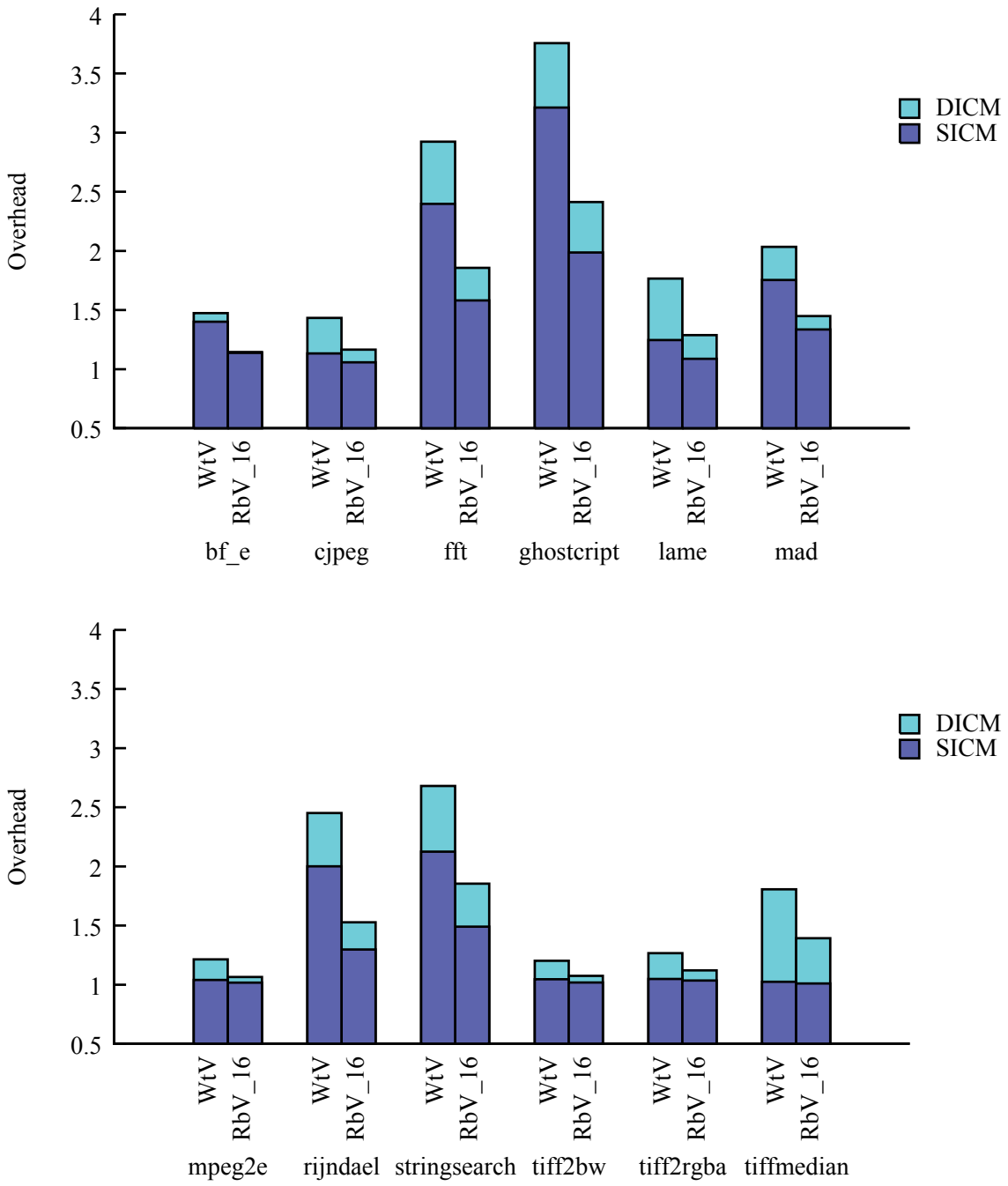


Figure 7.15 Performance Overhead Implications of Speculative Execution, Cortex M3, 1 KB

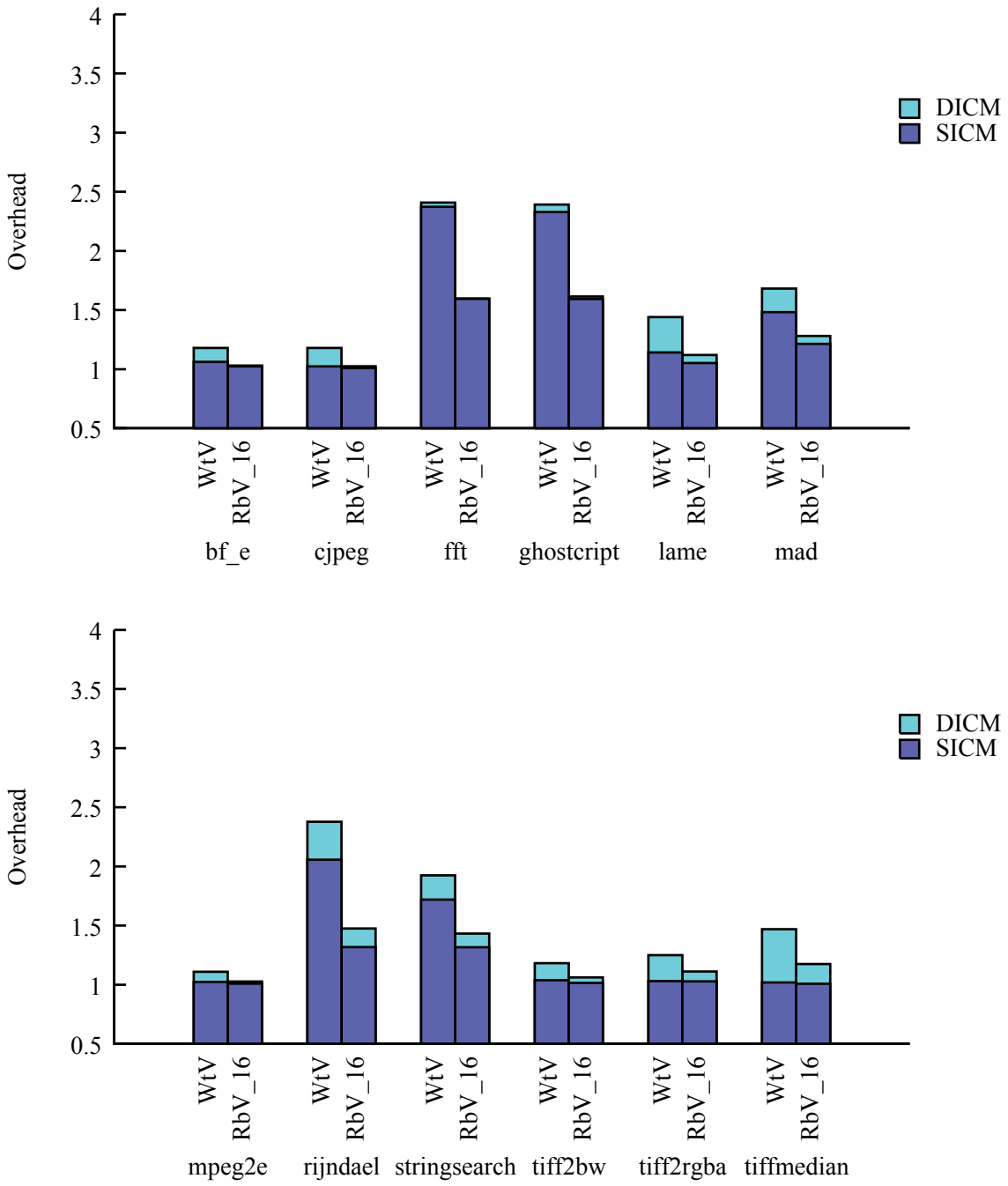


Figure 7.16 Performance Overhead Implications of Speculative Execution,

Cortex M3, 2 KB

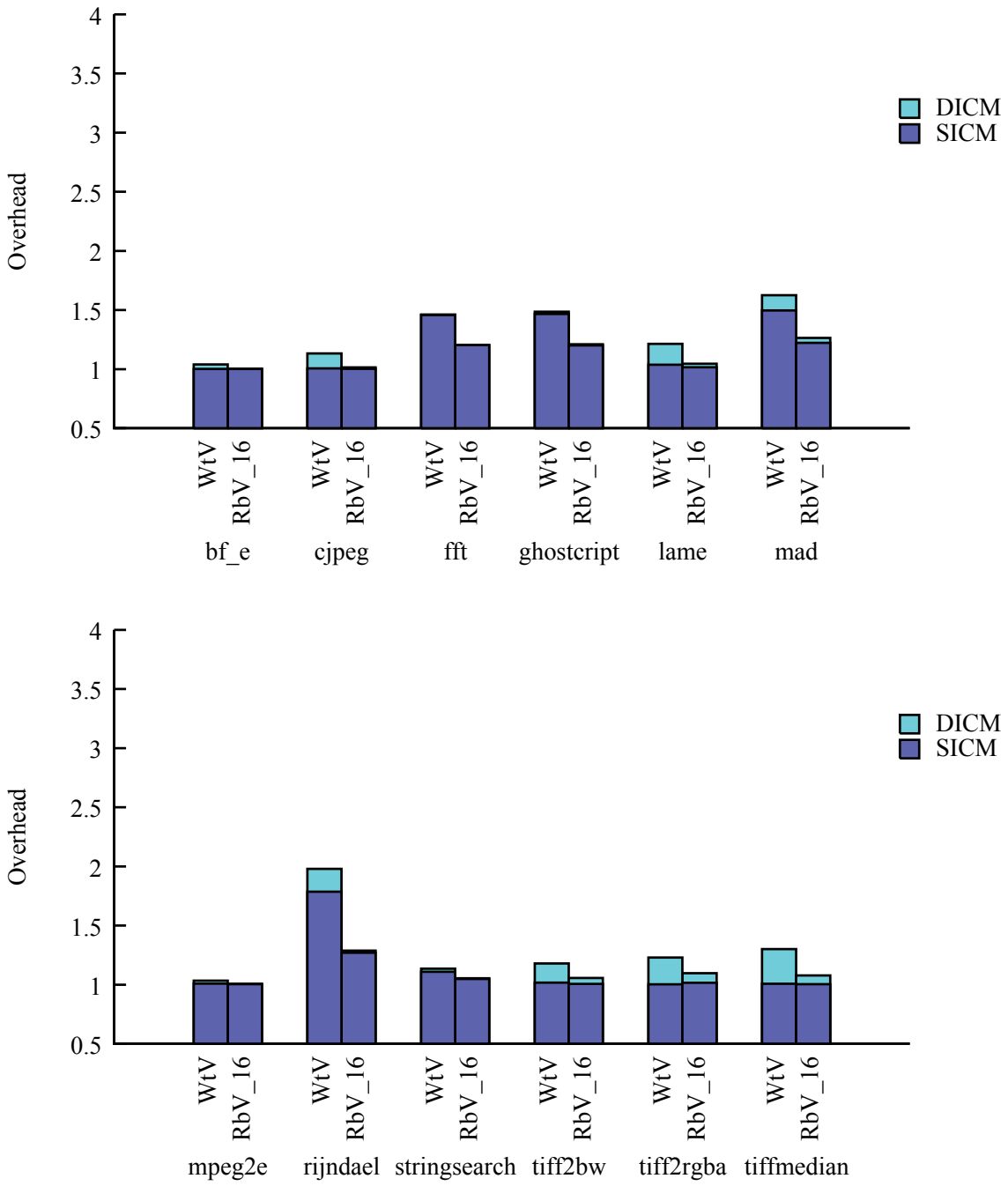


Figure 7.17 Performance Overhead Implications of Speculative Execution, Cortex M3, 4 KB

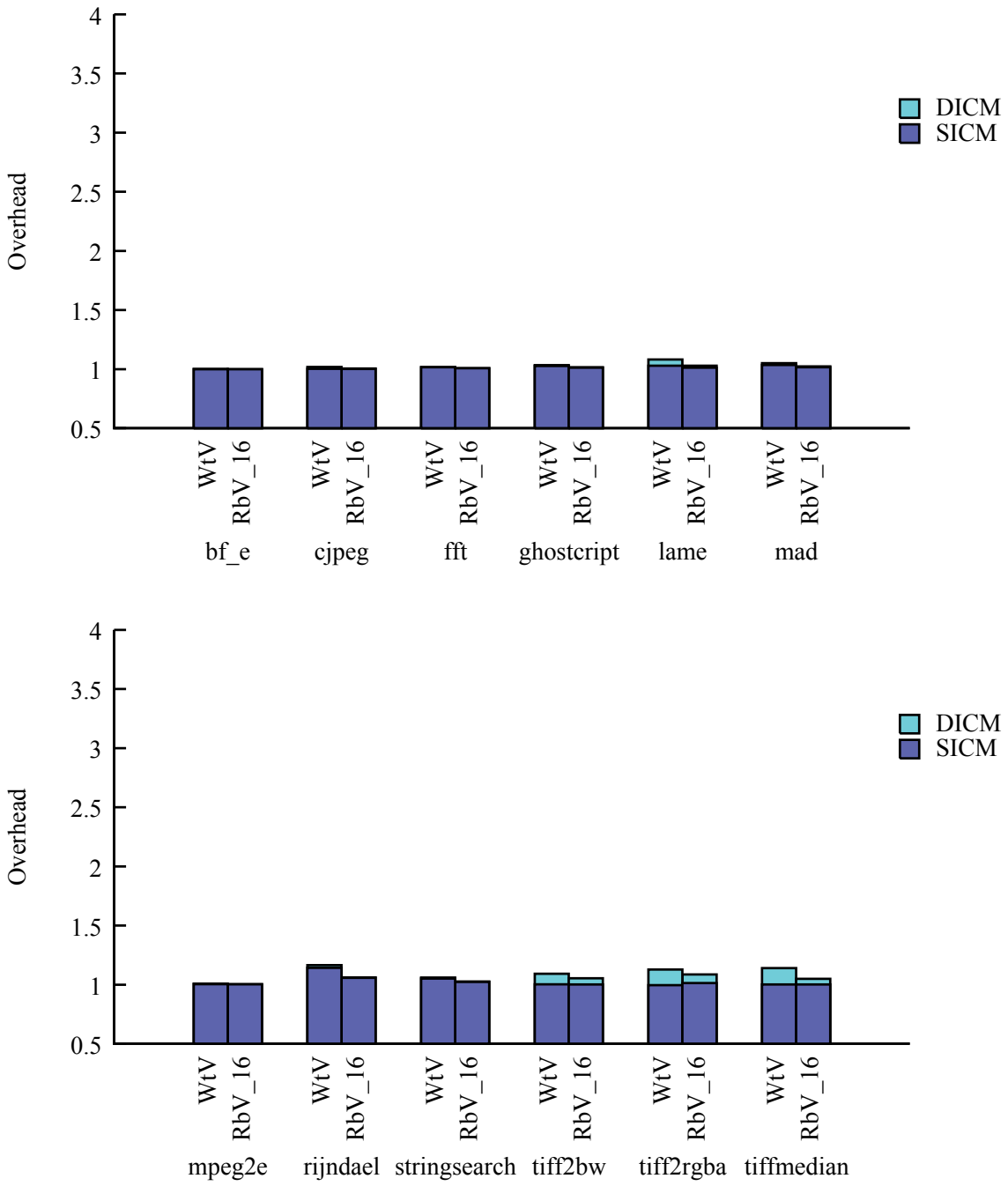


Figure 7.18 Performance Overhead Implications of Speculative Execution, Cortex M3, 8 KB

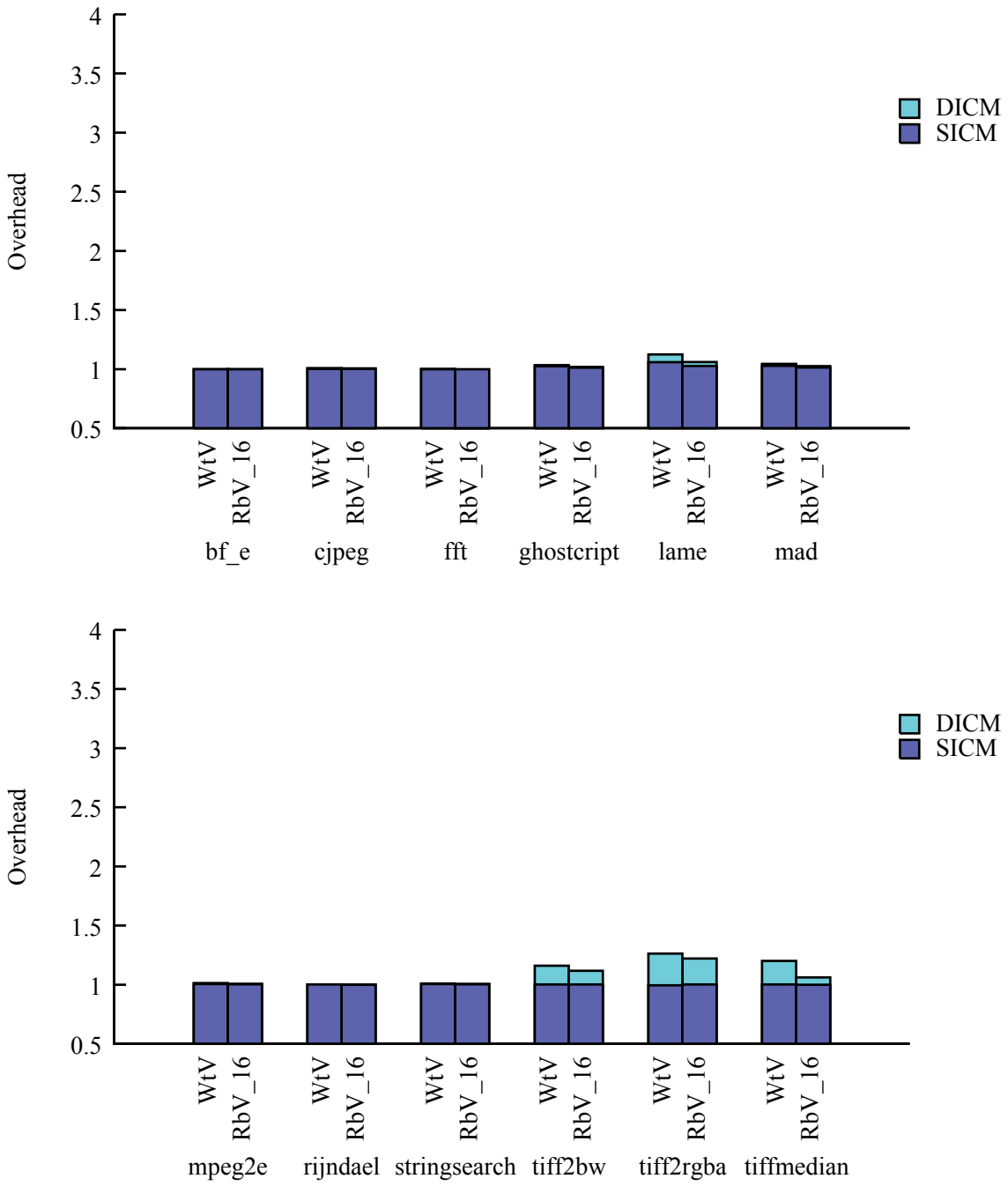


Figure 7.19 Performance Overhead Implications of Speculative Execution, Cortex A8, 16 KB

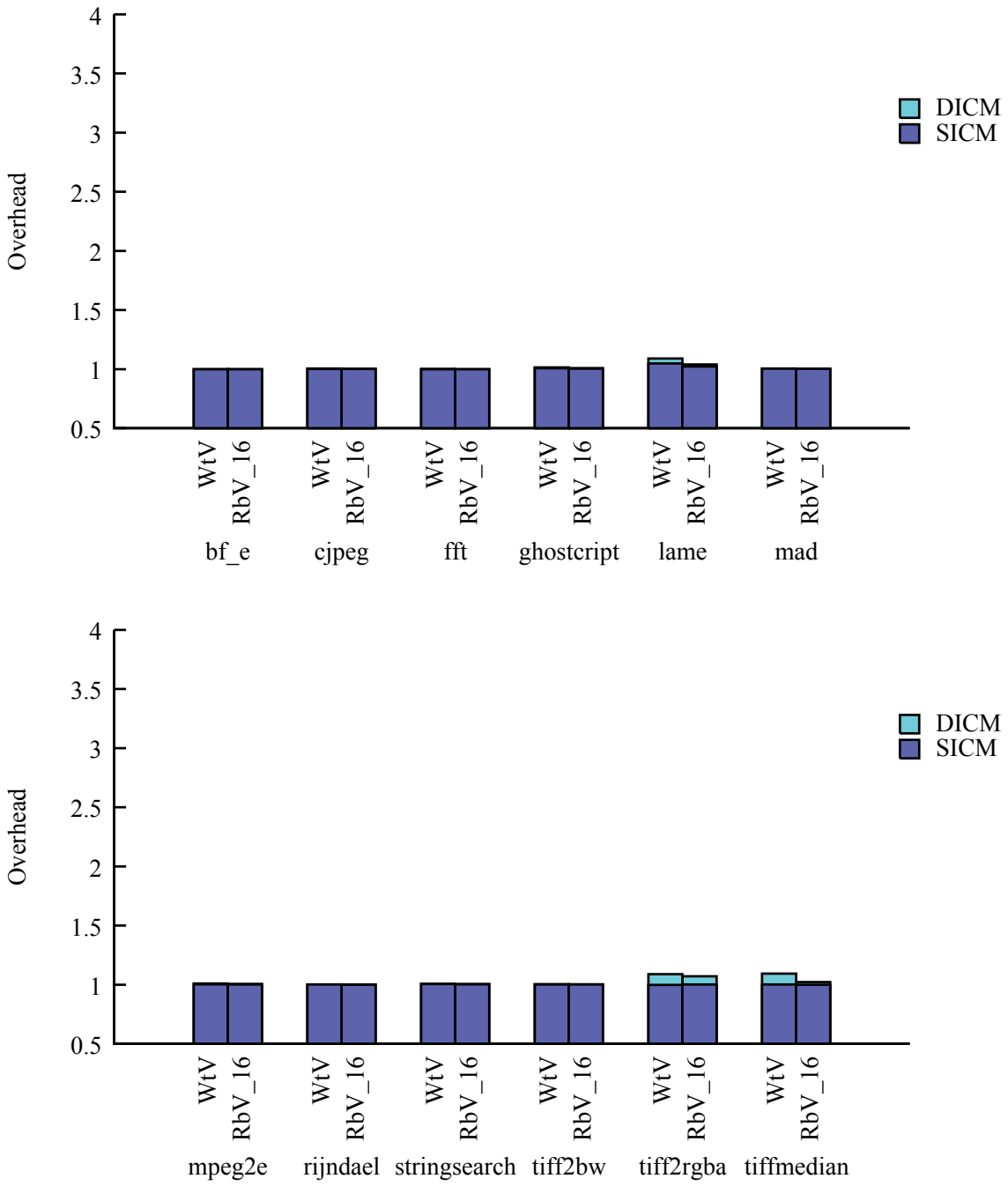


Figure 7.20 Performance Overhead Implications of Speculative Execution, Cortex A8, 32 KB

Table 7.11 Performance Overhead Implications of Speculative Execution, Cortex M3

Benchmark		1 KB		2 KB		4 KB		8 KB	
		SICM	Both	SICM	Both	SICM	Both	SICM	Both
bf_e	WtV	1.40	1.47	1.06	1.18	1.00	1.04	1.00	1.00
	RbV 16	1.14	1.14	1.02	1.03	1.00	1.00	1.00	1.00
cjpeg	WtV	1.13	1.43	1.02	1.18	1.01	1.13	1.00	1.02
	RbV 16	1.06	1.17	1.01	1.02	1.00	1.01	1.00	1.00
fft	WtV	2.40	2.92	2.37	2.41	1.46	1.46	1.02	1.02
	RbV 16	1.58	1.86	1.59	1.60	1.20	1.20	1.01	1.01
ghostcript	WtV	3.21	3.76	2.33	2.39	1.47	1.48	1.03	1.03
	RbV 16	1.99	2.41	1.59	1.61	1.20	1.21	1.01	1.02
lame	WtV	1.25	1.76	1.14	1.44	1.04	1.21	1.03	1.08
	RbV 16	1.09	1.29	1.05	1.12	1.01	1.04	1.01	1.03
mad	WtV	1.75	2.03	1.48	1.68	1.50	1.63	1.03	1.05
	RbV 16	1.33	1.45	1.21	1.28	1.22	1.26	1.02	1.02
mpeg2e	WtV	1.04	1.21	1.02	1.11	1.01	1.03	1.00	1.01
	RbV 16	1.02	1.06	1.01	1.03	1.00	1.01	1.00	1.00
rijndael	WtV	2.00	2.45	2.06	2.38	1.78	1.98	1.14	1.17
	RbV 16	1.30	1.53	1.32	1.47	1.27	1.29	1.06	1.06
stringsearch	WtV	2.13	2.68	1.72	1.92	1.11	1.13	1.05	1.06
	RbV 16	1.49	1.85	1.32	1.43	1.05	1.05	1.02	1.03
tiff2bw	WtV	1.05	1.20	1.04	1.18	1.02	1.18	1.00	1.09
	RbV 16	1.02	1.07	1.01	1.06	1.01	1.06	1.00	1.05
tiff2rgba	WtV	1.05	1.27	1.03	1.25	1.02	1.23	1.01	1.13
	RbV 16	1.03	1.12	1.03	1.11	1.00	1.10	1.00	1.09
tiffmedian	WtV	1.02	1.81	1.02	1.47	1.01	1.30	1.00	1.14
	RbV 16	1.01	1.39	1.01	1.17	1.00	1.08	1.00	1.05

Table 7.12 Performance Overhead Implications of Speculative Execution, Cortex A8

Benchmark		16 KB		32 KB	
		SICM	Both	SICM	Both
bf_e	WtV	1.00	1.00	1.00	1.00
	RbV 16	1.00	1.00	1.00	1.00
cjpeg	WtV	1.00	1.01	1.00	1.00
	RbV 16	1.00	1.01	1.00	1.00
fft	WtV	1.00	1.00	1.00	1.00
	RbV 16	1.00	1.00	1.00	1.00
ghostcript	WtV	1.02	1.03	1.01	1.01
	RbV 16	1.01	1.02	1.00	1.01
lame	WtV	1.06	1.12	1.05	1.09
	RbV 16	1.03	1.06	1.02	1.04
mad	WtV	1.03	1.04	1.00	1.01
	RbV 16	1.01	1.02	1.00	1.00
mpeg2e	WtV	1.00	1.01	1.00	1.01
	RbV 16	1.00	1.01	1.00	1.01
rijndael	WtV	1.00	1.00	1.00	1.00
	RbV 16	1.00	1.00	1.00	1.00
stringsearch	WtV	1.00	1.01	1.00	1.01
	RbV 16	1.00	1.01	1.00	1.00
tiff2bw	WtV	1.00	1.16	1.00	1.00
	RbV 16	1.00	1.12	1.00	1.00
tiff2rgba	WtV	1.00	1.26	1.00	1.09
	RbV 16	1.00	1.22	1.00	1.07
tiffmedian	WtV	1.00	1.2	1.00	1.09
	RbV 16	1.00	1.06	1.00	1.02

7.4.3.3.1 Optimal IVB Depth

In addition to demonstrating the usefulness of speculative execution, we would like to determine the optimal depth, or number of entries, for the instruction verification buffer. We isolate the effects of the IVB by choosing the CBC-MAC mode, which results in the longest verification latencies and thus will stress the IVB more than the other cipher modes. We again fix the architecture as Cortex M3 with 2 KB caches and use the benchmarks that are significant for that architecture, varying the IVB depth from eight to 64 in powers of two. Theoretically, there should be a performance increase going from WtV to an eight-entry IBV with further performance increases as the IVB

size increases. At some IVB size, the performance should level out as blocks may be verified and instructions issued without saturating the IVB.

The simulation results are presented graphically in Figure 7.21 and numerically in Table 7.13. The results demonstrate the dramatic increase in performance when going from WtV to RbV with a small, eight-entry RbV. However, the performance overhead shows no sensitivity to IVB depth; small IVBs perform as well as large IVBs. This indicates that, even using the CBC-MAC cipher mode, the secure blocks are being verified and their associated instructions retired before even a small IVB can be saturated. Thus, only small IVBs are necessary for this architecture, and transistors may be allocated for other uses on the chip.

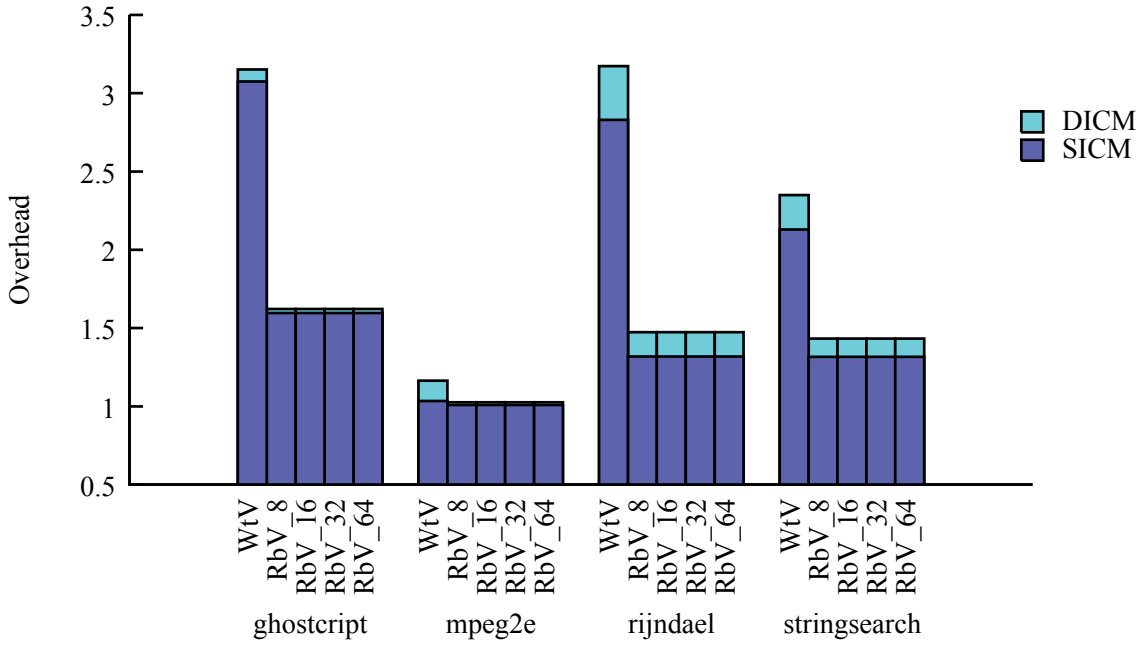


Figure 7.21 Performance Overhead Implications of IVB Depth

Table 7.13 Performance Overhead Implications of IVB Depth, Cortex M3, 2 KB

Benchmark		SICM	Both
ghostcript	WtV	3.08	3.15
	RbV 8	1.60	1.62
	RbV 16	1.60	1.62
	RbV 32	1.60	1.62
	RbV 64	1.60	1.62
	RbV 128	1.60	1.62
mpeg2e	WtV	1.03	1.16
	RbV 8	1.01	1.03
	RbV 16	1.01	1.03
	RbV 32	1.01	1.03
	RbV 64	1.01	1.03
	RbV 128	1.01	1.03
rijndael	WtV	2.83	3.17
	RbV 8	1.32	1.47
	RbV 16	1.32	1.47
	RbV 32	1.32	1.47
	RbV 64	1.32	1.47
	RbV 128	1.32	1.47
stringsearch	WtV	2.13	2.35
	RbV 8	1.32	1.43
	RbV 16	1.32	1.43
	RbV 32	1.32	1.43
	RbV 64	1.32	1.43
	RbV 128	1.32	1.43

7.4.3.4 Sequence Number Cache Size

All simulations up until this point have used sequence number caches that are 50% of the size of their associated data caches. We here explore the effects that varying the sequence number cache will have on overhead. We choose the PMAC cipher mode with embedded signatures and vary the signature cache sizes between 25%, 50%, and 100% of the associated data cache size. We predict that performance will increase as the signature cache size increases.

The simulation results are plotted in Figure 7.22 - Figure 7.27 and displayed numerically in Table 7.14 and Table 7.15. For most workloads, the results follow our theoretical projections. In many cases with the Cortex M3 architecture, increasing the sequence number cache size from 50% to 100% has less effect than going from 25% to 50%. We therefore conclude that sequence number cache sizes of 50% provide the optimal balance between performance and complexity when data caches are small ($\leq$ 8 KB). For large data caches ($>$ 8 KB), the results indicate minimal performance improvements with increasing sequence number cache size. Smaller sequence number caches, such as 25% of the data cache size, are acceptable in this case.

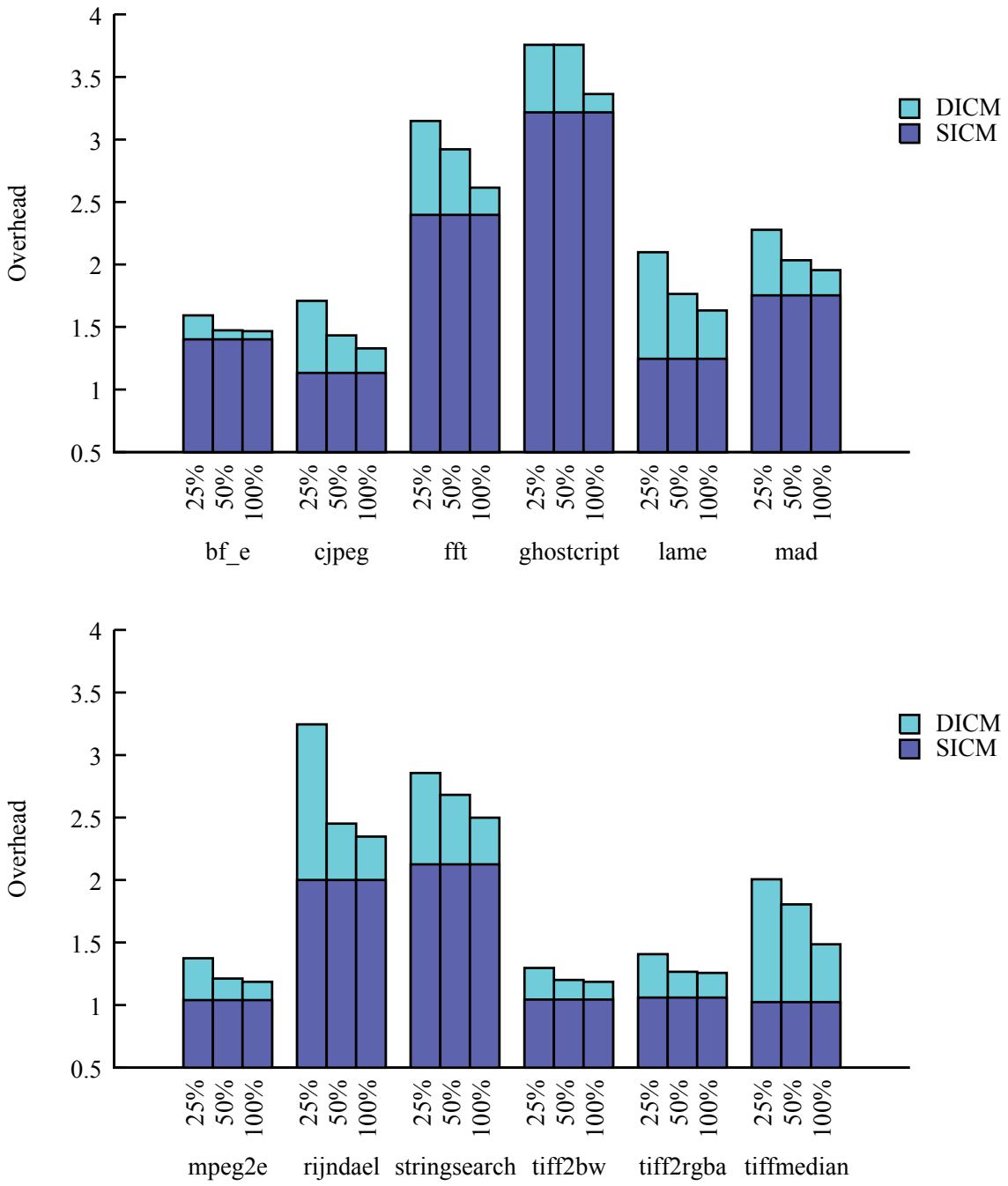


Figure 7.22 Performance Overhead Implications of Sequence Number Cache Size, Cortex M3, 1 KB

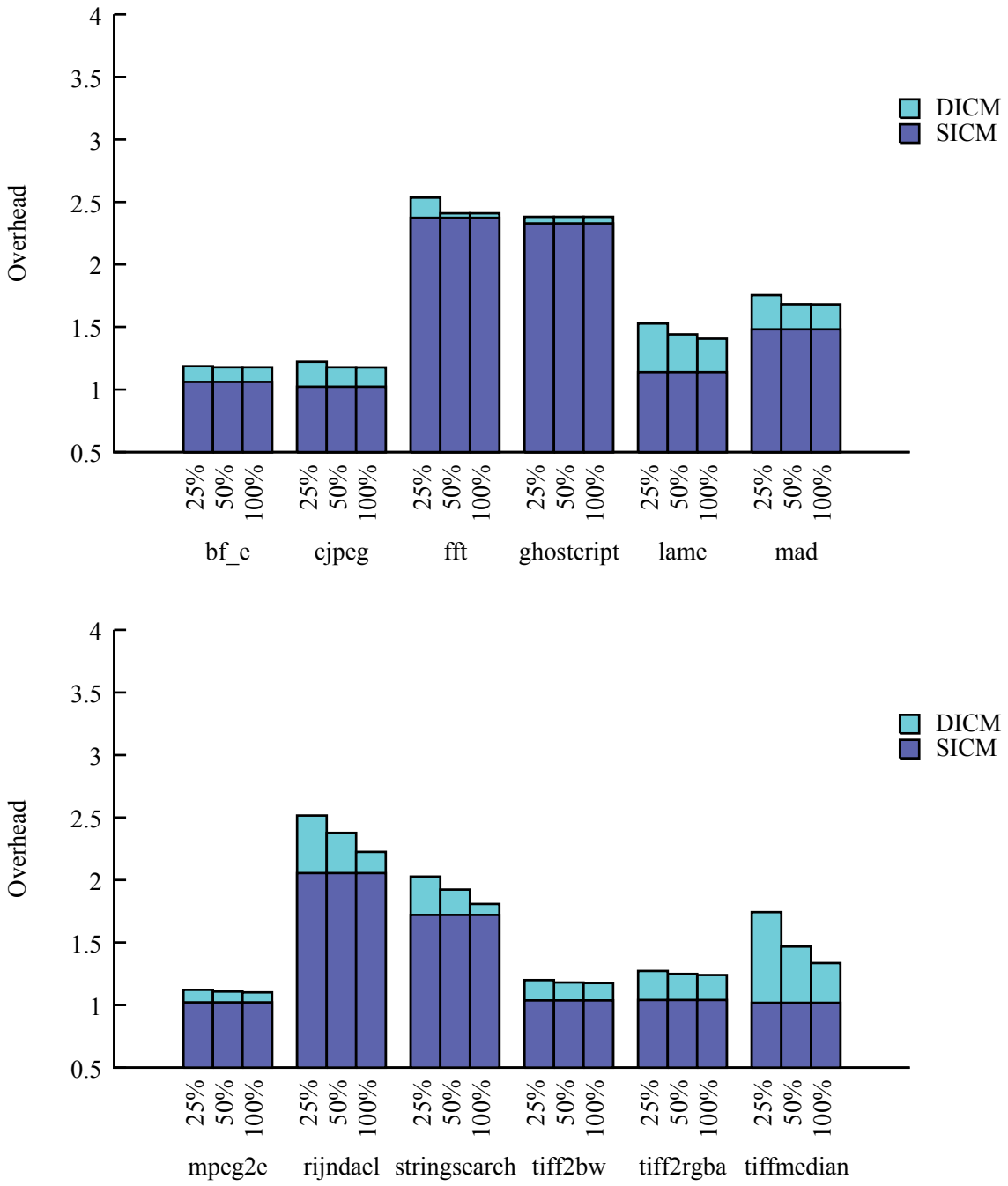


Figure 7.23 Performance Overhead Implications of Sequence Number Cache Size, Cortex M3, 2 KB

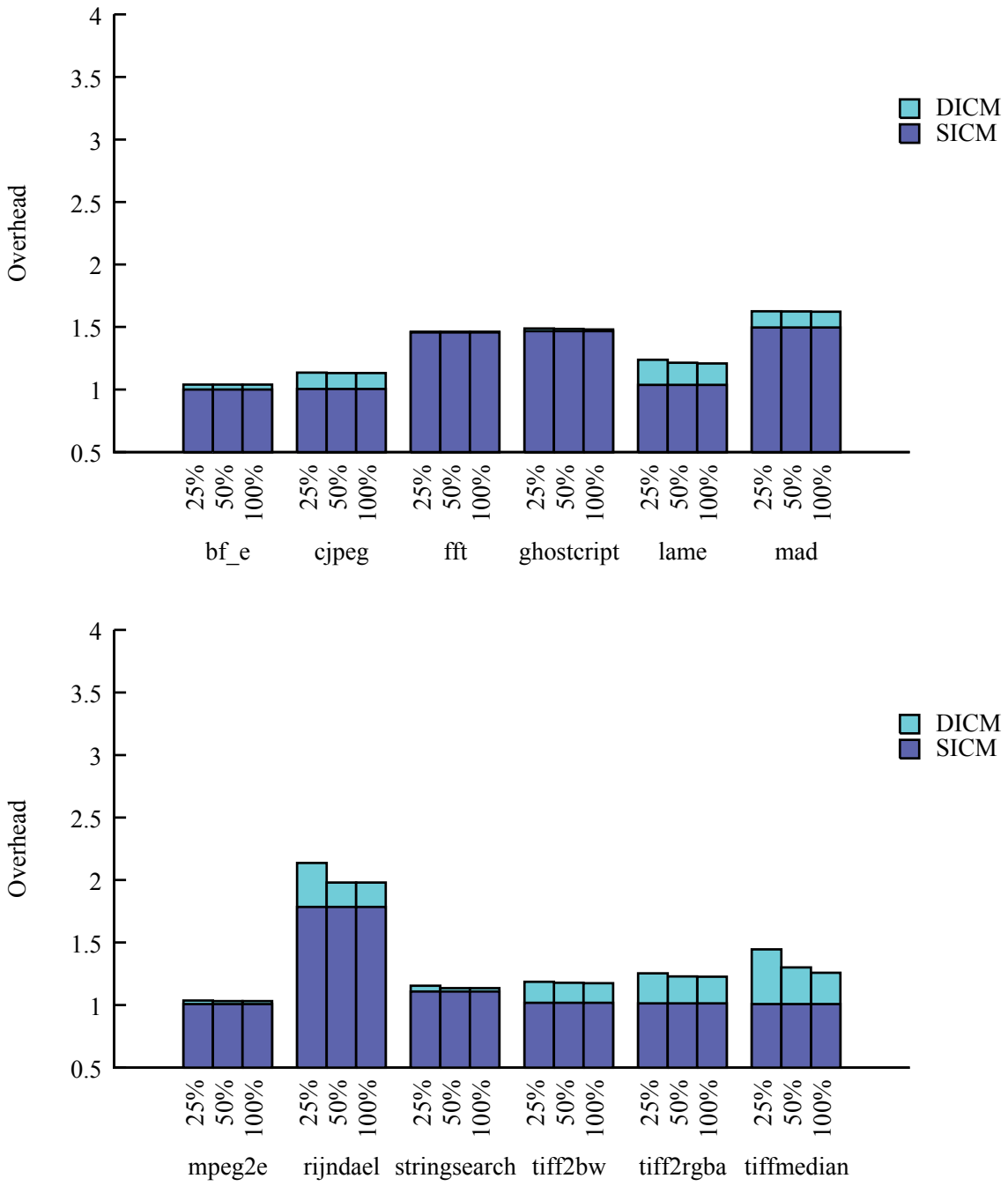


Figure 7.24 Performance Overhead Implications of Sequence Number Cache Size, Cortex M3, 4 KB

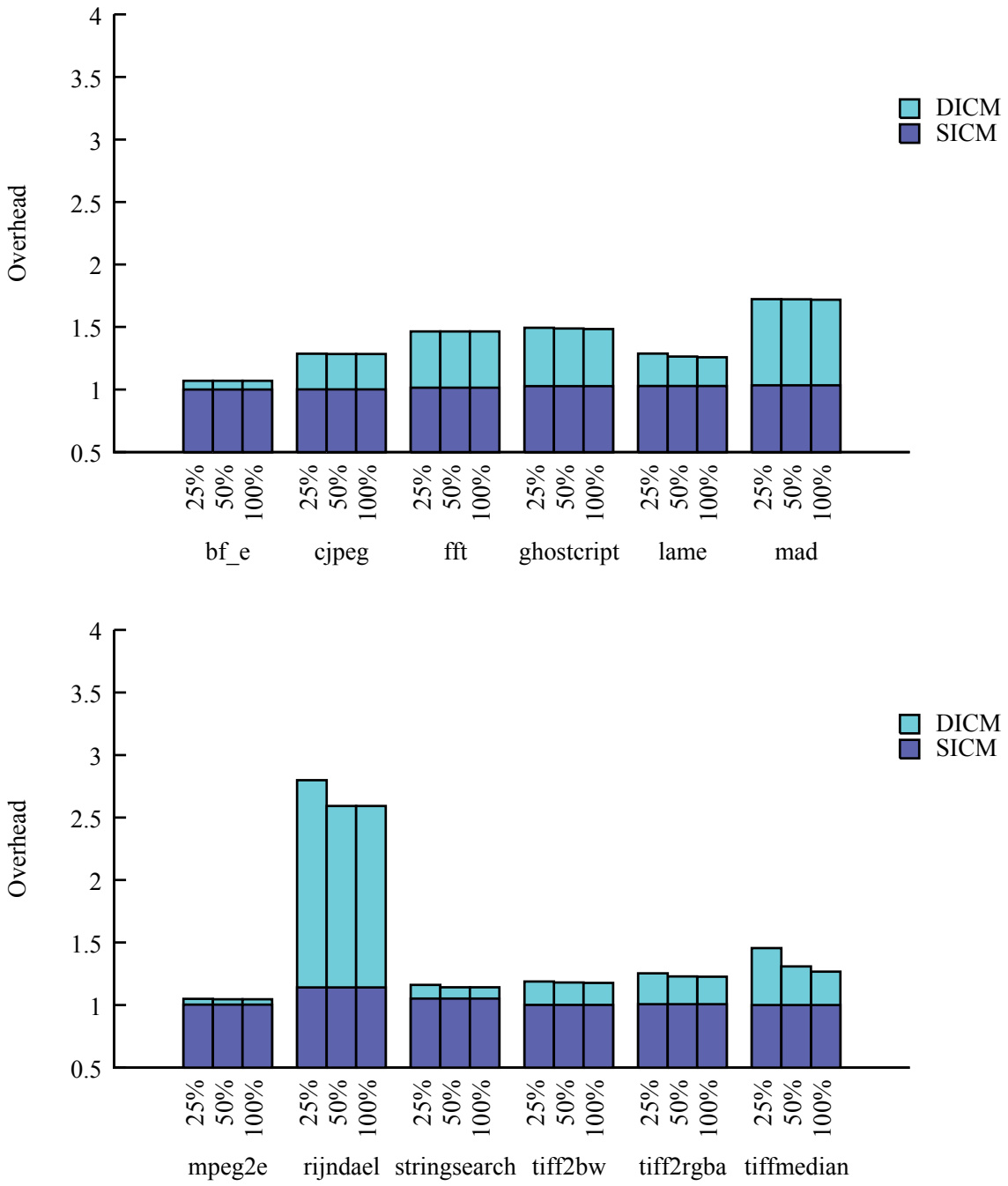


Figure 7.25 Performance Overhead Implications of Sequence Number Cache Size, Cortex M3, 8 KB

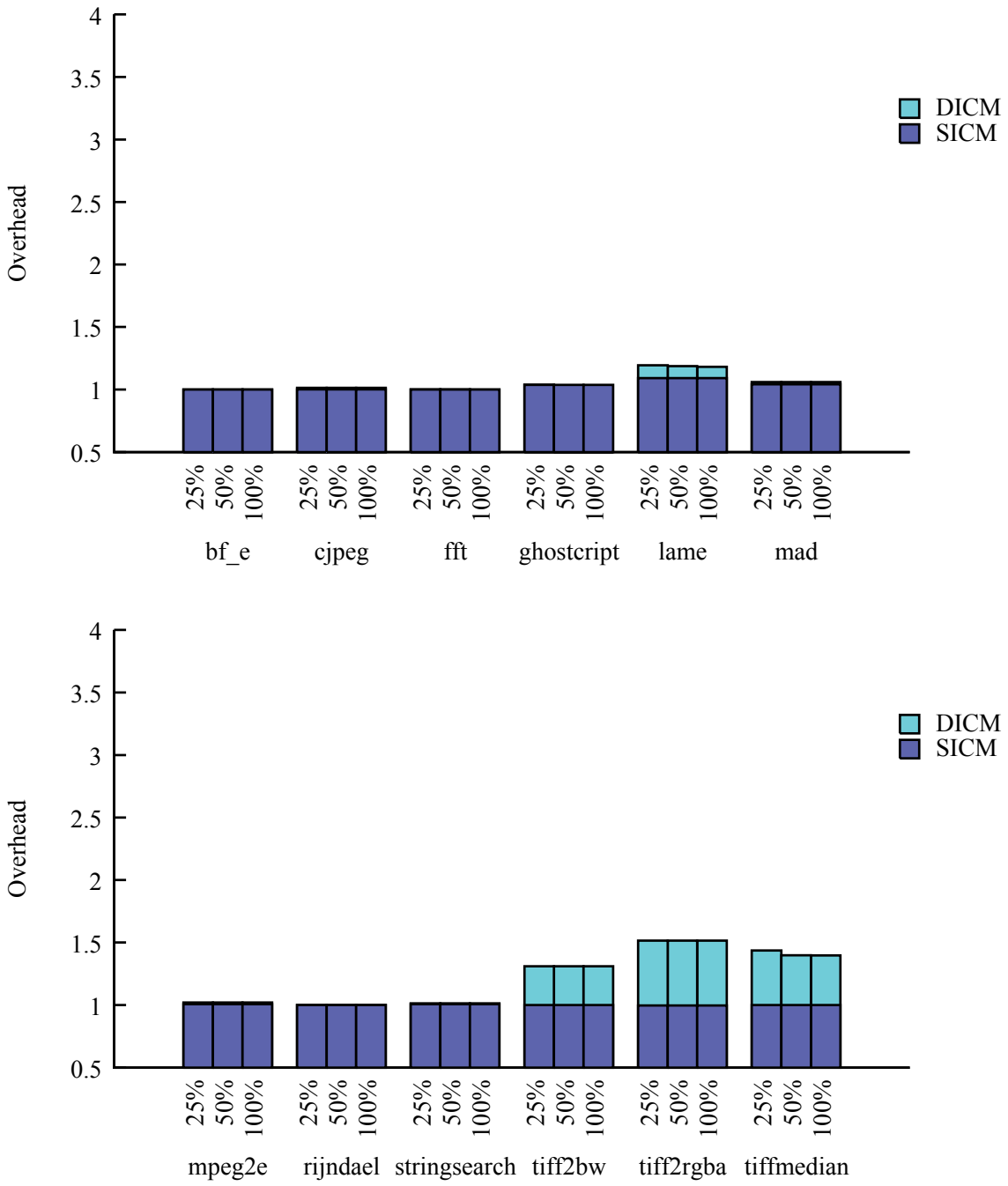


Figure 7.26 Performance Overhead Implications of Sequence Number Cache Size, Cortex A8, 16 KB

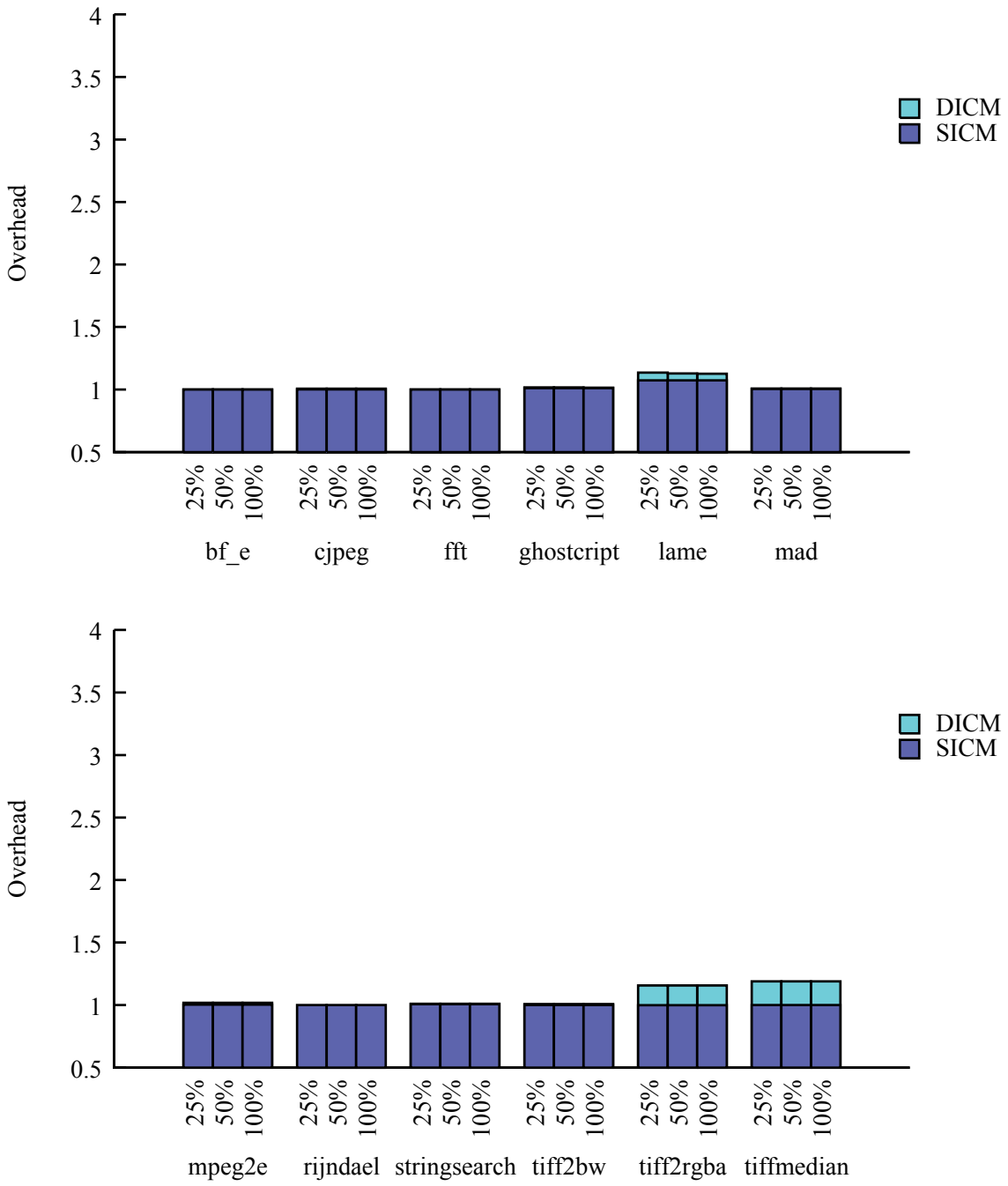


Figure 7.27 Performance Overhead Implications of Sequence Number Cache Size, Cortex A8, 32 KB

Table 7.14 Performance Overhead Implications of Sequence Number Cache Size, Cortex M3

Benchmark	Seqnum Cache Size	1 KB		2 KB		4 KB		8 KB	
		SICM	Both	SICM	Both	SICM	Both	SICM	Both
bf_e	25 %	1.40	1.59	1.06	1.19	1.00	1.04	1.00	1.07
	50 %	1.40	1.47	1.06	1.18	1.00	1.04	1.00	1.07
	100 %	1.40	1.47	1.06	1.18	1.00	1.04	1.00	1.07
cjpeg	25 %	1.13	1.71	1.02	1.22	1.01	1.13	1.00	1.29
	50 %	1.13	1.43	1.02	1.18	1.01	1.13	1.00	1.28
	100 %	1.13	1.33	1.02	1.18	1.01	1.13	1.00	1.28
fft	25 %	2.40	3.15	2.37	2.54	1.46	1.46	1.02	1.46
	50 %	2.40	2.92	2.37	2.41	1.46	1.46	1.02	1.46
	100 %	2.40	2.62	2.37	2.41	1.46	1.46	1.02	1.46
ghostcript	25 %	3.22	3.76	2.33	2.38	1.47	1.49	1.03	1.49
	50 %	3.22	3.76	2.33	2.38	1.47	1.48	1.03	1.49
	100 %	3.22	3.36	2.33	2.38	1.47	1.48	1.03	1.48
lame	25 %	1.25	2.10	1.14	1.53	1.04	1.24	1.03	1.29
	50 %	1.25	1.76	1.14	1.44	1.04	1.21	1.03	1.26
	100 %	1.25	1.63	1.14	1.41	1.04	1.21	1.03	1.26
mad	25 %	1.75	2.28	1.48	1.75	1.50	1.63	1.03	1.72
	50 %	1.75	2.03	1.48	1.68	1.50	1.63	1.03	1.72
	100 %	1.75	1.95	1.48	1.68	1.50	1.62	1.03	1.72
mpeg2e	25 %	1.04	1.37	1.02	1.12	1.01	1.04	1.00	1.05
	50 %	1.04	1.21	1.02	1.11	1.01	1.03	1.00	1.05
	100 %	1.04	1.19	1.02	1.10	1.01	1.03	1.00	1.05
rijndael	25 %	2.00	3.25	2.06	2.52	1.78	2.14	1.14	2.80
	50 %	2.00	2.45	2.06	2.38	1.78	1.98	1.14	2.59
	100 %	2.00	2.35	2.06	2.22	1.78	1.98	1.14	2.59
stringsearch	25 %	2.13	2.86	1.72	2.03	1.11	1.15	1.05	1.16
	50 %	2.13	2.68	1.72	1.92	1.11	1.13	1.05	1.14
	100 %	2.13	2.50	1.72	1.81	1.11	1.13	1.05	1.14
tiff2bw	25 %	1.05	1.30	1.04	1.20	1.02	1.19	1.00	1.19
	50 %	1.05	1.20	1.04	1.18	1.02	1.18	1.00	1.18
	100 %	1.05	1.19	1.04	1.18	1.02	1.17	1.00	1.18
tiff2rgba	25 %	1.06	1.41	1.04	1.27	1.01	1.25	1.01	1.25
	50 %	1.06	1.27	1.04	1.25	1.01	1.23	1.01	1.23
	100 %	1.06	1.26	1.04	1.24	1.01	1.23	1.01	1.23
tiffmedian	25 %	1.02	2.01	1.02	1.74	1.01	1.45	1.00	1.46
	50 %	1.02	1.81	1.02	1.47	1.01	1.30	1.00	1.31
	100 %	1.02	1.49	1.02	1.34	1.01	1.26	1.00	1.27

Table 7.15 Performance Overhead Implications of Sequence Number Cache Size, Cortex A8

Benchmark	Seqnum Cache Size	16 KB		32 KB	
		SICM	Both	SICM	Both
bf_e	25 %	1.00	1.00	1.00	1.00
	50 %	1.00	1.00	1.00	1.00
	100 %	1.00	1.00	1.00	1.00
cjpeg	25 %	1.00	1.01	1.00	1.01
	50 %	1.00	1.01	1.00	1.01
	100 %	1.00	1.01	1.00	1.01
fft	25 %	1.00	1.00	1.00	1.00
	50 %	1.00	1.00	1.00	1.00
	100 %	1.00	1.00	1.00	1.00
ghostcript	25 %	1.04	1.04	1.01	1.02
	50 %	1.04	1.04	1.01	1.02
	100 %	1.04	1.03	1.01	1.02
lame	25 %	1.09	1.19	1.07	1.13
	50 %	1.09	1.19	1.07	1.13
	100 %	1.09	1.18	1.07	1.13
mad	25 %	1.04	1.06	1.00	1.01
	50 %	1.04	1.06	1.00	1.01
	100 %	1.04	1.06	1.00	1.01
mpeg2e	25 %	1.01	1.02	1.00	1.02
	50 %	1.01	1.02	1.00	1.02
	100 %	1.01	1.02	1.00	1.02
rijndael	25 %	1.00	1.00	1.00	1.00
	50 %	1.00	1.00	1.00	1.00
	100 %	1.00	1.00	1.00	1.00
stringsearch	25 %	1.01	1.01	1.01	1.01
	50 %	1.01	1.01	1.01	1.01
	100 %	1.01	1.01	1.01	1.01
tiff2bw	25 %	1.00	1.31	1.00	1.01
	50 %	1.00	1.31	1.00	1.01
	100 %	1.00	1.31	1.00	1.01
tiff2rgba	25 %	1.00	1.52	1.00	1.16
	50 %	1.00	1.52	1.00	1.16
	100 %	1.00	1.52	1.00	1.16
tiffmedian	25 %	1.00	1.44	1.00	1.19
	50 %	1.00	1.40	1.00	1.19
	100 %	1.00	1.40	1.00	1.19

7.4.3.5 Double-Sized Protected Blocks

We also demonstrate the performance overhead effects of using double-sized protected blocks, that is, protected blocks whose size is twice that of the cache line. We have previously used 32 byte protected blocks, so double-sized protected blocks allude to protecting 64 bytes of data with one signature (and one sequence number in the dynamic case). We simulate using double-sized protected blocks on systems using the PMAC cipher and embedded systems, and compare the results to the same systems using single-sized protected blocks. We predict that, in the majority of workload cases, performance should be roughly the same. Some cases should exhibit better performance as a result of the prefetching behavior that using double-sized protected blocks entails. In some cases, however, this prefetching may do more harm than good, leading to cache pollution and degraded performance.

The results of these simulations are displayed graphically in Figure 7.28 - Figure 7.33 and numerically in Table 7.16 and Table 7.17. The simulation results mostly follow the theoretical projections. Note that using double-sized protected blocks when protecting instructions and static data nearly always yields the same or better performance as using single-sized protected blocks; protecting dynamic data shows greater sensitivity to protected block size. Overall, an improvement in performance is seen in most cases with smaller caches. Lower performance is observed in most cases with larger caches.

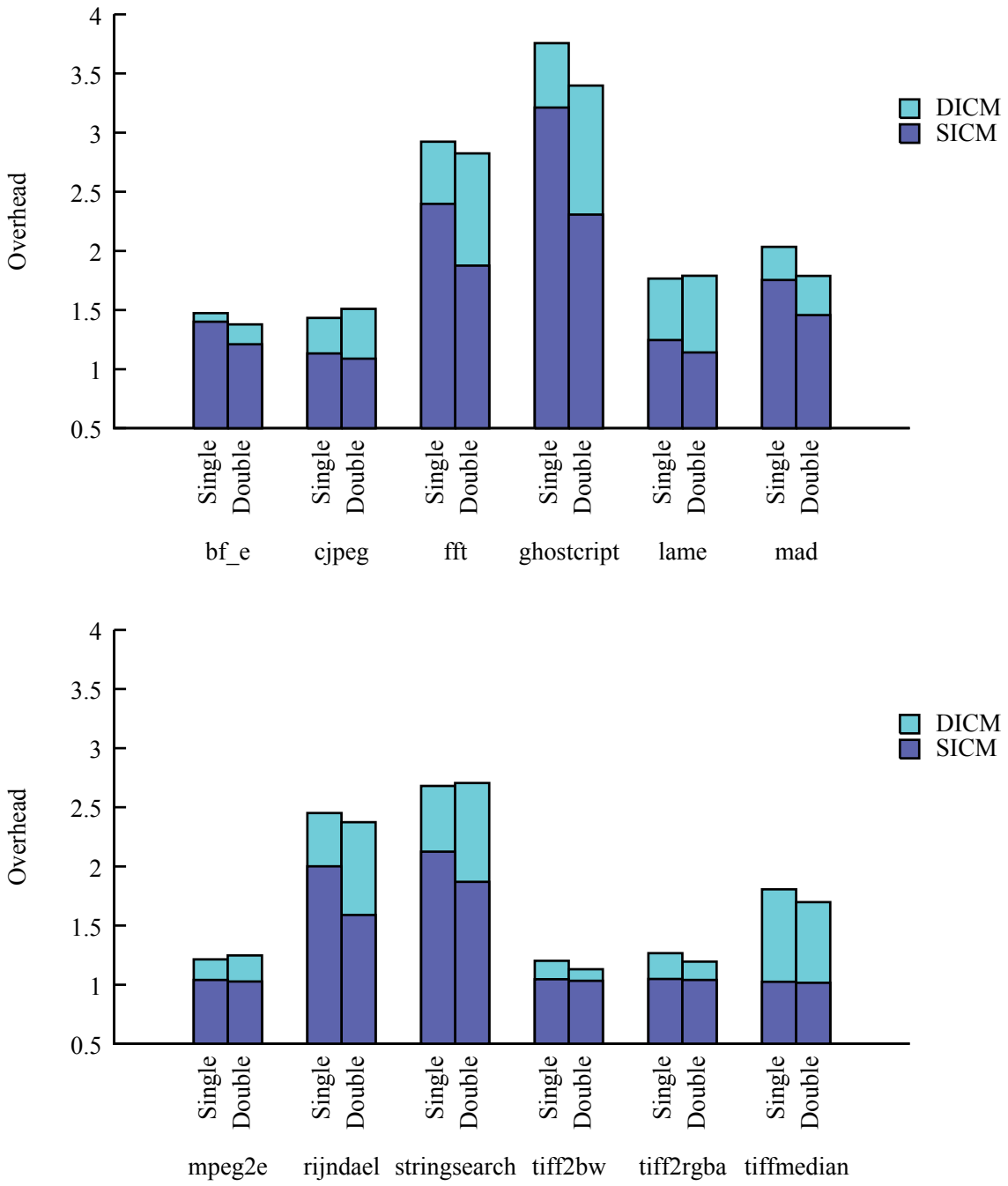


Figure 7.28 Performance Overhead Implications of Using Double-Sized Protected Blocks, Cortex M3, 1 KB

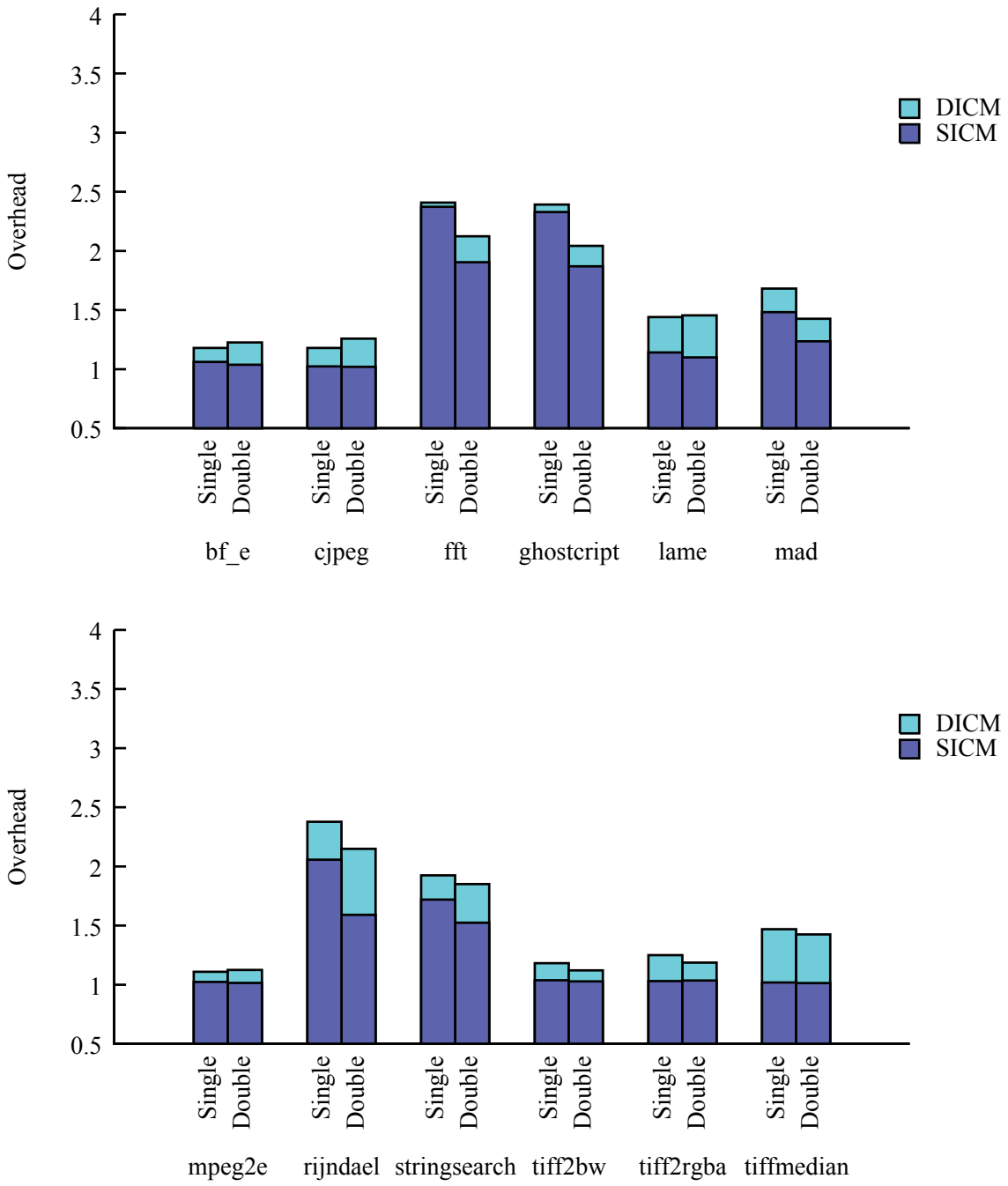


Figure 7.29 Performance Overhead Implications of Using Double-Sized Protected Blocks, Cortex M3, 2 KB

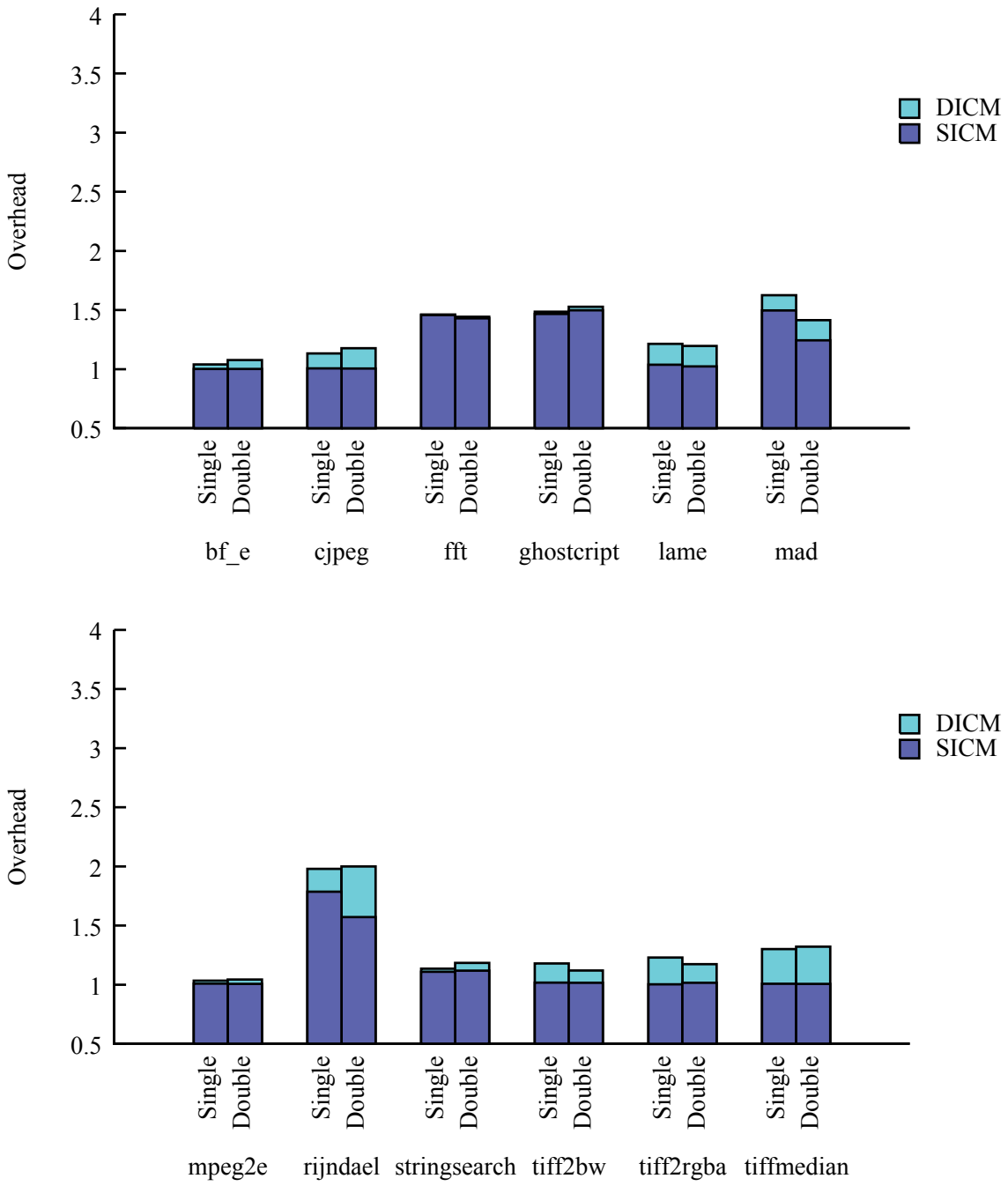


Figure 7.30 Performance Overhead Implications of Using Double-Sized Protected Blocks, Cortex M3, 4 KB

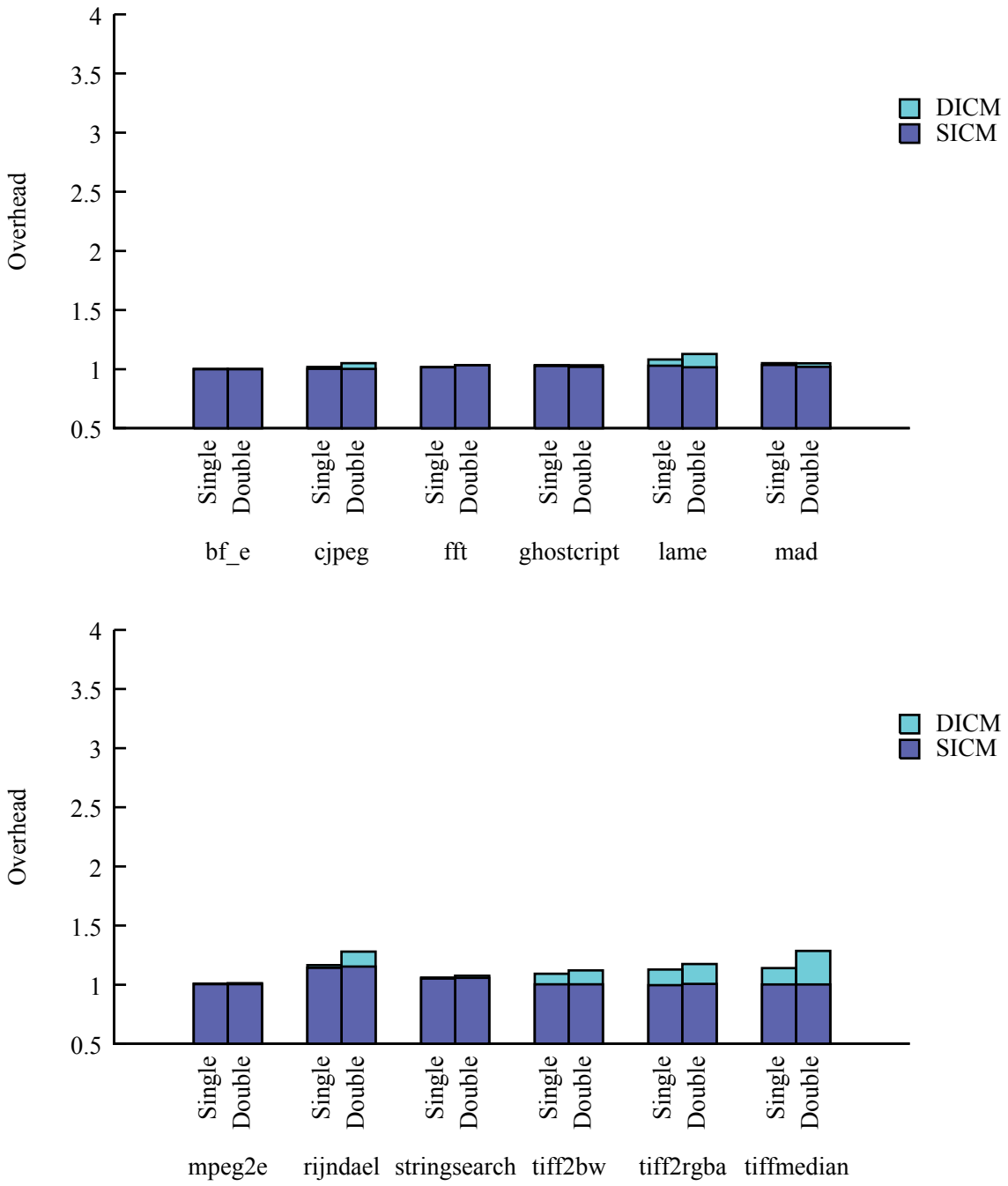


Figure 7.31 Performance Overhead Implications of Using Double-Sized Protected Blocks, Cortex M3, 8 KB

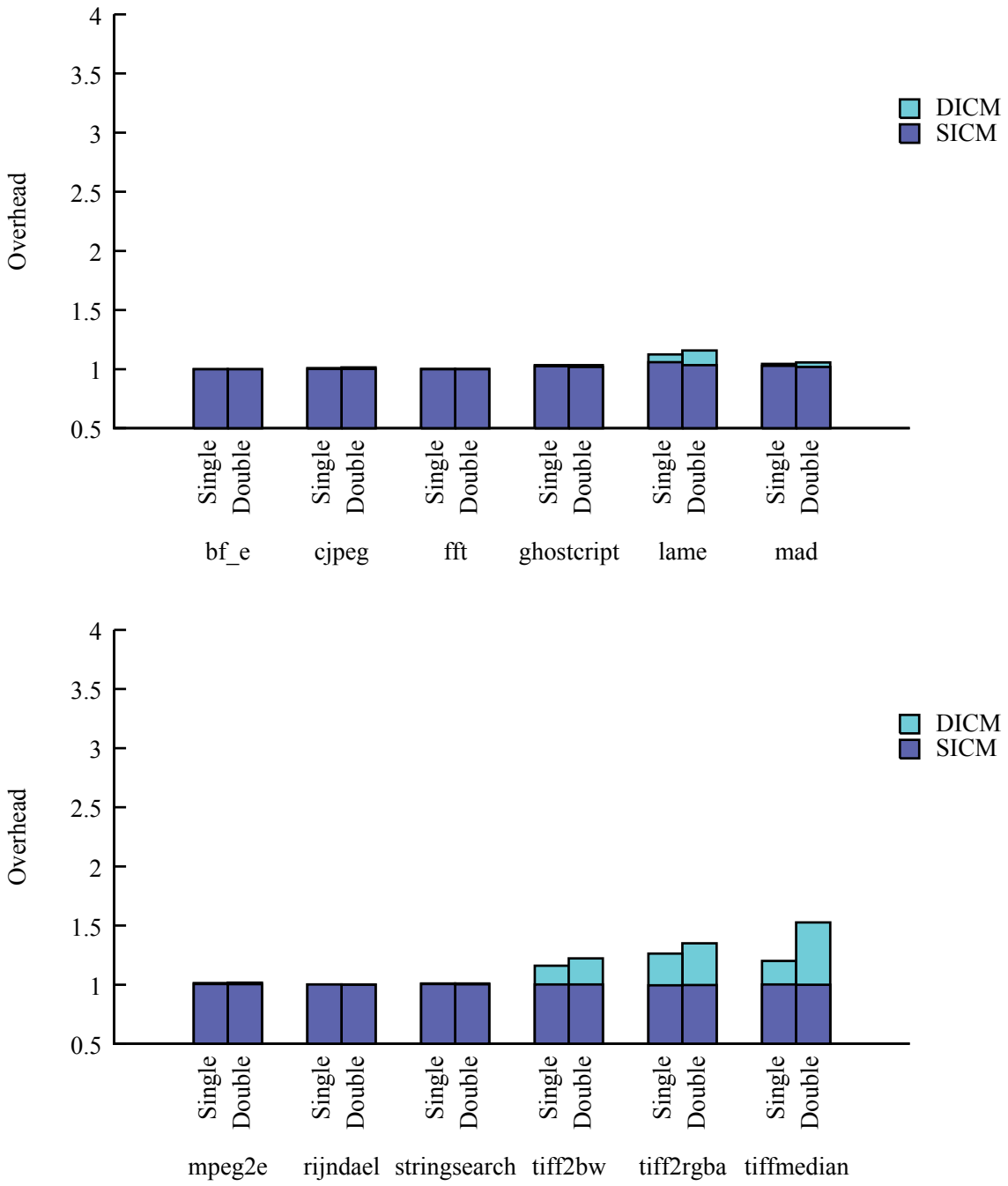


Figure 7.32 Performance Overhead Implications of Using Double-Sized Protected Blocks, Cortex A8, 16 KB

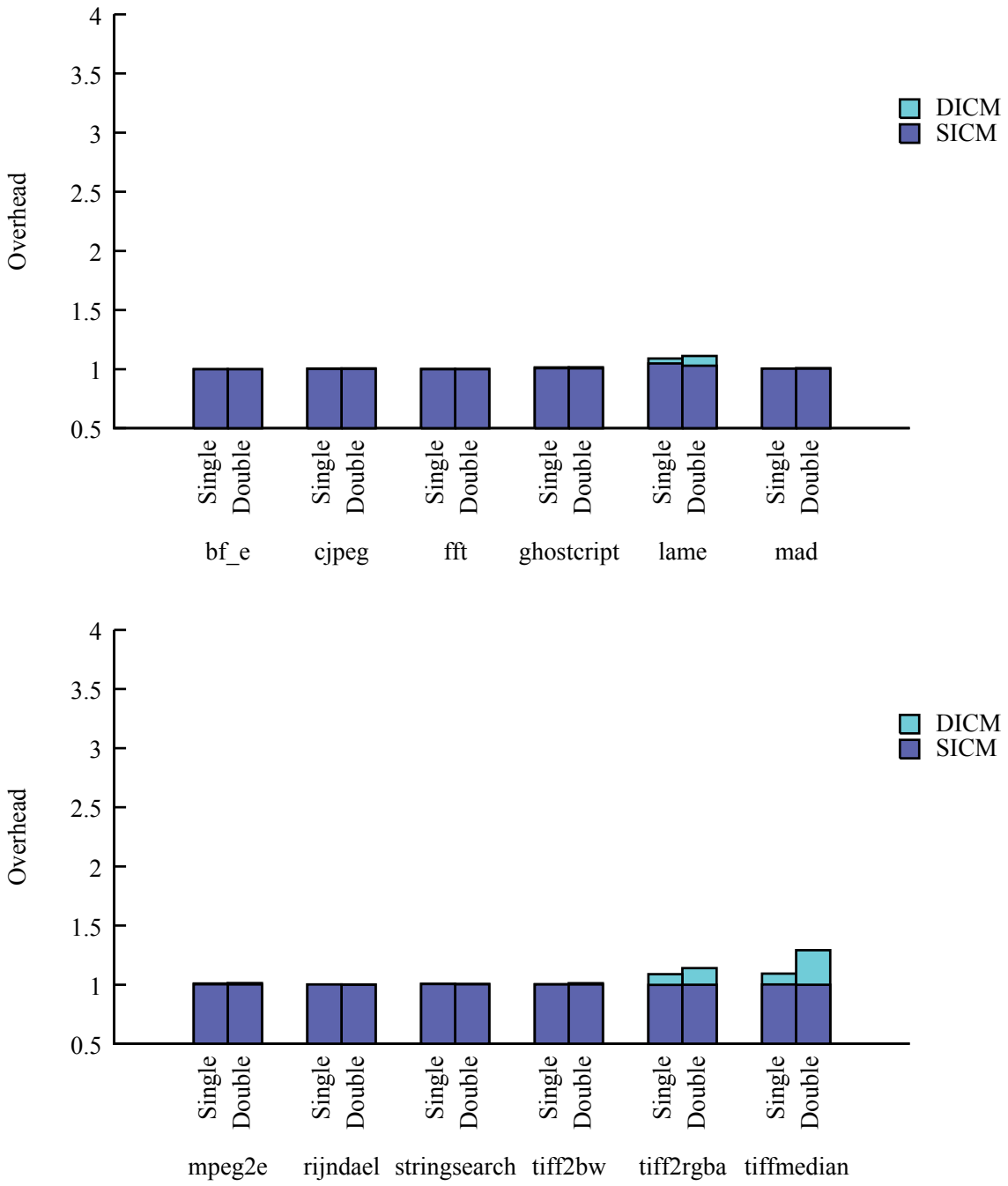


Figure 7.33 Performance Overhead Implications of Using Double-Sized Protected Blocks, Cortex A8, 32 KB

Table 7.16 Performance Overhead Implications of Using Double-Sized Protected Blocks, Cortex M3

Benchmark	Block Size	1 KB		2 KB		4 KB		8 KB	
		SICM	Both	SICM	Both	SICM	Both	SICM	Both
bf_e	Single	1.40	1.47	1.06	1.18	1.00	1.04	1.00	1.00
	Double	1.21	1.38	1.04	1.23	1.00	1.08	1.00	1.00
cjpeg	Single	1.13	1.43	1.02	1.18	1.01	1.13	1.00	1.02
	Double	1.09	1.51	1.02	1.26	1.00	1.18	1.00	1.05
fft	Single	2.40	2.92	2.37	2.41	1.46	1.46	1.02	1.02
	Double	1.88	2.82	1.90	2.12	1.43	1.44	1.03	1.03
ghostcript	Single	3.21	3.76	2.33	2.39	1.47	1.48	1.03	1.03
	Double	2.31	3.40	1.87	2.04	1.50	1.53	1.02	1.03
lame	Single	1.25	1.76	1.14	1.44	1.04	1.21	1.03	1.08
	Double	1.14	1.79	1.10	1.45	1.02	1.20	1.02	1.13
mad	Single	1.75	2.03	1.48	1.68	1.50	1.63	1.03	1.05
	Double	1.46	1.79	1.24	1.43	1.24	1.41	1.02	1.05
mpeg2e	Single	1.04	1.21	1.02	1.11	1.01	1.03	1.00	1.01
	Double	1.03	1.25	1.01	1.12	1.01	1.04	1.00	1.01
rijndael	Single	2.00	2.45	2.06	2.38	1.78	1.98	1.14	1.17
	Double	1.59	2.37	1.59	2.15	1.57	2.00	1.15	1.28
stringsearch	Single	2.13	2.68	1.72	1.92	1.11	1.13	1.05	1.06
	Double	1.87	2.71	1.52	1.85	1.12	1.18	1.06	1.08
tiff2bw	Single	1.05	1.20	1.04	1.18	1.02	1.18	1.00	1.09
	Double	1.03	1.13	1.03	1.12	1.02	1.12	1.00	1.12
tiff2rgba	Single	1.05	1.27	1.03	1.25	1.00	1.23	1.00	1.13
	Double	1.04	1.19	1.03	1.19	1.02	1.17	1.01	1.17
tiffmedian	Single	1.02	1.81	1.02	1.47	1.01	1.30	1.00	1.14
	Double	1.02	1.70	1.01	1.42	1.01	1.32	1.00	1.28

Table 7.17 Performance Overhead Implications of Using Double-Sized Protected Blocks, Cortex A8

Benchmark	Block Size	16 KB		32 KB	
		SICM	Both	SICM	Both
bf_e	Single	1.00	1.00	1.00	1.00
	Double	1.00	1.00	1.00	1.00
cjpeg	Single	1.00	1.01	1.00	1.00
	Double	1.00	1.01	1.00	1.01
fft	Single	1.00	1.00	1.00	1.00
	Double	1.00	1.00	1.00	1.00
ghostcript	Single	1.02	1.03	1.01	1.01
	Double	1.02	1.03	1.01	1.02
lame	Single	1.06	1.12	1.05	1.09
	Double	1.03	1.16	1.03	1.11
mad	Single	1.03	1.04	1.00	1.01
	Double	1.02	1.06	1.00	1.01
mpeg2e	Single	1.00	1.01	1.00	1.01
	Double	1.00	1.02	1.00	1.01
rijndael	Single	1.00	1.00	1.00	1.00
	Double	1.00	1.00	1.00	1.00
stringsearch	Single	1.00	1.01	1.00	1.01
	Double	1.00	1.01	1.00	1.01
tiff2bw	Single	1.00	1.16	1.00	1.00
	Double	1.00	1.22	1.00	1.01
tiff2rgba	Single	1.00	1.26	1.00	1.09
	Double	1.00	1.35	1.00	1.14
tiffmedian	Single	1.00	1.20	1.00	1.09
	Double	1.00	1.53	1.00	1.29

7.4.4 Analytical Model

We use the simulation results from the cipher choice evaluation to generate analytical models of our architecture's performance overhead. We first plot performance overhead versus the cache miss rate for a dataset of interest. Visual inspections of these plots suggest that performance overhead trends piecewise linearly with respect to cache miss rate. We therefore use linear regression to find approximate equations for each piecewise segment and the breakpoint between segments. The linear regression is

performed using Microsoft Solver, with Microsoft Excel used as a front-end. The resulting formulae are valid over the range of cache miss rates that we have simulated. We limit our analysis to the Cortex M3 architecture, as the Cortex A8 demonstrates little or no overhead in most cases. All Cortex M3 cache sizes are considered simultaneously, as the overhead from an individual cache miss is independent of cache size. We analyze each cipher mode separately, producing a plot and an equation for each cipher mode. Furthermore, since our simulated system has independent instruction and data caches, we treat the SICM and DICM modes independently.

7.4.4.1 SICM

The performance overhead incurred by protecting instructions and static data is plotted versus the number of instruction cache misses is plotted in Figure 7.34 and Figure 7.35. Equations (7.1), (7.2), and (7.3) were produced by piecewise linear regression, and can be used to predict the performance overhead y as a function of the instruction cache miss rate x for the CBC-MAC, PMAC, and GCM cipher modes, respectively. The cache miss rate is in units of misses per thousand instructions. These equations are valid for applications with instruction cache miss rates up to 160 misses per 1,000 instructions.

Note that the equations for the PMAC and GCM modes are very similar, and the breakpoints for all three modes are close to each other. Also of interest is that the linear functions for the higher cache miss rates have gentler slopes than their respective lower miss rate functions. This suggests that, during the execution of real programs, our security extensions incur a basic performance penalty up to a certain threshold. After that threshold is met, additional cache misses incur less penalty.

$$y = \begin{cases} 0.02412x + 1.00112 \ for \ x < 57.45560 \\ 0.01424x + 1.56859 \ otherwise \end{cases} \quad (7.1)$$

$$y = \begin{cases} 0.01523x + 1.00018 \ for \ x < 57.29303 \\ 0.00854x + 1.38369 \ otherwise \end{cases} \quad (7.2)$$

$$y = \begin{cases} 0.01517x + 1.00113 \ for \ x < 57.17616 \\ 0.00856x + 1.37908 \ otherwise \end{cases} \quad (7.3)$$

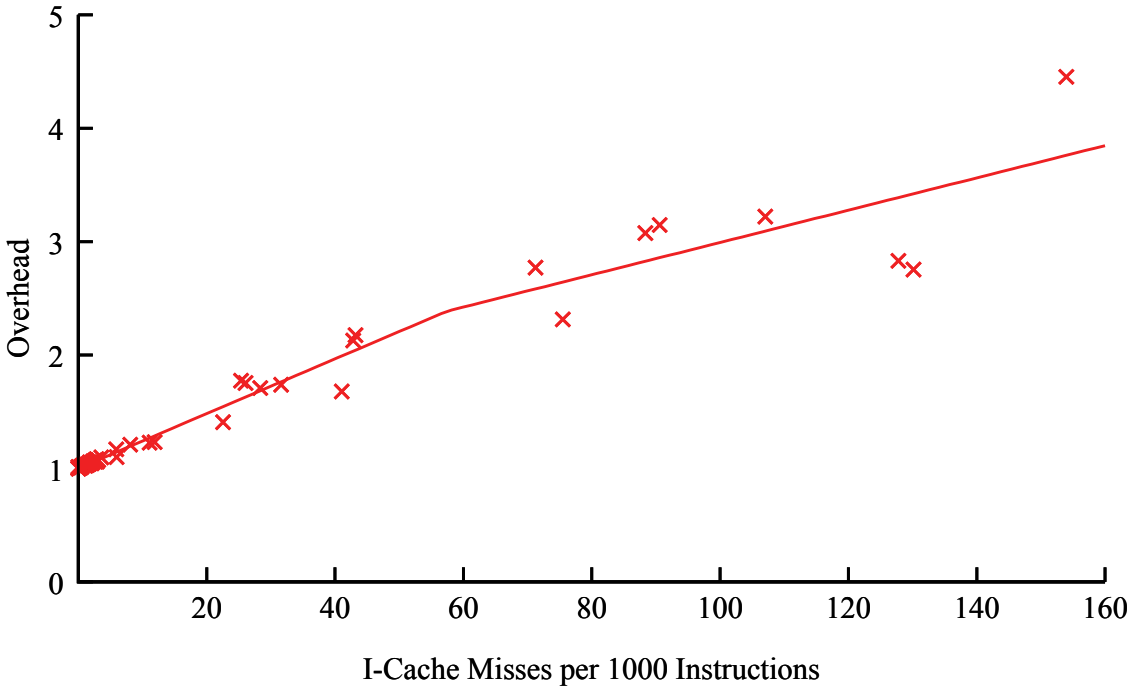


Figure 7.34 Analytical Model of SICM Performance Overhead, CBC-MAC

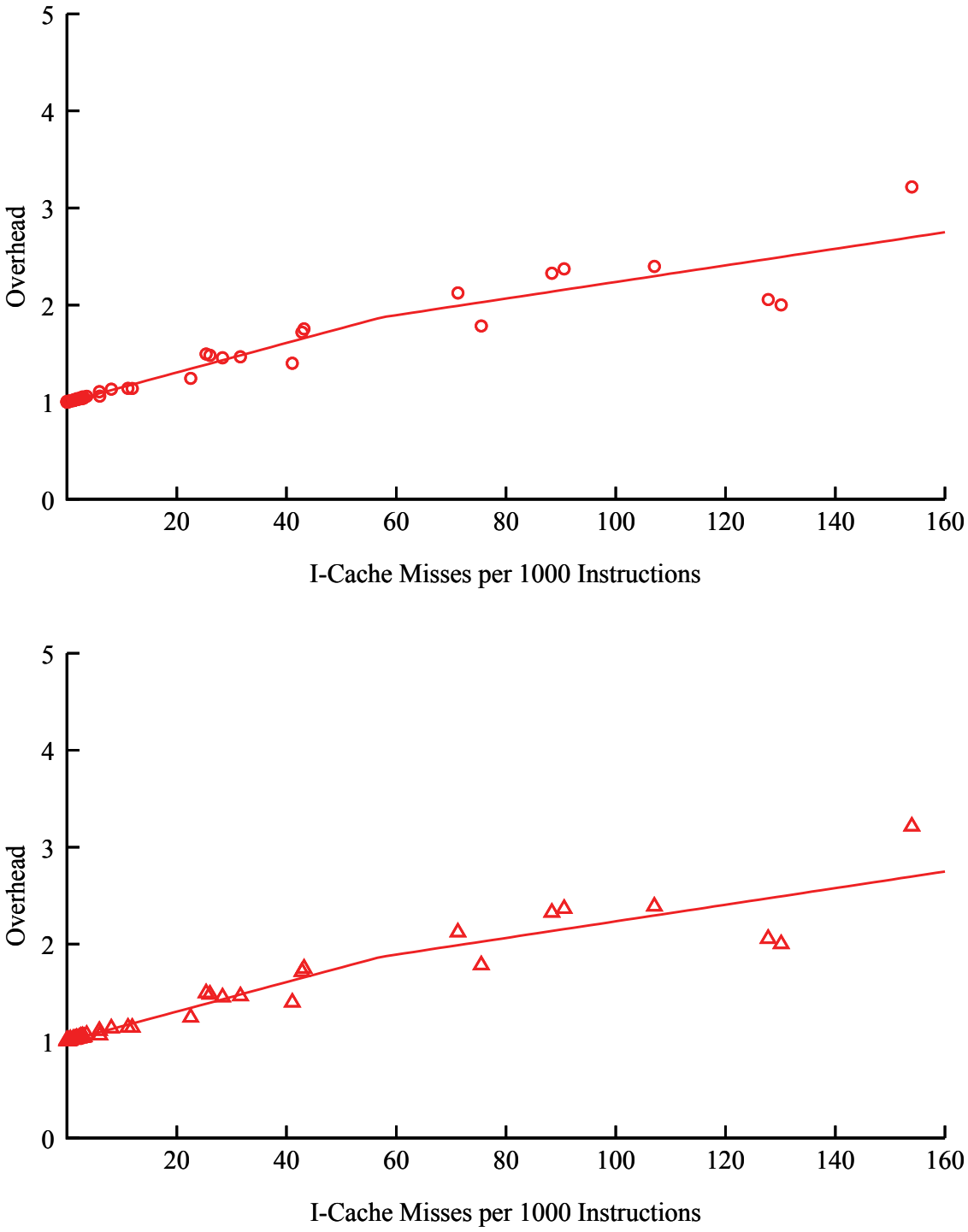


Figure 7.35 Analytical Model of SICM Performance Overhead, PMAC and GCM

7.4.4.2 DICM

The protection overhead incurred by protecting only dynamic data is plotted versus the number of data caches misses in Figure 7.36 and Figure 7.37. Equations (7.4), (7.5), and (7.6) may be used to model this performance overhead for applications with data cache misses up to about 140 per 1,000 instructions. We note a dramatic difference in slope for higher miss rates; the slope, in fact, is negative, indicating that past a certain cache miss rate, security becomes less costly. A rough analogy could be made with receiving a volume discount when purchasing a large number of items.

$$y = \begin{cases} 0.01551x + 0.97536 \; for \; x < 29.43342 \\ -0.00077x + 1.45468 \; otherwise \end{cases} \tag{7.4}$$

$$y = \begin{cases} 0.01009x + 0.99029 \; for \; x < 29.38261 \\ -0.00002x + 1.28733 \; otherwise \end{cases} \tag{7.5}$$

$$y = \begin{cases} 0.00652x + 0.99057 \; for \; x < 29.38261 \\ -0.00021x + 1.18826 \; otherwise \end{cases} \tag{7.6}$$

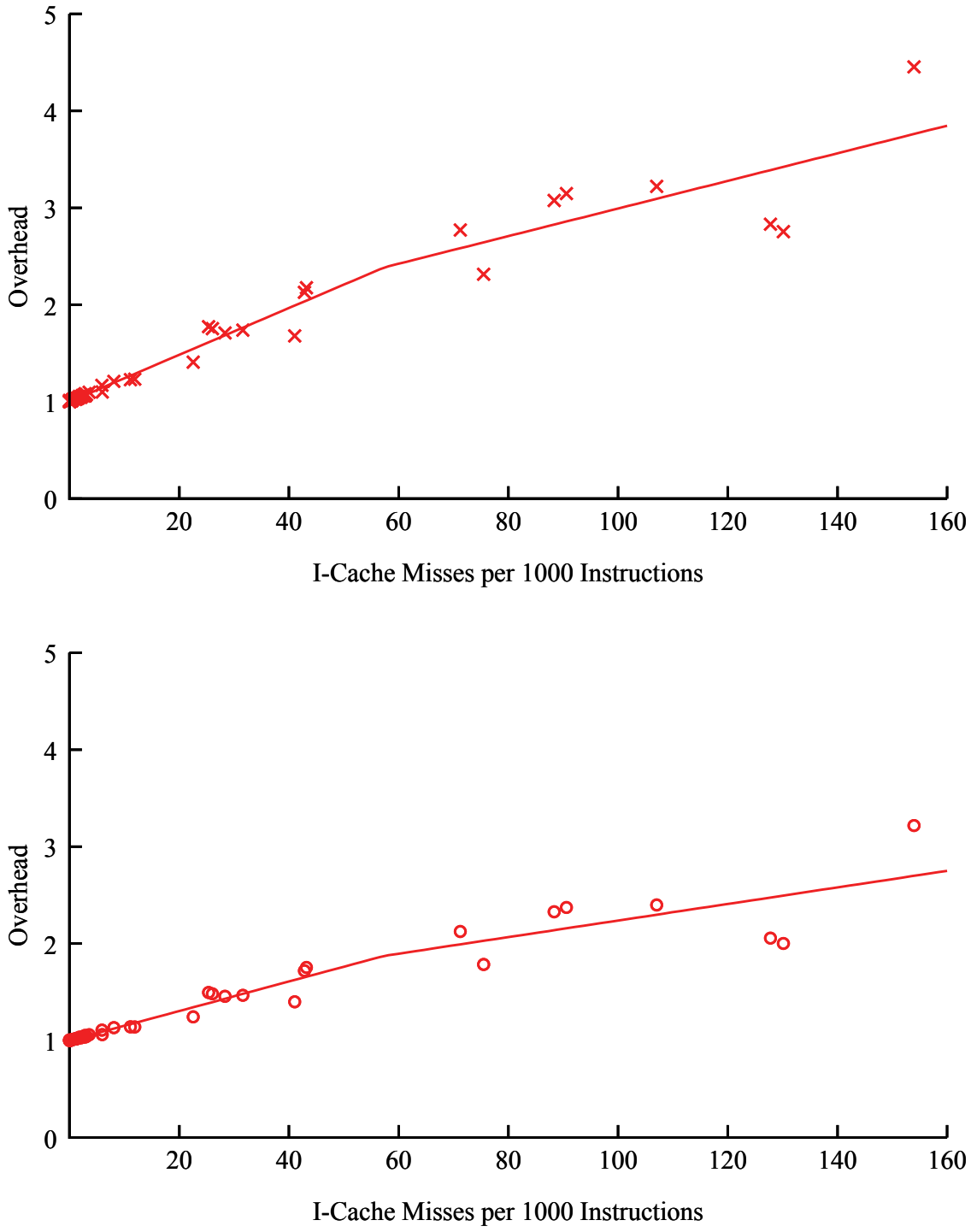


Figure 7.36 Analytical Model of DICM Performance Overhead, CBC-MAC and PMAC

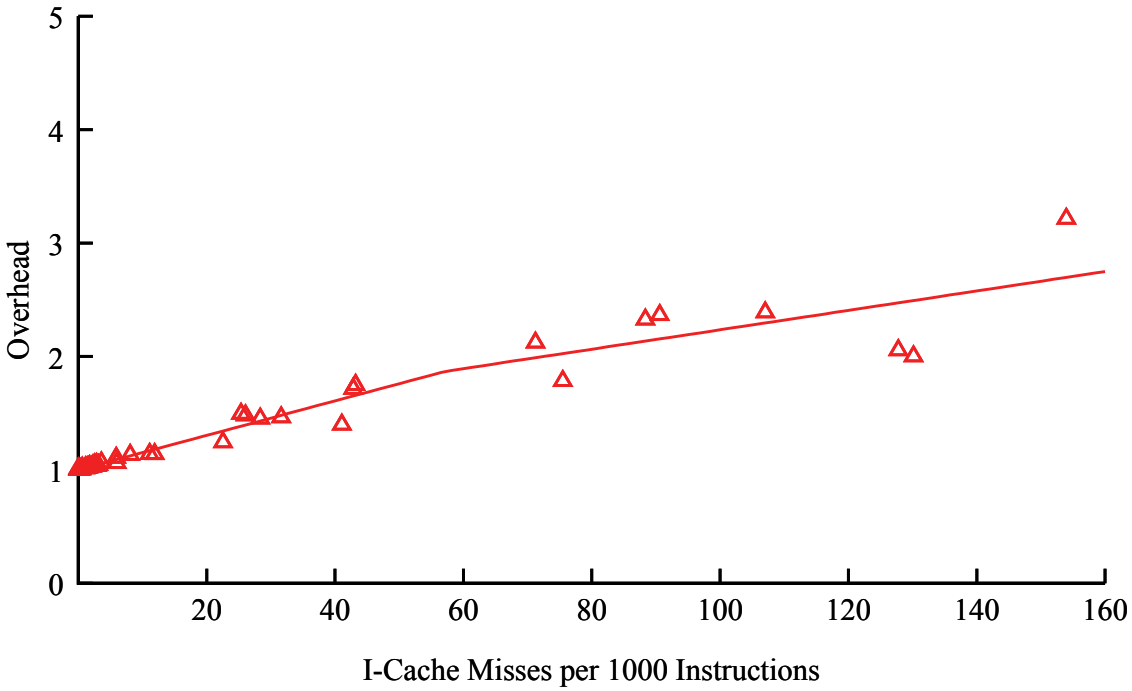


Figure 7.37 Analytical Model of DICM Performance Overhead, GCM

7.5 Comments

In this chapter, we have used a simulator to evaluate the performance of our proposed security enhancements. This evaluation shows that our enhancements are, for the most part, practical for implementation. However, some enhancements, such as signature victim caches, do not increase performance enough to justify their use. We have also evaluated the complexity and memory overheads of our enhancements analytically, and used the simulation results to develop an analytical model that can be used to produce a first-order prediction of performance overhead without having to run the simulator.

CHAPTER 8

AN FPGA SECURE PROCESSOR IMPLEMENTATION

The security extensions proposed in this dissertation should be feasibly implementable using existing technologies. To prove this, we have implemented a prototype of a subset of our security extensions using existing, inexpensive hardware. Our implementation is limited to protecting the integrity and confidentiality of data stored off-chip in a system on a programmable chip (SOPC). We here describe this implementation, including the application of various enhancements discussed in previous chapters and an evaluation of its performance.

8.1 Design Goals

Our implementation has three design goals: ensure a high level of security (integrity and confidentiality), make the security extensions transparent to the programmer, and keep the performance and complexity overheads as low as possible. We ensure confidentiality by encrypting secure data that is stored off-chip. The base implementation ensures integrity by generating cryptographically sound signatures for off-chip data and using those signatures to verify those data when they are brought on-chip. The security extensions are transparent to the programmer other than requiring a function call to initialize the security-related hardware resources. We minimize

performance overhead in our base implementation by overlapping cryptography with memory accesses and buffering verified blocks. Further enhancements include parallelizing encryption/decryption, and parallelizing signature generation.

In the earlier chapters of this dissertation, we assumed that the computer architect could modify the microprocessor, cache controller, TLB, and any other system components as necessary. However, in the SOPC arena, many of these components are implemented as binary intellectual property (IP) cores that cannot be modified. Therefore, a guiding principle of our prototype implementation of security extensions is it will not require the modification of any other cores in the system.

We have implemented these security extensions in a system based on the Altera NIOS II soft-core processor. The test system was implemented on a Cyclone II FPGA using Altera's Quartus II toolchain. The performance of our extensions is evaluated using both a targeted microbenchmark and a small suite of embedded system benchmarks running on the actual SOPC. Evaluation shows that parallelizing encryption/decryption and signature generation yields the best performance, but at the cost of increased complexity.

8.2 Basic Implementation of Security Extensions

This section describes how our basic implementation achieves our three design goals. We begin with a description of how our design achieves security. We then discuss the programming model for our design, and the memory architecture necessary to implement it. We finally discuss how these security extensions are implemented in a hardware resource called the Encryption and Verification Unit (EVU).

8.2.1 Achieving Security

As described in earlier chapters, the basic unit of secure data is a protected block. In systems with on-chip caches, the cache block size, or some multiple thereof, is a convenient protected block size. In our implementation, we chose a protected block size of 32 bytes. For our initial implementation we do not use data caches.

Our design uses cryptography to protect the integrity and confidentiality of data stored off-chip. Confidentiality is protected by encryption. Integrity is protected by generating a 16 byte signature for each protected block of data. We defend against replay attacks by associating a sequence number with each protected block, and using it in encryption/decryption and signature generation.

The confidentiality of data is protected by using the low-overhead OTP encryption scheme described in Section 5.1. Equation (8.1) shows how this encryption is performed. The 32 byte plaintext data block D is divided into two 16 byte sub-blocks $D_{0:3}$ and $D_{4:7}$, which are separately encrypted to form ciphertext sub-blocks $C_{0:3}$ and $C_{4:7}$. The 128-bit key used for pad generation is denoted as *KEY1*, $A(SB_i)$ is the address of sub-block i, SN is the protected block's sequence number, and SP is a secure padding function that generates a unique 128-bit value from the 32-bit address and 32-bit sequence number.

$$C_{4i:4i+3} = D_{4i:4i+3} \; xor \; AES_{KEY1}(SP(A(SB_i), SN)) \quad for \; i = 0..1 \tag{8.1}$$

Decryption is simply the reverse of this operation. The pads are calculated as in Equation (8.1), and then XORed with the ciphertext sub-blocks to produce the desired plaintext sub-blocks.

Signatures are generated using a modified version of the CBC-MAC mode. The protected block's signature *S* is calculated according to Equation (8.2). Another 128-bit key, *KEY2*, is used for signature generation. We also use the same secure padding function defined above, *SP*, operating on the block's address *A(SB)* and sequence number *SN*. The use of the block address prevents splicing attacks, the use of the block text prevents spoofing attacks, and the use of the sequence number prevents replay attacks. If the keys are generated randomly for each run, then cross-executable splicing attacks will also be prevented. The CBC-MAC approach used here differs from the approach in Section 5.3.1 in that the initial vector is not encrypted and the signature is calculated on ciphertext rather than plaintext. These changes are due to the limitations of the AES core used in our implementation; it is not pipelined, so adding an additional operation to encrypt the initial vector would significantly increase latency. This should not greatly effect the resulting cryptographic soundness of the signatures as long as the secure padding function assures uniqueness.

$$S = AES_{KEY2}\left[C_{4:7}\ xor\ AES_{KEY2}(C_{0:3}\ xor\ SP(A(SB), SN))\right] \tag{8.2}$$

If sequence numbers are stored off-chip, then they may be subjected to sophisticated replay attacks in which the sequence number is replayed as well as the protected block and its signature. This gives rise to the necessity of complex structures such as Merkle trees [30, 31] to protect the sequence numbers. Our design assumes that sequence numbers are stored in on-chip memory and are thus invulnerable to replay attacks, and require no additional protection.

When the programmer reads from or writes to secure data at runtime, the appropriate sequence number, encrypted protected block, and signature are fetched.

When the pads are available, the block is decrypted. As the two ciphertext sub-blocks become available, its signature is recalculated. If the calculated signature and fetched signature match, the block has not been subjected to tampering and the read or write operation can proceed. If the signatures do not match, a security violation has occurred and an interrupt is raised. More operational details are given below in Section 8.2.3.

In addition to preventing spoofing, splicing, and replay attacks, we must also prevent the programmer from inadvertently accessing uninitialized blocks. To that end, the sequence number value zero is reserved to indicate that its associated protected block is uninitialized. If a protected block's sequence number is zero, the programmer may write to it, but not read from it. If the sequence number is nonzero, then the programmer may both read from and write to the protected block. A read from an uninitialized block will result in an interrupt.

Whenever a protected block is written back to main memory, its sequence number must be incremented and new pads calculated to encrypt the block. Sequence number overflows are undesirable, as they lead to pad re-use. Our design uses 32-bit sequence numbers; should a particular target application have a strong likelihood of a sequence number rollover, the design may be modified to use 64-bit sequence numbers.

In our design, the two cryptographic keys *KEY1* and *KEY2* are hard-coded in our security extension hardware. For greater security, they could be randomly generated at runtime for each application using methods such as physical unclonable functions [29]. In that case, these keys must be stored in the process control block in an encrypted form in the event of a context switch. An additional hard-coded internal key would be needed, which would then be used to encrypt these keys before and decrypt them after a context

switch. Keys should never leave the chip in plaintext form. Hard-coded keys should only be used if the design will be protected by bitstream encryption.

8.2.2 Programming and Memory Model

An important design goal for these security extensions is that they be as transparent to the programmer as possible. To that end, our implementation does not require the programmer to use any special application programming interface (API) to read and store secure data. An initialization function must be called to initialize the necessary hardware resources (see Section 8.2.3 below). Thereafter, the programmer simply defines his or her pointers appropriately and uses them as normal.

This transparency is possible because of address mapping. A portion of the address space is set aside to physically store encrypted data. A similarly sized portion of the address space is mapped to the EVU. For instance, to read or write the n^{th} word of encrypted data, the programmer will read or write the n^{th} word in the EVU's address space. This transparency is illustrated in the code snippets in Figure 8.1. In the first snippet, OFFCHIP_MEM_BASE_ADDR defines the base address for off-chip memory. The second snippet accesses data relative to SECURE_DATA_BASE_ADDR, which defines the base address for accessing secure data via the EVU.

The memory architecture of our design is illustrated in Figure 8.2. The program text, heap, and stack are all stored in on-chip memory. Sequence numbers should also be stored on-chip. The figure depicts signatures as stored on-chip; they may also be stored in off-chip memory if desired. The shaded region in the address space contains the secure data in its encrypted form, which is physically stored off-chip.

The programmer may read data directly from the encrypted region, but the result would be a word of ciphertext. A direct write to this region would effectively constitute a spoofing attack, and would result in an interrupt the next time this secure data was properly accessed. Secure data should be accessed through an area of the address space assigned to the EVU. Addresses in this region are mapped to those in the encrypted data region, and the EVU handles all decryption and verification. If a block of secure data is no longer needed, its corresponding space in off-chip memory may be reclaimed for unsecured use. However, that block must not be treated as secure data thereafter.

```
/* This code writes data directly to off-chip
   memory in an insecure manner. */
void Array_Access_Insecure()
{
   int i;
   int *pArray;

   pArray = OFFCHIP_MEM_BASE_ADDR;

   for(i = 0; i < 16; i++)
      pArray[i] = i;
}

/* This code writes secure data using the EVU. */
void Array_Access_Secure()
{
   int i;
   int *pArray;

   Initialize_EVU();

   pArray = SECURE_DATA_BASE_ADDR;

   for(i = 0; i < 16; i++)
      pArray[i] = i;
}
```

Figure 8.1 Programmer's View of Securing Data in Off-Chip Memory

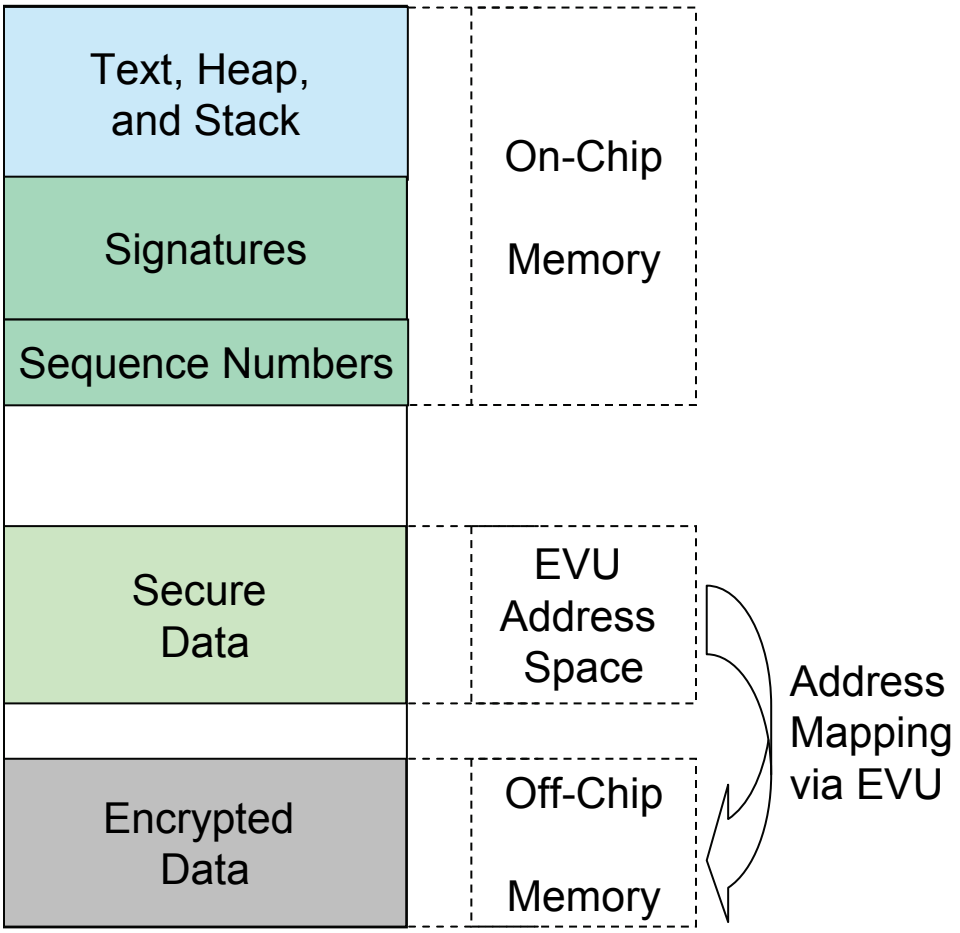


Figure 8.2 Memory Architecture

The maximum number of 32 byte protected blocks is determined by the amount of memory allocated to storing signatures and sequence numbers. Each protected block requires a 16 byte signature and a four byte sequence number. Thus the maximum number of protected blocks N_{PB} in a system is limited by Equation (8.3). In this equation, $Sz(M_{sig})$ and $Sz(M_{seqnum})$ are the sizes in bytes of the memory regions allocated for storing signatures and sequence numbers, respectively.

$$N_{PB} = \min\left(\frac{Sz(M_{sig})}{16}, \frac{Sz(M_{seqnum})}{4}\right) \tag{8.3}$$

Since signatures introduce the greatest memory overhead, the designer may wish to fix the size of the region of memory allocated to signatures, and then calculate the required sizes for the other memory regions. In our implementation, we chose to allocate eight kilobytes of memory for storing signatures. This allows us to have 512 protected blocks of 32 bytes each, for a total of 16 kilobytes of secure data. We thus require two kilobytes of on-chip memory for sequence numbers.

8.2.3 Implementation

The implementation of these security extensions must balance complexity and performance overhead, while at the same time not requiring the modification of any existing soft cores. To that end, the EVU is implemented as an on-chip peripheral attached to the bus. Other implementations are certainly possible, such as embedding the EVU functionality in a custom memory controller. The implementation strategy we choose, however, allows our design to be flexible and applicable to existing systems.

Figure 8.3 shows a block diagram of our implementation of an embedded system incorporating our security extensions. All components of the baseline system are unshaded, while the shaded components are added to implement the security extensions. The baseline system for this implementation is a simple 32-bit NIOS II system-on-a-chip. On-chip memories are used to store program instructions and data (heap and stack). A synchronous dynamic random access memory (SDRAM) controller provides access to off-chip memory. The system is generated using Altera's SOPC Builder, part of the Quartus II toolchain. The on-chip bus interconnects conform to the Altera Avalon standard [53], with loads and stores occurring at the word level.

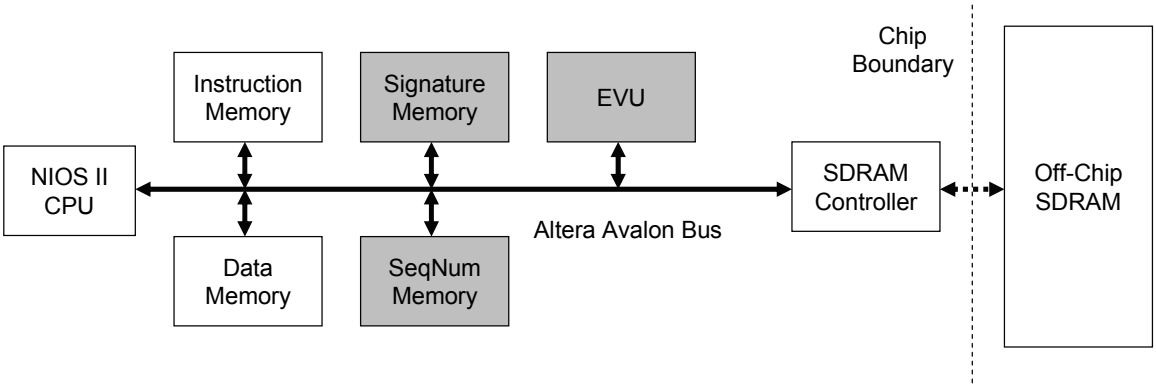


Figure 8.3 System-on-a-Programmable Chip Incorporating Security Extensions

The base system uses a simple NIOS II CPU with no data cache. In a NIOS II system with caches, cache lines are loaded and evicted via sequences of single-word accesses. The EVU would handle these like any other accesses.

The additional hardware to implement the security extensions consists of a discrete EVU peripheral, an on-chip memory for the sequence number table, and an on-chip memory for the signature table. Secure data is physically stored in its encrypted form in the off-chip SDRAM. (As mentioned earlier, signatures may also be stored off-chip if necessary.) The programmer may read directly from the SDRAM; however, if a location in the SDRAM containing secure data is read, encrypted data will be returned. SDRAM locations not used for storing secure data or signatures may be used to store non-sensitive plaintext data.

The internals and interfaces of the EVU are shown in Figure 8.4. In the upper left of this figure are the data and control registers for the EVU. Three data registers specify the base addresses of encrypted data in external memory, the signatures, and sequence

numbers. These should be set in the aforementioned initialization function. (The initialization function should also initialize the sequence number table to all zeros.) The control register allows the programmer to reset the EVU and clear the interrupt. An Avalon bus slave interface allows access to these data and control registers.

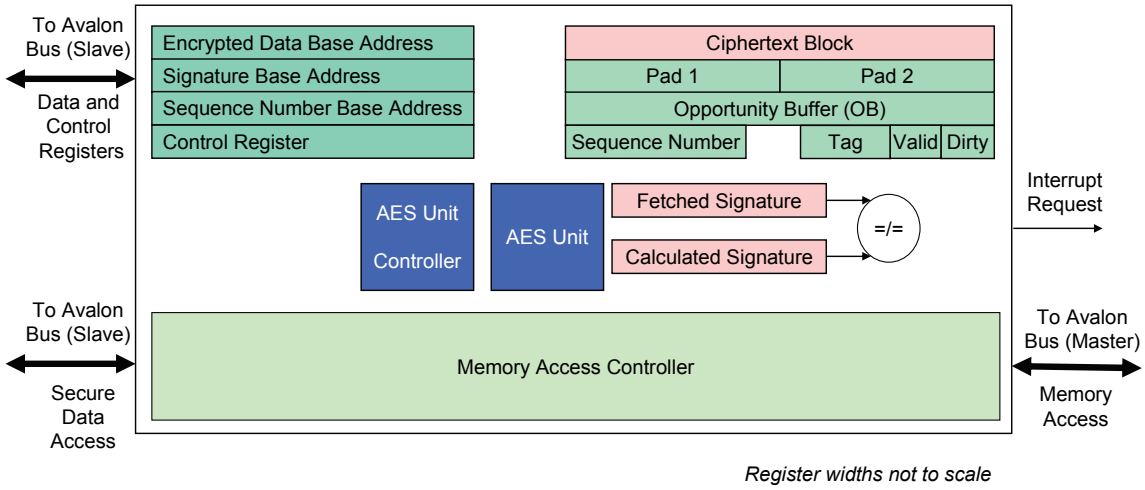


Figure 8.4 Block Diagram of the Encryption and Verification Unit

A second Avalon bus slave interface is shown in the bottom left of the figure. This is the interface that the programmer will use to access secure data. Therefore, the portion of address space allocated to this interface should be commensurate with the amount of protected data. This is achieved by setting the width of the address signal on the slave interface. Avalon slave interface address signals are actually word indices rather than actual addresses. In our sample system, we have 16 kilobytes of secure data, constituting 4,096 32-bit words. Thus, the address bus for this interface must be 12 bits wide to address all 4,096 words.

The memory access controller is a state machine responsible for fetching sequence numbers, signatures, and data blocks from memory and maintaining local buffers. The controller can access on-chip and external memories via an Avalon bus master interface. The EVU also contains an AES core and a state machine to control it. An interrupt interface allows interrupts to be raised by the memory access controller if the programmer tries to read from an uninitialized block or a fetched block and signature fails verification.

The upper right of the figure shows the various buffers used in the EVU. There are buffers for the fetched signature, calculated signature, the ciphertext block that has been read from memory or will be written to memory, the pads used to encrypt and decrypt the block, and the sequence number. An additional structure called the opportunity buffer attempts to reduce performance overhead by taking advantage of the locality of data accesses. Even though the processor will only read or write one word at a time, the entire protected block must be brought into the EVU in order to perform verification. This block is stored in the opportunity buffer as plaintext. Any further reads from or writes to the protected block while it is buffered can be done within the EVU, without having to access external memory. The block's address may be reconstructed from the opportunity buffer's tag. Its sequence number and the pads used to encrypt and decrypt it are also buffered.

When a word from a different block is requested, the block in the opportunity buffer must be evicted, along with its sequence number and signature. If the block is dirty, then it must be written back to external memory. The sequence number must be incremented and the pads recalculated before the plaintext block can be encrypted for

storage. The opportunity buffer's tag is used to calculate the addresses for the block to be written back, its sequence number, and signature.

Figure 8.5 and Figure 8.6 list the algorithms used for reading and writing words of secure data, respectively. Conditions that cause an interrupt to be raised are marked in italicized text. These algorithms reveal the latency hiding mechanisms used in the EVU. Whenever possible, cryptographic operations are done concurrently with memory operations to hide cryptographic latency. When writing to a protected block, new pads must be calculated once the sequence number has been incremented. As Figure 8.6 shows, the sequence number is only incremented when a block in the opportunity buffer is first marked dirty. Pad calculation is begun, and the processor is allowed to continue execution. If another secure read or write is initiated before the new pads have been calculated, the new access is stalled until the pads are completed.

```
Wait for any crypto operations from a previous access to complete.
Is buffer valid and does buffer tag match address?
  Yes: (read hit)
    Return word from buffer and exit.
  No: (read miss)
    Is buffer valid and dirty?
      Yes: (evict block from buffer)
        Encrypt block using buffered pads.
        Write sequence number and cryptotext block to memory.
        In parallel with memory write, calculate block signature.
        When signature is ready, write signature to memory.
        Continue with read miss operation.
      No: (do nothing, continue with read miss operation)
    Fetch sequence number from memory.
    Is sequence number nonzero?
      Yes: (block has been initialized)
        Read block and signature from memory.
        In parallel with memory accesses, calculate pads.
        Decrypt sub-blocks as pads and data are available.
        When block is fully available, calculate signature.
        Do calculated signature and fetched signature match?
          Yes: (everything is fine)
            Buffer block and pads; mark buffer valid and clean.
            Return word from buffer and exit.
          No: (security violation)
            Raise interrupt, mark buffer invalid, and exit.
      No: (trying to read an uninitialized block)
Raise interrupt, mark buffer invalid, and exit.
```

Figure 8.5 Algorithm for Secure Read

```
Wait for any crypto operations from a previous access to complete.
Is buffer valid and does buffer tag match address?
  Yes: (write hit)
    Latch word into buffer.
    Is buffer currently marked clean?
      Yes: (precompute pads for eventual writeback)
        Mark buffer dirty.
        Increment buffered sequence number.
        Start calculation for new pads, and exit.
      No: (do nothing, exit)
  No: (write miss)
    Is buffer valid and dirty?
      Yes: (evict block from buffer)
        Encrypt block using buffered pads.
        Write sequence number and cryptotext block to memory.
        In parallel with memory write, calculate block signature.
        When signature is ready, write signature to memory.
        Continue with write miss operation.
      No: (do nothing, continue with write miss operation)
    Fetch sequence number from memory.
    Is sequence number nonzero?
      Yes: (block has been initialized)
        Read block and signature from memory.
        In parallel with memory accesses, calculate pads.
        Decrypt sub-blocks as pads and data are available.
        When block is fully available, calculate signature.
        Do calculated signature and fetched signature match?
          Yes: (everything is fine)
            Buffer block and pads; mark buffer valid and dirty.
            Increment sequence number.
            Latch word into buffer.
            Start calculation for new pads, and exit.
          No: (security violation)
            Raise an interrupt, mark buffer invalid, and exit.
      No: (initialize the block)
        Set sequence number to 1.
        Start pad calculation.
        Load buffer with zeros; mark buffer valid and dirty.
        Latch word into buffer and exit.
```

Figure 8.6 Algorithm for Secure Write

8.2.4 Initial Performance Evaluation

The EVU's performance was profiled using built-in counters inside the EVU. The EVU's behavior on a read miss is of particular interest, as the actions taken on a read miss also occur on a write miss. The counters report that a read miss in the EVU's opportunity buffer takes about 74 clock cycles. Further analysis reveals that memory accesses complete long before the cryptographic operations, as depicted in Figure 8.7. This analysis assumes that sequence numbers and signatures are stored on-chip, while the protected blocks are stored in off-chip SDRAM.

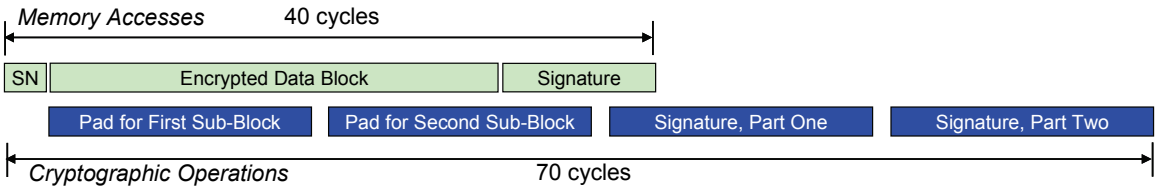


Figure 8.7 Performance Overhead on a Read Miss

8.3 Optimizations and Enhancements

All theoretical analysis in this dissertation, other than that in this chapter, has assumed a pipelined AES core. However, such cores may be prohibitively large to implement in reconfigurable logic. For instance, the simple open-source AES IP core used in this implementation [54] is not pipelined, and still contributes about half of the total complexity overhead of the EVU (see Section 8.4.1). Using a non-pipelined core requires all cryptographic operations to be performed sequentially, resulting in the cryptographic latency being on the critical path of an opportunity buffer miss.

As we have seen in Chapter 5, exploiting parallelism in cryptographic operations can decrease performance overhead. However, such optimizations would require either a more complex, pipelined AES core or another independent AES core acting in parallel. We choose the latter approach for lack of a pipelined AES core optimized for our target platform. The following sections describe how we exploit cryptographic parallelism to reduce performance overhead.

8.3.1 Parallelizing Pad Calculation

The first optimization we pursue is parallelizing pad calculation. Recall that the protected block is divided into two sub-blocks, which are decrypted and encrypted by XORing them with a precomputed pad, as in Equation (8.1). Each pad requires only the sub-block's address and the protected block's sequence number, and thus the pads may be calculated independently. We exploit this independence by initializing a second AES core and generating both pads concurrently. The resulting performance profile is shown in Figure 8.8. Comparing Figure 8.8 with Figure 8.7 shows that parallelizing pad calculation reduces the cryptographic latency from 70 clock cycles to 57 clock cycles.

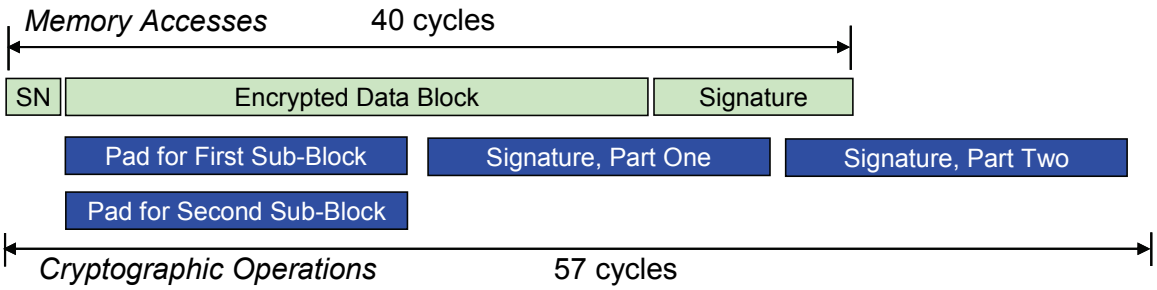


Figure 8.8 Performance Overhead on a Read Miss with Parallelized Pad Generation

8.3.2 Parallelizing Signature Generation

The cryptographic latency may be further reduced by parallelizing signature generation. The CBC-MAC technique, by its very nature, requires that the cryptographic operations required for signature generation be performed in sequence. Therefore, we modify the signature generation methodology to use a variation on the PMAC mode. Using this technique, signatures for each sub-block are calculated independently (Equation (8.4)) and then XORed together to form the signature for the protected block (Equation (8.5)). In these equations, $Sig(SB_i)$ is the signature for sub-block i, $C_{0:3}$ and $C_{4:7}$ are the two ciphertext sub-blocks, SP is the secure padding function defined above, $A(SB_i)$ is the address of sub-block i, SN is the protected block's sequence number, and S is the protected block's signature. Like our CBC-MAC implementation, this implementation of the PMAC mode differs from that discussed in Section 5.3.2 in that the initial vectors are not encrypted. As with our CBC-MAC variant, this should not significantly reduce the cryptographic soundness of our signatures as long as the initial vectors are unique.

$$Sig(SB_i) = AES_{KEY2}(C_{4i:4i+3}\ xor\ SP(A(SB_i), SN)) \quad (8.4)$$

$$S = Sig(SB_0)\ xor\ Sig(SB_1) \quad (8.5)$$

Applying these equations, the cryptographic operation required for each sub-block's signature may be started as soon as the sub-block's ciphertext is available from memory. We again take advantage of the presence of two independent AES cores to perform these operations concurrently. Figure 8.9 shows the resulting performance profile incorporating both parallelized pad generation and parallelized signature generation. Applying these techniques reduces the cryptographic latency to 47 clock cycles (as compared to 70 cycles with a single AES core and 57 cycles for parallelized pad generation alone).

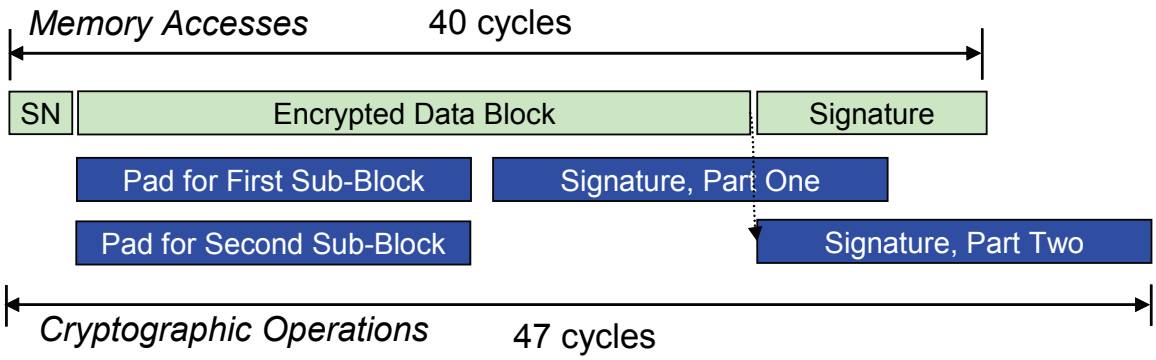


Figure 8.9 Performance Overhead on a Read Miss with Parallelized Pad and Signature Generation

8.4 Evaluation

This section evaluates the complexity and performance overheads introduced by the EVU in an actual SOPC. The implementation of our security extensions was

synthesized, placed, routed, and deployed on a Terasic DE2-70 [55], a low-cost development and education board. The DE2-70 includes an Altera Cyclone II 2C70 FPGA. The complexity overhead is evaluated using the output of the synthesizer, while the performance overhead is evaluated by running several benchmarks on the actual secure system.

8.4.1 Complexity Overhead

Three discrete components were added to the baseline system to implement the security extensions: the EVU, a 2 KB on-chip memory for the sequence number table, and an 8 KB on-chip memory for the signature table. Furthermore, we have three distinct EVU designs: an EVU with one AES core using CBC-MAC, an EVU with two AES cores using CBC-MAC with parallelized signatures, and an EVU with two AES cores using PMAC with parallelized signatures. The complexity overhead introduced by these components is shown in Table 8.1. The figures in the table are reported by the Quartus II tool. The first three rows in the table show the overheads for each of the EVU types. The first number in each cell is the overall figure for that design EVU, followed by the contribution of the AES cores in parenthesis. The final two lines show the overheads induced by the memories, which are constant regardless of which EVU is chosen.

Note that about half of the overhead induced by any given EVU design comes from its AES cores. The EVU itself takes advantage of dedicated logic registers to implement the opportunity buffer. The additional memories consume little in the way of logic cells, but do consume M4K blocks, which are on-chip RAM resources. Recall that signatures need not be stored on-chip; they may be stored in an off-chip memory if on-chip memory space is at a premium. The higher complexity of the PMAC EVU design

and CBC EVU design with parallelized pads is due to the presence of the second AES core. Also, the complexity of the PMAC EVU design is similar to that of the CBC EVU design with parallelized pads.

Table 8.1 Complexity Overhead

Component Name	Logic Cells	Dedicated Logic Registers	M4K Blocks
EVU – CBC	8,321 (5,031)	2,768 (658)	0
EVU – CBC with Parallelized Pads	13,514 (10,062)	3,403 (1,316)	0
EVU – PMAC	13,780 (10,062)	3,564 (1,316)	0
Sequence Number Memory (2 KB)	2	0	4
Signature Memory (8 KB)	2	0	16

8.4.2 Benchmarks

We run a suite of benchmarks to evaluate the performance overhead introduced by our security extensions and explore the design space. A microbenchmark is used to stress-test the system and evaluate its performance under a worst-case scenario. Four actual benchmarks for embedded systems are used to evaluate performance under a more realistic workload. Performance overhead is determined by dividing the number of clock cycles required to run the benchmark on hardware with a secure configuration of interest by the number of cycles required to run the same benchmark on a system without security extensions.

The worst-case performance overhead introduced by the security extensions is evaluated by running a microbenchmark to stress-test the system. The microbenchmark potentially introduces far greater overhead than an actual application. It reads and writes

to an eight kilobyte array in memory with a varying stride factor. When performing secure writes, a miss in the opportunity buffer will always cause a writeback. Baseline results are measured by reading and writing directly to SDRAM. Varying the stride factor allows the benchmark to vary the degree to which it takes advantage of the opportunity buffer. With a stride of one, it takes full advantage of the buffer, with an opportunity buffer miss every eighth access. With a stride of eight, an opportunity buffer miss occurs every access, thus allowing us to measure the average time required to fetch and verify a protected block from off-chip memory. Unless otherwise noted, neither the baseline nor secure systems contain data caches or any other performance enhancement mechanisms other than those in the EVUs being evaluated. This allows us to see the worst-case, bottom-line latencies. Therefore, the latencies reported from the microbenchmark are worse than they would be in a more realistic system containing one or more levels of data cache.

In addition to the microbenchmark, four actual benchmarks representing typical workloads for embedded processors were ported to run on the secure system. We chose two benchmarks, ADPCM and a cyclic redundancy check (CRC) algorithm, CRC32, from the MiBench suite [47]. We also chose two digital signal processing algorithms, a fast Fourier transform (FFT) and finite impulse response (FIR) filter, to use as benchmarks [56]. (Note that this FFT benchmark is different from that used in Chapter 7.) The benchmarks were modified to place buffers, working variables, and lookup tables in secure memory. Input and output files were read from and written to a personal computer using the Altera Host Filesystem driver. This introduced a source of uncertainty in runtimes, and so all performance data reported for these benchmarks are

averages across several runs. These benchmarks are profiled in Table 8.2. The table shows the average number of cycles required for execution in the unsecured case (with data placed in off-chip SDRAM), as well as the numbers of secure reads and writes, and the read and write opportunity buffer miss rates.

Table 8.2 Embedded System Benchmarks

Benchmark	Avg. Cycles Unsecured (millions)	Secure Reads	Secure Writes	OB Read Miss Rate [%]	OB Write Miss Rate [%]
ADPCM	248.80	26,661,720	14,044,719	17.97	14.87
CRC32	358.81	1,368,864	256	96.87	12.50
FFT	18.91	20,920	10,434	55.65	87.71
FIR	12.35	35,422	9,241	73.56	6.24

8.4.3 Effects of Cryptography Approaches

The suite of benchmarks was run on three secure systems incorporating EVUs with all three designs discussed in this chapter: CBC-MAC without any cryptographic parallelization, CBC-MAC with parallelized pad generation, and PMAC with parallelized pad and signature generation. The performance overheads experienced by these benchmarks are presented in Table 8.3. The first two sections of this table report the worst-case read and write overheads, respectively, as reported by the microbenchmark. The third section reports the overhead from the more realistic benchmarks.

As the table shows, the PMAC design consistently outperforms the other designs, as would be expected based on theoretical analysis. This holds for both the raw, worst-case overhead as reported by the microbenchmark and the more realistic benchmarks.

This suggests that the PMAC design should be used if its additional complexity relative to CBC-MAC with a single AES unit can be tolerated.

The two benchmarks with long runtimes, ADPCM and CRC32, exhibit very low overhead. ADPCM even takes advantage of the prefetching behavior of the opportunity buffer, and experiences a speedup. The CRC32 benchmark, even with its high opportunity buffer read miss rate, appears to amortize performance overhead over its runtime, and still exhibits negligible overhead when using a CBC EVU. The FFT and FIR benchmarks, on the other hand, have much shorter runtimes and high opportunity buffer miss rates, thus exhibiting a much greater sensitivity to overhead from security extensions.

Table 8.3 Performance Overhead Implications of EVU Design

Benchmark	Performance Overhead		
	CBC	CBC with Parallelized Pads	PMAC
Microbenchmark Read Accesses			
Miss Every 8th Access	0.94	0.92	0.90
Miss Every 4th Access	1.06	1.02	0.98
Miss Every 2nd Access	1.31	1.21	1.15
Miss Every Access	1.80	1.61	1.47
Microbenchmark Write Accesses with Writebacks			
Miss Every 8th Access	1.37	1.35	1.33
Miss Every 4th Access	1.73	1.68	1.63
Miss Every 2nd Access	2.44	2.35	2.24
Miss Every Access	3.85	3.68	3.46
Embedded System Benchmarks			
ADPCM	0.99	0.97	0.97
CRC32	1.02	1.01	1.00
FFT	1.28	1.27	1.26
FIR	1.14	1.09	1.07

8.4.4 Effects of Signature Location

We use the same suite of benchmarks to evaluate the performance overhead incurred by storing signatures in off-chip SDRAM rather than in an on-chip memory. We use the PMAC EVU design, as it has the lowest cryptographic latency and is thus more likely to show the effects of longer memory fetch times. The resulting performance overheads are shown in Table 8.4. The microbenchmark clearly shows an increase in performance overhead when signatures are moved off-chip, but it is relatively minor for all but the extreme worst case with a miss on every access. The embedded system benchmarks, however, exhibit very little sensitivity to performance overhead, incurring about the same amount of overhead regardless of signature location. These figures suggest that, for actual applications, storing signatures off-chip should not introduce prohibitive latencies. System designers may thus conserve on-chip memory resources when needed without suffering prohibitive performance overheads.

Table 8.4 Performance Overhead Implications of Signature Location

Benchmark	Performance Overhead	
	PMAC Signatures On-Chip	PMAC Signatures Off-Chip
Read Accesses		
Miss Every 8th Access	0.90	0.92
Miss Every 4th Access	0.98	1.02
Miss Every 2nd Access	1.15	1.23
Miss Every Access	1.47	1.65
Write Accesses with Writebacks		
Miss Every 8th Access	1.33	1.37
Miss Every 4th Access	1.63	1.71
Miss Every 2nd Access	2.24	2.41
Miss Every Access	3.46	3.80
Embedded System Benchmarks		
ADPCM	0.97	0.98
CRC32	1.00	1.00
FFT	1.27	1.27
FIR	1.07	1.07

8.4.5 Effects of Data Caching

The analysis presented in this chapter has assumed that the processor has no data cache. Many mid-range to high end embedded processors, however, will have one or more levels of data cache. We therefore use the benchmark suite to evaluate the performance of our security extensions in the presence of a data cache. The benchmarks were run in systems with cache sizes of 2 KB, 4 KB, and 8 KB, with and without security extensions. Recall from Section 4.1 that protected block size should be some multiple of cache block size for best performance. As our protected block size is already set at 32 bytes, we choose cache line sizes of 32 bytes as well.

The observed performance overhead is presented in Table 8.5. The microbenchmark operates on an 8 KB array, and thus will behave well for a cache size of 8 KB, but will cause severe thrashing for smaller caches. As the results from the

microbenchmark show, when an application is thrashing in the cache, the security extensions amplify the thrashing's deleterious effects. However, when an application is well-behaved with respect to the cache, the security extensions introduce negligible overhead.

Table 8.5 Performance Overhead Implications of Data Caching

Benchmark	Performance Overhead		
	PMAC 2 KB D-Cache	PMAC 4 KB D-Cache	PMAC 8 KB D-Cache
Read Accesses			
Miss Every 8th Access	1.11	1.12	1.00
Miss Every 4th Access	1.21	1.21	1.01
Miss Every 2nd Access	1.37	1.37	1.01
Miss Every Access	1.58	1.58	1.02
Write Accesses with Writebacks			
Miss Every 8th Access	1.38	1.40	1.01
Miss Every 4th Access	1.69	1.72	1.02
Miss Every 2nd Access	2.25	2.27	1.03
Miss Every Access	3.29	3.33	1.05
Embedded System Benchmarks			
ADPCM	0.91	0.95	1.00
CRC32	0.99	1.02	0.99
FFT	0.86	0.99	1.01
FIR	0.93	0.93	0.97

The more realistic benchmarks have relatively small working data sets, so they are more well-behaved in these small data caches than the microbenchmark. The effects of the EVU should therefore be negligible in these benchmarks. However, the benchmarks all exhibit a speedup when running with the EVU with smaller caches, and performance overheads approaching unity as the cache size increases. We can conclude that the performance of systems with a data cache and an EVU is comparable to that of

systems with only a data cache, but the uncertainty introduced by using the Altera Host Filesystem driver prevents us from drawing any further conclusions.

8.5 Comments

The implementation documented in this chapter proves that the sign-and-verify security extensions described in this dissertation can be feasibly implemented in low-cost embedded systems. Existing technology allows security extensions to be implemented right now in systems utilizing soft-core processors; designers of such systems need not wait for security features to be included in future generations of microprocessors. Furthermore, the performance overhead results from the optimizations explored in this chapter bear out the theories that were described above in Chapter 5. They demonstrate that the theory applies in actual hardware, not just simulations.

This chapter should contain sufficient information to allow the interested reader to design their own security extensions using the hardware description language of their choice. However, we offer the source code of our implementation as an electronic appendix to this dissertation. The basic principles and optimizations presented in this dissertation and used in our implementation may be easily adapted for use with other soft-core processors.

CHAPTER 9

RELATED WORK

In this chapter, we briefly survey several architectural techniques that have been proposed to support software and data integrity and confidentiality. Security may be approached from both the software and hardware perspectives. Software techniques may be classified as static (relying on the detection of security vulnerabilities in code at design time) and dynamic (adding code to enhance security at runtime). A survey of static and dynamic software techniques may be found in [4]. Hardware techniques rely primarily on hardware to ensure security, often with some degree of software support. This chapter focuses on hardware techniques, as our proposed security architectures are hardware-oriented.

Several non-comprehensive hardware techniques have been put forth to address common types of attacks. Xu *et al.* [57] and Ozdoganoglu *et al.* [58] propose using a secure hardware stack to defend against stack buffer overflow attacks. Tuck *et al.* [59] suggest using encrypted address pointers. Suh *et al.* [60] and Crandall and Chong [61] propose that all data coming from untrusted channels be tagged and not allowed to be used as a jump target. Barrantes *et al.* [62] randomize a processor's instruction set to make attacks more difficult. Some techniques address side channel-attacks on software cryptography, such as Wang and Lee's proposal [63] to partition caches to thwart cache

miss analysis and Ambrose *et al.*'s [64] injection of random code to defeat power analysis attacks.

Our approach, however, is intended to be more comprehensive than the proposals mentioned above. Therefore, we more thoroughly examine proposals that are similarly comprehensive in both the uniprocessor and multiprocessor domains. Uniprocessor solutions may be further divided into proposals from academia, which are well documented, and proposals from industry, which are not as well documented due to their proprietary nature. We finally examine solutions targeting reconfigurable logic. Our research primarily involves uniprocessor systems, with a focus on embedded systems such as might be implemented in reconfigurable logic, so the uniprocessor and reconfigurable logic topics are most salient for this dissertation.

9.1 Uniprocessor Proposals

Most secure processor research to date has focused on systems with a single microprocessor. This type of system encompasses many general purpose computing systems and embedded systems. In this section, we examine comprehensive proposals for securing uniprocessor systems from both the academic and commercial sectors.

9.1.1 Academic

Ragel and Parameswaran [65] introduce an architecture for verifying code integrity. The compiler calculates a checksum for each basic block. Special instructions are inserted at the beginning of each basic block to load its checksum into a dedicated register. The checksum is independently calculated as the block executes, and when an instruction that alters control flow is encountered, the calculated checksum is compared

with the loaded checksum. If they mismatch, then the block has been subjected to tampering. This approach requires both software and hardware support, including compiler modifications and adding custom instructions to the instruction set. Also, it only targets instruction integrity, and does not address data integrity or any form of confidentiality.

The execute-only memory (XOM) architecture proposed by Lie *et al*. [66] provides an architecture meeting the requirements of integrity and confidentiality. Main memory is assumed to be insecure, so all data entering and leaving the processor while it is running in secure mode is encrypted. This architecture was vulnerable to replay attacks in its original form, but that vulnerability was corrected in [67]. The drawbacks to this architecture are its complexity and performance overhead. XOM requires modifications to the processor core itself and to all caches, along with additional security hardware. This architecture also incurs a significant performance overhead, by its designers' estimation, of up to 50%.

The high overhead of XOM is reduced by the architectural improvements proposed by Yang *et al*. [34]. They only address confidentiality, as their improvements are designed to work with XOM, which already addresses integrity concerns. They propose to use a one-time pad (OTP) scheme for encryption and decryption, in which only the pad is encrypted and then XORed with plaintext to produce ciphertext, or with ciphertext to produce plaintext. They augment data security by including a sequence number in the pad for data blocks, and require an additional on-chip cache for said sequence numbers. While their scheme greatly improves XOM's performance, it inherits its other weaknesses.

Gassend *et al*. [31] propose to verify untrusted memory using a tree of hashes. They only address integrity, suggesting that their architecture can be added to a system such as XOM, which will handle confidentiality concerns. The use of a hash tree introduces significant bandwidth overhead, which is alleviated by integrating the hash mechanism with system's caches. However, their integrity-only overhead is still high, with a maximum of 20% for the most efficient architecture they propose.

Lu *et al*. [68] propose a similar architecture, using a message authentication code (MAC) tree. MACs are computed for each cache block, incorporating its virtual address and a secret application key. For higher level nodes, MACs are computed using those from the lower level and a random number generated from thermal noise in the processor. They propose to enhance performance by caching MAC data on the chip. This MAC tree architecture does show an improvement over the hash tree proposed by Gassend *et al*., but it still introduces an average performance overhead of between 10% and 20%.

Platte and Naroska [41] describe another tree-based sign-and-verify system for protecting the integrity of code and data, also protecting the values of registers during traps to the operating system. They treat dynamically generated code in the same manner as dynamic data, but do not allow the use of dynamically linked libraries. Their design only addresses integrity, and does not ensure confidentiality. Furthermore, verification is not immediate; data block verification is only guaranteed to complete by the next sequence call or context switch. This opens a window of vulnerability during which malicious instructions may execute unchecked. Due to the securing of registers, the compiler and operating system must be modified to utilize added instructions for accessing secure data. No performance overhead analysis is presented.

Elbaz *et al.* [69] develop a technique for performing decryption and integrity checking at the same time. They take advantage of the spreading property of the AES algorithm, whereby every bit in a plaintext block influences every bit in the corresponding ciphertext block. Every block of protected data is appended with a random nonce before each encryption. The nonces are stored on-chip, and when a protected block is decrypted, the resulting plaintext nonce is compared with the stored nonce. If the nonces match, the block is safe for use. An average simulated overhead of 4% is reported. This approach requires a method for generating nonces that is at once random yet also deterministic enough to guarantee that the same nonce will never by generated twice. It also requires an on-chip resource to store a table of expected nonces; however, this also eliminates the need for a tree-like structure in memory. This architecture is extended in [70] to support off-chip nonce storage. In the extended architecture, the nonce consists of the protected block address and counter value. A tree-like structure is used to protect the counter values. Their approach introduces a 100% memory overhead, and no performance evaluation is presented.

Suh *et al*. [71] propose an architecture that addresses confidentiality and overall integrity. Their architecture uses one-time pad (OTP) encryption to provide confidentiality with relatively low overhead. However, since their cryptographic functions take a timestamp as an input, they propose that the entire protected memory be re-encrypted on the unlikely event of a timestamp counter rollover. To reduce overhead from integrity checking, they propose to construct a log of memory accesses using incremental multiset hashes. They assume that a program produces meaningful, signed outputs either at the end of its execution or at discrete intervals during execution. Their

architecture verifies the hashed memory access sequences only when those outputs are produced. Since verification occurs infrequently, it introduces negligible overhead. The major drawback is that tampering is not immediately evident, leaving the system potentially vulnerable between verifications.

The work of Milenković *et al*. [4, 26, 72] provides the foundation for the research documented in this dissertation, and introduced many of the elements used in this work. Their proposed architecture addresses only the integrity of instructions, and involves signing instruction blocks during a secure installation procedure. These signatures are calculated using instruction words, block starting addresses, and a secret processor key, and are stored together in a table in memory. At runtime, these signatures are recomputed and checked against signatures fetched from memory. The cryptographic function used in the architecture is a simple polynomial function implemented with multiple input shift registers. The architecture is updated in [73] and [74], adding AES encryption to increase cryptographic strength and embedding signatures with instruction blocks rather than storing them in a table. This architecture remains vulnerable to splicing attacks, since signatures in all programs use the same key.

Drinić and Kirovski [24] propose a similar architecture to that of Milenković *et al*., but with greater cryptographic strength. They use the CBC-MAC cipher, and include the signatures in the cache line. They propose to reduce performance overhead by reordering basic blocks, so that instructions that may not be safely executed in a speculative manner are not issued until signature verification is complete. The drawback to this approach is that it requires significant compiler support, and may not consistently

hide the verification overhead. Furthermore, their architecture does not address confidentiality, and is vulnerable to replay and splicing attacks.

A joint research team from the Georgia Institute of Technology (GA Tech) and North Carolina State University (NCSU) has proposed several secure processor designs. Yan *et al.* [42] describe a sign-and-verify architecture using Galois/Counter Mode cryptography. They protect dynamic data using split sequence numbers to reduce memory overhead and reduce the probability of a sequence number rollover. A tree-like structure is used to protect dynamic data against replay attacks. Rogers *et al.* [33] lower the overhead of the design by restricting the tree structure to only protect sequence numbers. They claim an average performance overhead of 11.9%. This overhead may be artificially low as they use "non-precise integrity verification," which allows potentially harmful instructions to execute and retire before they are verified.

9.1.2 Commercial

Microprocessor vendors Intel and Advanced Micro Devices (AMD) have each introduced features to prevent buffer overflow attacks. Intel calls their feature the Execute Disable Bit [75], which prohibits the processor from executing instructions that originate from certain areas of memory. AMD's No Execute (NX) Bit [76] is very similar to Intel's Execute Disable Bit. The NX bit is stored in the page table, and is checked on translation look-aside buffer (TLB) misses. Both Intel and AMD allow software to disable this functionality.

International Business Machines (IBM) has developed the SecureBlue architecture [77]. Like the academically-proposed techniques described above, it relies

on cryptography to ensure integrity and confidentiality of both software and data. SecureBlue is intended to be incorporated into existing microprocessor designs.

ARM markets the TrustZone security architecture [78], designed to augment ARM microprocessors. It relies on both hardware and software support. The hardware component uses cryptography to address integrity and confidentiality, allowing the processor to run in either a secure or non-secure mode. The software support includes the TrustZone Monitor, which augments the operating system and provides an application programming interface (API) for secure programs.

Maxim (formerly Dallas Semiconductor) manufactures the DS5250 secure microprocessor [79]. The DS5250 is designed to serve as a co-processor for embedded systems with traditional, non-secure microprocessors. Maxim proposes that the co-processor perform security-sensitive functions while the primary processor performs less sensitive operations. The DS5250 contains a non-volatile on-chip memory that is erased if physical tampering is detected. This memory is used to store the processor's secret key, and can also be used to securely store other sensitive data. The DS5250 can also access external memory, using cryptography to ensure the integrity and confidentiality of such accesses.

Secure Machines proposes an architecture to secure entire embedded computer systems, such as those contained in cellular telephones [80]. Their architecture targets the whole system, and ensures secure off-chip communications with peripherals. However, this security requires that all chips used in the system be a custom-made matched set sharing the same keys and containing security state machines called

hardware secure controllers. A secure kernel running on the microprocessor interacts with the hardware secure controllers, but details are not specified.

9.2 Multiprocessor Proposals

Researchers have also explored secure multiprocessor system designs. However, the added complexity of multiprocessor systems makes these designs difficult and costly to evaluate. We look at a few secure multiprocessor proposals in this section.

Shi *et al.* present a scheme for bus-snooping multiprocessor systems [81]. The basic architecture is sign-and-verify, like many of the above uniprocessor systems. They propose two security domains, with the boundary at the Northbridge memory controller. All incoming data (including executable code) is encrypted and signed using so-called vendor keys. The memory controller decrypts and verifies the data, and then re-encrypts and re-signs using system keys. All chips in the system must be matched sets, each containing cryptographic hardware and a set of secrets common across chips (including the system keys). Data is decrypted and verified when brought on a given chip. A sequential authentication buffer is used to allow speculative execution in parallel with data verification. The authors claim a performance overhead of 5% when running SPLASH2 benchmarks.

Zhang *et al.* also target bus-snooping multiprocessors [82]. They assume an existing method for securing external memory (such as one of the sign-and-verify systems described above) and focus on cache coherence messages. Processors in the system are divided into groups, each with a unique ID. Messages for each process are tagged with group and process IDs, requiring extra lines on the bus. All messages are encrypted using an OTP scheme and signatures are generated using CBC-MAC. A

global message counter is used to serialize the messages, and each processor keeps a circular buffer of precomputed pads for quick encryption and decryption. For a given message, one processor supplies the signature and all other processors recalculate the signature and verify the message individually. For maximum security, each message is verified. The chaining nature of the CBC scheme can be used to verify batches of messages at a time, increasing performance at the expense of decreased security. Their simulations predict that protecting messages in this manner adds an additional 2.03% overhead, above and beyond the overhead required for protecting external memory. Bus traffic also increases by 34%.

Lee *et al.* address the protection of cache coherence messages in distributed shared-memory systems [83]. Their goal is to provide security regardless of the interconnect system. They apply GCM cryptography, with a single authority assigning ranges of counters (initial vectors) to individual processors. When a processor receives a counter assignment, it precomputes the pads needed for GCM and stores them in a queue to accelerate the encryption of outgoing messages. The other processors precompute the same pads and cache them to accelerate the decryption of incoming messages. Recently used pads are also cached in case a block is received and then sent out again unmodified. The authors claim an average overhead of around 4% for protecting coherency messages. They admit a weakness in that control messages are not protected.

Patel *et al.* [84] propose to use a monitor processor to ensure the integrity of programs executing on a multiprocessor system-on-a-chip. The compiler maps out all possible execution paths for critical code, generating a constraint database of valid paths and minimum/maximum allowable execution times for each basic block. At runtime, one

processor is used as a monitor while others execute the programs. At the beginning and end of each secured basic block, its executing processor reports flow and execution time data to the monitor via a first-in-first-out queue. The monitor processor checks these data against the constraints database. If the flow is invalid, or if the basic block took too much or too little time to execute, the program has been compromised. This approach has several admitted weaknesses, including reliance on a static analysis of program code that may not accurately profile data-dependent execution paths. It does not address code confidentiality; neither does it protect data. Reported performance overheads range between 6.6% and 9.3%.

Rogers *et al.* from the GA Tech-NCSU team further extend their earlier design into the multiprocessor arena [85], addressing distributed shared-memory systems. In their design, each processor maintains its own tree for dynamic data protection. Sequence numbers and timestamps are communicated among processors in addition to blocks of data and their signatures. Coherence messages containing data blocks are also protected by an additional message signature. Like this team's earlier work, verification is non-precise, which may lead to security vulnerabilities.

9.3 Proposals Targeting Reconfigurable Logic

A few researchers have targeted the reconfigurable logic domain. Wang *et al.* [86] developed a cryptographic coprocessor on an FPGA to accelerate cryptographic functions in an embedded system. Zambreno *et al.* [87, 88] propose to use an FPGA as an intermediary, analyzing all instructions fetched by a processor. It calculates checksums for basic blocks using two different methods, such as a hash on the code and the list of registers used by instructions, and compares the two checksums at the end of

the basic block. The level of security provided by this approach is an open question, and requires extensive compiler support, including the insertion of dummy instructions, to establish the appropriate "register stream." This leads to a rather high overhead of around 20%, and only supports instruction integrity and confidentiality (by means of optional encryption).

Suh *et al.* [29] developed and implemented the AEGIS secure processor on an FPGA. They describe physical unclonable functions (PUFs) to generate the secrets needed by their architecture. Memory is divided into four regions based on whether it is static or dynamic (read-only or read-write) and whether it is only verified or is both verified and confidential. They allow programs to change security modes at runtime, starting with a standard unsecured mode, then going back and forth between a mode supporting only integrity verification and a mode supporting both integrity and confidentiality. They also allow the secure modes to be temporarily suspended for library calls. This flexibility comes at a price; their architecture assumes extensive operating system and compiler support.

CHAPTER 10

CONCLUSION

This dissertation has laid out the basic principles for implementing a secure processor, presenting a sign-and-verify architecture for protecting the integrity and confidentiality of software, static data, and dynamic data. We have also discussed many challenges that a computer architect will face when implementing a secure processor, and explored the various design options for meeting those challenges. We have also introduced enhancements to reduce performance latency relative to that caused by a naïve implementation of security extensions.

The performance overhead of our secure processor design has been evaluated using a cycle-accurate simulator. This simulator allowed us to examine the effects of the various design choices as if they were implemented in modern embedded processors and prove that security can be ensured without incurring excessive performance overhead. Utilizing our simulator, properly configured for the appropriate microprocessor architecture, would allow computer architects to make informed decisions when implementing our security enhancements in an actual processor design.

Furthermore, we have successfully demonstrated a prototypical implementation of our security enhancements using a soft-core embedded processor in actual hardware. Our implementation proves that our secure processor design concepts are sound, and may be

feasibly implemented in real systems. Security need not wait for future generations of microprocessors; it can be implemented at the hardware level using existing technologies.

The field of secure processor research is quickly maturing, as evidenced by the multiple academic proposals and industrial offerings discussed above. This means that future advances in secure processor designs will likely be incremental in nature. As cryptography evolves, secure processor designs should evolve along with it, embracing newer, more secure cryptographic standards while still adhering to the basic established principles of preserving integrity and confidentiality. Advances in hardware process and fabrication will also influence secure processor development; more transistors will allow more elaborate security hardware to be included on-chip. Chip designers must use those added transistors wisely to ensure that security extensions are not detrimental to system performance.

Given the breadth of the field of computer security, approaches above the hardware level must also be employed. However, we believe that secure processors will be an important part of overall solutions to computer security challenges. This dissertation has treated the subject of secure processors in detail, in hopes of contributing to making tomorrow's computer environments safer for all users.

APPENDIX

SIMULATOR AND PROTOTYPE IMPLEMENTATION SOURCE CODE

This dissertation includes an electronic appendix in the accompanying CD-ROM. The electronic appendix is comprised of two zip archives, containing the source code for the simsec-outorder simulator described in Section 7.2 and the EVU hardware implementation described in Chapter 8. This printed appendix describes the contents of those archives.

The zip archive `simsec-outorder_source_pack.zip` contains source code, documentation, and configuration files for the simsec-outorder simulator. The source code, which may be found in the archive's `src` directory, is a patch to the SimpleScalar/ARM suite. Therefore, SimpleScalar/ARM must be downloaded from the SimpleScalar website [89] and unpacked before applying the source patches contained in the simsec-outorder source pack. Instructions for applying the patch, compiling, and running simsec-outorder are included in a file called `README.txt`, which is at the top level of the archive. It also documents the configuration options for simsec-outorder and gives several example command lines for various configurations. The sample command lines reference the configuration files located in the archive's `conf` directory, which may

be used to simulate architectures based on ARM Cortex-M3 and Cortex-A8 cores with varying caches sizes.

The zip archive `EVU_source_pack.zip` contains source code and documentation for our EVU implementation. At the top level, the archive contains several files and a directory called `EVU`. The `EVU` directory contains the bulk of the source code for the EVU implementation, in very-high-speed integrated circuit hardware description language (VHDL) for the hardware component and C for the associated device driver. The source is released under the GNU Lesser General Public License, version 2.1, the text of which is included in a file at the top level of the archive. The file `README.txt` gives instructions on how to incorporate the EVU into an existing Quartus II project. There are also VHDL files for three EVU designs at the top level: an EVU using the CBC-MAC mode, an EVU using the CBC-MAC mode and calculating OTP pads in parallel, and an EVU using the PMAC mode and parallel pads. The desired EVU VHDL file should be copied into the `EVU` directory and there renamed to `EVU.vhdl`, per the instructions in `README.txt`. The top level of the archive also contains a C source code file, `EVU_demo.c`, which demonstrates how to use the EVU in an application.

REFERENCES

[1] NIST, "National Vulnerability Database," <http://nvd.nist.gov/home.cfm> (Available November, 2009).

[2] BSA-IDC, "2008 Piracy Study," <http://global.bsa.org/globalpiracy2008/studies/globalpiracy2008.pdf> (Available November, 2009).

[3] J. Turley, "The Two Percent Solution," <http://www.embedded.com/story/OEG20021217S0039> (Available December, 2007).

[4] M. Milenković, "Architectures for Run-Time Verification of Code Integrity," Ph.D. Dissertation, Electrical and Computer Engineering Department, University of Alabama in Huntsville, 2005.

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

[6] Anonymous, "Once upon a free()," <http://www.phrack.org/issues.html?issue=57&id=9#article> (Available August, 2007).

[7] R. Elbaz, D. Champagne, C. Gebotys, R. Lee, N. Potlapally, and L. Torres, "Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines," *Transactions on Computational Science IV*, vol. 5430, May 2009, pp. 1-22.

[8] N. R. Potlapally, A. Raghunathan, S. Ravi, N. K. Jha, and R. B. Lee, "Satisfiability-Based Framework for Enabling Side-Channel Attacks on Cryptographic Software," in *Proceedings of the 15th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT '97)*, Konstanz, Germany, 1997, pp. 37-51.

[9] P. Kocher, "Cryptanalysis of Diffie-Hellman, RSA, DSS, and Other Systems Using Timing Attacks," in *Proceedings of the 15th Annual International Cryptology Conference (CRYPTO '95)*, Santa Barbara, CA, USA, 1995, pp. 171-183.

[10] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Proceedings of the 19th Annual International Cryptology Conference (CRYPTO '99)*, Santa Barbara, CA, USA, 1999, pp. 388-397.

[11] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults," *Cryptology*, vol. 14, February 2001, pp. 101-119.

[12] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the Power of Simple Branch Prediction Analysis," in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, Singapore, 2007, pp. 312-320.

[13] M. Milenković, A. Milenković, and J. Kulick, "Microbenchmarks for Determining Branch Predictor Organization," *Software Practice & Experience*, vol. 34, April 2004, pp. 465-487.

[14] C. Percival, "Cache Missing for Fun and Profit," in *Proceedings of BSDCan 2005*, Ottawa, Canada, 2005, pp. 1-13.

[15] D. J. Bernstein, "Cache-timing attacks on AES," <http://cr.yp.to/antiforgery/cachetiming-20050414.pdf> (Available May, 2009).

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

[17] ISO/IEC, "Information technology - Security techniques - Message Authentication Codes (MACs)," ISO/IEC 9797-2:2002, June 2002.

[18] M. Bellare, J. Kilian, and P. Rogaway, "The Security of the Cipher Block Chaining Message Authentication Code," *Journal of Computer and System Sciences*, vol. 61, December 2000, pp. 362-399.

[19] J. Black and P. Rogaway, "A Block-Cipher Mode of Operation for Parallelizable Message Authentication," in *Proceedings of the 21st Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2002)*, Amsterdam, Netherlands, 2002, pp. 384-397.

[20] J. H. An, Y. Dodis, and T. Rabin, "On the Security of Joint Signature and Encryption," in *Proceedings of the 21st Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2002)*, Amsterdam, Netherlands, 2002, pp. 83-107.

[21] M. Bellare and C. Namprempre, "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm," in *Proceedings of the 19th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2000)*, Bruges, Belgium, 2000, pp. 531-545.

[22] D. A. McGrew and J. Viega, "The Galois/Counter Mode of Operation (GCM)," Cisco Systems and Secure Software, January 2004.

[23] Jetstream Media Technologies, "JetAES Fast: High Speed AES Core," <http://www.jetsmt.com/us4s/JetAES_4F_1675317.pdf> (Available December, 2009).

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

[25] A. Rogers, M. Milenković, and A. Milenković, "A Low Overhead Hardware Technique for Software Integrity and Confidentiality," in *Proceedings of the 25th International Conference on Computer Design (ICCD)*. Lake Tahoe, CA, USA, 2007, pp. 113-120.

[26] M. Milenković, A. Milenković, and E. Jovanov, "A Framework for Trusted Instruction Execution via Basic Block Signature Verification," in *Proceedings of the 42nd Annual ACM Southeast Conference*, Huntsville, AL, USA, 2004, pp. 191-196.

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

[28] Y. Wang, H. Zhang, Z. Shen, and K. Li, "Thermal Noise Random Number Generator Based on SHA-2 (512)," in *Proceedings of the 4th International Conference on Machine Learning and Cybernetics*, Guangzhou, China, 2005, pp. 3970-3974.

[29] G. E. Suh, W. O. D. Charles, S. Ishan, and D. Srinivas, "Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA-32)*, Madison, WI, USA, 2005, pp. 25-36.

[30] R. Merkle, "Protocols for Public Key Cryptography," in *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 1980, pp. 122-134.

[31] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA-9)*, Anaheim, CA, USA, 2003, pp. 295-306.

[32] A. Rogers, "Low Overhead Hardware Techniques for Software and Data Integrity and Confidentiality in Embedded Systems," Masters Thesis, Electrical and Computer Engineering Department, University of Alabama in Huntsville, 2007.

[33] B. Rogers, S. Chhabra, Y. Solihin, and M. Prvulovic, "Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40)* Chicago, IL, USA, 2007, pp. 183-196.

[34] J. Yang, L. Gao, and Y. Zhang, "Improving Memory Encryption Performance in Secure Processors," *IEEE Transactions on Computers*, vol. 54, May 2005, pp. 630-640.

[35] P. Rogaway, "PMAC - A Parallelizable MAC - Background - Rogaway," <http://www.cs.ucdavis.edu/~rogaway/ocb/pmac-bak.htm> (Available November, 2009).

[36] C. Parr, "Implementation Options for Finite Field Arithmetic for Elliptic Curve Cryptosystems," in *Proceedings of the 3rd Workshop on Elliptic Curve Cryptography (ECC '99)*, Waterloo, Canada, 1999.

[37] L. Song and K. Parhi, "Efficient Finite Field Serial/Parallel Multiplication," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, Chicago, IL, USA, 1996, pp. 72-82.

[38] G. Orlando and C. Paar, "A Super-Serial Galois Fields Multiplier for FPGAs and its Application to Public-Key Algorithms," in *Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, USA, 1999, pp. 232-239.

[39] G. Zhou, H. Michalik, and L. Hinsenkamp, "Improving Throughput of AES-GCM with Pipelined Karatsuba Multipliers on FPGAs," in *Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools, and Applications* Karlsruhe, Germany, 2009, pp. 193-203.

[40] W. Shi and H. Lee, "Accelerating Memory Decryption with Frequent Value Prediction," in *Proceedings of the ACM International Conference on Computing Frontiers*, Ischia, Italy, 2007, pp. 35-46.

[41] J. Platte and E. Naroska, "A Combined Hardware and Software Architecture for Secure Computing," in *Proceedings of the 2nd Conference on Computing Frontiers*, Ischia, Italy 2005, pp. 280-288.

[42] C. Yan, B. Rogers, D. Englender, Y. Solihin, and M. Prvulovic, "Improving Cost, Performance, and Security of Memory Encryption and Authentication," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA-33)*, Boston, MA, USA, 2006, pp. 179-190.

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

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

[45] ARM, "Cortex-M3 Technical Reference Manual," 2008.

[46] ARM, "Cortex-A8 Technical Reference Manual," 2008.

[47] 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 *Proceedings of the 4th Annual IEEE Workshop on Workload Characterization*, Austin, TX, USA, 2001, pp. 3-14.

[48] 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.

[49] 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.

[50] IP Cores, "GCM/AES MACsec (IEEE 802.1AE) and FC-SP Core Families," <http://www.ipcores.com/macsec_802.1ae_gcm_aes_ip_core.htm> (Available December, 2009).

[51] Jetstream Media Technologies, "JetGCM: High and Ultra High Speed AES-GCM Cores," <http://www.jetsmt.com/us4s/JetGCM_1_1576754.pdf> (Available December, 2009).

[52] V. Uzelac and A. Milenković, "A Real-Time Trace Compressor Utilizing Double Move-to-Front Method," in *Proceedings of the 46th Annual Conference on Design Automation (DAC)*, San Francisco, CA, USA, 2009, pp. 738-743.

[53] Altera, "Avalon Interface Specifications," <http://www.altera.com/literature/manual/mnl_avalon_spec.pdf> (Available January, 2009).

[54] H. Satyanarayana, "AES128," <http://www.opencores.org/projects.cgi/web/aes_crypto_core/> (Available August, 2008).

[55] Terasic Technologies, "Altera DE2-70 - Development and Education Board," <http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=39&No=226> (Available September, 2008).

[56] P. M. Embree and D. Danieli, *C++ Algorithms for Digital Signal Processing*: Prentice Hall PTR, 1999.

[57] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer, "Architecture Support for Defending Against Buffer Overflow Attacks," in *Proceedings of the Workshop on Evaluating and Architecting System Dependability (EASY-2)*, San Jose, CA, USA, 2002, pp. 50-56.

[58] 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," *IEEE Transactions on Computers*, vol. 55, October 2006, pp. 1271-1285.

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

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

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

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

[63] Z. Wang and R. B. Lee, "New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks," *ACM SIGARCH Computer Architecture News*, vol. 35, May 2007, pp. 494-505.

[64] J. A. Ambrose, R. G. Ragel, and S. Parameswaran, "RIJID: Random Code Injection to Mask Power Analysis based Side Channel Attacks," in *Proceedings of the 44th Annual Conference on Design Automation (DAC)*, San Diego, CA, USA, 2007, pp. 489-492.

[65] R. G. Ragel and S. Parameswaran, "IMPRES: Integrated Monitoring for Processor Reliability and Security," in *Proceedings of the 43rd Annual Conference on Design Automation (DAC)*, San Francisco, CA, USA, 2006, pp. 502-505.

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

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

[68] C. Lu, T. Zhang, W. Shi, and H. Lee, "M-TREE: A High Efficiency Security Architecture for Protecting Integrity and Privacy of Software," *Journal of Parallel and Distributed Computing*, vol. 66, September 2006, pp. 1116-1128.

[69] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, M. Bardouillet, and A. Martinez, "A Parallelized Way to Provide Data Encryption and Integrity Checking on a Processor-Memory Bus," in *Proceedings of the 43rd Annual Conference on Design Automation (DAC)*, San Francisco, CA, USA, 2006, pp. 506-509.

[70] R. Elbaz, D. Champagne, R. B. Lee, L. Torres, G. Sassatelli, and P. Guillemin, "TEC-Tree: A Low Cost, Parallelizable Tree for Efficient Defense against Memory Replay Attacks," in *Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2007)*, Vienna, Austria, 2007, pp. 289-302.

[71] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors," in *Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36)*, San Diego, CA, USA, 2003, pp. 339-350.

[72] M. Milenković, A. Milenković, and E. Jovanov, "Using Instruction Block Signatures to Counter Code Injection Attacks," *Computer Architecture News*, vol. 33, March 2005, pp. 108-117.

[73] M. Milenković, A. Milenković, and E. Jovanov, "Hardware Support for Code Integrity in Embedded Processors," in *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2005)*, San Francisco, CA, USA, 2005, pp. 55-65.

[74] A. Milenković, M. Milenković, and E. Jovanov, "An Efficient Runtime Instruction Block Verification for Secure Embedded Systems," *Journal of Embedded Computing*, vol. 4, January 2006, pp. 57-76.

[75] Intel, "Execute Disable Bit and Enterprise Security," <http://www.intel.com/technology/xdbit/index.htm> (Available December, 2009).

[76] A. Zeichick, "Security Ahoy! Flying the NX Flag on Windows and AMD64 To Stop Attacks," <http://developer.amd.com/articlex.jsp?id=143> (Available August, 2007).

[77] IBM, "IBM Extends Enhanced Data Security to Consumer Electronics Products," <http://www-03.ibm.com/press/us/en/pressrelease/19527.wss> (Available August, 2007).

[78] T. Alves and D. Felton, "TrustZone: Integrated Hardware and Software Security," *I.Q. Publication*, vol. 3, November 2004, pp. 18-24.

[79] MAXIM, "Increasing System Security by Using the DS5250 as a Secure Coprocessor," <http://www.maxim-ic.com/appnotes.cfm/appnote_number/3294> (Available August, 2007).

[80] G. Perrotey and P. Bressy, "Embedded Devices: Security Implementation," <http://www.secure-machines.com/fichiers/technical%20white%20Paper-Fev2006.pdf> (Available July, 2008).

[81] W. Shi, H. Lee, M. Ghosh, and C. Lu, "Architectural Support for High Speed Protection of Memory Integrity and Confidentiality in Multiprocessor Systems," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, Brasov, Romania, 2004, pp. 123-134.

[82] Y. Zhang, L. Gao, J. Yang, X. Zhang, and R. Gupta, "SENSS: Security Enhancement to Symmetric Shared Memory Multiprocessors," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, San Francisco, CA, USA, 2005, pp. 352-362.

[83] M. Lee, M. Ahn, and E. J. Kim, "I2SEMS: Interconnects-Independent Security Enhanced Shared Memory Multiprocessor Systems," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, Brasov, Romania, 2007, pp. 94-103.

[84] K. Patel, S. Parameswaran, and S. L. Shee, "Ensuring Secure Program Execution in Multiprocessor Embedded Systems: A Case Study," in *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, Salzburg, Austria, 2007, pp. 57-62.

[85] B. Rogers, C. Yan, S. Chhabra, M. Prvulovic, and Y. Solihin, "Single-Level Integrity and Confidentiality Protection for Distributed Shared Memory Multiprocessors," in *Proceedings of the 14th International Symposium on High Performance Computer Architecture (HPCA-14)*, Salt Lake City, UT, USA, 2008, pp. 161-172.

[86] C. Wang, J. Yeh, C. Huang, and C. Wu, "Scalable Security Processor Design and Its Implementation," in *Proceedings of the IEEE Asian Solid-State Circuits Conference (A-SSCC 2005)*, Hsinchu, Taiwan, 2005, pp. 513-516.

[87] J. Zambreno, A. Choudhary, R. Simha, B. Narahari, and N. Memon, "SAFE-OPS: An Approach to Embedded Software Security," *ACM Transactions on Embedded Computer Systems*, vol. 4, February 2005, pp. 189-210.

[88] J. Zambreno, D. Honbo, A. Choudhary, R. Simha, and B. Narahari, "High-Performance Software Protection Using Reconfigurable Architectures," *Proceedings of the IEEE*, vol. 94, February 2006, pp. 419-431.

[89] SimpleScalar LLC, "SimpleScalar Version 4.0 Test Releases," <http://www.simplescalar.com/v4test.html> (Available January, 2010).