Architectural Support for Protecting Memory Integrity and Confidentiality

A Thesis
Presented to
The Academic Faculty

by

Weidong Shi

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy Thesis

College of Computing
Georgia Institute of Technology
August 2006

Architectural Support for Protecting Memory Integrity and Confidentiality

Approved by:

Dr. Hsien-Hsin Sean Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Mustaque Ahamad
College of Computing
Georgia Institute of Technology

Dr. Doug Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Gabriel H. Loh
College of Computing
Georgia Institute of Technology

Dr. Sung Kyu Lim
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: April 19, 2006

To my parents and my wife.

ACKNOWLEDGEMENTS

I am taking this opportunity to thank individually, all those who have assisted me in one way or the other with my Ph.D Project.

First, I would like to thank my advisor, Dr. Hsien-Hsin Sean Lee, for the support and guidance I received from him. Without his supervision, this work would not have been a reality. In fact, without his willingness to adopt me after I had lost direction in academia, I would have quitted school. I would also like to thank Dr. Mustaque Ahamad, Dr. Doug Blough, Dr. Gabriel H. Loh, and Dr. Sung Kyu Lim for agreeing to be on my thesis committee and reviewing my thesis.

I would also like to extend my gratitude to all the people of the MARS lab for the help given during the various stages of my thesis including Fayez Mohamood, Dong Hyuk Woo, Richard Yoo, Chinnakrishnan Ballapuram. A very special thanks to Mrinmoy Ghosh for editing and commenting my papers multiple times, and Taeweon Suh for providing help on cryptologic synthesis.

I would like to thank Chenghuai Lu, Guofei Gu, and Tao Zhang for the help or suggestions given for shaping or re-shaping some of the ideas that finally lead to this thesis. Specifically, I would also like to thank Dr. Alexandra Boldyreva for her invaluable help and critical comments on many ideas presented in the thesis.

Especially, I thank Dr. Kenneth M. Mackenzie for introducing me to the research of computer architecture and converting me into a disciple of this wonderful world. Thanks are also due to all the professors who have helped or guided me during my time in Georgia Tech including Dr. Karsten Schwan, Dr. Santosh Pande, Dr. Richard M. Fujimoto, and Dr. Wenke Lee.

I also thank all the other professors and students I have done research with or co-authored with during the time of my Georgia Tech presence including Indrani Paul, Dr. Kalyan S. Perumalla, Dr. Josh Fryman, Jun Li, Weiyun Huang, Dr. Youtao Zhang, Dr.

Jun Yang, Dr. Trevor N. Mudge, Ashley Thomas, Laura Falk, Xiaotong Zhuang.

Most importantly, I would like to thank a very special person, Yang Lu, for all the family support and tolerance I received during the course of my education. Last, thanks for my German shepherd, Xena for all the fun, laugh, and enjoyment she brought when I was bored by writing the thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

This thesis describes efficient design of tamper-resistant secure processor and cryptographic memory protection model that will strength security of a computing system. The thesis proposes certain cryptographic and security features integrated into the general purpose processor and computing platform to protect confidentiality and integrity of digital content stored in a computing system's memory. System designers can take advantages of the availability of the proposed security model to build future security systems such as systems with strong anti-reverse engineering capability, digital content protection system, or trusted computing system with strong tamper-proof protection.

The thesis explores architecture level optimizations and design trade-offs for supporting high performance tamper-resistant memory model and micro-processor architecture. It expands the research of the previous studies on tamper-resistant processor design on several fronts. It offers some new architecture and design optimization techniques to further reduce the overhead of memory protection over the previous approaches documented in the literature. Those techniques include prediction based memory decryption and efficient memory integrity verification approaches. It compares different encryption modes applicable to memory protection and evaluates their pros and cons. In addition, the thesis tries to solve some of the security issues that have been largely ignored in the prior art. It presents a detailed investigation of how to integrate confidentiality protection and integrity protection into the out-of-order processor architecture both efficiently and securely. Furthermore, the thesis also expands the coverage of protection from single processor to multi-processor.

CHAPTER I

INTRODUCTION

Recently, there is a growing interest in creating trusted, tamper-resistant systems that combine the strength of advanced security hardware designs and secure operating systems to fight against both software and hardware tampers on computing systems [37, 61, 60, 73, 74, 38, 77, 75]. Such silicon based security systems aim at solving various problems in the security domain including anti-reverse engineer, virus/worm detection, system intrusion prevention, digital content protection, software digital rights protection, software privacy, etc. From the industry side, there have been a number of ongoing efforts that try to embed security features into both the general purpose and application specific processor chips to provide a safer computing environment to counter network intrusions, digital information theft, virus, and copy right violations. These include 1) the combined industry effort of trusted computing alliance called TCPA [4] and the related OS known as NGSCB [8] that try to implement trust management and authentication at silicon level; 2) the industry initiative to use security enhanced video processor to provide hardwired digital right protection on video content called secure video processing [2]; 3) the thrust to use specially designed digital right aware audio chip to protect copy right of music content [1]. Due to the diversity of threat model assumptions and security requirements, it is impossible to design a one-fit-all solution that will address all the security and digital right issues with one single solution. A typical approach is that depending on the security goal and threat model, hardware designers try to design a set of basic tamper proof security primitives or services implemented at silicon level that enable system software developers and content providers to design a proper solution that can meet the targeted security goal and effectively address the threats. A hardware security feature or primitive that is absolutely necessary under one threat model may not apply to other scenarios or applications that have different security goals and threat models.

Some critical primitives of future security systems are memory encryption and integrity protection that provide protection on program and data confidentiality and integrity. For many security systems, memory encryption often lies at the center of protection. Apart from hardware cryptography based tamper resistant systems [37, 61, 60] designed to prevent physical tamper of data, memory encryption can also be used for creating security systems where software based exploits are major concerns or constructing network security systems to handle code injection attacks. For example, an interesting application is to use memory encryption against remote code injection attack [15]. However, since system memory encryption and integrity protection are often components of a security system, we do not elaborate on how to create a specific trust computing or security system for a specific goal such as anti-reversing of military embedded systems using memory encryption in this thesis. Such systems can be developed on top of the memory protection model and secure processor architecture presented in this thesis.

In this thesis, we will focus on designing effective and efficient tamper proof memory protection primitives and secure processor architecture that will strength security of a computing system against a range of software and physical tamper at both system and platform level.

The thesis deals with computing systems and consumer computing platforms that may be subject to physical tamper. It assumes that the computing platform or system could be at the hand of hackers and the hackers may have the knowledge and skill set to launch relatively sophisticated physical tamper to compromise the sensitive information stored in the system memory. The concerned attacks include memory device spoof, bus signal interception aimed for information theft, physical memory eavesdrop, bus and device interface signal replay, etc. These attacks often aim at defeating or bypassing digital right protection or stealing sensitive data stored in the system memory. Counter measurements to those threats often require tamper proof or tamper resistant hardware features or primitives such as hardware facilitated memory encryption and integrity verification. However, The proposed protection model is not a panacea to solve all the security issues. Particulary, the thesis does not focus on attacks such as micro-probing directly on the die or other side-channel exploits e.g.

differential power analysis [70]. These attacks can be addressed by security countermeasures at the packaging level or circuits design level [70]. Such countermeasures are general to all the approaches that use silicon hardware to build a security system, thus orthogonal to the study of this work.

Bearing these research targets in mind, the thesis explores many security issues, design trade-offs, hardware design issues for implementing tamper resistant processor architecture and memory protection model at both the platform and micro-architecture level.

1.1 Thesis Organization

In chapter 2, we will discuss the objectives and challenges of designing a tamper-resistant memory model and computing platform. Furthermore, in the same chapter, we also describe some related work of using silicon based solution for creating secure computing system. In chapter 3, we describe several encryption modes applicable to memory protection and each one's pros and cons. Next in chapter 4 and chapter 5, we introduce two prediction based optimization techniques that significantly reduce the latency overhead of memory decryption. In chapter 6, we present a MACTree scheme for protecting integrity of system memory and preventing memory replay attacks. Then in chapter 7, we introduce and compare several approaches of integrating memory decryption and integrity verification into high performance out-of-order processor pipeline and the consequent security implications. Chapter 8 extends memory protection to SMP (symmetric multi-processor) shared memory and discusses the new memory protection challenges in SMP environment. Finally, chapter 9 concludes the thesis.

CHAPTER II

CHALLENGES: MEMORY SECURITY AND SECURE MICRO-PROCESSOR

Depending on the types of applications, their operating environment, business model or usage model, the requirements on content protection and the definition of security would be different. A system that is secure or a protection measure that is sufficient for one type of applications could bring security catastrophe when applied to a different type of applications or used in a different environment. For example, some of the biggest security concerns for enterprise computing may be accountability, viruses, access control, and network intrusion. Due to the nature of enterprise computing, security is far more likely been compromised by software based attacks or corrupted insiders than by hardware-based tampering such as sophisticated eavesdrop attacks involving a complicated logical analyzer. However, for consumer computing products, as indicated by the history, hardware-based tampering has caused wide spread security breaches of the game consoles including compromising security protection through user installed various types of hardware cheating or spoofing devices.

2.1 Applications

To give more details, we present four different application scenarios ranging from military embedded system, game consoles, to distributed computing and examine their respective security requirements.

First, high-tech military systems or advanced weapon systems are increasingly dependent on complicated computer software. One of the many security concerns on high-tech military application systems is that they may fall into the hand of people who are not so trusted. If unprotected, the system along with its software can be studied to come up counter measurement or counter system. Furthermore, un-trusted parties can reverse engineer and design copied version of the system. Both are security nightmares that should

be prevented regardless of the cost. Though self-destroy mechanism provides an alternative solution, it is far from being reliable and robust.

Second, software piracy has haunted software industry for decades. Many solutions have been proposed in the past to fight against software piracy. As indicated by the cases of XBOX security breach [28], achieving software protection on consumer platform is far more difficult than expected due to some specific attacks. One is software patching that rips off the security component from a software application. This attack often involves reverse-engineering and sometimes run-time de-assembling of the original software. Second, hardware modification or device spoof such as installing a BIOS spoof device or MOD-chip to break security protection. The third one is platform emulation. Illegal emulation violates software right and bypasses security protection by running illegitimate copies through a software machine emulator (for example playing a PSX game on a PC). To develop a working emulator, it often first requires reverse-engineer of a computing platform, which typically comprises reverse-engineer of the basic BIOS or system software.

Third, privacy and secrecy of mobile software agents and mobile data has been intensively studied recently [65]. In many cases, the mobile software to be protected is not a stand-alone process, but a piece of program, called mobile agent. How to execute a piece of mobile code on a host machine without potentially exposing or disclosing both the software and its data is a great challenge.

Fourth, internet based multi-player video gaming is growing rapidly. However, online multi-player gaming since the first day of its success has been mauled by rampant, sometimes wide spread "cheating"s [52]. Many of the cheating techniques involves reverse-engineer the client game software, modifying either the client code or data (so called authoritative clients) so that players using the hacked client will have advantages over others. The worst scenario is that the hacked clients or patches can be downloaded online, which clearly jeopardizes the entire business of online video game industry. How to prevent reverse engineer of the game client and protect against tampering on the client game code and data is vital for this emerging market.

Most of the discussed security requirements can not be met by today's computing platform. Although researchers have tried to tackle some of the security requirements through software based protection, the solutions are not tamper-proof and far from being satisfactory. They often leave security holes to the well-knowledged hackers who can break the protection through either hardware based physical tamper or software reverse engineering.

It is important to point out that the best practice of security hardware design is not to come up specific hardware features for each type of application, but to find out a set of basic security components behind the diversified security requirements. These basic security components can be implemented as trusted hardware security primitives. Various domain or application specific security requirements can be enforced through proper combination or usage of these trusted hardware security primitives.

2.2 Threat Model

Physical tamper is a great threat to digital content confidentiality or integrity if can be launched. It is in general not possible to protect a computing device from physical tamper using pure software based solutions. Often some security measures have to be imbedded at hardware level. Such solution is justified only when physical attack is a real threat to data confidentiality for a computing platform and its applications. Some examples are game consoles or military embedded systems. Computer hackers often have many techniques at their disposal to compromise secret information the designers try to hide in either software or hardware. If necessary, they can construct specialized hardware or even specially designed printed circuit boards or cracking/spoofing devices to recover protected secret information. Some typical techniques include front side bus trace analysis, memory trace analysis, hardware spoofing devices, and signal replay devices, etc.

- **Signal eavesdrop** An adversary can collect sensitive information through logging and analyzing software execution traces. Traditional protection model such as virtual memory or process memory space isolation would not defend against such exploit because the attack occurs directly on the physical buses. Front side buses, peripheral

buses, chip inter-connects, and external pins of integrated circuit chips are all potential targets of eavesdrop. Any information exposed by the external pins or transmitted over the buses are vulnerable. As to the feasibility of tracing physical bus traffic, it could be achieved by using an interposer board [30] and multi-channel high frequency logic analyzer. As shown by [28], a skillful computer hacker may build custom tracing or eavesdrop devices using cheap commodity components. Signal eavesdrop also assists the adversary to launch more sophisticated surgical kind of attacks where the adversary tries to alter or remove either a software or hardware security component with certain hypothesis about the underlying security system or secret and observe the consequences to prove the hypothesis.

- **Device spoof.** Apart from signal probe and eavesdrop, an adversary can directly alter a printed circuit board, replace an onboard device with some spoof device in order to bypass or subvert security measures. The famous MOD-chip attack on the game consoles belongs to such category. MOD-chip unravels security protection implemented in BIOS boot software through spoofed BIOS device and signal hijacking.

- **Signal Replay.** Furthermore, an adversary can launch physical signal replay attacks to defeat security protections. This attack is often used in combination with the signal eavesdrop and device spoof attacks where an eavesdrop device first intercepts important signals and later replay the logged signals through a spoof device. For example, lots of consumer hackers use this kind of attack to play copied games. First, the consumer hacker will install some eavesdrop and spoof device. Then the hacker will run a legitimate version of some game and eavesdrop some important security code and stores the information in the spoof device. Next the hacker will make a copy of the original software and load it. Assume that there are some copy protection associated with the software to prevent the security code from being copied. During loading, the spoof device will replay the security code to give a false image that the system is reading the original released software.

Throughout this thesis, it is assumed that external physical buses, chip-interconnects, and external pins are unprotected and subject to potential malicious tamper. The physical RAM itself is unprotected. An adversary could eavesdrop or overwrite the memory content directly without involving the processors. Note that the thesis does not try to solve all the physical attacks, especially those attacks outside system or platform protection level such as compromising a security key through timing-analysis [33], power analysis [46] that uses the power profile to infer security key bits, electromagnetic analysis that recovers sensitive data through electromagnetic radiation analysis [66], micro-probing or screening that requires de-packaging and extremely expensive devices. Most of those attacks can be mitigated using secure practice of circuit design and packaging [32], which is orthogonal to the study of this thesis. The platform and system level security measures can be combined directly with those circuit and packaging level security practices to deliver system with strong security strengths.

An effective way to fight against the aforementioned eavesdrop and spoof attacks at platform level is memory encryption. Hardware cryptography based tamper resistant systems have been proposed to address data confidentiality protection under this threat model [60, 37, 61, 73, 74]. Though memory encryption is a promising direction to counter many physical attacks at platform level, it is still unclear how memory encryption can be designed efficiently. The tradeoff between security and performance is not well understood. As shown in our study [59], ill-conceived designs of memory encryption can have security holes exposed to the hackers and end up with very weak or in the worst case no security protection.

2.3 Related Work

2.3.1 Smart Cart and Security SOC

Smart card chip is a security chip based on system-on-chip design that integrates a number of security funcitonalities, limited storage, and micro-processor into one chip. Smart card chip is a stand-alone tamper-proof device. It has been used successfully in a number of applications including GSM mobile telephones, DirectTV, EchoStar satellite receivers, and

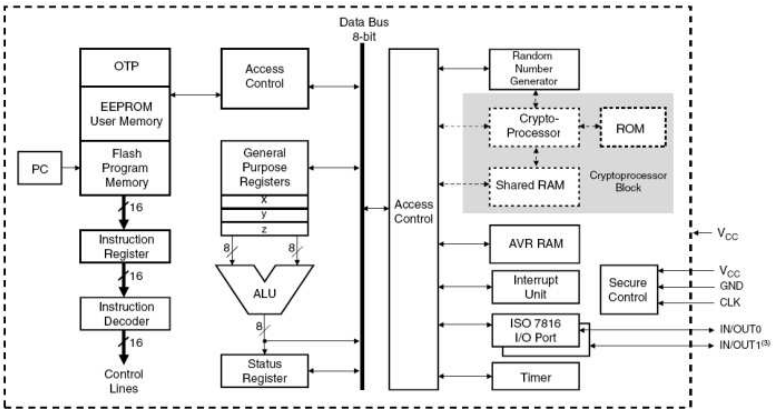


Figure 1: Smart Cart Security Chip

the American Expres Blue card. Figure 1 shows a block diagram of a typical smart card chip. Smart card chip achieves tamper-proof protection by integrating all the units of a platform into a single chip - system on chip. It is very difficult for physical attacks that rely on probing of chip inter-connect and external interface to break smart card security. Other types of exotic side-channel attacks such as power analysis, electromagnetic analysis are also preventable with a careful chip design practice when kept these attacks in mind.

Smart card attains tamper-proof by providing a SoC single chip solution that integrates simple micro-processing element, memory, crypto engines into the same chip. This way of building secure and trusted system does not apply to the scenario where a complex platform has to be constructed using discrete components. For a complete high end platform, to apply the same smart card concept, it would be to put everything including Gig bytes of RAM, processor, north bridge, etc into one single chip, which is impossible under today's technology.

2.3.2 TCPA - Trusted Computing Platform Alliance

Software protection and trusted computing are among the most important issues in the area of computer security. Traditionally, the protections on software are provided through a trusted OS. The OS implements certain mechanisms to ensure that the applications are protected and the information spaces from different applications are isolated. Consequently, the software protection can be achieved since malicious applications will not be able to

access others' data without the permission of the OS. The trusted computing is ensured as well since an application is protected from tampering by others using the underlying OS. To improve the security model, some tamper resistant device is embedded into computer architecture to ensure the loaded operating system is trusted. From there, a chain of trusted applications are executed, each depending on the underlying layer. A typical example of such computer architecture is the TCPA [4] and the related OS known as NGSCB [8]. TCPA relies on a tamper-proof device called TPM (trusted platform module) for providing three main security functionalities.

- Protected storage and protected capabilities. TPM is a stand-alone smart card alike chip. It provides secure internal persistent storage that can be used to seal sensitive data such as symmetric keys for encrypted file, hashes of configuration information and etc. TCPA stores sensitive data into shielded location and those data can be accessed with protected capabilities.

- TCPA provides the necessary means for remote attestation. The TPM chip can report the data it stores by signing the data with a unique key that is generated and certified by the TPM chip. Each TPM chip has a unique public private key pair. An external remote party can verify the data signed by a TPM chip using public private key scheme.

- Integrity auditing. TPM can gather important configuration information of a platform, store the data, compute an integrity hash and report the result to an external third party upon request.

TCPA provides authentication and data signing services. It facilitates the so called trusted boot functionality by using the TPM chip to verify the integrity of the BIOS and bootstrap code. It can be used for digital content protection and digital right management given the assumption that the end users would not launch a physical tamper on the platform, which might be true for enterprise applications and environment. Though the TPM chip itself is tamper-proof, the TCPA platform as a whole is not. TCPA is not designed

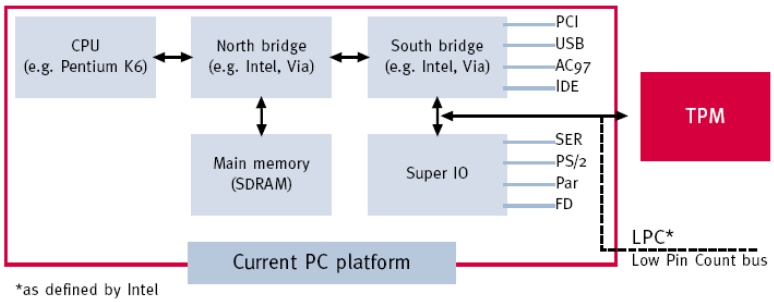


Figure 2: TPM and TCPA Platform

to protect digital content confidentiality against physical tamper. When physical tamper based information theft is a concern, TCPA would not provide sufficient protection because the entire platform outside the TPM is vulnerable. As mentioned above, TCPA and the TPM chip are designed to provide a tamper-proof signing services, which is infrequent in a software's execution life span. They are not meant to protect the entire software's memory space and all the intermediate software execution results. TPM as a specialized remote peripheral device sitting on the south bridge has neither the access nor the computing power to provide comprehensive protection to the software running in the CPU with the code and data stored in the system memory.

For complete protection, the TPM engine has to be integrated into the CPU and acts as a security agent between the henceforth secured CPU and the un-trusted memory. The result would be a tamper resistant processor, we will discuss in detail in the next subsection.

2.3.3 Tamper-Proof Platform And Micro-architecture

Integrating crypto services directly into the micro-processor to build tamper-resistant system is an effective way to fight against physical tamper on software and data confidentiality [60, 37, 61, 73, 74]. Such tamper-proof system uses memory encryption to provide a secure environment where software can be executed in such a way that it is almost impossible to be duplicated (copy-protection), altered (integrity), or reverse engineered (software confidentiality). Data and software confidentiality is often protected through encryption of both the executable image and the associated data. Integrity of software is often ensured through layered integrity verification such as an authentication tree [60, 61].

A typical secure processor architecture model comprises of a tamper-resistant processor, external memory and peripherals. Naturally, the protection boundary is drawn between the processor and the external hardware units. Hardware units, like registers and on-chip caches, are protected from any possible attack while the remaining hardware units such as the external memory and peripherals are considered vulnerable to the physical attacks. Besides the aforementioned hardware, the secure computing model also includes a small trusted program, e.g., XVMM in *XOM* [37] and secure kernel in *AEGIS* [61]. The trusted program will be called secure kernel hereafter. The secure kernel has a higher privilege level than any other program including the regular operating systems and is responsible for performing encryptions/decryptions for the protected applications when their data are crossing the protection boundary. The secure kernel is also responsible for maintaining sensitive data that resides in private memory and registers during context switches. Note that confidentiality and integrity of "process context" are protected by the secure processor. Here process context refers to all per-process information that need confidentiality or integrity protection including but not limited to register values, page table, dirty cache lines, hash tree nodes, etc.

Most of the proposed systems support separate protection on software confidentiality and integrity. In XOM [37], a per-process encryption key (triple-DES) is used to decrypt software on-the-fly, while AEGIS [61] uses AES [20]. One major difference between AEGIS and XOM is that AEGIS employs an on-chip hash tree to verify the integrity of entire process space also in the execution time, thus preventing memory replay attack. As studies in [73] indicate, block cipher based systems can incur substantial performance penalty. Systems using encryption schemes similar to one-time pad (OTP) with relaxed integrity check [60, 73] are proposed because they support faster software execution. Alternative solutions also aimed for better performance such as encrypting only small amount of carefully selected instructions, called software slices, can also be found in the literature [74].

The objective of this thesis is to expand the research of the previous studies on tamper-resistant processor design on several fronts. The thesis tries to solve some of the security issues that have been largely ignored in the prior art. Furthermore, it offers new architecture

optimization techniques to further reduce the overhead of the previous approaches and improve the efficiency of tamper resistant processor. Finally, it also expands the coverage of protection from the single processor to multi-processor.

CHAPTER III

BASIC ENCRYPTION MODES AND MAC

For protecting randomly accessed memory, several standard encryption modes are viable and some of them have been studied for secure processor design. One is the simple traditional *electronic code book* mode, also called the ECB mode, another one is the CBC mode *cipher-block chaining*, a third one is *counter* mode [17]. Applying these modes involves splitting memory space into equal size blocks (often the same size as L2 cache line) and encrypting each memory block separately to support random access of encrypted memory. The CBC mode based secure processor design can be found in some earlier systems such as [61] and *counter-mode* based systems are discussed in some recent publications [73, 60]. Note that all the applications of encryption modes to secure processor design are confined to the granularity of a memory block instead of the entire memory space.

From the security point of view, both the CBC mode and the *counter mode* are secure to provide software confidentiality. However, many encryption modes such as the CBC mode, *counter-mode* and stream cipher based modes allow an adversary to alter protected data/code to any value at their choice under known plaintext attack by flipping individual bits of the encrypted information, called malleability. One specific attack that exploits this weakness is presented in [58]. A stream cipher is a cipher in which the input data are encrypted one bit (sometimes one byte) at a time. Most stream ciphers consist of a pseudorandom number generator (PRNG) and a XOR gate. The PRNG is initialized with a key, and outputs a sequence of bits known as a keystream. Encryption consists of XORing the plaintext bits with the corresponding bits of the keystream; decryption consists of XORing the ciphertext bits with the corresponding keystream bits. The conclusion is that memory encryption based on malleable encryption modes should be used with extra rigorous integrity protection to avoid disclose of the protected information.

One major overhead of hardware cryptography based protection on software confidentiality is increased memory latency because memory blocks fetched from memory have to be decrypted first before they can be used. For direct memory encryption, the encrypted memory block (also called ciphertext), has to be fetched before a long decryption process can start. The increased overall memory latency can cause substantial loss of performance. The critical path of memory decryption consists of standard block cipher, such as AES (advanced encryption standard) [20] or Triple-DES (data encryption standard) [49]. Real hardware implementations of block ciphers show that it is often relatively easy to improve the throughput of block ciphers through pipelining [19, 26] but the latency of block ciphers is more difficult to reduce.

In this thesis, we propose architectural techniques and optimizations to reduce memory decryption overhead. One is called *one-time-pad prediction* or *sequence number(time-stamp, counter) prediction* or *encryption pad prediction* that combines prediction technique and hardware cryptography to address the latency issue of *counter-mode* encrypted memory. The basic idea is that a secure-processor predictable counter(or sequence number, time-stamp) is used to generate a secure processor predictable encryption pad (or OTP) deterministically for memory decryption before the sequence number (time-stamp, or counter) is loaded from memory. Applying the same idea to other encryption modes to reduce the memory decryption latency overhead is a great challenge because unlike the *counter-mode* that inherently supports pre-computing of decryption pads, direct memory encryption modes such as the ECB mode or the CBC mode use strait-forward sequential invokes of block ciphers. In this thesis, we also propose a unique *ciphertext prediction* technique that significantly hide decryption latency of directly encrypted software/data. The technique is based on "value" prediction and speculative **encryption** of predicted data values to hide decryption latency. Those optimization techniques will be described in details in the next two chapters. Before that, we will discuss in this chapter different encryption modes and their application to memory protection first.

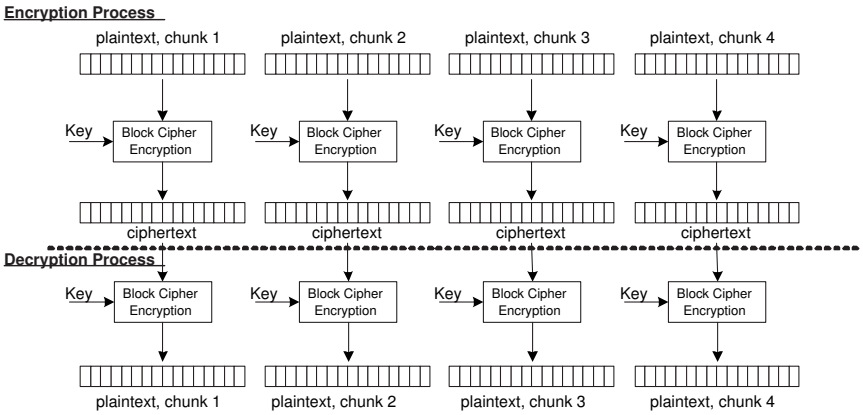


Figure 3: Standard ECB mode (electronic code block)

3.1 Encryption Mode

There are several standard encryption modes that are candidates for memory encryption, for example, the ECB mode, the CBC mode, the OCB mode, and the counter mode mode. Different encryption modes not only have different security strength against specific attacks but also have dramatic differences in terms of speed, efficiency, area cost and performance under hardware implementation.

There are two desirable security features of memory encryption,

First, it should not be feasible for an adversary to guess plaintext of some memory block from its ciphtertext given that the adversary knows plaintext and ciphtertext of some other memory block. In anther word, identical plaintexts stored in different memory blocks do not produce identical ciphertexts.

Second, it should not be feasible for an adversary to guess plaintext of some memory block given that the adversary can choose plaintext of some other memory block. Almost all program applications today require some user inputs/data or command line inputs/data or other sources of data that an adversary can manipulate. This security feature basically requires that given the possibility that an adversary can control some data sources freely, the adversary should not be able to guess secret information in other memory locations through manipulating these data sources and compare the result ciphertexts.

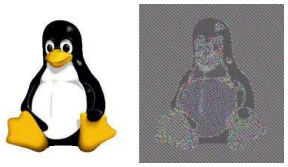


Figure 4: Patterns in ECB Encryption, Courtesy of Wikipedia.com

3.1.1 The Electronic Code Block (ECB) Mode

The ECB (electronic code block) mode is a standard encryption/decryption mode of using block ciphers. Assume that each cache line size memory block is split into four chunks (64-byte block into 4 16-byte chunks or 32-byte block into 4 8-byte chunks). Figure 3 shows how each four chunks of plaintext are encrypted under the ECB mode using block cipher and how decryption is carried out. Candidates of block cipher are Tripe-DES [49], AES [20] and other encryption standards. One disadvantage of the ECB mode is that identical memory blocks are encrypted to identical ciphertext blocks. This violates the first security requirement set in the previous subsection. This means that the ECB mode by default provides only weak protection on data patterns. One example to demonstrate the degree to which the ECB can reveal patterns of data is shown in figure 4. The right image is created from the left image using the ECB mode encryption. One way to fix this problem is to concatenate every memory block with an RV (random bit vector) including its virtual address and encrypt the result bit string. One major disadvantage of this approach is the memory overhead. Assume that the RV is 64-bit (32-bit virtual address plus another 32-bit random vector [61]), For 128-bit cipher, the overhead would be at least another 50%. In a less secure setting, assume that the RV is 32-bit including only the virtual address, for 64-bit cipher, the additional overhead would be at least 50% and for 128-bit cipher, the overhead would be at least another 25%.

3.1.2 The Cipher Block Chaining (CBC) Mode

Assume that each cache line size memory block is broke into four chunks (64-byte block into 4 16-byte chunks or 32-byte block into 4 8-byte chunks). Figure 5 shows how each four chunks of plaintext are encrypted under the CBC mode using block cipher and how

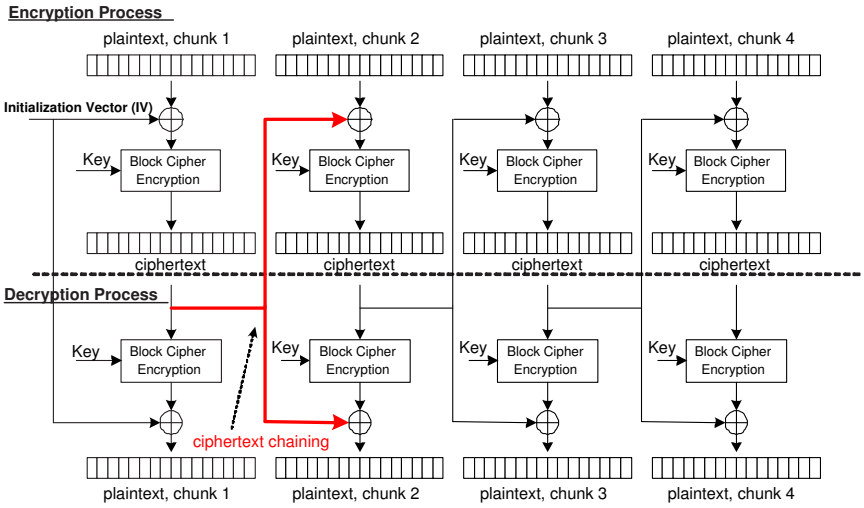
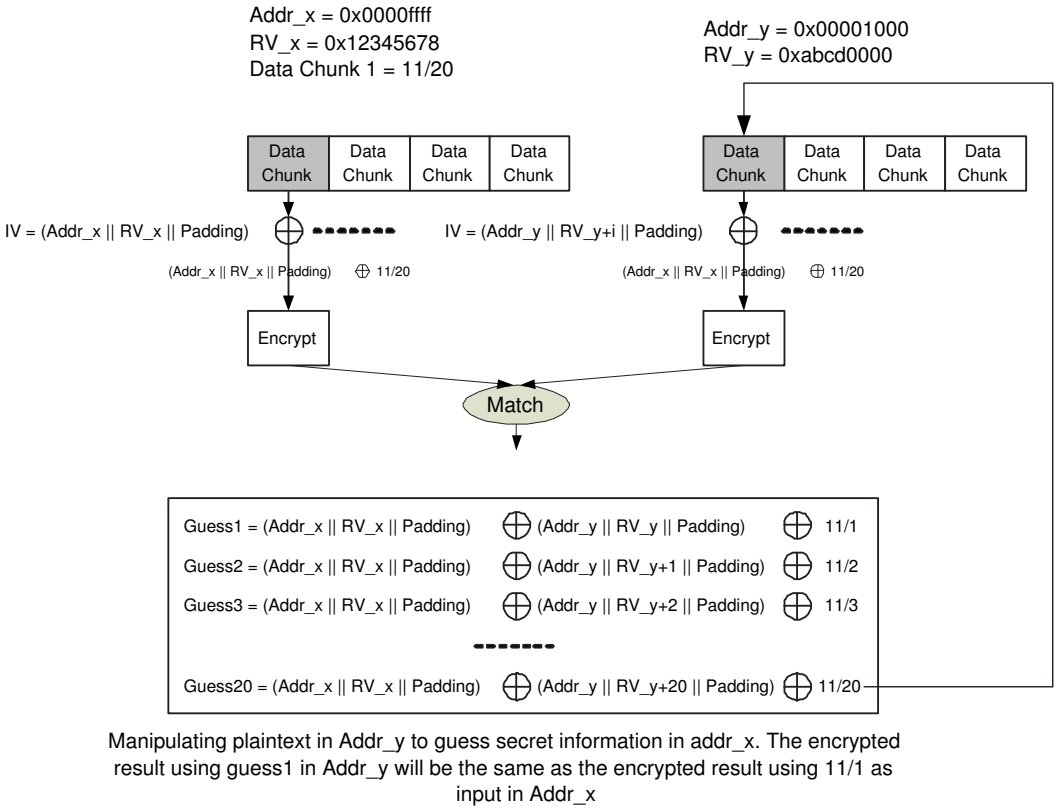


Figure 5: Standard CBC mode (cipher block chaining) example



Manipulating plaintext in Addr_y to guess secret information in addr_x. The encrypted result using guess1 in Addr_y will be the same as the encrypted result using 11/1 as input in Addr_x

Figure 6: Example showing that second property is not satisfied by CBC Mode

decryption is carried out. In the CBC mode, each chunk of a block of plaintext is XORed with the previous ciphertext chunk before being encrypted as shown in figure 5. the CBC mode supports encryption of any number of memory chunks. But to facilitate random memory access, the CBC mode is preferred to be carried out on each cache line size memory block (4 or 8 chunks). Candidates of block cipher are Tripe-DES [49], AES [20] or other encryption standards. The CBC mode based memory protection can be found in [61] with AES as block cipher. To prevent disclosure of memory patterns, the same data value when stored to different memory addresses must produce different ciphertext. This is achieved in [61] by using an IV (initial vector) specific to each memory block. An IV consists of memory block address, a random bit vector (RV) and bit padding. However, it is worth pointing out that the CBC mode in [61] does not satisfy the second desired security requirement. Because the address is XORed with plaintext of a memory block, an adversary is able to guess information stored in a memory block in certain situations if the adversary can manipulate plaintext data in some other memory blocks. One example is given in figure 6. Assume that there is a piece of secret information stored in the first chunk of data in memory block x. Further assume that the piece of data belongs to a finite set, for example, date when a military action will be launched. Given the condition that the adversary is able to manipulate plaintext in some other memory block, say block y, the adversary can use the plaintexts shown at the bottom as guesses and let the secure processor to encrypt the guesses. If the adversary can find one of the encrypted guesses in memory block y having the same value as the ciphertext in memory block x, then the adversary can conclude that the date used to generate the guess is the same as the date in memory block x (note that RV is incremented after each memory update as suggested in [61]).

The security of the CBC mode against the adversary who has access to chosen plaintexts is provided in [9][1]. It is proved that if the underlying block cipher is secure, the CBC mode offers secure encryption in the sense that the ciphertext is random. However, the

[1] A chosen plaintext attack is any form of cryptanalysis which assumes that an adversary has the capability to choose arbitrary plaintexts to be encrypted and obtain the corresponding ciphertexts.

CBC mode is vulnerable to an attack called, *matching ciphertext attack* [41]. It is recommended that the CBC mode should not be used alone on block ciphers whose block length is 64-bit. This motivated the research to combine or interleave standard modes, where the modes themselves are considered as primitives [11]. However, mode interleaving and combination substantially increases the latency of encryption/decryption. The CBC mode is less vulnerable to the *matching ciphertext attack* for block cipher whose block length is 128 bit. But, the CBC mode has another weakness. It is vulnerable to *chose ciphertext attack* [2]. In another word, the CBC mode is chosen-ciphertext malleable. A malleable mode means that it allows one to flip bits in the ciphertext and the flipped bits can induce flipped corresponding bits in the plaintext. Non-melleable is always a desired security property.

From performance perspective, the CBC mode has a major disadvantage. It requires sequential invoke of block cipher through a dependency chain. This greatly increases the latency overhead and severely limits its efficiency for hardware facilitated memory protection. Note that though some security weakness of the CBC mode may be fixed with additional protection on integrity, it still has the performance disadvantage.

3.1.3 The Offset Code Block (OCB) Mode

The OCB (*offset codeblock*) mode was introduced in [54]. Its strong achieved security level is proved in [53]. Here we only give brief description on how the OCB mode works and its security properties and advantage in performance. Interested readers can refer to [53] for details.

From the security perspective, the OCB mode provides higher security protection than the most standard modes including the ECB mode and the CBC mode. The OCB mode is non-malleable under *chosen-ciphertext attack*, which cannot be achieved by the CBC mode. The OCB mode also belongs to the set of *authenticated encryption modes* that support decryption and authentication at the same time. Figure 7 illustrates how the OCB mode encrypts and decrypts a memory block (4 chunks) and how integrity code is computed. The nonce is a bit string similar to the IV (initial vector) in the CBC mode. For memory block

[2] A chosen ciphertext attack is an attack on a cryptosystem in which the cryptanalyst chooses ciphertext and causes it to be decrypted

encryption, the nonce can consist of virtual address of cache line, concatenated with some 64-bit RV (random vector) and padding.

The OCB mode based memory encryption defined in this way satisfies both the security requirements listed at the beginning of this chapter. Since virtual address concatenated with the RV is used as nonce, the weakness of having detectable ciphertext patterns is avoided. This prevents an adversary from guessing the encrypted data value based on the encrypted values from other memory blocks, therefore satisfying the first security requirement. For example, a large number of memory blocks may use zero as its data value. Under the OCB mode based design using virtual address as nonce, zero will have different ciphertext for different memory address. Because R is address dependent (obtained from the encrypted IV that incudes cache line virtual address) and un-predictable (produced by a block cipher), the attack described in figure 6 that violates the second desired security requirement does not hold for the proposed OCB based design neither. Similar to [61], the nonce/RV associated with dynamic data blocks is incremented each time the corresponding dirty data block is evicted from the on-chip cache [3]. Also note that the L in figure 7 is a secret pseudo-random bit string computed by encrypting a constant [54, 53].

From the performance side, the OCB mode achieves high level parallelism for decrypting/encrypting data chunks. Because decryption/encryption of each data chunk is fully independent of other data chunks, the OCB mode is fully paralleziable and suitable for high performance hardware implementation. Fully parallelizable means that each chunk of data can be encrypted or decrypted at the same time.

Detailed explanation of each step of the OCB mode operation such as 2L and the + operation requires some background knowledge in number theory and cryptography and interested readers can refer to the original OCB paper for details. The OCB mode is a better choice for direct memory block encryption not only because it is more secure but also because it can deliver better performance. It is worth pointing out that the nonce or IV in both the the CBC and the OCB mode "needs **not** to be random, unpredictable, or

[3] note hundreds of years are needed for 64-bit RV to wraparound under a few hundred MHz or 1 GHz clock rate

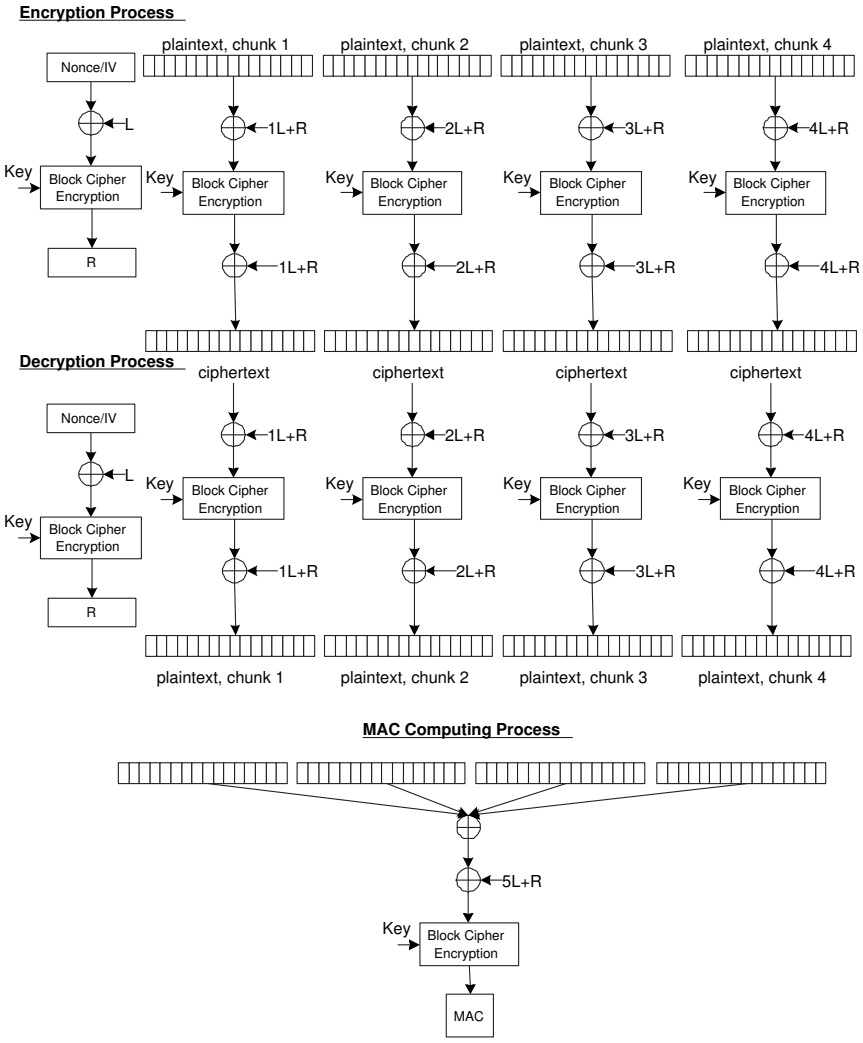
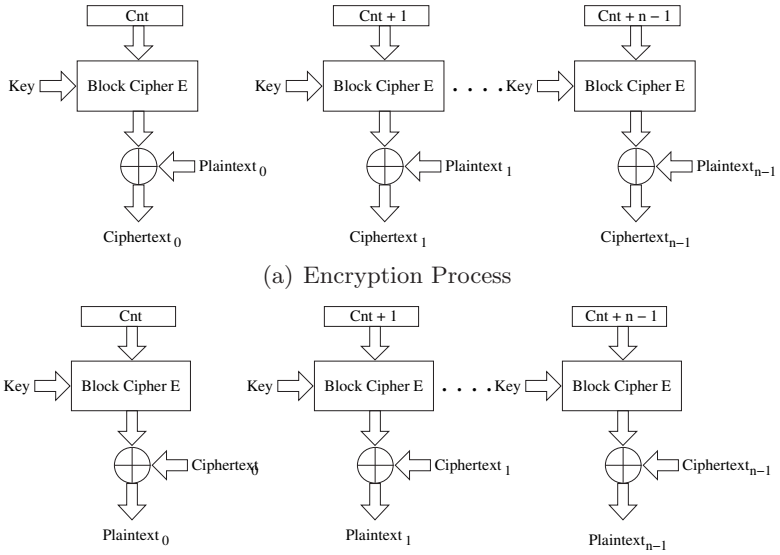


Figure 7: OCB Based Protection

secret" [54].

3.1.4 Counter Mode

Counter mode encryption is a common symmetric-key encryption scheme [17]. It uses a block cipher (e.g. AES [20]), a keyed invertible transform that can be applied to short fixed-length bit strings. To encrypt with the counter mode, one starts with a plaintext *P*, a counter *cnt*, a block cipher E, and a key. An encryption bitstream (OTP) of the form E(key, cnt) || E(key, cnt+1) || E(key, cnt+2) ...|| E(key, cnt+n-1) is generated as shown in figure 8. This bitstream is XORed with the plaintext bit string P, producing the encrypted string ciphertext C. To decrypt, the receiver computes the same pad used



(a) Encryption Process

(b) Decryption Process

Figure 8: Counter Mode Encryption

by the sender based on the same counter and key, XORs the pad with C, then restores the plaintext P. P is padded, if necessary, to facilitate the OTP length. Counter mode is known to be secure against chosen-plaintext attacks, meaning the ciphertexts hide all partial information about the plaintexts, even if some a priori information about the plaintext is known. This has been formally proved in [9]. Security holds under the assumptions that the underlying block cipher is a pseudo-random function family (this is conjectured to be true for AES) and that a new unique counter value is used at every step. Thus a sequence number, a time stamp or a random number can be used as an initial counter. Note that the counter does not have to be encrypted. As most encryption modes, counter mode is malleable and thus is not secure against chosen-ciphertext attacks. For example, flipping one bit in a ciphertext results in the flipped bit in the plaintext. Also, counter mode does not provide authentication (integrity) of the data. For these reasons an additional measure such as message authentication code (MAC) should be used. If a secure MAC is applied to the counter mode ciphertext during the encryption and is verified during the decryption process, then the resulting scheme provides authentication and integrity, is non-malleable and is secure against the chosen-ciphertext attacks. This is formally proved in [10]. As pointed out by [40], most other perceived disadvantages of counter mode are invalid, and

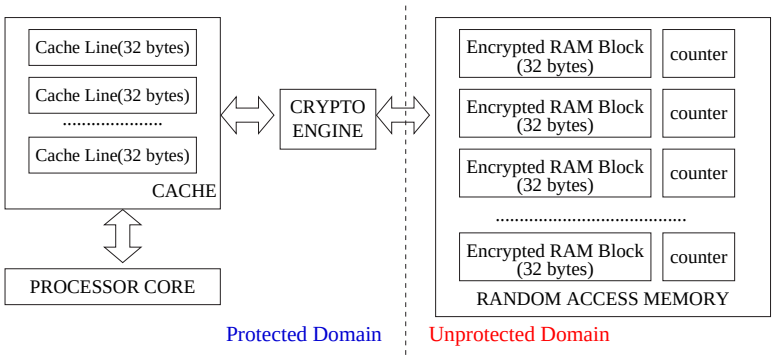


Figure 9: Concept of Counter Mode Security

are caused by lack of knowledge.

Memory encryption schemes based on the counter mode were employed for its high efficiency by some prior proposed security architectures [60, 73]. In these schemes, a counter[4] is associated with each cache line size memory block of the physical RAM as shown in figure 9. Whenever a cache line is evicted from the secure processor, the corresponding counter is incremented and the result is forwarded to the crypto-engine to generate the OTP for the evicted cache line. Meanwhile, the corresponding counter value in the memory is updated. When an encrypted memory block is fetched, the corresponding counter must be fetched first from the memory to regenerate the OTP for decryption. As we mentioned earlier the counters do not need to be encrypted because the security strength of counter mode does not rely on their secrecy [9]. Unlike other direct memory encryption schemes that serialize cache line fetching and decryption process, the potential advantage offered by counter mode is that it overlaps the OTP generation with fetching of the encrypted program and data [60, 73]. However, to compute the OTP for a fetched cache line, the counter needs to be fetched from the memory first, eliminating much of the advantage of OTP pre-computation.

3.2 Comparison of Different Memory Decryption Speedup Techniques

In addition to choosing encryption modes that achieve the best balance in terms of performance, area, and security, computer architecture researchers also tried to experiment with

[4]In many encryption mode descriptions the *counter* is sometimes referred to as *sequence number*, *nonce*, *initial vector*. We will use *sequence number* or *counter* interchangeably throughout this thesis.

different architectural level design optimizations to help reduce the extra memory latency caused by memory decryption. We will briefly discuss two of these techniques, sequence number caching and memory pre-decryption.

3.2.1 Sequence number caching

Caching sequence numbers or time stamps required for decryption was proposed in [60, 73]. As shown in [60, 73], sequence number caching can effectively reduce decryption latency overhead of counter mode based memory encryption schemes. However, sequence number caching requires significant chip area overhead as the size of sequence number cache grows to hundred of KB for achieving high sequence number cache hit rate. Ideally, for every virtual memory page present in the TLB, if all of the page's sequence numbers are cached, the system would have 100% sequence number cache hit rate and have the smallest decryption latency overhead. But such kind of system will have very high cost in area.

3.2.2 Memory Prefetch and Pre-decryption

Prefetch was intensively studied for hiding memory latency [7, 14, 67, 68]. When memory is encrypted, a prefetched memory block can be pre-decrypted. There are many challenges of using prefetch to reduce latency overhead of direct encrypted memory. First, memory prefetch/ pre-decryption still conducts memory read and decryption operations sequentially. The combined latency of memory access and decryption can be twice as much as a simple memory read. To hide such long latency, prefetch requests need to be issued long enough prior to the usage of the prefetched/pre-decrypted data. This can be a challenge for many prefetch techniques and the accuracy of prefetch may decrease if it has be issued much earlier. Second, prefetch/pre-decryption can cause cache pollution if the pre-decrypted data are stored in the regular caches. Third, prefetch/pre-decryption can increase workload on the front side bus and memory controller if there are too many unnecessary memory prefetch requests issued.

3.3 Integrity Verification

Integrity verification is a critical component of secure processor design. Integrity verification, achieved by employing message authentication codes (MAC) [43], guarantees the detection of any unauthorized data modification. The MACs are stored along with each encrypted memory block such as a cache line. Similar to the case of encryption, there are many approaches and standards for generating a MAC, for example, HMAC [34], CBC-MAC [12], to name a few. For each dirty writeback, the plaintext of the dirty cache line must be re-encrypted and stored with its updated MAC value. It is possible to conduct decryption and integrity verification concurrently. Some encryption modes called *authenticated encryption* go even further to combine decryption and integrity verification into one process. Such examples include LAPM [31] and XCBC [24]. Authenticated decryption has less implementation and design complexity but makes decryption latency roughly the same as authentication latency. In general, in secure processor, authentication takes longer time than decryption because in theory authentication can only be initiated after data is fetched from memory. In secure processor design, the existence of various side-channel exploits on software confidentiality requires close-coupling between integrity verification and memory decryption. To issue decrypted instructions or data to a superscalar processor pipeline without integrity verification may put software and data confidentiality at risk.

3.4 Implementation

Implementation of hardware crypto engines is relatively straightforward. Most commercial crypto engines support standard ciphers such as Triple-DES (TDES), Rijndael-AES and standard MAC schemes. The trade-offs between performance, area, and energy consumption have been studied. In general, when comparing against a full-fledged processor implementation, the overhead of a crypto engine in terms of area and power consumption has been shown to be rather insignificant.

3.4.1 Cipher

A pipelined implementation of TDES can achieve about 20Gbits/sec using roughly 55K gates. Considering that the critical path of a single stage of TDES implemented as a customized design using dynamic logic [55], and a 0.18 μ process, is about 2nsec, the total latency will be about 96nsec. The throughput can be further increased significantly when multiple pipelines are deployed to exploit the concurrency.

The Rijndael cipher can process data blocks of 128, 192, or 256 bits by using key lengths of 128, 196 and 256 bits. It is based on a round function, which is iterated 10 times for a 128-bit length key, 12 times for a 192-bit key, and 14 times for a 256-bit key. Each round consists of four stages. For high throughput and high speed hardware implementation, Rijndael is often unrolled and with each round pipelined into multiple pipeline stages (4-7) to achieve high decryption/encryption throughput [42, 27, 26]. As shown in [27], the total area of unrolled and pipelined Rijndael is about 100K - 150K gates to achieve a throughout of 15-20Gbit/sec. Based on our own synthesized Verilog implementation, each decryption round of pipelined AES-Rijndael takes less than 5nsec using 0.18μ standard cell library and each encryption round takes even less time. The area overhead of encryption process is also small as it requires less number of gates. In this study, unless specified, the default latency is 80ns for the 256-bit AES.

Since the implementation of a crypto engine occupies a very small area, the extra power consumption is also very low. According to both industry standard and our implementation, it is estimated that a crypto engine consumes about tens of mW when active. In fact, many mobile devices such as PDAs and mobile phones already support hardware crypto-engines for encrypted wireless communication with very low energy overhead.

3.4.1.1 Encryption Mode

Encryption mode has major impact on secure processor performance. A few recent published works have provided detailed evaluation and comparison of the ECB mode, the CBC mode, and the counter mode [62, 73, 57]. In all the studies, counter mode delivers the best performance because it allows pre-computation of decryption pads in parallel with memory

fetch by encrypting a sequence number or time stamp. Once the encrypted data arrives, decryption will be performed by XORing the decryption pad and the encrypted data just fetched. With the assistance of sequence number caching [73] or prediction [57], decrypting counter mode encrypted data incurs only very small latency overhead. Furthermore, counter mode allows critical word to be accessed and decrypted first before the rest of a data block is fetched or decrypted. Our reference implementation of encryption mode is based on [57]. The OCB based design can use any block cipher with proven security. Typical choices are Triple-DES (TDES) and Rijndael-AES.

3.4.1.2 Integrity Verification

Integrity verification based on MAC is often standard operation. But variation of different MAC approaches can have significant impact on verification latency. Though it is plausible to carry out integrity verification and decryption concurrently, unlike counter mode, integrity verification must wait until the data is fetched. Furthermore, integrity verification has to be conducted on the entire data block. As a result, latency of integrity verification is often longer than decryption. In the reference implementation, we use standard HMAC [34] for protecting integrity of data blocks stored in the external RAM. The default size of MAC is 64 bits. The reference HMAC uses standard SHA-256 algorithm [50]. Simulation study is based on Verilog implementation of SHA-256 [64], synthesized using Synopsys. This design is totally asynchronous and has a gate count of 19,000 gates. The latency for this design is 74ns for 512 bits of padded input (padding with the required padding in SHA-256).

In addition to per-data block based integrity verification, secure processor sometimes also applies a hash tree or MAC tree for preserving the overall memory space integrity and preventing replay attacks of data blocks. This causes substantially additional amount of latency overhead. The CHTree scheme in [62] constructs an m-ary hash tree where m is the number of child nodes per parent node has and is equal to the size of the cache line divided by the size of hash values. A typical value of m is 4. To verify a data block, it takes $log_m(L)$ hashing computations, assuming the entire memory space comprising L total

Table 1: Processor model parameters

Parameters	Values
Fetch/Decode width	8
Issue/Commit width	8
L1 I-Cache	DM, 8KB, 32B line
L1 D-Cache	DM, 8KB, 32B line
L2 Cache	4way, Unified, 64B line, write back cache 256KB and 1M
L1/L2 Latency	1 cycle / 4 cycles (256KB), 8 cycles (1M)
Memory Bus	200MHz, 8B wide
Memory Latency	X-5-5-5 (core clocks) X depends on page status
CAS latency	20 mem bus clocks
Precharge latency (RP)	7 mem bus clocks
RAS-to-CAS (RCD) latency	7 mem bus clocks
Triple-DES latency	96ns, 48 stages
AES latency	80ns

data blocks. The reference implementation uses a m-ary MAC tree instead of hash tree for improved performance and security. A default value of m is either 4 or 8. This means that a new MAC is calculated over every 4 or 8 MACs. Recursively, each node of the MAC tree is a MAC of 4 or 8 child MACs. On top of the MAC tree is a root MAC of 64 bits. The root MAC is a signature of the entire memory space of a software application.

3.5 Simulation Framework

Our simulation framework is based on SimpleScalar [13] running SPEC2000 INT and FP Alpha binaries compiled with -O3 option. We integrated a more accurate DRAM model [25] to improve the system memory modelling, in which bank conflicts, page miss, row miss are modelled based on the PC SDRAM specification. The architectural parameters used for performance evaluation are listed in Table 11. Each benchmark is fast-forwarded and simulated at representative places according to SimPoint [56] for 400M instructions in performance mode. During fast-forwarding, L1 cache, L2 cache, and frequent value tracking are simulated. We subset the simulations for those with high L2 misses and memory throughput requirements.

The simulation environment and setup herein discussed is the default experiment setting for all the studies across this thesis. If a performance evaluation is conducted under a different processor model or configuration, the difference of setup will be highlighted and listed separately.

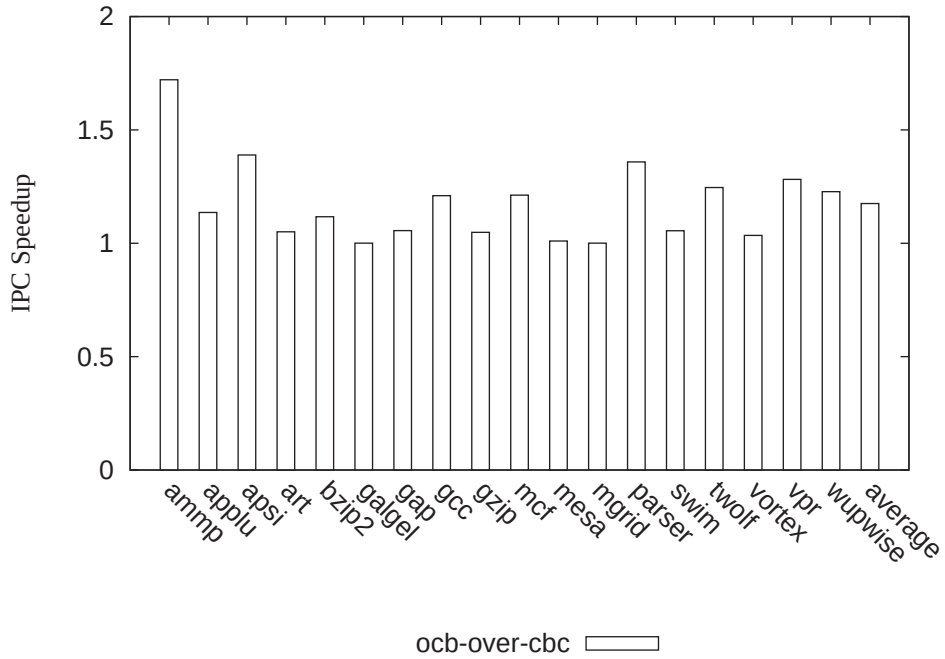


Figure 10: Normalized IPC under CBC or OCB, 256K L2

3.6 Performance Comparison of Encryption Modes

Here we summarize performance results of comparing different encryption modes. The results are collected on two L2 cache settings, 256K and 1M. Small L2 cache is critical because protection on data confidentiality is not only deemed for very high end machines but commodity platforms as well. Majority sold processors for regular users have L2 cache of only 256K or even less. Note that very large L2 size is not appropriate for evaluating SPEC2000 benchmarks because the entire working set of most SPEC2000 benchmarks can be fit into a very large L2.

3.6.0.3 Performance of OCB vs. CBC

First, we compared performance of the *cipher-blocking chaining, the CBC* mode based memory encryption vs. the *offset codeblock, OCB* based memory encryption. As addressed before, the OCB mode is not only more secure but also friendly for parallel processing of multiple memory chunks under large cache line size. We used AES as the underlying cipher. Under AES, each line is divided into 4 chunks. Figure 10 shows the IPC speedup of the OCB mode over the CBC mode under 256K L2 and figure 11 shows the IPC speedup

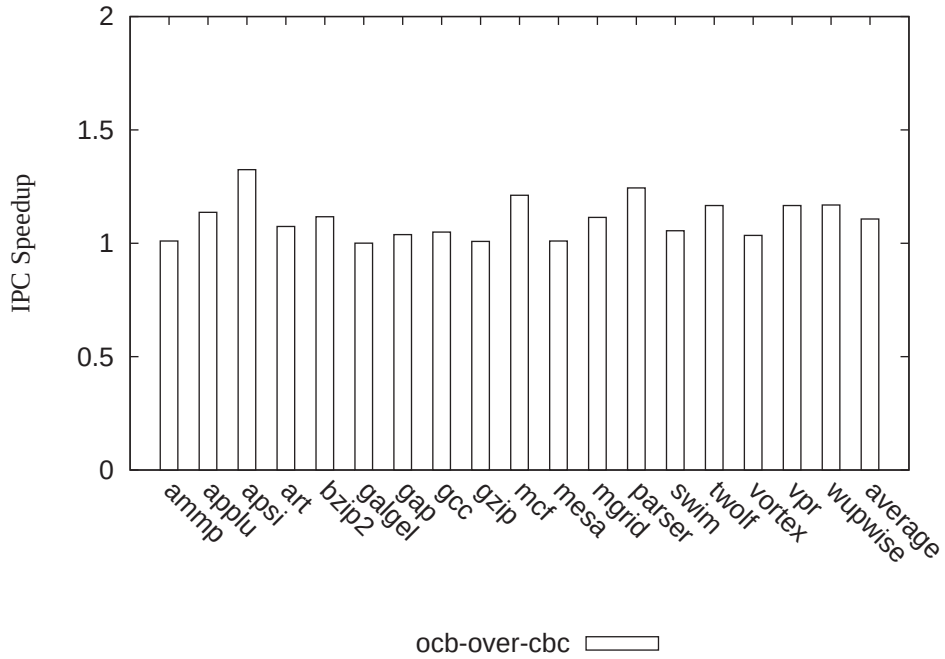


Figure 11: Normalized IPC under CBC and OCB , 1M L2

under 1M L2. As shown by both figures, for all the benchmarks, the OCB based approach achieves higher IPC performance than the CBC mode. Under 256K L2, the average IPC speedup is over 17%. Aside from using 256K L2, we also compared the OCB vs. the CBC under 1M L2. Similar to the scenario of 256K L2, the OCB outperforms CBC, the IPC speedup is about 11%. The main advantage of the OCB over the CBC is that it allows critical memory block to be decrypted first whereas under the CBC mode, all the memory blocks have to be decrypted sequentially.

3.6.0.4 Performance of Counter Mode vs. OCB

We also compared performance of the *counter* mode based memory encryption vs. the *offset codeblock, OCB* based memory encryption. Both the *counter mode* and the OCB mode support parallel memory block decryption and allow critical memory block to be decrypted first. Furthermore, counter mode allows decryption process to be paralleled with memory fetch. This means that under counter mode, part or sometimes all of the decryption latency can be overlapped with memory fetch latency. This provides additional performance advantage. However, to attain such benefit, sequence numbers or counters have to be somehow available to the secure processor. If the sequence numbers or counters

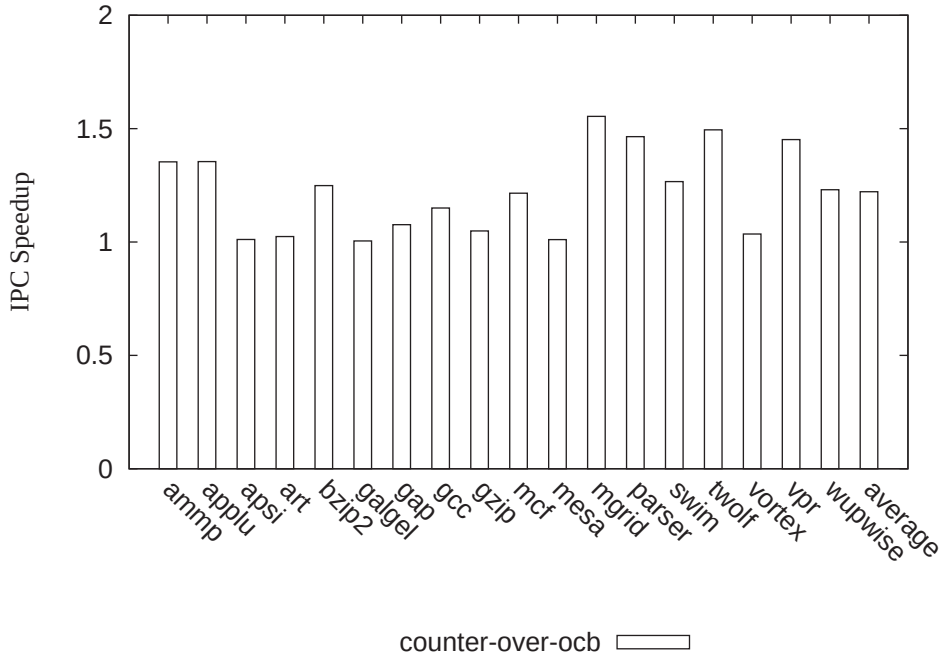


Figure 12: Normalized IPC under CBC or OCB, 256K L2

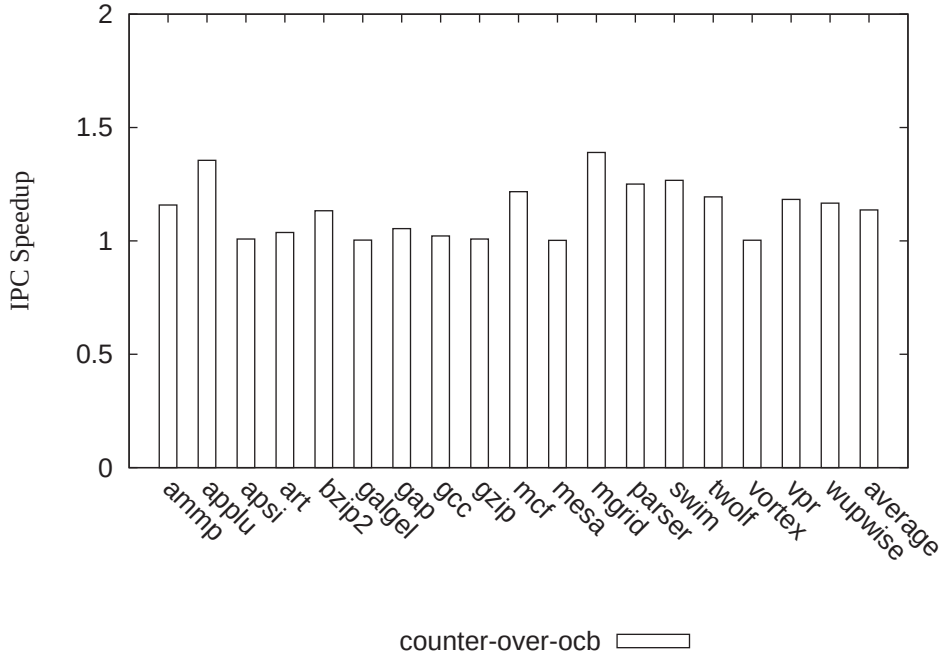


Figure 13: Normalized IPC under CBC and OCB, 1M L2

have to be fetched also from the memory, counter mode will no longer have the attractive performance edge over the OCB mode. In this evaluation, we assume that the sequence numbers are perfectly cached under counter mode.

Figure 12 shows the IPC speedup of counter mode over the OCB mode under 256K L2 and figure 13 shows the IPC speedup under 1M L2. As shown by both figures, for all the benchmarks, counter mode based approach achieves higher IPC performance than the OCB mode given that the sequence numbers are cached. Under 256K L2, the average IPC speedup is about 22%. Similar to the scenario of 256K L2, under 1M L2 setting, counter mode also outperforms the OCB mode. The IPC speedup is about 13%.

3.7 Conclusions

In this chapter, we compared the pros and cons of several encryption modes for encrypting system memory. The results show that both the OCB and the counter mode outperform the CBC mode because of support for parallel decryption and support of decrypting the critical memory block first. Between the counter mode and the OCB mode, the counter mode provides additional performance advantage because it allows decryption latency to be overlapped with memory fetch latency given that the sequence numbers or counter values are cached. In the next two chapters, we will introduce some new optimization solutions different from caching and pre-decrypting for reducing memory decryption latency overhead that are based on prediction techniques.

CHAPTER IV

COUNTER PREDICTION

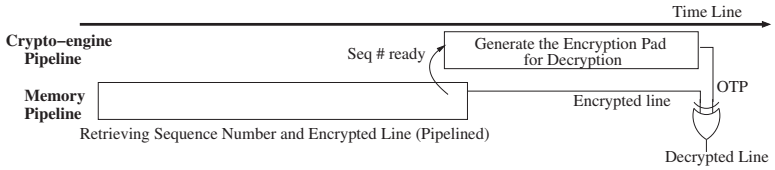
In [60, 73], a small sequence number (or counter) cache is used to cache counter values on-chip to exploit the advantage of counter mode architecture. Profile studies show that sequence number cache hit rate does not grow steadily with its size. One possible explanation of this "plateau" effect of the sequence number cache is that the sequence number cache may contain (multiple) very large working sets . A small cache of several Kbytes may be enough to capture the first working set, but other working sets might be too large to be captured by a sequence number cache of tens or even hundreds of Kbytes. In other words, the area cost to improve the hit rate via simple caching can be prohibitively high.

We propose an alternative solution via prediction and precomputation to hide memory decryption latency more effectively with minimal area overhead. The premise is that in the counter mode, predictable counters are often used to generate the encryption OTP deterministically using some standard encryption function. Hence one can speculate the counter value and pre-compute the OTP before the counter is loaded from memory. It is important to point out that even though the counter value is deterministic and predictable, the cryptographic function used to generate the OTP is not. These functions involve a secret key only known to the secure processor itself. It is computationally infeasible for an adversary to predict the OTP even with a known counter value.

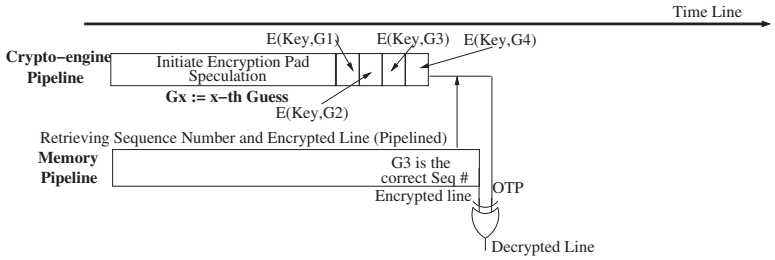
4.1 OTP Prediction and Precomputation

In this section we explain the concept behind OTP prediction and pre-computation[1]. We first explain the technique using our proposed regular sequence number prediction algorithm and we point out a potential performance issue for the regular sequence number prediction. Next, we discuss a simple modification to the regular prediction scheme to redress the

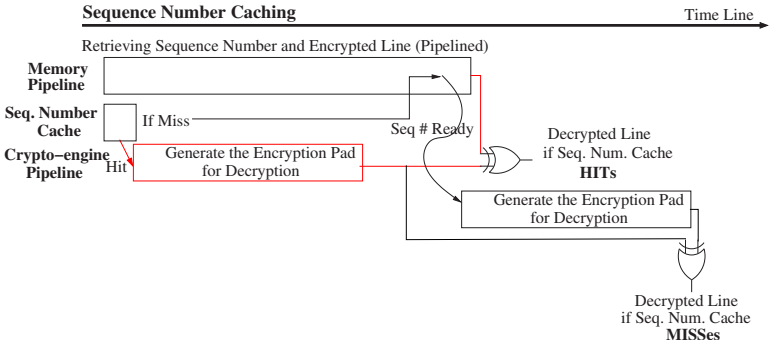
[1]Hereafter we use OTP prediction to represent OTP prediction and pre-computation.



(a) Baseline



(b) OTP Prediction and Precomputation



(c) Sequence Number Caching

Figure 14: Timeline Comparison of Different OTP Computation

potential performance problem. It is also important to note that *OTP prediction* can be tied with any scheme based on the counter mode or stream cipher such as those discussed in [60, 73].

4.1.1 Regular OTP prediction

The concept of OTP prediction and pre-computation can be understood from the time-lines shown in figure 14. We assume that the memory access latency and the encryption OTP generation latency are comparable[2]. We also assume a fully-pipelined encryption and decryption AES crypto-engine. The input block of the AES is a concatenation of a 64-bit virtual address and a 64-bit sequence number. For 32-bit architecture, the virtual address is padded to 64-bit. The timeline of figure 14(a) shows a scenario when fetching a cache

[2]We later justify this assumption in Section 4.3.

line in a baseline security architecture without any support to accelerate the decryption process. It is obvious that the crypto-engine pipeline sits idle for the whole time between the time when the request is sent to the memory and the time when the sequence number is returned. Now suppose that in our architecture the sequence number of a given cache line is predictable and only depends on the page the cache line is associated with. We will justify our claim later. figure 14(b) shows the timeline of our framework. A certain number of sequence number guesses, G1, G2, ..., Gx, can be tried and passed to the crypto-engine for pre-computing their corresponding OTPs while the actual memory request is being sent to memory. Since the AES cipher is pipelined, the predicted and precomputed OTPs will be available around the time the actual sequence number is obtained from memory. The precomputed OTPs are represented as E(Key,G1),E(Key,G2)... in the figure 14(b). When the actual sequence number returns, we check it against all the guessed numbers. If one of them matches, in our case G3, we already have the encryption pad E(key,G3) for G3. We can now directly obtain the plaintext by XORing the pre-computed OTP with the encrypted cache line fetched. The rationale of our scheme is to utilize the idle time of the crypto-engine pipeline for pre-computing several OTPs and thus hide the memory access latency if one of the speculations succeeds.

To demonstrate its difference from prior art, figure 14(c) shows the sequence number caching technique. A sequence number cache stores a finite set of sequence numbers for evicted lines. If a request hits the sequence number cache, the sequence number is obtained and the OTP generation can begin before the cache line returns from memory. For a sequence number cache miss, the decryption process will be serialized similar to the baseline scenario. One issue of sequence number caching is the hit rate, which can be substantially reduced when the working set is large or in-between context switches. Another issue is the potential large hardware overhead dedicated to the sequence number storage for achieving decent hit rates. It should be kept in mind that the area overhead of our scheme is minimal compared to the caching scheme because we only need a small buffer to store the pre-computed OTPs. We will show in our results that the benefit we receive with our optimized design cannot be achieved even with a very large sequence number cache. Our scheme is

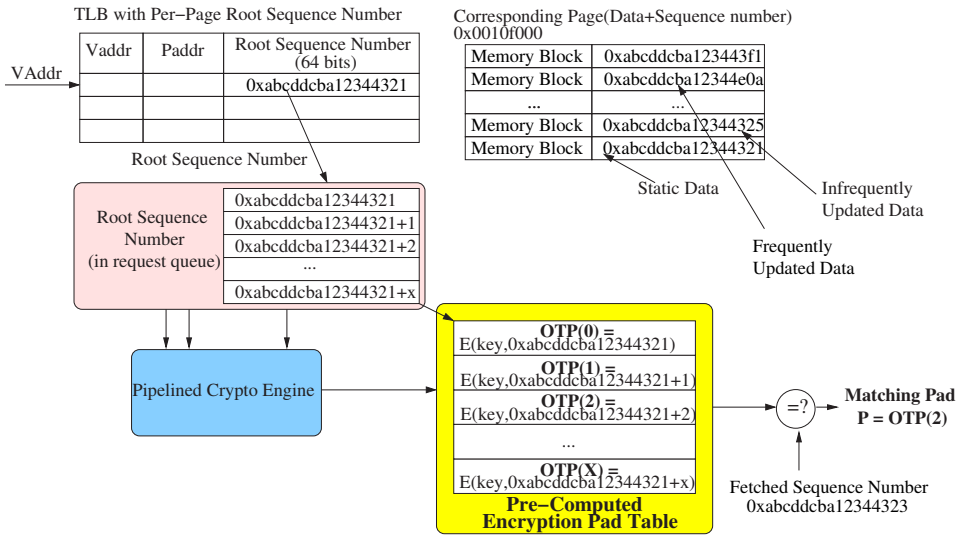


Figure 15: OTP Prediction

also complementary to the sequence number cache. We can combine them in a design to gain benefits offered by both. Now we elaborate our prediction architecture in details.

Figure 15 shows the design of our OTP prediction and precomputation mechanism. A root OTP sequence number is assigned to each virtual memory page by a hardware random number generator each time the virtual page is mapped to a physical one. All the cache lines of the same page use the same root OTP sequence number for their initial values. Each TLB entry is tagged with the root sequence number of the corresponding page. Whenever a dirty line is evicted from the secure processor, the sequence number associated with the corresponding line is incremented. For each missing cache line, a request for the line itself and its associated sequence number is sent to memory. Simultaneously, the prediction logic takes the root OTP sequence number associated with the virtual page, and inserts a few sequence number guesses into the *request queue*. The pipelined crypto-engine takes each request and computes its corresponding encryption OTP. Upon the receipt of the correct sequence number from memory, the processor compares it with the set of sequence number predictions. If a match is found, then the corresponding pre-computed OTP is used to decrypt the fetched memory data. If no match can be found, the crypto-pipeline will take the newly received sequence number and computes its OTP, same as baseline. In summary, OTP prediction effectively hides the latency of encryption pad generation by speculatively

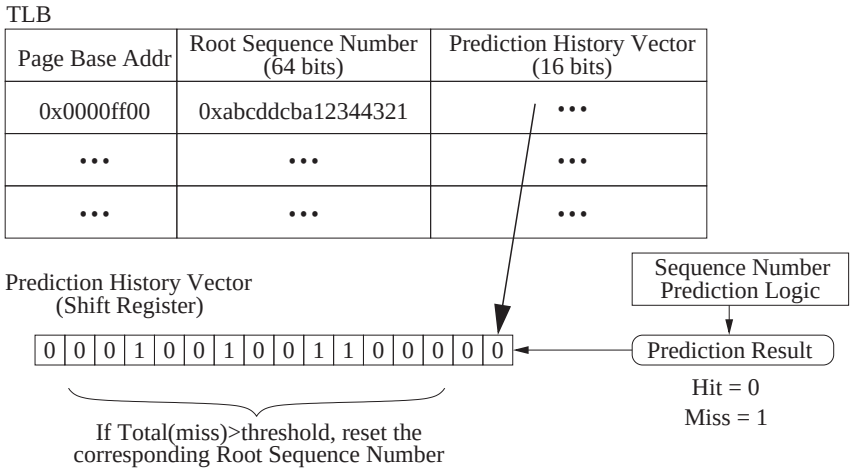


Figure 16: Prediction Tracking and Sequence Number Resetting

pre-computes encryption pads using speculated sequence numbers.

Sequence number prediction is based on the observation, that during the whole lifetime a physical memory page is bound to a virtual memory page, many of its cache lines are only updated a very small number of times. Our profiling study of SPEC benchmarks indicates that many lines are rarely updated during the entire process lifetime, in other words, when a cache line missing the L2 cache, its sequence number is very likely to be within a small range of the first time initialized random sequence number associated with that memory page. Regular OTP prediction is designed to predict sequence numbers associated with static and infrequently updated data.

4.1.2 Adaptive OTP prediction for frequently updated data

There is a concern about the performance of OTP prediction over a large time window of execution. It is reasonable to suspect that prediction rate may drop as data are frequently updated. To address this issue, a dynamic prediction rate tracking and sequence number reset mechanism is proposed. The purpose of this mechanism is to identify those virtual pages with low prediction rate caused by frequent memory updates and reset its page root sequence number to a new random value so that high predictability can be maintained.

Prediction tracking is performed in hardware using a scheme as follows. There is a 16 bit prediction history vector (PHV) for each memory page. The PHV records hit or miss of the last 16 sequence number prediction on cache lines of the associated page. Every time

a cache line is loaded from memory, the PHV of that page is updated by shifting the new prediction result into the vector (1 for misprediction and 0 for hit). When the total number of mispredictions of the last 16 predictions is greater than a threshold, the root sequence number associated with that page will be reset to a new randomly generated number. After reset, blocks of the involved page will use this new number for OTP generation next time when it is evicted from the L2.

The adaptive predictor requires the ability to test whether a sequence number used by a cache line is counted based on the current root sequence number. Note that this function does not have to be 100% accurate because a wrong test result will only cause reset of the sequence number. A simple implementation is to use the distance between a sequence number and the current root sequence number as a criteria. To decide whether a sequence number started its count from the current root sequence number, its distance to the current root is calculated. If the distance is negative or too large, the sequence number is considered counting from an old root sequence number. If a mismatch is detected, the corresponding cache line will reset its sequence number to the current root sequence number.

4.2 Security Analysis

We discuss a few issues about OTP prediction security in this section.

- Security is guaranteed by the security analysis provided in [9] since we did not modify the counter mode encryption scheme itself, but rather showed how to decrease the long decryption process latency via architectural techniques.
- In OTP prediction, different memory blocks of the same page may use the same sequence number. This, however will not weaken security. When the OTP is generated, the address is used together with the sequence number and a prefix-padding. Since each memory block has a different address, the resultant OTP will be different for different memory blocks. Knowing OTP of a particular memory block does not reveal any information about the OTPs of other blocks of the same page.
- The root sequence number is set and reset using a hardware random number generator. For the sequence number to wrap-around, it has to be incremented 2^{64} times. This

equals hundreds of years under the current processor clock speed.

4.3 Simulation Methodology and Implementation

4.3.1 Simulation framework

Our simulation framework is based on SimpleScalar 3.0 running SPEC2000 INT and FP benchmark programs compiled with -O3 option. We implemented architecture support for *OTP Prediction* and root sequence number history over SimpleScalar's out-of-order Alpha simulator.

The architectural parameters used for performance evaluation are listed in table 4.3.1. To model OTP prediction faithfully, we added memory profiling support to SimpleScalar that keeps track of memory transactions for evaluating OTP prediction, such as number of times a memory block is evicted from the L2 cache, the sequence number assigned to each virtual page, and etc. Each benchmark is fast-forwarded at least 4 billion instructions and simulated in a representative place according to SimPoint [56] for 400M instructions in performance mode. During the fast-forwarding, the L1 cache, the L2 cache, the sequence number cache, the sequence number prediction mechanism are simulated. The profiled memory status is also updated during the fast-forwarding. To study the performance sensitivity of the OTP prediction optimization, we also run each benchmark in a simplified mode that simulates the memory hierarchy and the OTP prediction for 8 billion instructions. We used 16 bits for the prediction history window. By default, the sequence number of each virtual page is reset if the number of prediction misses over the last 16 is greater than or equal to 12. The prediction depth, that is the number of guesses generated for each missing sequence number, is set to 5. We also use a prediction swing of 3 for context-based prediction to be discussed in Section 4.5.4. Per-page root sequence numbers require small storage space. Given a 64-bit sequence number and 256 page entries cached, the total cost of storing root sequence numbers is about 2KB. Also, to simulate the effect of OS and system, dirty lines of caches are flushed every 25million cycles. Furthermore, we subset the SPEC simulations for those with high L2 misses. All the benchmarks are simulated under the security setting that data confidentiality must be protected.

Table 2: Processor model parameters

Parameters	Values
Frequency	1.0 GHz
Fetch/Decode width	8
Issue/Commit width	8
L1 I-Cache	DM, 8KB, 32B line
L1 D-Cache	DM, 8KB, 32B line
L2 Cache	4way, Unified, 32B line, Writeback, 256KB and 1MB
L1 Latency	1 cycle
L2 Latency	4 cycles (256KB), 8 cycles (1MB)
Memory Bus	200MHz, 8B wide
Memory Latency	X-5-5-5 (core clocks) X depends on page status
CAS latency	20 mem bus clocks
Precharge latency (RP)	7 mem bus clocks
RAS-to-CAS (RCD) latency	7 mem bus clocks
AES latency	14 rounds with an initial and a final round, 6 stages 1ns each 96ns
Sequence number cache	4KB, 128KB, 512KB (32B line)
Prediction History Vector	16 bit
PHV threshold	12
Prediction depth (used by all predictions)	5
Prediction swing (used by context-based only)	3

4.4 Evaluation of adaptive OTP prediction

In this section we summarize the prediction rates and IPC results of adaptive OTP prediction. The results are collected for two L2 cache sizes, 256KB and 1MB. Choosing a 256KB L2 cache is not only representative for many contemporary machines but also appropriate for our evaluation given the SPEC benchmark suite is known to have relative small working set.

4.4.1 OTP prediction rate over large execution time

One concern about OTP prediction is that its performance over execution time or its capability to predict dynamic data. To answer this question, we simulated performance of *OTP prediction* over a relatively large time window, 8 billion instructions. We also used two different sequence number cache sizes, 128KB, and 512KB, for our reference comparison. Figure 17 shows the sequence number hit rates for different sequence number cache sizes and our OTP prediction. As the results indicate, the hit rate of sequence number cache is relatively low even with a decent sized 128KB sequence number cache. The results also indicate that sequence number prediction can achieve good prediction rate over a long

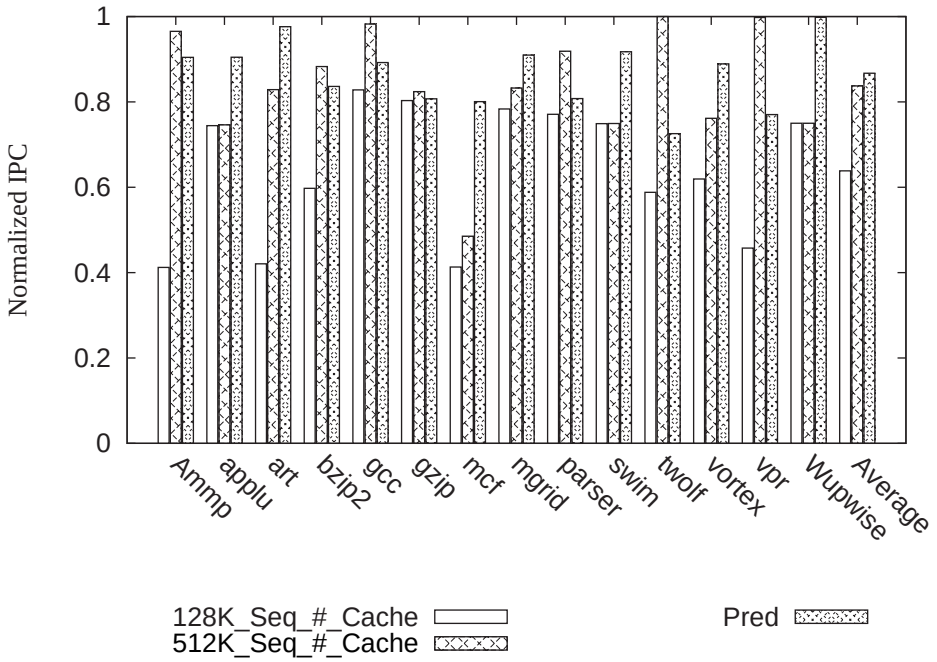


Figure 17: Sequence Number Hit Rates, 256KB L2, 8 billion instructions

execution window. The average prediction rate is 82%, higher than that of a 128KB or a 512KB sequence number cache. Figure 18 shows the same comparison with a 1MB L2. Similarly, OTP prediction also achieves a better performance than sequence number caching for a fairly large L2. The average prediction rate is 80% compared to 57% for a 128KB sequence number cache.

The results also verify the initial assumption that sequence numbers tend to have large working set. One possible explanation is that for the sequence number cache to perform well, the processor has to miss on the same memory block many times within a short time window before the sequence number is evicted from the sequence number cache. Due to the temporal locality and memory working set, a processor rarely repeats missing on the same memory block many times in a short duration of time. This illustrates the limitation of using caching for improving performance since the area cost to cache the complete working set can be prohibitively high.

Figure 19 breaks the total number of hits into three categories, 1) hit both, a sequence number that is in the sequence number cache and can be predicted; 2) prediction only, a sequence number that is missing in the sequence number cache, but can be predicted; 3)

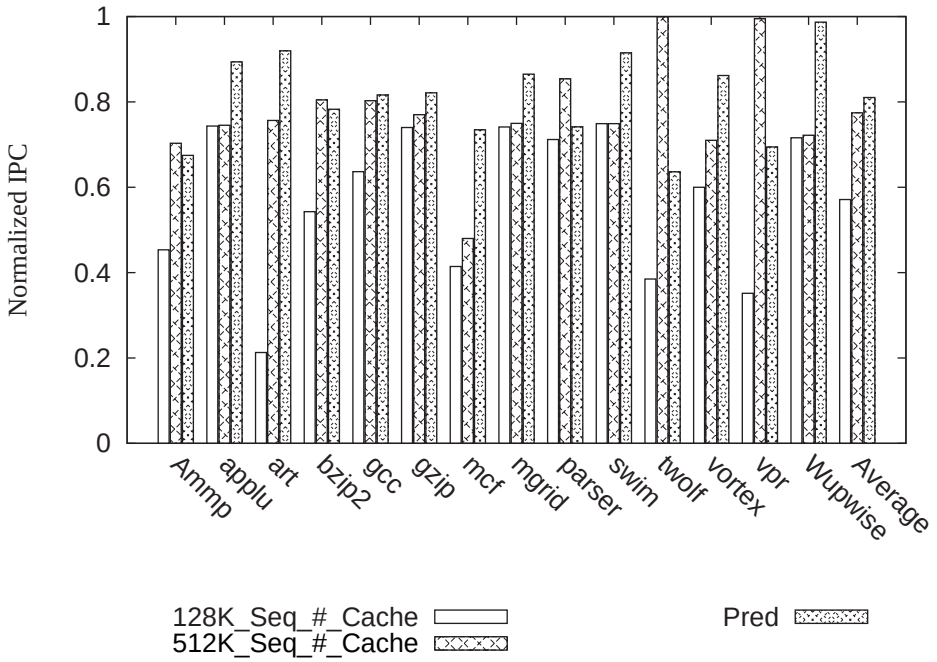


Figure 18: Sequence Number Hit Rates, 1MB L2, 8 billion instructions

sequence cache only, sequence number cannot be predicted but available in the cache. The results are collected from simulation using a 32KB sequence number cache plus prediction. As seen from the figure, OTP prediction can uncover more performance opportunities lost by the sequence number caching scheme.

4.4.2 IPC improvement using OTP prediction

Increasing prediction rate has a great performance impact on memory-bound benchmarks. For example, without OTP prediction, the average IPC of the selected benchmarks only reaches 82% of IPC of an oracle scenario where every sequence number is cached. In particular, bzip2, mcf, mgrid, twolf, and vpr have their ratios in the range of 60% to 80%. If OTP prediction can achieve ideal 100% prediction rate, the potential performance improvement would be in the range of 20% to 40%.

Figure 20 and figure 21 show normalized IPC performance of large sequence number caches vs. adaptive OTP prediction. The IPC is normalized to the oracle case. As shown, OTP prediction can effectively improve performance. On average, IPC is increased by 18% and 11% for a 256KB L2 and a 1MB L2, respectively. For ten of the fourteen SPEC2000 benchmarks, the improvement is in the range from 15% to 40%. Six benchmarks have

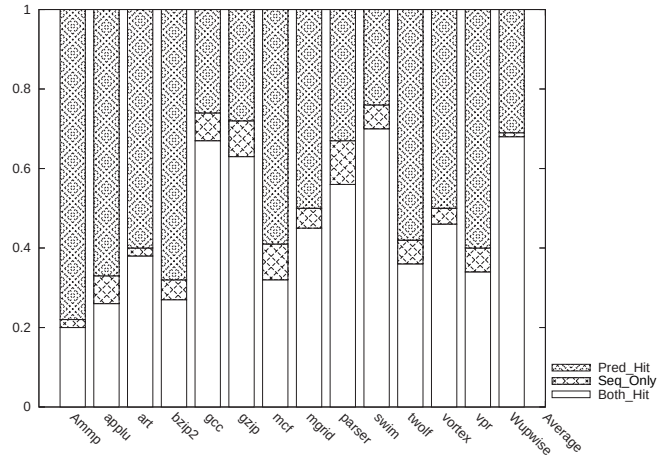


Figure 19: Breakdown of Contribution of Sequence Number Cache, and OTP Prediction

their improvements over 20% and two over 30%. For every benchmark, OTP prediction outperforms a 128KB sequence number cache. For average IPC, *OTP prediction* even performs better than a very large 512KB sequence number cache. The results clearly show the advantage of OTP prediction over a pure sequence number caching.

Consistent with the prediction rate results, sequence number prediction is more effective than caching for overall performance. To achieve similar performance using caching, the required sequence number cache size needs to be unreasonably large, larger than a typical unified L2 cache.

4.5 Optimizing OTP Prediction

Although adaptive OTP prediction can handle both infrequently and some frequently updated data, our study of prediction rate shows that there is still room for improvement. In this section, we propose and investigate some unique optimizing techniques to increase the prediction performance for frequently updated data.

4.5.1 Profiling misprediction

First, we conduct profiling studies on OTP misprediction. We found there are two main contributors. One is *prediction depth* which is equivalent to the number of predictions can be made for each missing line. Profiling studies reveal that some lines are evicted far more often than the others and they are often outside the prediction range. Increasing the prediction

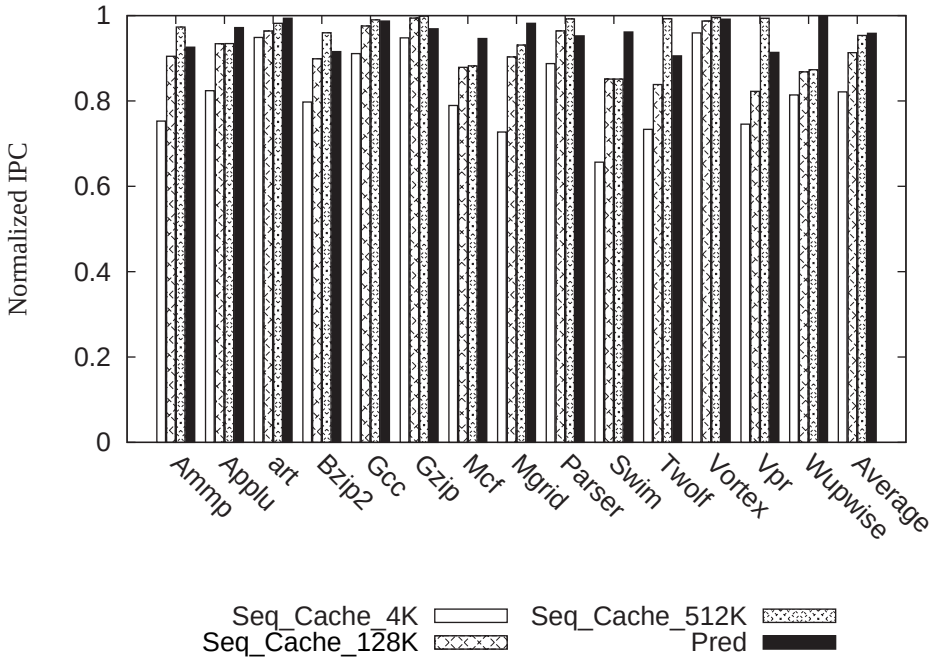


Figure 20: Normalized IPC Under Different Sequence Number Cache Sizes(4KB, 128KB and 512KB) vs OTP Prediction, 256KB L2

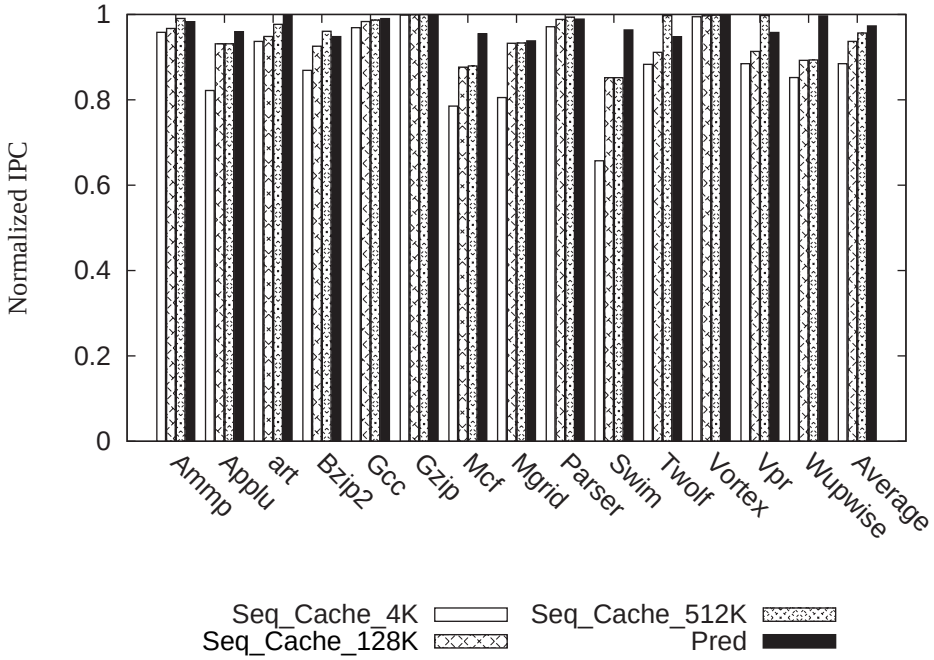


Figure 21: Normalized IPC Under Different Sequence Number Cache Sizes(4KB, 128KB and 512KB) vs OTP Prediction, 1MB L2

depth, i.e. more predictions per line, does not solve the problem as too many predictions will overload the crypto-engine and could lead to negative impact on performance. The second contributor is due to the reset of the root sequence number. After the per-page root sequence number is reset, all the future predictions on cache lines of the same page will use the new root sequence number instead of the old ones that causes predictions of lines using old root sequence numbers to fail.

4.5.2 Two-level prediction

To reduce mispredictions caused by short prediction depth, we introduce a novel range prediction technique to be used in combination with the regular OTP prediction. Regular OTP prediction is good at predicting sequence numbers that are not updated frequently. For example, if the prediction depth is 8 and a cache line has been evicted 23 times. It is not possible for the regular OTP prediction to predict correctly. However, if we divide the distance between each sequence number to its root sequence number into multiple ranges, for instance, four ranges, [1, 8], [9, 16], [17, 24], [25, ∞] and have OTP predictions generated only under a particular range for each sequence number, it will greatly increase the hit rate of OTP prediction without adding pressure to the crypto-engine pipeline. We call this design, *Two-level OTP Prediction*, with the first level predicting the possible range of a sequence number and the second level using regular OTP prediction in that range. To implement a two-level OTP prediction, range information associated with each cache line needs to be encoded and stored. In fact, the cost of the first level prediction is small. For example, assume that there are four ranges, this information can be encoded with only 2 bits. Under a 4KB page and 32-byte lines, the cost to store range information for all the 128 lines of a page is only 256 bits.

When accessing a sequence number, the secure processor will look up the range prediction table, where each entry of the table stores the range information for all lines in a page. Then the retrieved range information is used by the regular OTP prediction where a number of predicted sequence numbers are inserted to the *prediction queue*. The starting

predicted sequence number is *root sequence number + range lower bound*, and the last predicted sequence number is *root sequence number + range lower bound + prediction depth*. When a line is evicted from the processor, its associated entry in the range prediction table is updated.

The additional cost of range prediction is relatively small considering a 64 entry table costs only about 2KB with the benefit of quadruple the effective prediction depth.

4.5.3 Root sequence number history

Per-page root sequence number reset is important to maintain a satisfactory OTP prediction performance. However, resetting and discarding old root sequence numbers may cause predictions on sequence numbers based on discarded root numbers to fail. To prevent misprediction caused by resetting, we introduced a sequence number memoization technique that keeps a certain number of old root sequence numbers (usually very small, 1 or 2 at most) as history. When predicting a missing sequence number, predictions based on both the new root sequence number and the old sequence number(s) are generated. It is important to note that when a dirty cache line is being evicted, the writeback is always encrypted using a new OTP based on the current root sequence number.

4.5.4 Context Based Prediction

Context-based prediction is another OTP prediction optimization that can significantly improve the OTP prediction rate. The idea of context-based prediction is very simple. We use a register called *Latest Offset Register (LOR)* that records the offset sequence number of the most recent memory access (= sequence number - root sequence number). For a new memory fetch, aside from the regular predictions based on the per-page root sequence number, the context-based OTP prediction mechanism also generates a few more predictions based on the LOR value. In the context-based prediction, two sets of prediction are made. The first is our regular prediction using the prediction depth (*pred_depth*). Assume that root(addr) returns the root sequence number associated with fetched data, so the prediction of sequence numbers falls in the range of $[root(addr), root(addr) + pred\_depth]$, in other words, $(pred\_depth + 1)$ predictions are generated. The second prediction set uses

the LOR value with a *prediction swing (pred_swing)*. The prediction falls in the range of $[Max(root(addr) + LOR - pred\_swing, root(addr)), root(addr) + LOR + pred\_swing]$ with a maximum of $2 \cdot pred\_swing + 1$ predictions to be made. In our simulation, the *pred_depth* and *pred_swing* are set to 5 and 3 respectively.

The context-based prediction has far less overhead than Two-level prediction because it requires only one extra register rather than a range table. Comparing with the regular OTP prediction scheme, the additional cost is almost negligible. However, the context-based prediction does increase the workload on the OTP engine because it generates more predictions. As will be shown, the context-based prediction in fact has the best prediction rate among all three prediction schemes.

4.6 Performance Evaluation

In this section, we present performance evaluation of the proposed optimizations for improving OTP prediction rate. We do not report the results for the root sequence number history method since it shows only marginal improvement over the regular OTP prediction scheme.

4.6.0.1 Two-level and Context based predictions

We use a 4-bit range predictor for each memory block and there are 64 entries in the range prediction table (about 4KB size). The prediction rates of two-level prediction are shown in figure 22 and figure 23. We can see a significant improvement over the regular OTP prediction. The average prediction rate of two-level prediction is almost 96% with a 256KB L2 and 95% with 1MB. Comparing with results in figure 17 and figure 18, two-level OTP prediction using only 4KB range prediction table actually outperforms both the 128KB and 512KB sequence number cache settings. However, the best prediction performance is achieved by the context-based prediction. For nearly all benchmark programs, the context-based prediction attains almost perfect prediction rate.

The prediction rate using a large L2 is often smaller than the prediction rate using smaller L2 size. Since a large L2 typically reduces memory traffic, the number of predictions made with a larger L2 is usually far less than that of a smaller one. In terms of absolute

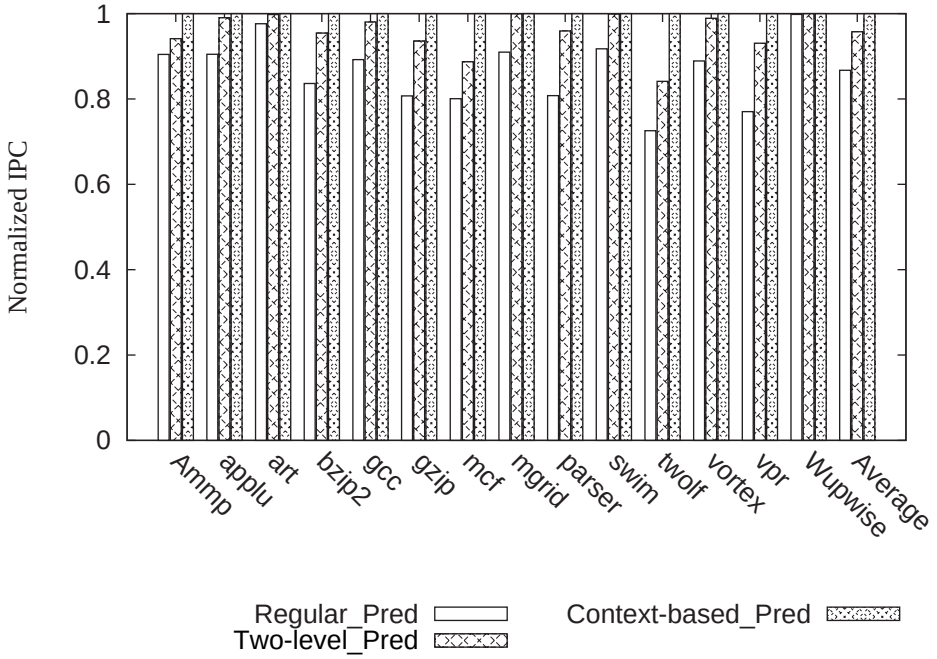


Figure 22: Hit Rate of Two-level Pred vs. Context-based Pred vs. Regular Pred, 256KB L2, 8 billion instructions

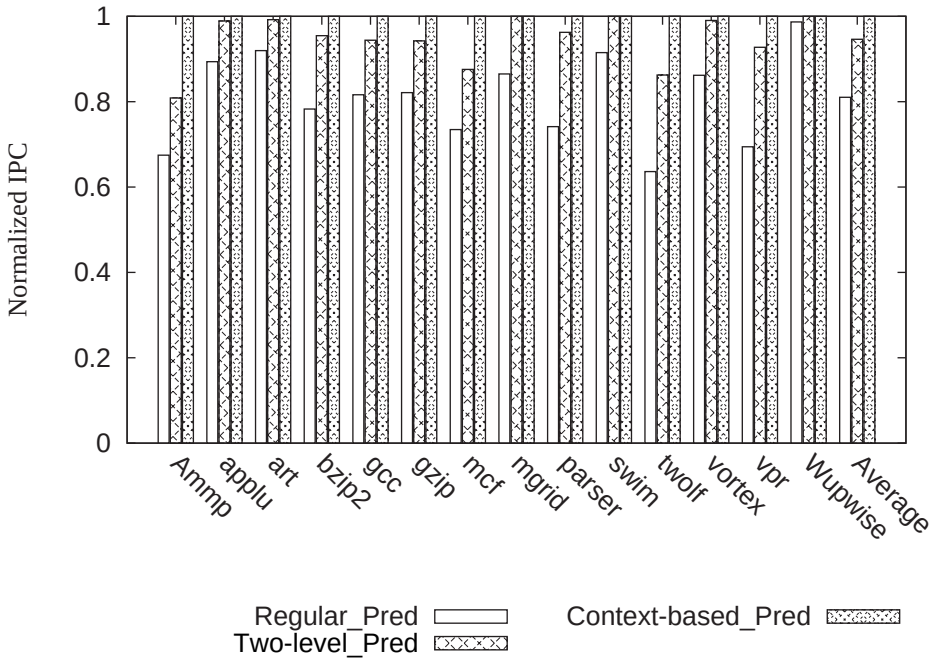


Figure 23: Hit Rate of Two-level Pred vs. Context-based Pred vs. Regular Pred, 1MB L2, 8 billion instructions

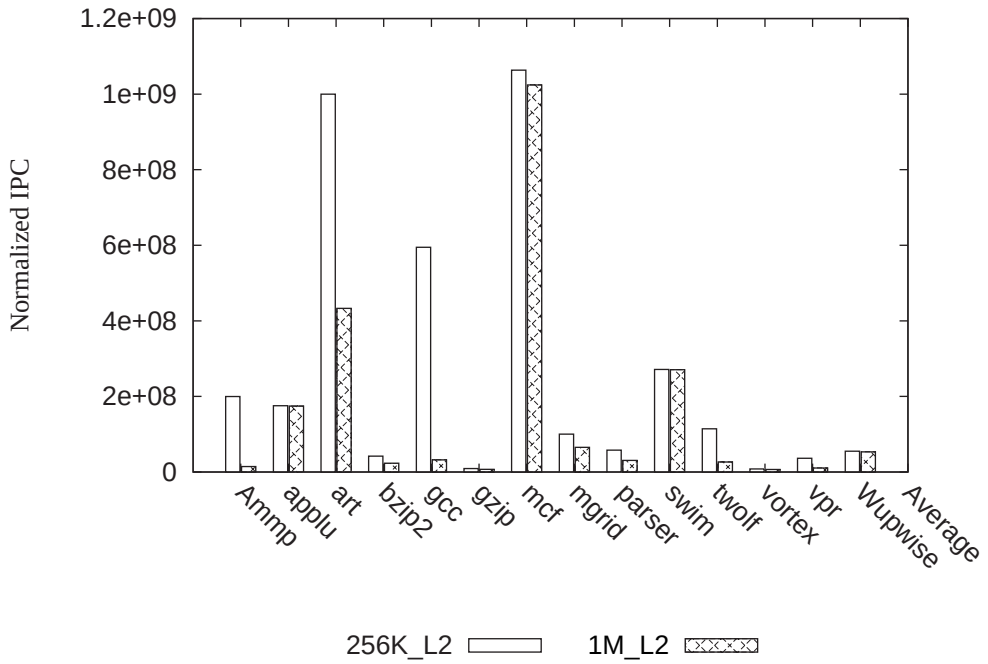


Figure 24: Number of Predictions under 256KB vs. 1MB L2

number of miss predictions, larger L2 has less number of overall misses than smaller L2. Figure 24 shows the absolute number of predictions.

Figure 25 and figure 26 show normalized IPC results for the two-level and the context-based predictions. Both the two-level and the context-based predictions achieve better performance over their regular counterpart. Using 256KB L2, for some benchmarks such as ammp, bzip2, twolf, and vpr, the improvement is about 7%. With 1MB L2, the improvement is about 4% for a number of benchmarks including applu, bzip2, mgrid, swim, twolf and vpr. For most benchmarks, the context-based prediction outperforms two-level prediction. Note that the context-based prediction also generates more predictions than two-level prediction. Considering the extra range table cost of the two-level prediction, the context-based prediction is a better choice because on average, it delivers better performance with far less hardware requirement.

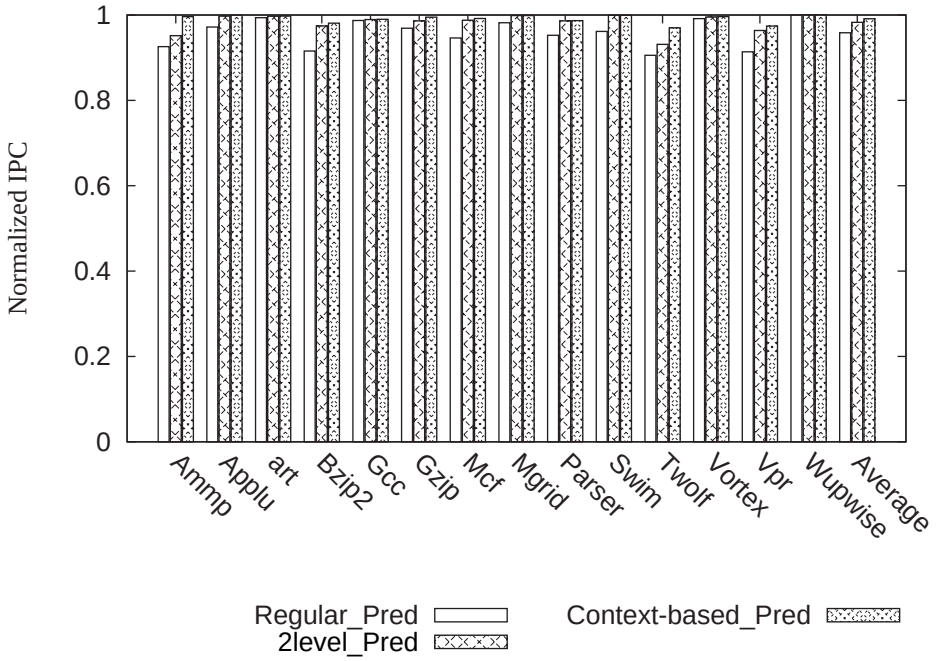


Figure 25: Normalized IPC of Two-level Pred vs. Context-based Pred vs. Regular Pred, 256KB L2

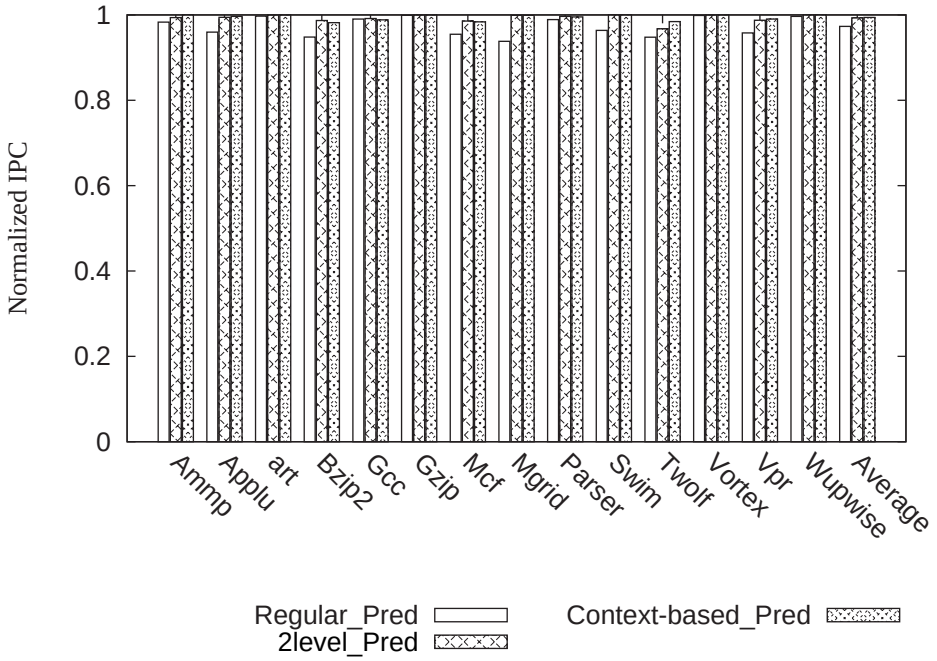


Figure 26: Normalized IPC of Two-level Pred vs. Context-based Pred vs. Regular Pred, 1MB L2

4.7 Conclusions

This chapter describes an OTP prediction and precomputation mechanism for hiding decryption latency in counter mode security architectures. In addition, we propose and evaluate several optimization techniques to further improve the performance of OTP prediction. The proposed adaptive OTP prediction improves the performance of memory-bound applications significantly over the prior sequence number caching method with a much smaller area overhead. Without any extra on-chip cache, our adaptive OTP prediction can achieve an average of 82% OTP prediction rate for both the infrequently and frequently updated memory. For several memory-bound SPEC benchmark programs, the IPC improvement is in the range of 15% to 40%. In addition to the regular prediction technique, a two-level OTP prediction is proposed to further improve the prediction rate from 82% to 96%. Finally, we propose a context-based OTP prediction that performs even better than the two-level prediction. It also outperforms the caching scheme with a very large 512KB sequence number cache. The two-level and the context-based predictions can provide additional 7% IPC improvement on top of the regular OTP prediction. To summarize, with a minimal area overhead, our techniques succeed in achieving significant performance improvement for counter mode encrypted memory architectures.

CHAPTER V

VALUE PREDICTION

Cipher text prediction is another technique to hide latency overhead of encrypted memory. It is based on frequent value prediction [39, 72] and specifically designed for memory encryption that uses parallelizable direct memory encryption modes such as the well known OCB mode [54]. Unlike the *counter mode*, modes of direct memory block encryption themselves do not support any pre-computing.

In this chapter, we explain value prediction and how this technique may be applied to predict cyphertext. We will also describe how cyphertext prediction is applied to two popular block ciphers, namely the Triple DES and the AES.

5.1 Value Prediction

Previous studies have shown the existence of predictable or frequent values among data fetched from the external memory [39, 72]. For SPEC2000 benchmark suite, up to 80-90% fetched dynamic data are from a small set of application dependent data values (8 to 64 most frequent values). Memory profiling results in [72] also show that on average, about 40% of the data stored in the entire memory space of an application are frequent values. The existence of predictable and frequent values enables novel yet viable prediction techniques for reducing latency overhead associated with direct encrypted memory. Here we propose *ciphertext speculation* and *MAC speculation* that effectively hide decryption and integrity verification latency by *speculatively encrypting* predictable values or pre-computing MACs for predictable values and match the results against fetched encrypted data block or MACs. In this study, we used a similar dynamic frequent value tracking mechanism as proposed in [72], in which the most frequent values are dynamically determined. It can capture most frequently encountered values in-between working set changes or software context switches.

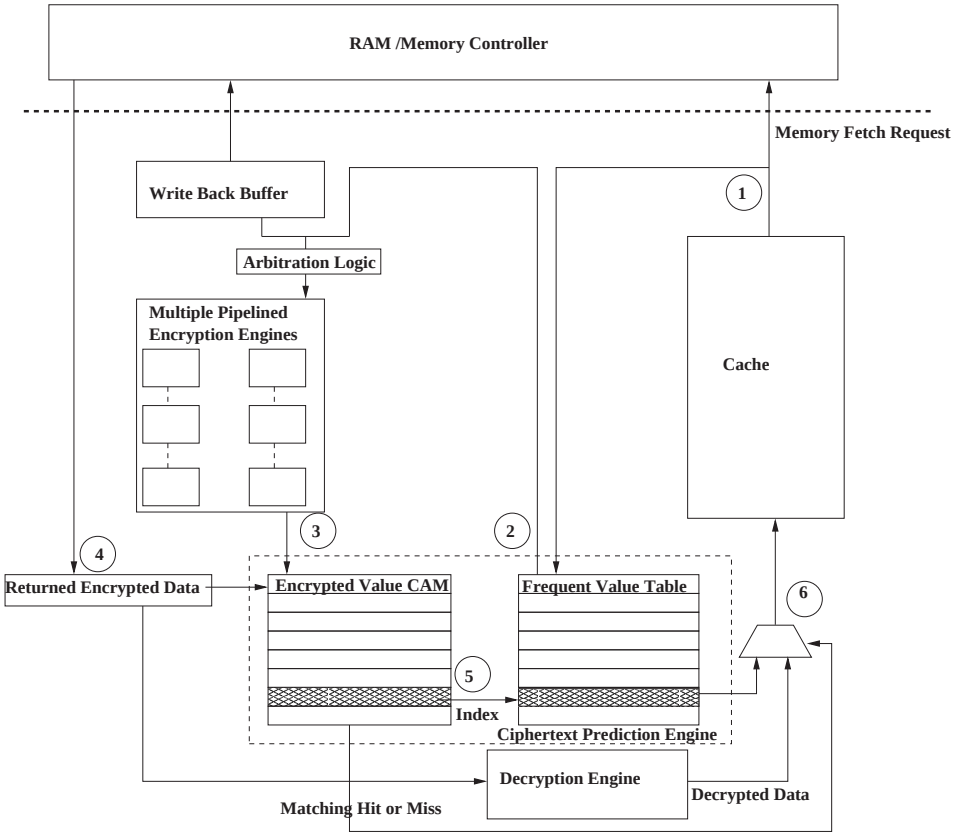


Figure 27: Ciphertext Speculation Mechanism

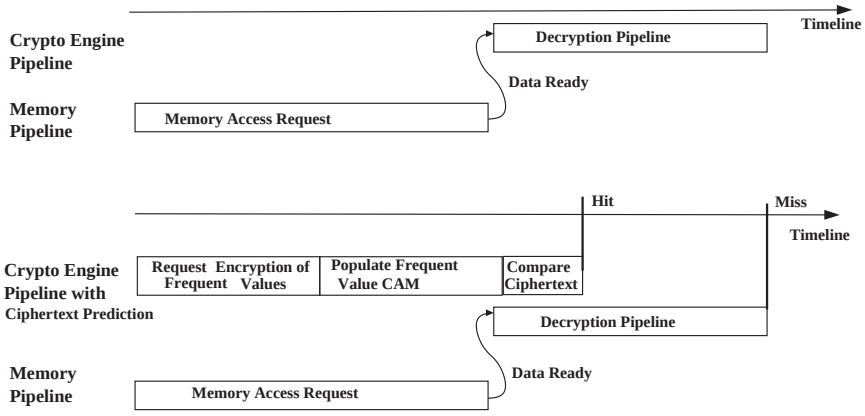


Figure 28: Timeline of Ciphertext Speculation

5.2 Ciphertext Speculation

The rationale of *ciphertext speculation* includes the follows.

- The decryption latency of block cipher can be significant. It ranges from 50ns to a couple of hundreds of nsec depending on the cipher, design methodology, area constraint, and fabrication process [42, 26, 55, 19].
- Decryption of direct encrypted memory contains serialized operations including demand fetch from memory and decryption of encrypted memory block.
- A large portion of data stored in memory and fetched from memory are predictable values [72].
- It is easier to increase and maximize block cipher throughput than to reduce its latency [26].
- For some block cipher such as Rijndael AES, encryption process is faster than decryption process.

Figure 27 illustrates the principle of *ciphertext speculation*. To enable the prediction, a few new microarchitectural functional blocks are introduced including a frequent value table (FVT), Pipelined Encryption Engines, and an Encrypted Value Content-Addressable Memory (CAM). The FVT keeps the top N most frequently used data values managed with an LRU policy. In our experiments, N is either 8, 16 or 32. The ciphertext speculation mechanism is illustrated in the timeline given in Figure 28. When data miss the on-die L2 cache and needs to be fetched from memory, a fetch request is issued to the memory controller and at the same time, a request is also posted to the ciphertext speculation engine (① in Figure 27). The engine will read all the frequent values from the FVT, and send the data together with the fetch address to the encryption engine (②), which will encrypt them and store the resulting ciphertexts in the CAM (③). Note that the encryption engine is pipelined and replicated, thus multiple encryptions can be performed concurrently to accelerate the value speculation. When the missed encrypted cache line arrives, it is compared against the ciphertext waiting in the CAM (④). If a match is found, the original frequent value that corresponds to the matched ciphertext is returned and no decryption

is needed (⑤ and ⑥). Observing the timeline we should note that increasing the size of the FVT does not necessarily lead to a better performance. Even though more hits in the CAM will occur when more frequent values are kept inside the FVT, however, it needs more time to encrypt these values, thus delaying the value speculation of subsequent misses when there are several back-to-back L2 misses. Ciphertext predication applies to memory encrypted using parallelizable encryption modes such as the one proposed by XOM [37].

In the next couple of sections, we consider two popular block cipher algorithms and discuss how our *ciphertext speculation* applies to each of them.

5.2.1 Triple-DES/DES

The Data Encryption Standard (DES) has been a symmetric encryption standard for two decades [48]. It encrypts a block of 64-bit message using a 56-bit secret key. In practice, a multiple encryption scheme based on DES, called Triple-DES is widely used [49] for improved security. Triple-DES applies DES three times to achieve better effective key length. Security of multiple encryption is proved in [45] and Triple-DES is the NIST approved replacement for DES in 1999. Applying *ciphertext speculation* to Triple-DES/DES is straightforward because the prediction is made on each 64-bit data block, the same size of Triple-DES/DES' blocks. For each 64-byte cache line, the line is divided into 8 64-bit chunks and each chunk is encrypted using parallelizable encryption modes with Triple-DES as the block cipher. The granularity of value prediction is made for every fetched 64-bit data. So, the size of each predicted ciphertext is also 64-bit. With pipelined design, all the 8 chunks of the same cache line can be encrypted or decrypted simultaneously.

5.2.2 AES

Advanced Encryption Standard (AES) encrypts a 128-bit block with variable key lengths of 128-bit, 196-bit, or 256-bit. Since value prediction is performed for every 64-bit data[1], to predict each 128-bit ciphertext, combination of frequent values has to be used. For example, if the number of 64-bit frequent/predictable values used is 8, then there are 64 possible

[1]Frequent value can also be dynamically tracked for 16 or 32-bit or even 128-bit values. But 64-bit is close to the block size of AES encryption (128-bit). Profiling results show that using 128-bit as units of frequent values generate poor frequent value profile.

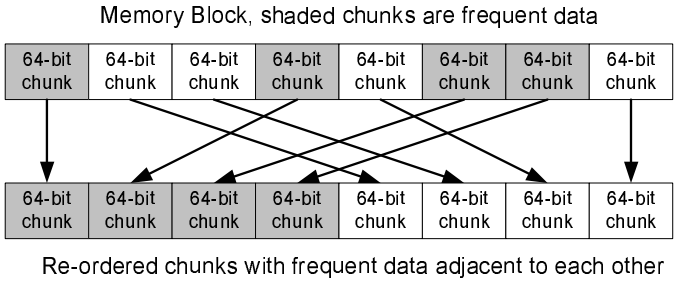


Figure 29: Data Chunk Re-ordering

combination of the eight frequent values, which gives 64 128-bit ciphertext speculations. Although such a prediction scheme may work, the prediction rate would drop because it requires both 64-bit values of each 128-bit chunk to be frequent or predictable. To solve this problem, we propose a data chunk re-ordering mechanism that re-orders the eight data chunks so that all chunks of more predictable values are grouped and encrypted together. Figure 29 illustrates how the data chunk re-ordering is performed for each cache line. For this example, there are eight 64-bit data chunks and four of them are frequent values (shaded boxes). If a prediction is made for every two chunks, none of the four 128-bit ciphertext can be predicted correctly. Nevertheless, if the chunks are re-ordered as shown below the original data block, two of the four 128-bit ciphertext will be predictable.

To restore decrypted chunks back to the original order, a secure processor has to maintain additional information. One choice is to store a bitmap of frequent values. For a 64-byte cache line, the bitmap requires 8 bits to encode whether any of the eight 64-bit chunk is a frequent value.

The bitmap associated with each memory block is encrypted also when stored in the external memory. Since the bitmap contains only 8 bits per cache line, so bitmaps of several memory blocks can be encrypted and stored together. Decrypted data chunks or correctly predicted data chunks cannot be used if the secure processor cannot determine their original order before the corresponding bitmap is decrypted. This performance problem can be tackled in three ways. First, a small cache can be applied to cache bitmaps in the secure processor. The overhead to cache bitmaps is very small. Considering a data TLB of 128 entries and 4KB page size, to cache the frequent value bitmaps for all the 128 pages requires

an 8KB cache. In fact, instead of requiring more power or space, such frequent value bitmap cache may save power or space. Since frequent values have less entropy, significant portion of a cache line may be turned off or put into a lower power state knowing that many of the chunks are frequent values. For example, given eight frequent values, if the frequent value bitmap indicates that all the chunks of a cache line are frequent values, it requires only 24 bits (3 bits x 8) instead of 512 bits (64 bits x 8) for storing the cache line. Second, bitmaps can be prefetched using simple hardware prefetch technique.

In summary, *ciphertext speculation* is practical for both AES based and Triple-DES based direct memory block encryption. For the AES based design, frequent value bitmap cache is preferred for better performance results. Considering that frequent value bitmap is also a compression technique, introducing such bitmap cache may actually make the secure processor more power efficient.

5.3 MAC Speculation

Similar to the ciphertext speculation, a secure processor can also speculatively compute MACs for frequent value chunks. This requires each cipher size chunk to have it own MAC. One performance advantage of assigning a MAC to each chunk is that it supports parallel authentication of each individual memory chunk. When combined with parallelizable encryption mode, a secure processor can decrypt and authenticate critical fetched chunk of a cache line first before its trailing chunks. As aforementioned, a simple and secure design is to have a secure processor to issue instructions and data only after they are authenticated. Such design will not have the risk of disclosing sensitive information due to maliciously altered instructions or data. To implement MAC speculation, a secure processor will compute along with each speculated ciphertext to speculate its corresponding MAC value. The speculated MAC will be matched against the MAC fetched from the memory. If both the speculated ciphertext and the speculated MAC match with the fetched ciphertext and the MAC, the secure processor knows that the fetched value is a frequent value and there is no additional decryption and MAC verification required. Given that a large percentage of fetched memory blocks contain frequent values, such technique can significantly reduce the

decryption and authentication overhead.

5.4 Security Analysis

It is important to point out that the proposed prediction technique is completely an architectural optimization. It does not change the security strength of the underlying encryption modes and the MAC integrity check schemes, nor does it modify the original encryption modes and the MAC schemes.

Some audience may have concerns that since there are so many frequent values in the applications, memory encryption may not be able to provide sufficient protection against statistical analysis of ciphertext. It is important to point out that proper application of encryption modes will even out the frequency distribution. In theory, once encrypted, the encrypted data will be evenly distributed, in other words, without any frequent value bias. Thus frequent values in plaintext do not manifest itself in ciphertext. To further clarify the confusion, even though our technique is named *ciphertext speculation*, by no means it implies that the ciphertext or the plaintext is predictable by adversaries. Predictions are made only by a secure processor within the secure boundary because the secure processor has the complete knowledge of necessary information such as a secret key used for encryption for creating ciphertext. Also note that we are not proposing any new security model or new secure processor design paradigm but simply provides architecture enhancementon top of existing security architectures to address the latency issues associated with memory decryption and authentication.

5.5 Comparison With OTP Prediction

Cipher text prediction is another prediction based technique to hide latency overhead of encrypted memory. It is based on the frequent value prediction [39, 72] and specifically designed for memory encryption that uses parallelizable direct memory encryption mode such as the well known OCB mode [54]. Different from the ciphertext prediction, the OTP prediction does not predict the ciphertext itself.

The commonality of both techniques is that, both of them use idle cycles of pipelined

Table 3: Compare counter prediction with Ciphertext speculation

Technique	Supported Mode	What is predicted	Source of predictability
Counter prediction & Decryption pad precomputing	counter-mode	sequence numbers/pads	predictability of memory update frequency
Ciphertext/MAC prediction	all modes	ciphtertext itself	frequent data chunk

Table 4: Processor model parameters

Parameters	Values
Frequency	1.0 GHz
Fetch/Decode width	8
Issue/Commit width	8
L1 I-Cache	DM, 8KB, 32B line
L1 D-Cache	DM, 8KB, 32B line
L2 Cache	4way, Unified, 64B line, write back cache 256KB and 1MB
L1/L2 Latency	1 cycle / 4 cycles (256KB), 8 cycles (1MB)
Memory Bus	200MHz, 8B wide
Memory Latency	X-5-5-5 (core clocks) X depends on page status
CAS latency	20 mem bus clocks
Precharge latency (RP)	7 mem bus clocks
RAS-to-CAS (RCD) latency	7 mem bus clocks
Triple-DES latency	96ns, 48 stages
AES latency	80ns

crypto engines to speculatively compute either pads (*the OTP prediciton*) or ciphertexts (*the ciphertext prediction*) themselves that can hide decryption latency of encrypted memory. But the two techniques also differ from each other significantly. Different from the ciphertext prediction, the *OTP prediction* does not predict the ciphertext itself. Table 3 shows some difference between the *OTP prediction* and the ciphertext prediction.

5.6 Simulation Framework

Our simulation framework is based on SimpleScalar [13] running SPEC2000 INT and FP Alpha binaries compiled with -O3 option. We implemented architecture support for dynamic frequent value tracking [72] and our *ciphertext speculation* scheme. The architectural parameters used for performance evaluation are listed in Table 5.6. Each benchmark is fast-forwarded and simulated at representative places according to SimPoint [56] for 400M instructions in performance mode. During the fast-forwarding, the L1 caches, the L2 cache, and the frequent value tracking are simulated. We subset the simulations for those with high L2 misses and memory throughput requirements.

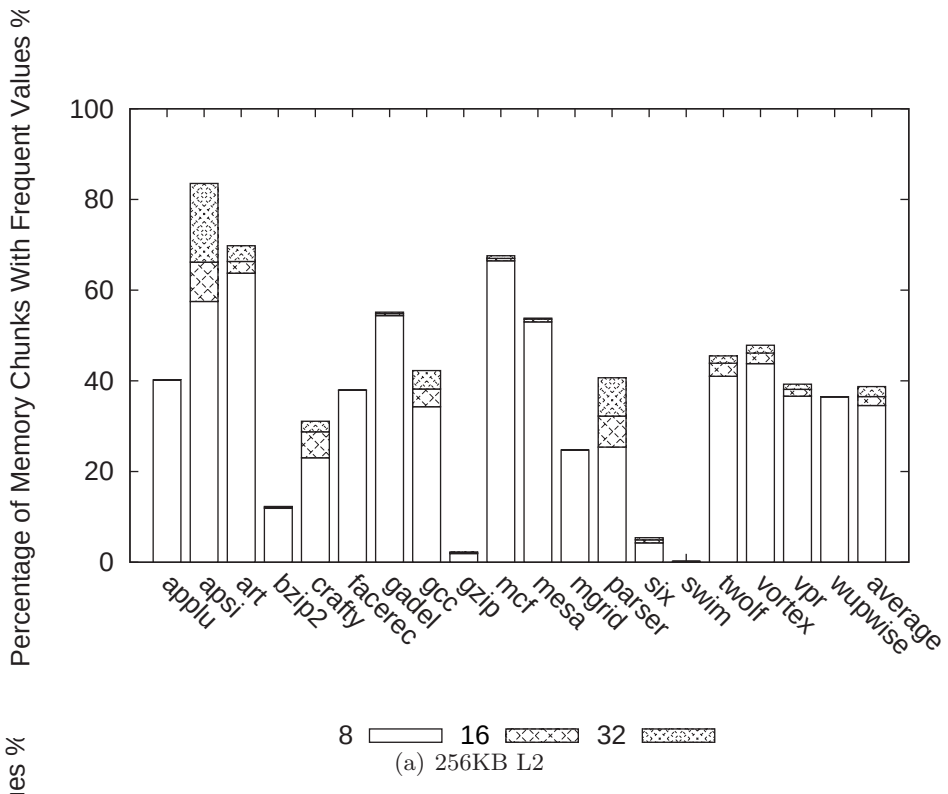
5.7 Performance Analysis

In this section we summarize and analyze performance results of our *ciphertext speculation* technique. The results are collected for two L2 cache settings, 256KB and 1MB. The choice of a smaller L2 cache is critical for our analysis, because protection on data confidentiality is not only deemed for high-end systems but commodity platforms as well. Note that a very large L2 size for SPEC2000 may be inappropriate in evaluating our technique since the entire working set of most SPEC2000 benchmarks can easily fit into a very large L2.

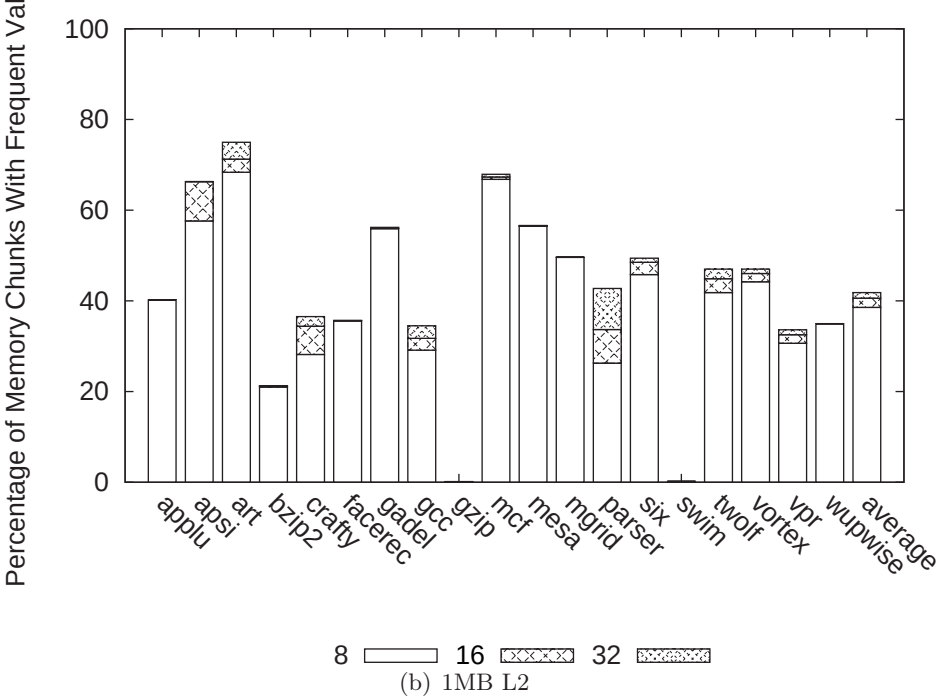
5.7.1 Frequent Values

Figure 30 shows the value predictability of each 64-bit memory chunk fetched due to L2 misses using 8, 16, and 32 frequent values. For several benchmarks, the prediction rate on average is over 40% or higher. Some benchmarks such as 164.gzip shows poor prediction rate. Note that the relationship between ciphertext predictability and IPC is rather complicated and by no means proportional to the ciphertex predictability. It is not guaranteed that every 64-bit data chunk fetched to L2 will be accessed because memory fetch is based on cache line size. For example, a piece of predictable data might be fetched into the processor because data adjacent to it mapped to the same cache line is accessed by the processor. Despite being predictable, the predictability of this un-used data may contribute little to the performance.

Figure 31 shows the advantages of using data chunk re-ordering for frequent value prediction. When the hardware uses a block cipher whose encryption size is 128-bit and one 64-bit frequent value is encrypted together with a 64-bit non-frequent value, the predictability is lost. Since a memory line often contains several frequent value data chunks, the hardware will re-arrange the data chunks so that frequent value data chunks are adjacent to each other. If a block cipher encrypts two 64-bit frequent value data chunks, both of them can be predicted. Figure 31 compares the ratios of predictable data chunks out of the total number of frequent value data chunks with and without the data chunk re-arrangement under 128-bit encryption unit size. According to the figures, without any re-arrangement, few 64-bit frequent value data chunks can be predicted under 128-bit block

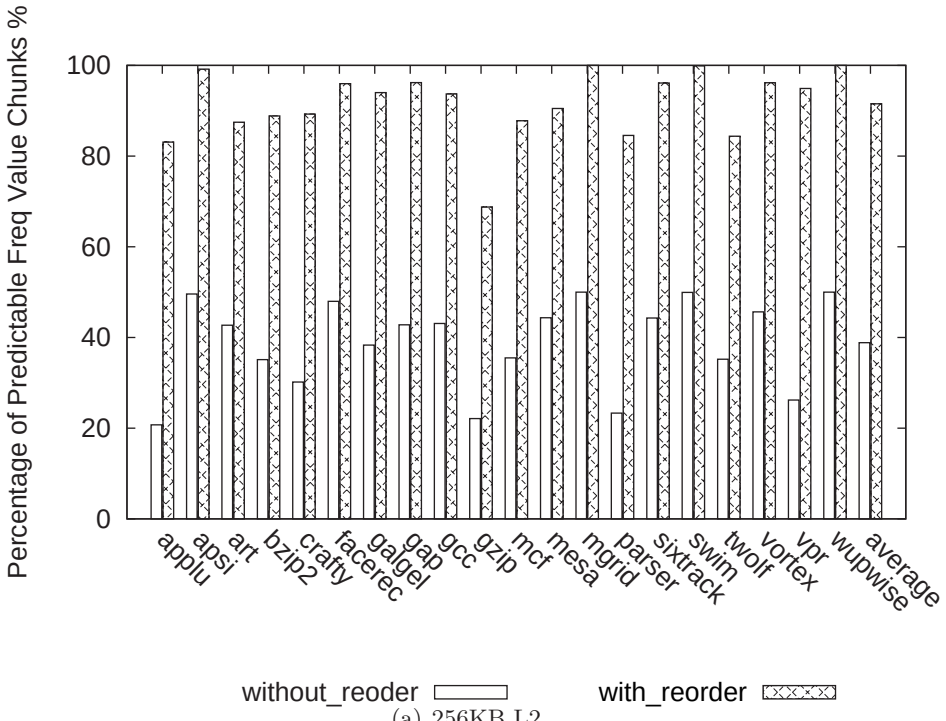


(a) 256KB L2

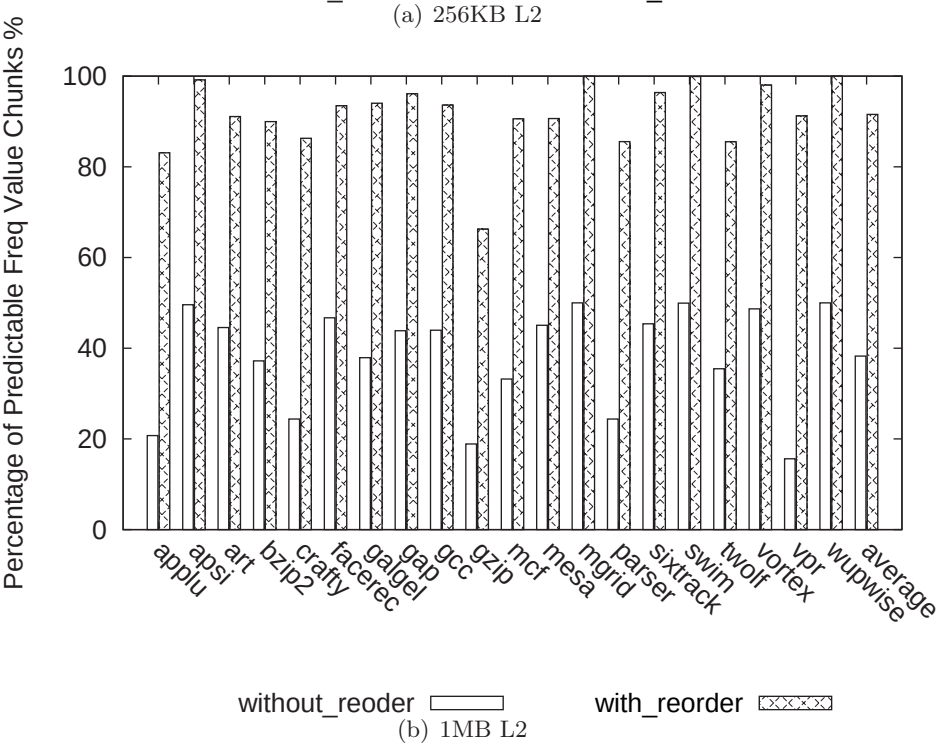


(b) 1MB L2

Figure 30: Percentage of frequent value memory chunks by keeping top 8, 16, 32 64-bit frequent values



(a) 256KB L2



(b) 1MB L2

Figure 31: Percentage of Predictable Data Chunks Over All Frequent Value Data Chunks Under 128-bit Block Cipher

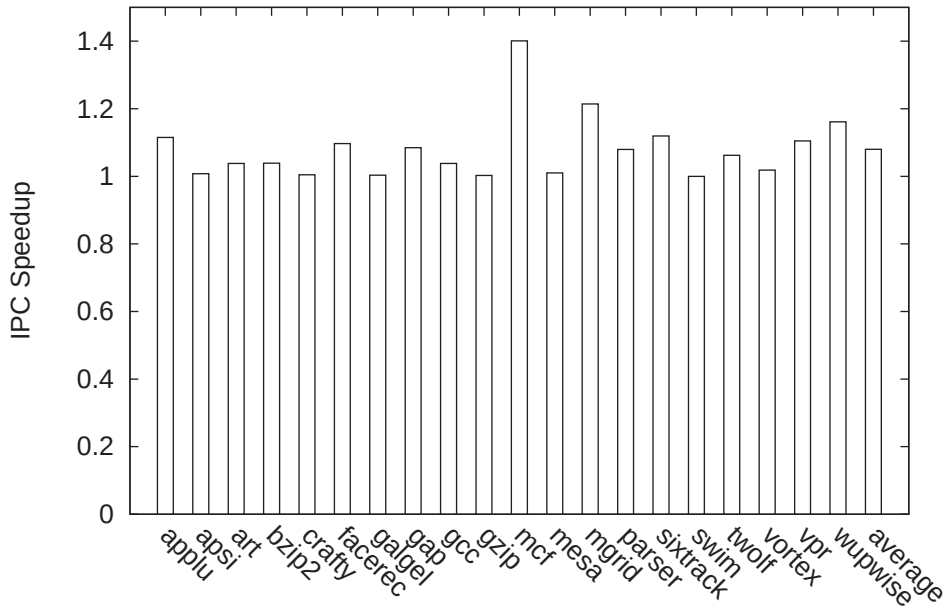
cipher. But when the re-arrangement is applied, over 95% 64-bit frequent value data chunks are predictable.

5.7.2 Ciphertext and MAC Speculation

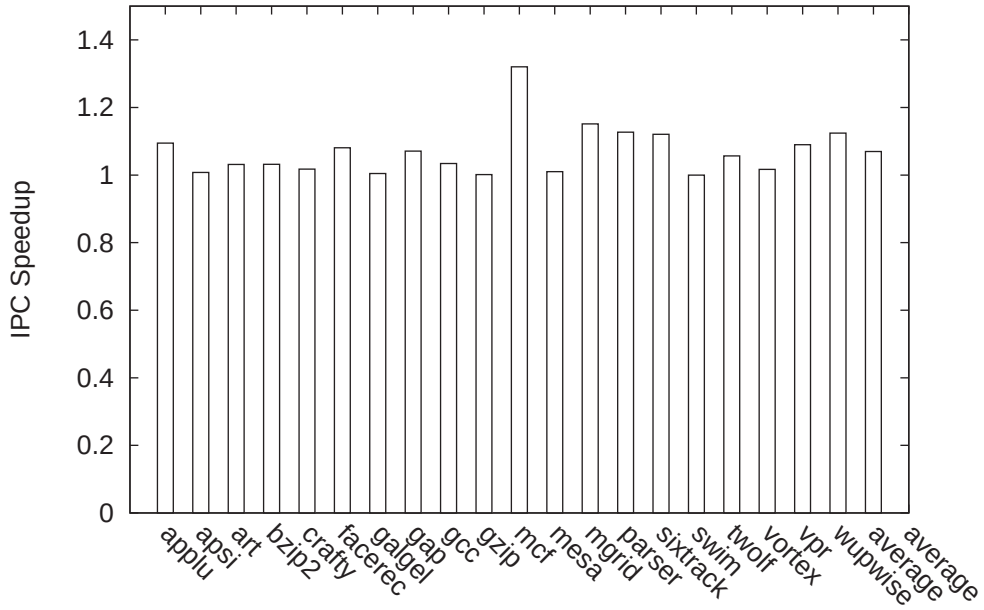
Under parallelizable encryption modes, *ciphertext and MAC speculation* can be performed independently for each chunk of a cache line. In this section, we investigate the performance impact of *ciphertext and MAC speculation*. For TDES, a prediction is made on per 64-bit memory chunk of a cache line; for AES, each prediction is made on per 128-bit memory chunk using memory chunk re-ordering. We use a set of 8 64-bit frequent memory chunks dynamically tracked with an approach similar to that in [72].

The performance improvement of *ciphertext and MAC speculation* is shown in figure 32 and figure 33 for TDES and AES, both directly encrypt memory. Apparently, *frequent value based speculation* improves performance for most benchmarks under both cipher conditions. For some memory-bound applications such as 181.mcf, 175.vpr, 172.mgrid, the improvement is more significant. For about eight benchmarks, the IPC speedup is over 10% for both TDES and AES. In general, using 64-bit speculation chunk achieves more performance improvement than the 128-bit speculation chunk. This is because 128-bit based speculation generates more speculations and is more likely to saturate the crypto engine with speculative encryption requests. When the L2 is increased to 1MB, the IPC improvement, as expected, decreases. But still for six benchmarks, the speedup is at least 10% and some of them attain almost 30%. Note that when the cache size is increased to 1MB, most of the memory bound SPEC2000 benchmark programs are no longer memory bound.

Figure 34 studies the performance impact of *MAC speculation* on memory encryption using the counter mode encryption. Unlike direct encryption modes, the counter mode receives less benefit from *ciphertext speculation*. That is because in the counter mode, pre-computation of decryption pads is the dominating factor of decryption performance. But the counter mode can still take advantage of frequent value based *MAC speculation* by significantly reducing authentication latency. The results indicate that many benchmark programs achieve performance increase of more than 5% and several of them have

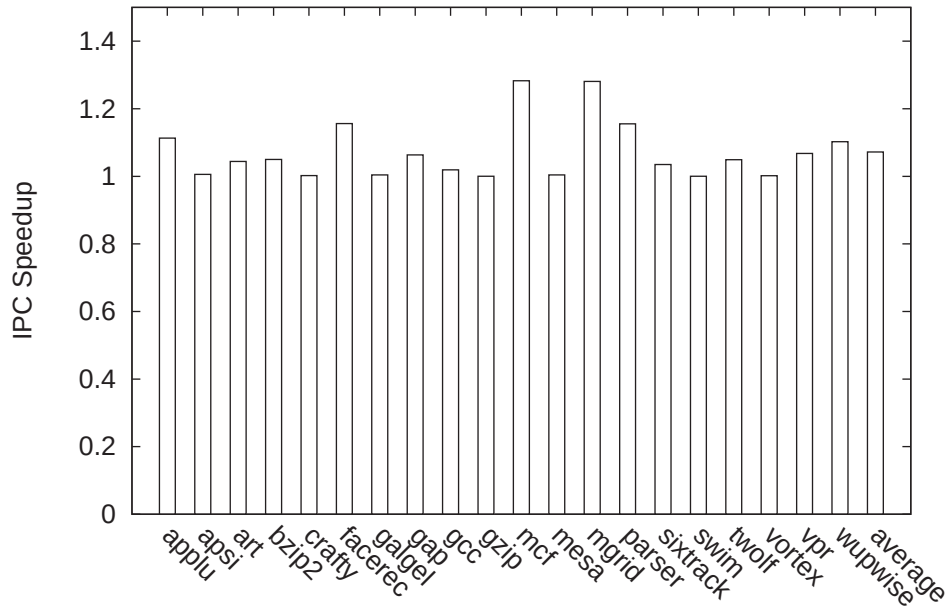


(a) TDES, 256KB L2

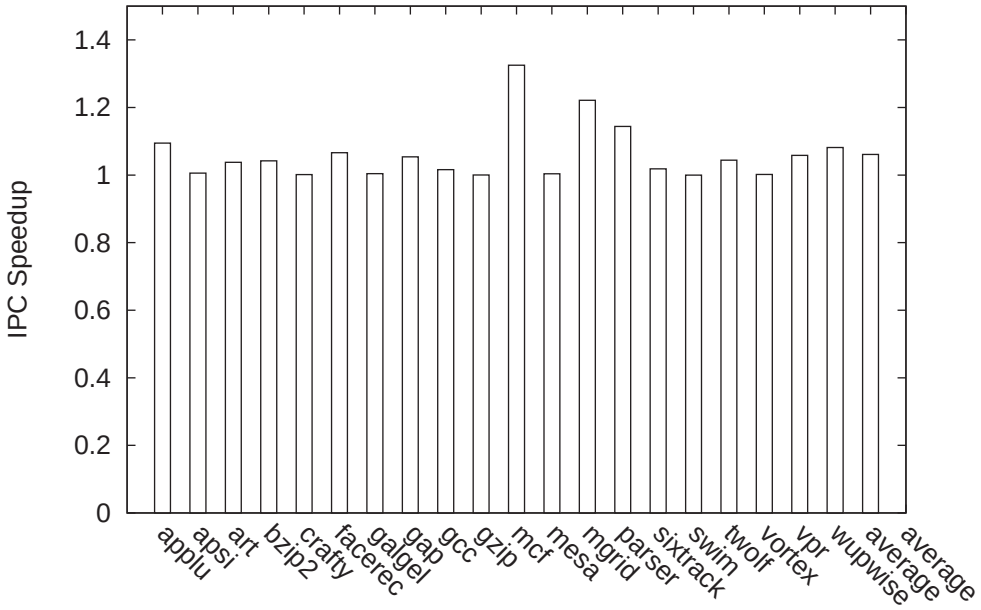


(b) AES, 256KB L2

Figure 32: IPC Speedup Using Ciphertext and MAC Speculation For Direct Memory Encryption, 256K L2

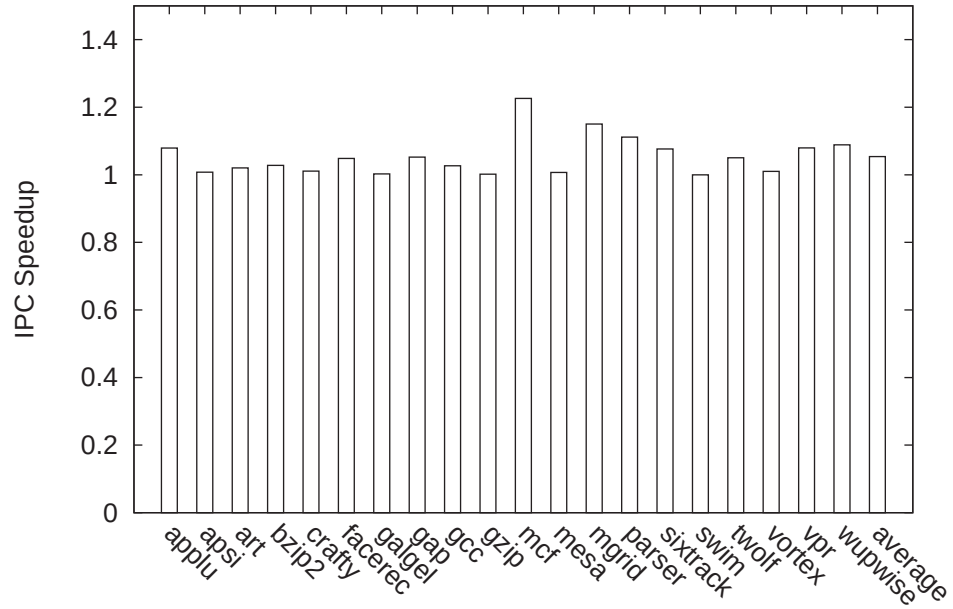


(a) TDES, 1MB L2

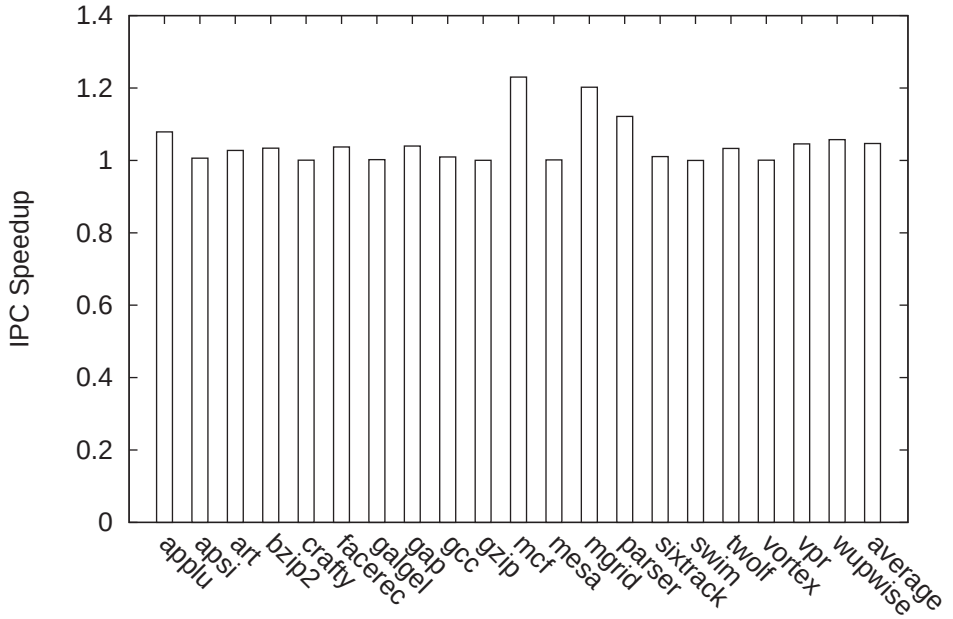


(b) AES, 1MB L2

Figure 33: IPC Speedup Using Ciphertext and MAC Speculation For Direct Memory Encryption, 1M L2



(a) 256KB L2



(b) 1MB L2

Figure 34: IPC Speedup Using Ciphertext and MAC Speculation For Counter Mode)

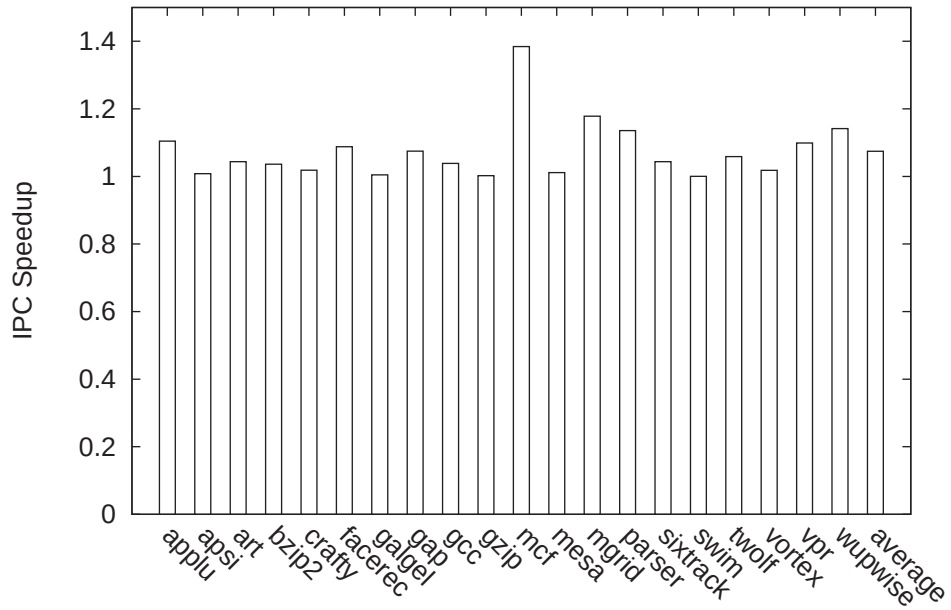
IPC speedup over 10%. Under 1M L2 size, some benchmarks such as 181.mcf, 172.mgrid, 197.parser also attain significant IPC speedup from 15% to 20%.

5.7.3 Sensitivity of Memory Latency

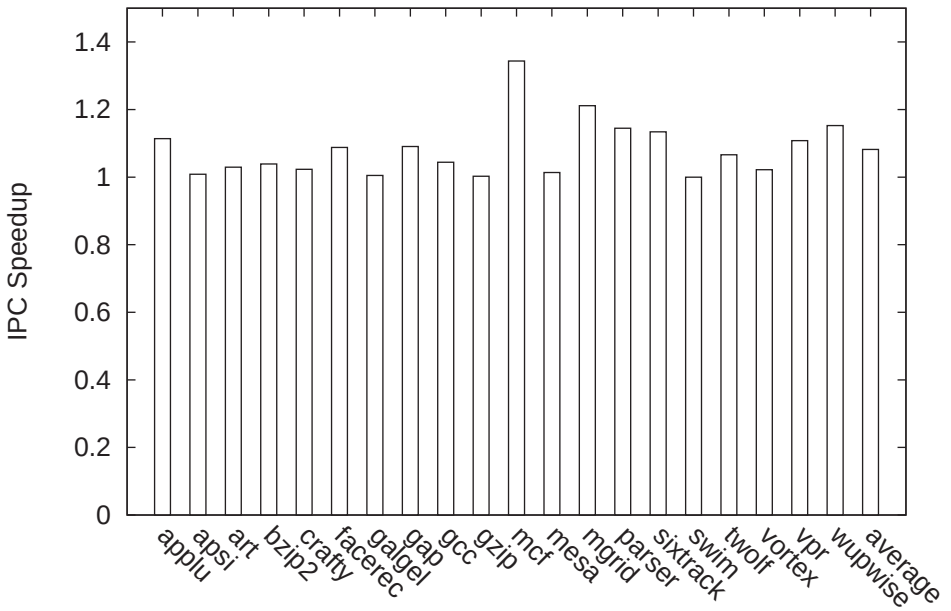
Essentially, frequent value based ciphertext and MAC speculation is a latency hiding technique. In this subsection, we evaluate its effectiveness under different settings of memory speed. Since our simulation is based on detailed SDRAM model, there is no single absolute memory fetch latency. The relative speed between CPU and memory is captured by the CPU-to-memory clock ratio. We experimented three different CPU memory clock ratio settings. They are 1:4, 1:5, and 1:6. Higher ratios infer longer memory fetch latencies. Figure 32 and figure 33 show the results under 1:5 ratio. Figure 35 shows the impact of memory latency on the effectiveness of ciphertext and MAC speculation under 1:4 and 1:6 CPU-to-memory clock ratios. First, the results indicate that ciphertext and MAC speculation is quite effective under all the three settings. For each benchmark, the differences of IPC improvement rates under the three settings are relatively insignificant. This shows robustness of the proposed frequent value based prediction technique. Second, the results show that on average ciphertext speculation improves IPC more under larger CPU memory clock ratios. The average IPC speedup of all the benchmarks under 1:4 ratio is about 7.5% and the average IPC speedup of all the benchmarks under 1:6 ratio is close to 9%.

5.7.4 Number of Frequent Values

To evaluate the performance sensitivity with respect to the size of the Frequent Value Table (FVT), we increased the number of frequent values, i.e. the number of entries, stored inside the FVT from 8 to 16 and 32. Ideally, more entries should produce more predictions of the missed ciphertext. As explained earlier, however, they can also throttle the performance due to much more encryption work that needs to be done for each L2 miss when a series of back-to-back L2 misses occur. Figure 36 shows the IPC results for TDES encrypted memory. As indicated, there is little improvement with more entries in the FVT In some cases, the performance was even reduced a little bit because the number of predictions is too large. The reason is that even though adding more predictions improve prediction rate, but



(a) CPU-to-Memory Ratio 1:4



(b) CPU-to-Memory Ratio 1:6

Figure 35: Effect of Memory Speed Relative to CPU Clock Speed, 256KB L2

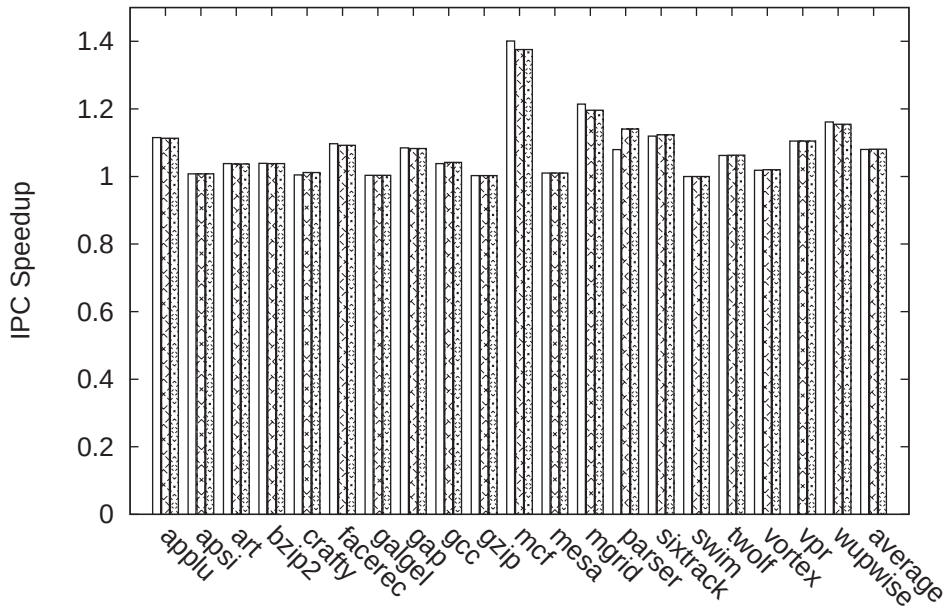


Figure 36: Effect of number of guesses/per chunk on performance with TDES encrypted memory, 256K L2

at the cost of increasing workload on the speculative encryption engine, which potentially increases the latency to generate encrypted frequent values for the succeeding misses. For AES based scheme, the cost and workload to use more frequent values is higher than TDES based scheme because it uses combination of frequent values. Since increasing the number of frequent values and predictions does not have a significant performance advantage, we did not evaluate its effect on the AES based scheme.

5.8 Conclusion

Minimizing the latency overhead of memory decryption and integrity verification is a crucial issue for designing a high performance secure processor. We propose novel latency-hiding techniques — *frequent value ciphertext speculation* and *frequent value MAC speculation* to hide decryption and MAC authentication latency. *Ciphertext speculation* reduces or eliminates the decryption latency by speculatively encrypting frequent values and matching their ciphertext results with the fetched one. Our simulation profile indicates that on average over 40% fetched data values are frequent values. By exploiting these properties, the decryption latency for secure processors using direct memory encryption modes can be substantially

reduced. We also propose *MAC speculation* which pre-computes MAC for frequent values and can accelerate authentication process by comparing the speculated MAC with the corresponding MAC fetched from the memory. *MAC speculation* improves performance for all memory encryption schemes including the counter mode security architecture that supports parallel MAC verification. As shown in our experiments, memory bound benchmark programs show significant IPC improvements using *ciphertext speculation* and *MAC speculaiton* with speedup ranging from 10% to 30%.

CHAPTER VI

EFFICIENT MEMORY INTEGRITY VERIFICATION

For guaranteeing a tamper-proof computing environment, any unauthorized modification attempt to the applications running on the system must be detected via a robust integrity verification mechanism. In addition to its effectiveness, the hardware support for a tamper-proof computing environment needs to be transparent to the users. Toward this goal, we propose a scheme called *MACTree integrity protection*. We describe the integrity tree construction and analyze its security and efficiency in this chapter.

6.1 MACTree Construction

The proposed integrity checking scheme, *MACTree*, uses 32-bit Message Authentication Code (MAC) nodes to construct the m-ary hash tree used in prior art. Note that a conventional MAC requires at least 128-bit to ensure a sufficient security level. However, using 128-bit MAC nodes suffers a substantial performance loss due to a large number of hash computations and storage overhead. Therefore, MACTree has to be signed carefully to attain the best balance between performance and security.

A MACTree is constructed by the following steps.

- A 32-bit MAC value for each cache line is generated as shown in Figure 37. First of all, an initial 256-bit MAC value is generated using the SHA-256 hash function [50] by concatenating the cache line data, its virtual address, and the secret key of the application as inputs. A new 32-bit MAC is then computed by XORing the eight evenly-divided chunks;

- All 32-bit MACs form one layer of nodes in the MACTree and are stored linearly as shown in the left-hand side of Figure 38. For this layer, a new MAC line is made by concatenating (m-1) consecutive MACs together and padded with a 32-bit *Random*

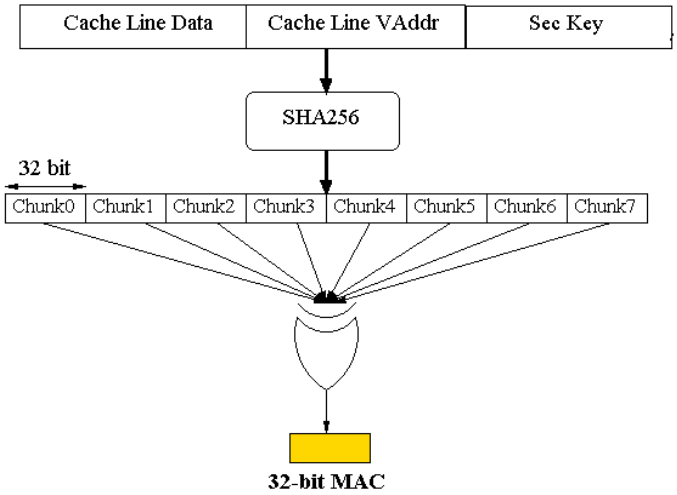


Figure 37: 32-bit MAC Generation for a Cache Line

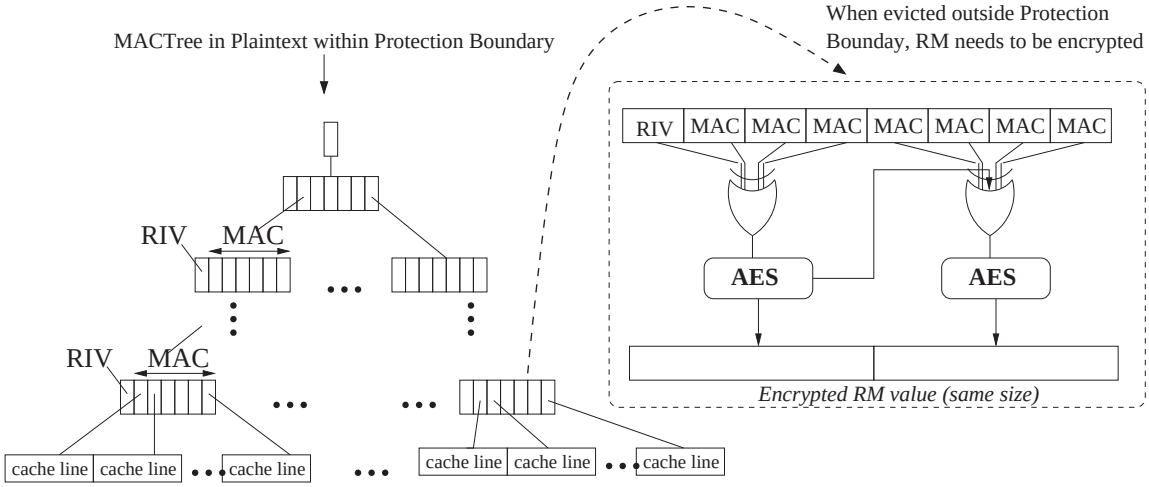


Figure 38: Structure of the MACTree in plaintext within protection boundary. Note that a hash value needs to be encrypted when being evicted out of the protection domain.

Initial Value (RIV) which is generated by a random number generator using thermal noise in the processor core [29].

- Similar to the method described in Figure 37, a new 32-bit MAC value for the next level in the MACTree is computed by concatenating the new MAC line and the secret key of the application as the inputs to the SHA-256 function.

- Repeat the last two steps until a root MAC is generated.

Whenever a MAC line is to be cast out of protection boundary, the MAC line is encrypted by the AES block cipher as illustrated in the dashed box of Figure 38. Note that the *root*

MAC is always kept inside the processor once the program enters the security environment to avoid any potential tampering.

6.2 Security Analysis

The CHTree scheme [23] in AEGIS was designed for protecting the integrity of an application by constructing an m-ary hash tree. The integrity check of a cache line using the hash tree costs $log_m(L)$ hashing computations, assuming there are L data cache lines in the application. The latency overhead is substantial according to simulation results. MACTree scheme differs from the CHTree scheme in three aspects: 1)32-bit MAC values are used instead of 128-bit hash values; 2)The MACtree is encrypted using the secret key of a given application; 3) Each MAC line is computed from its (m-1) children lines with a random initial value generated randomly by circuits. These features reinforce and guarantee the security of the proposed MACTree scheme albeit the MAC's shorter bit length of 32.

Using our proposed MAC instead of simple hash values can prevent an adversary from producing a collision using independent computing devices. For instance, with a 32-bit hash value, an adversary can combine arbitrary instructions into a cache line to generate a matched 32-bit hash value. This task can be performed in any computer as computing hash values does not need any assistance from the victim computer. When the same hash value of a target instruction cache line is found with the worst case of 2^{32} tries [3], the adversary can replace the line with his fabricated instructions. Now, the integrity check will fail to detect such an attack. MACTree does not suffer from such an attack because there is a secret key involved in deriving the MAC value. This secret key is processor and application specific.

An adversary has to launch brute force crack on the victim machine. This is one of the major differences between MACTree and CHTree in terms of security strength. An adversary could modify cache line data of a victim processor directly without touching the MACTree. But the chance for this activity to be undetected by the processor architecture is 2^{-32}. To enhance security, the processor of MACTree architecture will stop the execution of the application once an integrity verification failed, prohibiting the adversary from further

attempted attacks. Moreover, the security can introduce artificial small amount of delay between integrity verification failure and resumed execution when the number of integrity verification failures is frequent. Albeit the size of MAC is only 32-bit, if only one minute minimal resume latency is introduced for frequent integrity failures under brute force crack, it would take hundreds of years for an adversary to comprise MACTree protection, which is sufficient in terms of security strength.

One possible threat of using 32-bit MACs is the higher probability of aliasing. That is, given two cache lines, the possibility that they have the same MAC value is 2^{-32}. Although the probability of aliasing appears to be small, it might become a loophole when a huge number of cache lines and their MACs are generated and compared by the adversary given he can access data and the associated MACs in the external memory. The MACTree prevents this type of attacks by associating an RIV with each MACTree line and encrypting the line with the application's secret key. Since the MACTree is encrypted, an adversary will be unable to tell whether their plaintext MACs are the same.

6.2.0.1 Efficiency Analysis

Here we analyze the advantage of the MACTree over the CHTree. First of all, a 256-bit cache line can hold 7 nodes in the MACTree versus 2 nodes in the CHTree. Hence the height of the tree is reduced to $log_7 L$ nodes from $log_2 L$, or a 180% reduction of the number of nodes needing to be examined. Nonetheless, the MACTree incurs decryption overheads caused by encryption. Assume that the decryption delay is T_d and the memory access delay is T_m. The overhead caused by memory access and decryption for integrity check is $log_7 L \times (T_d + T_m)$ in the MACTree against $log_2 L \times T_m$ in the CHTree scheme. Let $T_d = \frac{T_m}{2}$, similar to the assumption used in other secure processor architectures, the MACTree has 47% less overhead than the CHTree. Even when $T_d = T_m$, the MACTree has 35% less overhead. The MACTree scheme can be further improved by caching nodes within the protected boundary. With the same cache size, the MACTree can hold 3.5 times more nodes.

Table 5: Processor Model Parameters

Parameters	Values
Frequency	1.0 GHz
L1 I/D Cache	DM, 8KB, 32B line
L2 Cache	4W, Unified, 32B line, 256KB, 0.5MB, 1MB, 2MB
L1 Latency	1 cycle
L2 Latency 256KB/512KB/1MB/2MB	6/8/10/12 cycles
Memory Latency	X-5-5-5 (cpu cycles) X depends on mem page status
Memory Bus	200 MHz, 8B wide
Fetch/Decode Width	8
Issue/Commit Width	8
Branch Predictor	Perfect
LSQ/RUU Size	64/128
AES/SHA-256 Latency	80ns/80ns
MAC Length	32 bits
RIV/MAC	64 bits
RM Cache	4W, 8KB, 32B line
MAC Cache	4W, 8KB, 32B line
RM Cache Lat.	6 cycles
MAC Cache Lat.	6 cycles

6.3 Performance Analysis

6.3.1 Memory overhead

Integrity checking schemes need additional memory space to store the checksums. In the MACTree scheme, the additional memory space needed is approximately $\frac{1}{(m-1)}$ of the data memory space with an m-ary balanced MACTree. As we use a 256-bit cache line and a 32-bit MAC value, the memory overhead is about 15%. In contrast, a 128-bit hash value used in the CHTree scheme will result in a 100% memory overhead. The M-TREE encryption scheme uses a 64-bit RM value for each cache line, leading to 25% memory overhead for encryption, larger than 12% in a direct block cipher that uses 32-bit random initial values, but can achieve a better performance.

6.3.2 Simulation framework

Our simulation work is based on SimpleScalar [6] running Alpha binaries compiled with -03 option. Each benchmark is fast-forwarded according to SimPoint's suggestion [56]. During fast-forwarding, L1 and L2 caches were warmed up. table 11 summaries the architectural and microarchitectural parameters. Both SPEC2000 INT and FP benchmarks were used for our evaluation.

6.3.3 Unverified Instruction Speculation

Speculative execution can be employed to further improve the performance for the MACTREE secure processors. The results of unverified instructions can be speculatively consumed by dependent instructions without stalling the pipeline. However, all the unverified instructions and their dependent instructions must not be committed before the corresponding integrity checks are completed. The Reorder Buffer can be used to satisfy this need with a minor modification which adds the verification completion as a condition for retirement. In other words, in addition to branches, the machine states induced by instructions with pending integrity checks are also considered speculative.

6.3.4 Performance Results

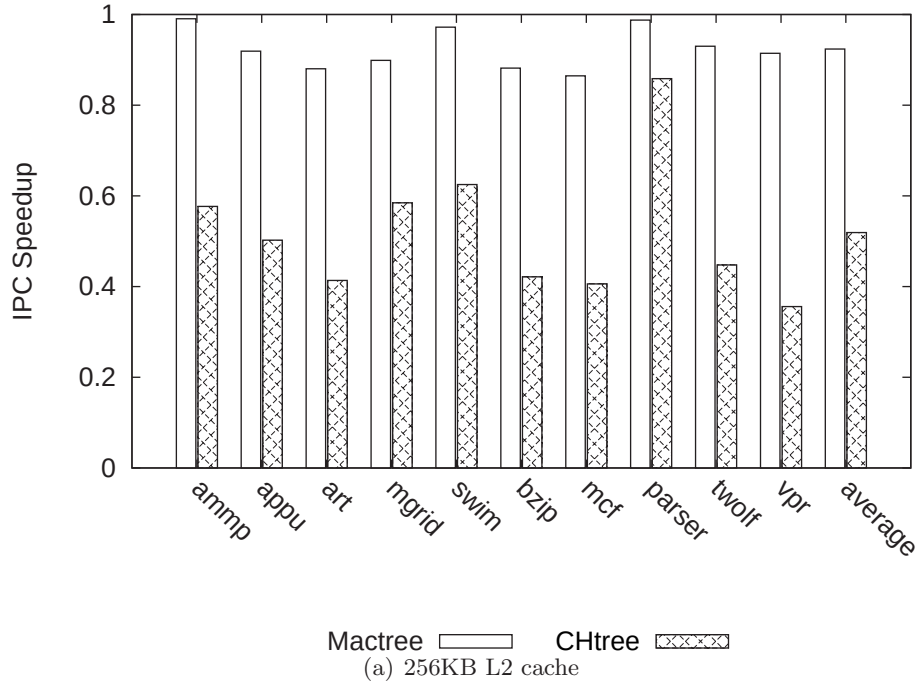
In this subsection, we briefly summarized the performance results.

6.3.4.1 Performance comparison with the CHTree

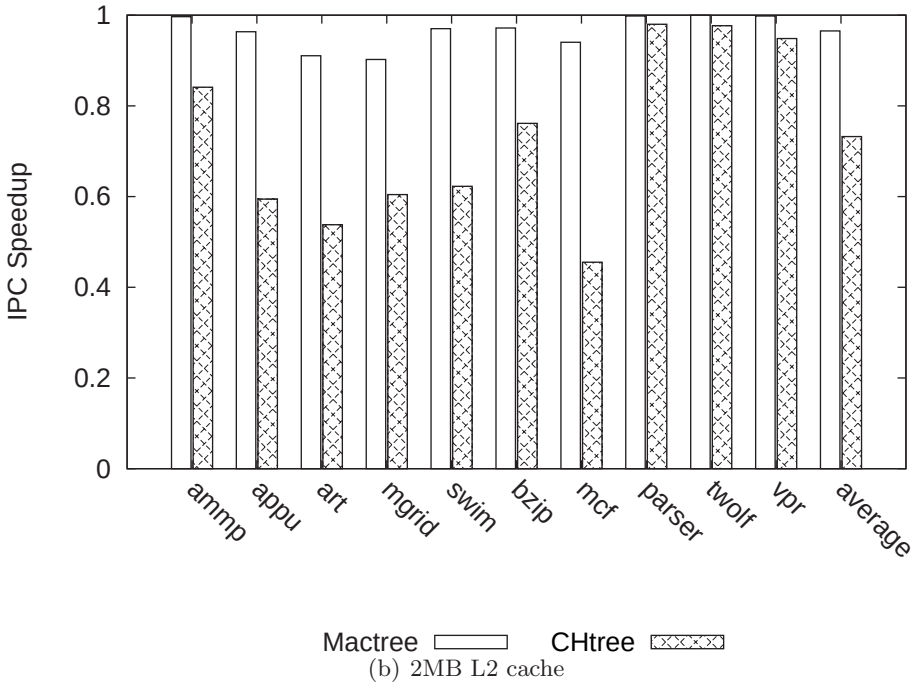
Figure 39 compares the performance of the MACTree and the CHTree under two different L2 cache sizes. The IPC numbers were normalized to the baseline[1]. The results clearly show the performance advantage of the MACTree scheme over the CHTree. The performance overhead over the baseline for the MACtree scheme is 8% on average and up to 14% in the worst case, while the CHTree scheme has 50% slowdown on average and as much as 60% overhead in the worst case with a 256KB L2 cache. Even with a 2MB L2, the performance degradation of the CHTree is reduced to 35% in the worst case and 11% on average, still less efficient than the MACTree scheme which shows an overhead of 5% on average and 10% in the worst case.

The performance advantage of the MACTree is twofold. As shown in Figure 40(a), with an 8K MAC cache and a 256KB L2 cache, the MACTree has a much less number of MAC cache accesses than the CHTree. It can be seen that the number of accesses to the MAC cache in the MACTree are reduced to less than half of those in the CHTree for all cases. On the other hand, since the same cache capacity will hold more nodes for the MACTree,

[1]Baseline has the same configuration with no security feature.

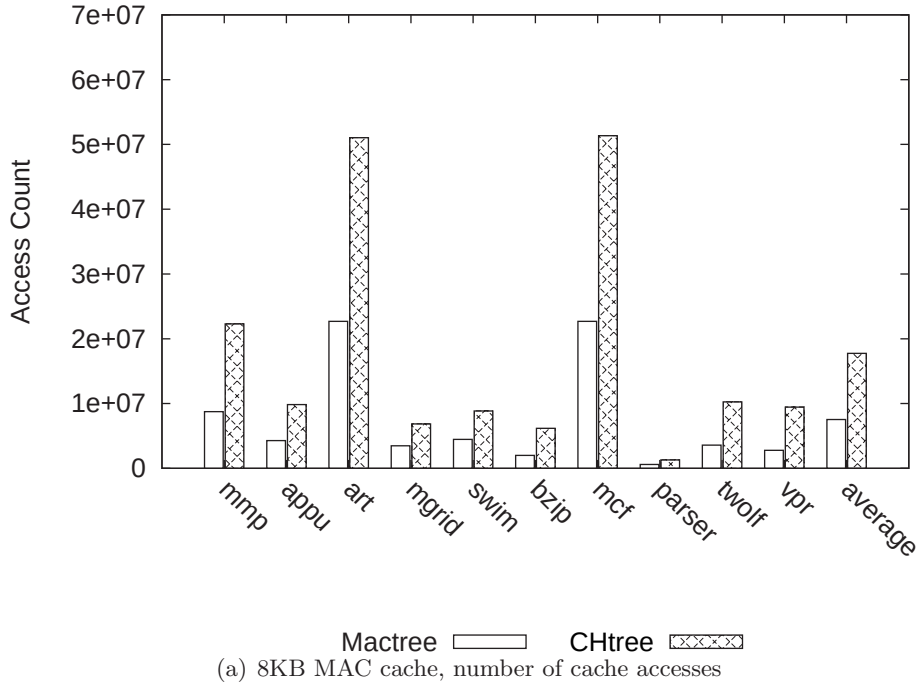


(a) 256KB L2 cache

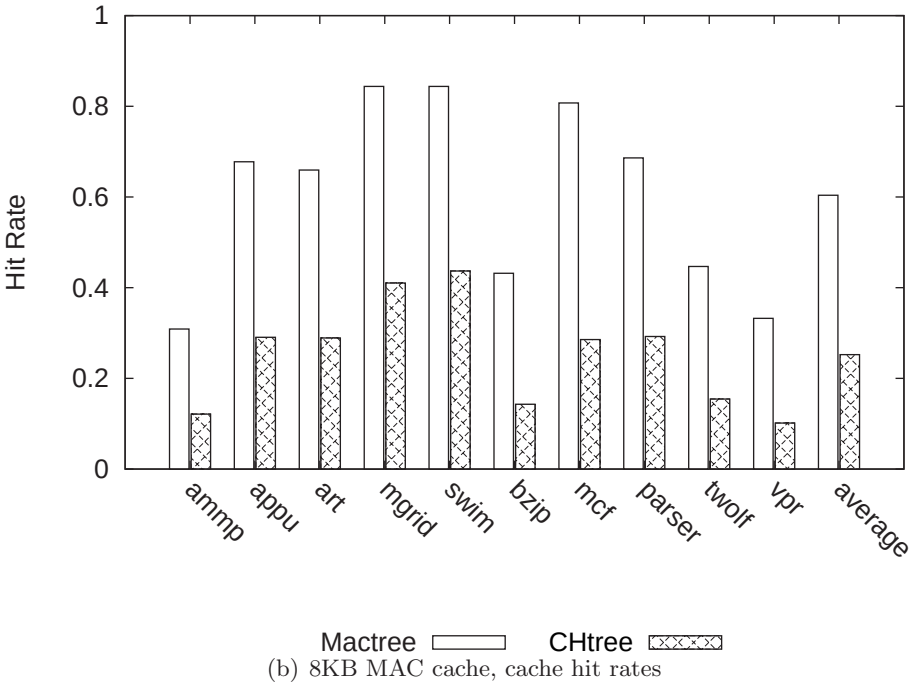


(b) 2MB L2 cache

Figure 39: Normalized IPC for MACTree and CHTree with 8K MAC cache



(a) 8KB MAC cache, number of cache accesses



(b) 8KB MAC cache, cache hit rates

Figure 40: MAC cache accesses/hit rates (256KB L2)

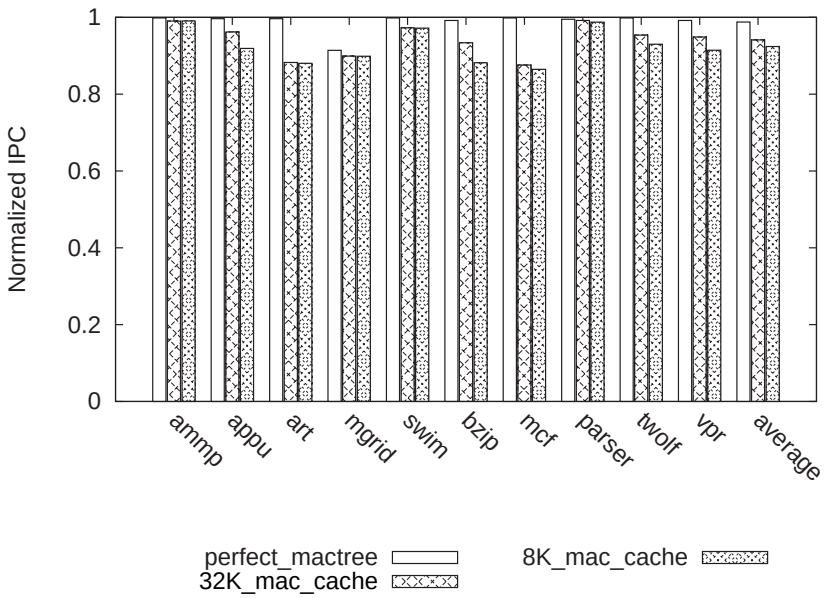


Figure 41: Performance sensitivity of MAC cache sizes of MACTree (256KB L2)

the cache hit rate is also significantly higher as shown in Figure 40(b).

6.3.4.2 Performance sensitivity of the MAC cache size

Now we investigate the performance sensitivity of using different MAC cache sizes for the MACTree. The normalized IPC results are shown in Figure 41 for four difference MAC caches including a perfect MAC cache, a transparent security support representing the baseline. As shown, even for a MAC cache as small as 8KB, the performance with integrity check can approach closely to the baseline. The 8-32KB MAC cache can hold a sufficient number of nodes for the MACTree in order to exploit temporal locality and thus achieve a high hit rate.

6.4 Conclusions

In this chapter, we described a MACTree processor architecture that implements integrity protection for providing a tamper-resistant and tamper-evident computing environment. We analyzed security and efficiency of the proposed integrity checking scheme and against the existing techniques and show that the MACTree architecture offers a significant performance improvement as well as security advantages over the prior art. Based on our simulation results, it is shown that the MACTree scheme suffers much less performance degradation

in data integrity verification than the previously proposed CHTree scheme. Our worst case overhead of 14% is substantially lower than the worst case overhead of 60% reported in the published literature. In conclusion, the MACTree integrity verification design offers several performance advantages over the existing secure processor architectures while retaining a high level of security for protecting applications. In the next chapter, we will continue to discuss how to combine integrity checking logic with decryption logic both efficiently and securely in secure processor architecture.

CHAPTER VII

DECOUPLING INTEGRITY VERIFICATION AND DECRYPTION

It has been known to the security community that memory fetch is an information leakage channel that if exploited by the hackers can potentially disclose sensitive information and data. At the very beginning, the presence of such an information disclose channel in secure processor does not seem to impose an imminent or significant threat [37]. At least, the potential disclose of sensitive information through program control flow that can be partially recovered through tracking of memory fetch is addressable in accordance with large body of compiler research that use static analysis and strong typed flow safe langauge to mitigate the problem [16, 47]. However, in-depth study shows another picture. Improper disassociation of decryption and authentication imposes a far more greater threat to software and data confidentiality. Such disassociation potentially allows unverified or unauthenticated programs to be speculatively executed and allows them to leave memory footprints. It hands to the hackers a valuable tool for them to tamper the protected data or software and disclose whatever information at their wish through the memory fetch side channel. Static compiler analysis and safe language provide virtually no help to counter this exploit because it tampers binary code and occurs at runtime.

In this chapter, we explore the design space of decryption and authentication disassociation with the objective to find a solution that is secure, fast, and simple to implement. In terms of how to integrate authentication and decryption into a modern out-of-order processor pipeline, we study both the trade-off between the security and the complexity, and the trade-off between the security and the performance. We investigate and evaluate a range of designs from *authentication-then-issue*, *authentication-then-commit*, *authentication-then-write*, and *authentication-then-fetch*. Under *authentication-then-issue*, a secure processor does not issue instructions whose integrity has not been fully verified. In addition,

authentication-then-issue requires that fetched data be issued as operand only after the secure processor has authenticated their integrity. *Authentication-then-issue* is a conservative solution that allows almost no decryption and authentication disassociation and has negligible design complexity. The downside is that it may incur significant performance overhead. *Authentication-then-commit* is a straight-forward solution that speculatively issues unverified instructions and data to the processor pipeline and commits the finished instructions only after both the instruction itself and its operands are authenticated. Though seemingly secure, *authentication-then-commit* leaves significant security loopholes in an out-of-order processor because the speculatively executed memory fetches may disclose sensitive data through the memory fetch address side-channel if they are based on the altered instructions or data. Another solution offering even weaker security is *authentication-then-write*. Under this design, all permanent changes to memory state have to be made based on the results derived from authenticated instructions and operands. Though this design guarantees the authenticity of the results produced, it leaves significant security holes and puts confidentiality of both data and software at great risk. *Authentication-then-fetch* allows bus cycles to be granted to a memory fetch if only all the instructions and data that the memory fetch depends on according to data and control dependency have been authenticated and verified.

7.1 Memory Fetch as Information Disclosing Channel

Whenever there is a security system, there will be attempts to break it. History proves that it is often the case of underestimating the hacker's dedication and skill to compromise security protection instead of the other way around. Considering that secure processor is designed for countering physical tamper and one of the main applications is anti reverse-engineering of military embedded or security system, its security strength should never be taken for granted and in-depth study of potential exploits and risk assessment are not simply optional. For reverse engineering military embedded systems and software, motivated opponents may mobilize significant resources and expertise.

In a secure processor, since both the software and data are encrypted, memory fetch becomes one main source of information to the hackers. The reason is that memory fetch

address shown on the front side interface between a secure processor and memory module is not encrypted. Encrypting fetch address is neither practicable nor necessary because as long as the system uses regular unprotected DRAM, the adversaries can always eavesdrop at the final interface where fetch address is not encrypted. The disclosed fetch address in some extreme scenario allows a hacker to re-construct program control flow and even to recover sensitive data such as security keys [75, 76]. Compiler researchers have investigated this issue with a focus on static program analysis and flow safe language design [16, 47]. Such intellectual efforts though helpful for understanding how this side-channel may be exploited by the hackers to compromise security in a secure processor environment, they are far from being sufficient to guide secure processor design practice because most of the compiler research have been carried out under completely different assumptions and objective.

7.1.1 Threat Model

Almost all the previous compiler research adopts a natural execution threat model where memory fetch addresses (both data and code fetch) during natural execution of a program may comprise a side-channel for leaking sensitive information. Given this threat assumption, most past research have focused on compiler based static analysis. In those studies, authenticity and integrity of software and data are less a security concern. For secure processor design, the story would be different. For attaining high performance, a modern out-of-order processor may speculatively issue and execute instructions, speculatively fetch instructions and data. Such aggressive speculative execution when combined with secure processor design featuring disassociation of decryption and authentication may bring potential new security risks. Adversaries may deliberately alter software or data in some specific way and the altered software or data if executed or used speculatively by an out-of-order processor pipeline may disclose sensitive information through the memory fetch side channel. This more active form of exploit presents a more serious threat because static compiler analysis or safe programming practice lends almost no help to mitigate the post compilation runtime tampering.

Given that software and data are protected with secure processor using encryption, there

are potentially several ways to trick the secure processor to disclose sensitive data through the memory fetch side-channel. Most of these exploits involve tampering and altering of the protected software or data. An adversary may resort to one or a combination of the following techniques to achieve such a purpose.

7.1.1.1 Bit flip

Most standard encryption modes including the CBC mode, the counter mode and many other modes are malleable. A malleable encryption mode is a mode where flipping certain bits in cipher-text will induce certain bits of the decrypted plaintext to flip. In the counter-mode, flipping a particular cipher-text bit will make the same bit of the corresponding plaintext to flip. In the CBC mode, flipping a cipher-text bit will induce flipping of plaintext bit at certain offset depending on the encryption unit size of the underlying cipher. Through bit flipping, an adversary may transform a piece of protected software or data so that the altered code or data will either directly or indirectly disclose sensitive information to the side-channel. It is a standard practice to provide non-malleability to malleable encryption mode by adding authentication [18]. However, disassociation between decryption and authentication in an out-of-order processor leaves security holes that is big enough for the hackers to maneuver. Authentication and integrity verification may lag behind decryption with a latency ranging from tens of cycles to even hundreds of cycles. This provides an execution window where tens of unverified instructions or even more may be speculatively executed without being verified. In general, those speculatively executed instructions are not allowed to modify processor and memory state before they reach the commit stage. However, memory fetches are not state change operations. A standard out-of-order processor will grant bus cycles to speculative memory fetches before the commit stage. This makes memory fetch a side-channel.

An adversary can accelerate bit flipping attack using frequency analysis. According to the profile study [36, 72], application's memory space comprises significant amount of predictable or frequent plaintext values. An adversary can exploit the disproportional distribution of binary values to speed up bit flipping attack. For example, significant amount of

memory data in an application's memory space are zeros. Using this fact, an adversary can tamper a piece of ciphertext bits through bit flipping with the assumption that its plaintext are zeros. If the percentage of zero values is 30%, which is true for many applications, then the adversary will have 30% tampering success rate.

7.1.1.2 Replay attack

Apart from bit flipping, an adversary may use replay attack as a tampering method. Under this attack, an adversary replaces a piece of protected data with replayed old information. For example, if a protected function generates a random seed each time it is invoked and has the seed encrypted when stored in the external RAM. An adversary may log the encrypted seed. Next time, when the function is called again, the adversary may replace the new encrypted seed with the logged old seed to force the program to behave in a way predicable by the adversary. Replay attack imposes many security risks. Some general solutions to mitigate replay attack are Merkel hash tree or MAC tree [44] that authenticates the entire RAM as a whole. When authentication is decoupled from decryption, so is Merkel hash tree or MAC tree verification. Again the lag between data is decrypted and authenticated by Merkel tree or MAC tree provides rooms for the adversaries to exploit.

7.1.2 Exploit Taxonomy

In this subsection, we will briefly describe some of the exploits of the memory fetch side-channel for breaking data confidentiality protection provided by a secure processor. It is worth pointing out that the list given below is by no means comprehensive and they may represent only a small portion what an adversary can do by exploiting the fetch address side-channel. Except natural disclose that has been described in early this section, all other exploits are described below.

7.1.2.1 Pointer Conversion

The basic idea of *pointer conversion* exploit is to convert a piece of encrypted sensitive data into pointer such that its value will be automatically disclosed to the side-channel. There maybe many variations of this exploit. Here we give only one example, called *"link-list*

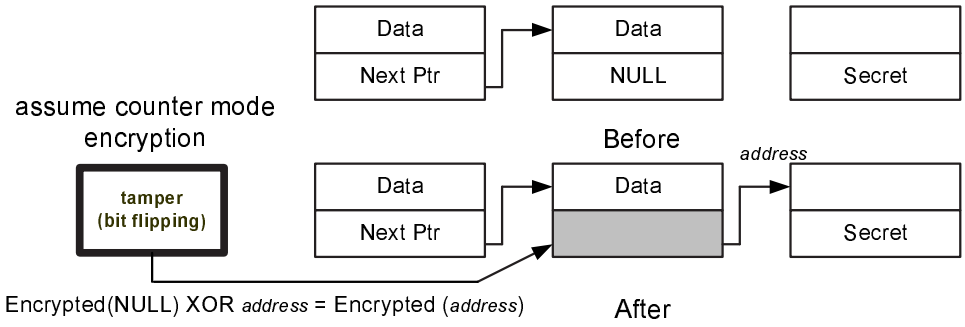


Figure 42: Point Conversion Exploit

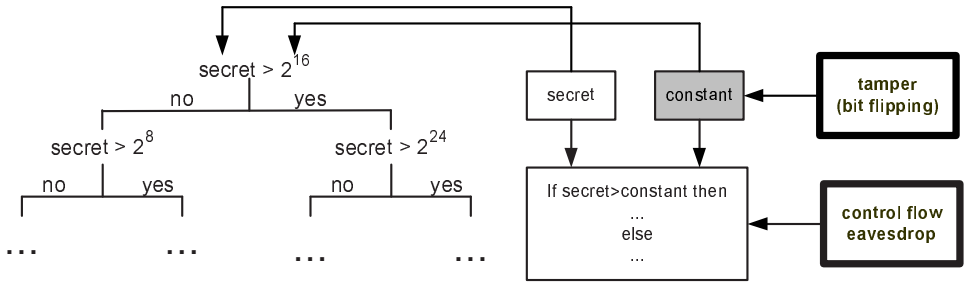


Figure 43: Binary Search Tamper

attack" to illustrate how to recover encrypted data by only altering program data. Link-list is widely used in software program. One property of link-list is that the last node is always terminated with a NULL pointer. Assume that the adversary knows where the link-list ends (the last node). Then, the NULL pointer becomes a known plaintext. Further assume that there is a secret data value x stored in memory location l, which the adversary wants to recover. The adversary can alter the NULL pointer into *l - node size + 4* so that the secret data becomes a node pointer, see Figure 42. When the link-list is traversed, the program will try to use the secret data as a node pointer and issue a corresponding memory load which may reveal its value to the side-channel. There are many ways for an adversary to recover memory locations of sensitive data. For example, an adversary can run a local copy of the same system and discover the likely position where sensitive data may be stored [35]. An adversary may also reverse engineer the software first and find out memory location of sensitive data based on de-assembled codes.

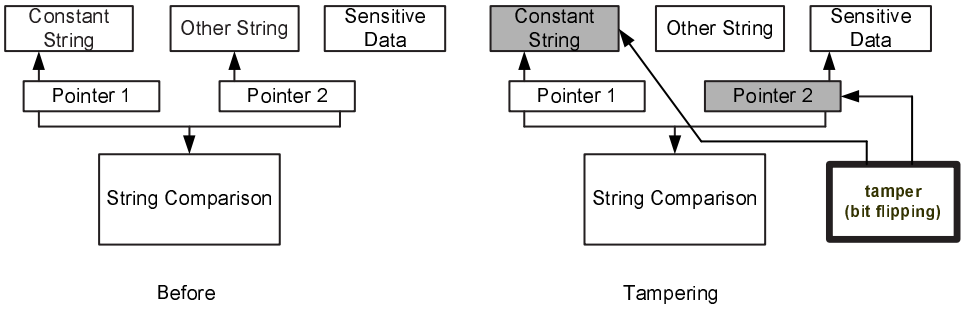


Figure 44: Binary Search Exploit Based on String Comparison

7.1.2.2 Binary Search

If there is some comparison that compares some secret data with some constant stored in memory and the constant value is known by an adversary. The adversary may launch so called *binary search* exploit. The adversary may alter the constant value in power-of-2 basis and eavesdrop how the modification will affect the comparison result (control flow). If the secret data is 32-bit long, according to the principle of binary search, at most $\log_2(2^{32})$ = 32 trials are enough to recover the sensitive data. Figure 43 illustrates such process. Alternatively, the adversary may tamper string comparisons and string constants. An adversary may combine this exploit with the *pointer conversion* exploit to force sensitive data to be compared with some constant, see Figure 44 for an example.

Binary search exploit requires tampering of constant or string values. To launch bit flipping attack on constant or strings, an adversary must first recover their plaintext values. This would not be too difficult because lots of strings or constants are either outputs that can be eavesdropped or input supplied by the users.

7.1.2.3 Disclosing Kernel

Disclosing kernel is a short piece of malicious code that will disclose possibly arbitrary data to the side-channel. The simplest *disclosing kernel* comprising only two RISC instructions is one that loads some arbitrary data into a secure processor, then use the data as fetch address. An adversary may insert a *disclosing kernel* into either code space or data space by tampering either code or data using bit flipping. Then by altering one more instruction or function return address, the adversary can hijack program control to the *disclosing kernel*. A

slightly sophisticated version of *disclosing kernel* such as one that has a loop can potentially disclose the entire application's memory space to the side-channel.

Inserting a *disclosing kernel* into an application's code space is in fact much easier than people thought. An adversary needs to first guess or recover a short sequence of encrypted instructions' plaintext whose size is large enough to hold the *disclosing kernel*. This in fact is not a very hard task. Instructions even in encrypted format are highly predictable or recoverable due to the following reasons.

- Invariant code sequence. Compiler always does code generation in a predictable way. The binary codes produced by a compiler comprise many short code sequences that are either invariant or predictable. Such invariant or predictable short code sequence can be found in program's entry point, function epilogue or prologue, compiled loop structure and etc. An adversary can replace one of those predictable or invariant code sequences with a *disclosing kernel*.

- Frequency analysis. A program compiled in typical RISC format contains 7% nop, 28% load/store, and 6.6% branch instructions [36]. An adversary has 7%, 28%, and 6.6% success rate of guessing some randomly encrypted instruction to be a nop, a load/store or a branch. In another word, if the adversary flips the bits of some encrypted instruction to transform the decrypted instruction into a load or branch instruction with the assumption that the original decrypted instruction is a nop, the adversary has on average 7% success rate. A straight forward attack is to replace a NOP with a control transfer instruction to hijack program flow into a *disclosing kernel*.

- Instruction reverse engineering. Instructions even encrypted can be reverse engineered by an adversary. For example, it is not too difficult for the adversary to reverse engineer branch instructions because of their memory access footprint and launch the bit flipping attacks to transform the reverse-engineered branches into something else. As long as an adversary can reverse engineer a short sequence of instructions, it is enough to stick into a *disclosing kernel*.

Opcode	Number			
Opcode	RA	Disp		
Opcode	RA	RB	Disp	
Opcode	RA	RB	Function	RC

Figure 45: Alpha Instruction Format

Inserting a *disclosing kernel* into application's data space could also be a very simple task because of the existence of frequent data values [72]. Research shows that large percentage of data values are zeros. An adversary may have a very high success rate of inserting a disclose kernel into a counter mode encrypted memory space by simply XORing the *disclosing kernel* with ciphertext whose plaintext is most likely to be zeros. If a *disclosing kernel* is inserted into data space, the adversary needs to hijack the program control flow into the *disclosing kernel*.

7.1.2.4 Data-to-code/Code-to-code Conversion

Program execution, especially executing of branches or data fetch instructions often leaves memory footprint that can be exploited by an adversary to reverse engineer and recover encrypted sensitive data or instructions. The objective of *data-to-code conversion* or *code-to-code conversion* exploit is to transform sensitive data or instructions that do not have memory footprint into instructions that have through either brute force or random sub instruction bit flipping. As a result, part or complete of the sensitive data or the original instructions can be disclosed.

This attack exploits some of the vulnerabilities of the RISC instruction set. They are,

- Regular Size. All the instructions have the same length and in many cases they are short, 16 bits, 24 bits, or 32 bits;
- Regular Format. The instructions are well formatted (e.g. fixed opcode field) for reducing decoding logic complexity. For example, bit[31:26] of Alpha ISA is fixed as the opcode, see figure 45.
- Reduced ISA Number. RISC philosophy advocates a small set of instructions instead

Table 6: Exploit Comparison

Exploit	Tamper	Prevention Requirement
Natural Disclose	recover control flow from execution trace	flow safe language, compiler static analysis
Pointer Conversion	data tamper, convert data into pointer	authenticate fetch address before granting bus cycle
Binary Search	data tamper, alter constant value	branch/memory fetch shall depend on authenticated data/code
Disclosing Kernel	data/code tamper, jump into malicious code kernel	authenticate fetch address before granting bus cycle
Data-to-code/ Code-to-code Conversion	data/code tamper, sub-instruction tamper	authenticate branch before granting bus cycle

of a large number of complex instructions. This also reduces the search space of brute-force attack.

One risk associated with RISC is that it allows brute force and incremental guess using sub instruction bit flipping. In *data-to-code* or *code-to-code conversion* exploit, an adversary divides each instruction size data into fields (opcode, operand one, operand two, and etc.) and launches brute-force guess piece by piece on each field of the targeted instruction. For example, if the opcode is only 6 bits long, it requires only 2^6=64 trial and error to transform a random encrypted data value into a jump instruction. The rest bits of the data are treated as displacement and its value can be eavesdropped through the fetch address side-channel. In another word, *data-to-code or code-to-code conversion* exploit may allow an adversary to recover an encrypted data value or instruction in the order of 64 trials in ideal case. A detailed description and demonstration of this exploit can be found in [59].

It is important to keep in mind that some of the aforementioned exploits apply equally to the degenerated scenarios such as only confidentiality of data or only confidentiality of code is protected.

7.1.3 Impact of Virtual Memory Translation

The aforementioned exploit cases represent an ideal situation where sensitive data always shows up directly as memory fetch address. Many high performance processors use virtual address translation. A piece of sensitive data when used as fetch address may map into invalid address space and trigger memory translation exception. It is worth pointing out that many embedded processors or micro-controllers do not use virtual memory and those

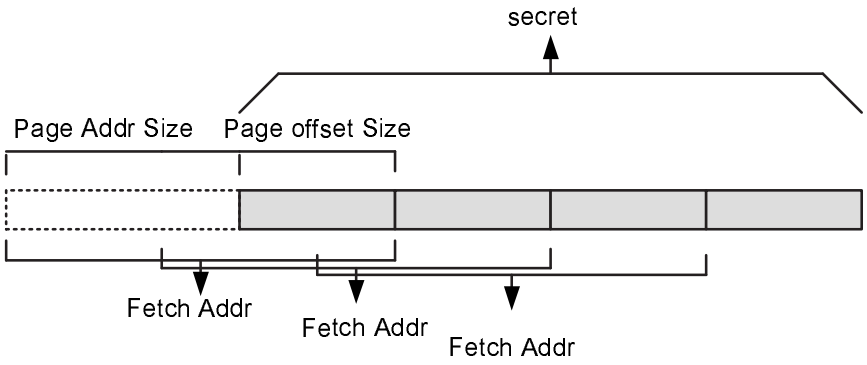


Figure 46: Shift Window

exploits can be applied straight-forwardly. On the other hand, the aforementioned exploits such as *pointer conversion*, *disclosing kernel* and etc are still effective when a secure processor uses virtual memory address translation. First, an adversary may try to disable virtual address translation if it is possible. Second, an adversary may try to tamper the address translation table to fool the system and avoid translation fault. Third, many processors throw exception and log the faulty address. For example, Windows OS throws a window that displays the invalid address to the user. If this is the case, an adversary can recover the sensitive data easily by reading the log or displayed address. If all the above do not work, an adversary can use the following two techniques,

7.1.3.1 Shift window and page address mask

For most processors, the page size is at least 4K bytes. This means that the lower 12 bits of a 32-bit address will not be affected by address translation. An adversary can use a shift window to recover 12 bits each time, see Figure 46. For the high bits, the adversary can mask them out to make sure that the result can always be translated. Given that a piece of sensitive data is stored in a register, the adversary can mask out or transform the page address before the data is used as fetch address. This requires tampering of binary instructions. As aforementioned, binary codes of applications comprise instructions that can be recovered or reverse engineered. After reverse engineer, an adversary can mold the recovered instructions into other instructions such as instruction to mask out or transform the page address. A *disclosing kernel* may also mask out the page address first and then

uses the result as fetch address.

7.1.3.2 Brute force or random page address tampering

If all the aforementioned techniques fail, the adversary can resort to brute force or random page address tampering. For example, in *pointer conversion* exploit, the adversary may either randomly or systematically flip ciphertext bits that map to page address. Assume that the page address size is 20 bits and the application has 100MB memory space mapped with virtual address translation. Using brute force or random page address tampering, the adversary on average has a chance of one out of about 40 ($2^20/(100MB/4KB)$) trials to have some random data correctly translated. Considering the disproportional distribution of frequent or predictable values, an adversary can speed up the process by playing with frequent or predictable values.

7.1.4 Effect of Cache

The existence of on-chip cache may impact the effectiveness of some of the exploits aforementioned. Cache reduces a program's memory footprint. However, an adversary can use the following techniques to reduce cache's effect on tampering,

- Disable cache. Most processors allow a local user to disable cache through resetting BIOS.
- Uncachable memory. An adversary can tamper with memory attributes and mark certain memory range as uncachable.
- Fake coherent signal. Most of today's processors support bus snoop based cache-coherence protocol. It is possible for an adversary to insert fake DMA (direct memory access) requests causing a processor to mark a cache line as shared and write the data back each time it is updated or reload a tampered data value. Such attack involves building a device that is able to insert false coherent memory read/write to the system memory. One way of doing this is to use a cheap FPGA board designed for peripheral device development and load it with logic that keeps sending coherent DMA requests to the targeted memory address.

- Attack before cache warm up. Most of the attacks and exploits can be launched at the very beginning of application execution. Since the cache has not been warmed up, tampered program execution will leave desired memory footprint in this case.

7.2 Authentication Architecture

It is not the objective of this chapter to discuss how to design or choose a MAC standard for integrity protection. This issue has been addressed in a number of recent publications [63, 62]. The main focus of this chapter is to investigate different schemes of integrating integrity verification logic or unit into an out-of-order processor pipeline. In this section, we will provide detailed information how to tie authentication result with instruction execution and discuss the security and performance implications of each method in the context of memory fetch side-channel exploits. We treat the specific MAC verification logic as a black box that for each piece of fetched data returns a binary verification result.

7.2.1 Authentication Queue

Figure 47 is a block diagram of an out-of-order processor pipeline augmented with integrity verification and cryptographic capabilities. For every block of code or data fetched from memory, the secure processor decrypts the information and verifies its integrity. There are two main reasons contributing to the latency disparity between decryption and authentication. First, secure processor can use encryption modes such as the counter mode that allows pre-computation and prediction [57] to reduce overall fetch-decryption latency while integrity verification can not be started until the data is fetched or decrypted. Second, in addition to regular MAC verification, the secure processor may also employ MAC tree or Merkel hash tree to prevent replay attacks, which further increases the authentication latency. We assume that authentication or integrity verification in general is falling behind decryption with latency from tens to hundred of clock cycles. For each fetched block of data or code, the secure processor sends a request to a component, called authentication queue. The integrity verification unit authenticates data in the request order and broadcasts the authentication result. Since each request is associated with a queue entry, the entry index provides a way to identify authentication requests. There is a register, called "LastRequest

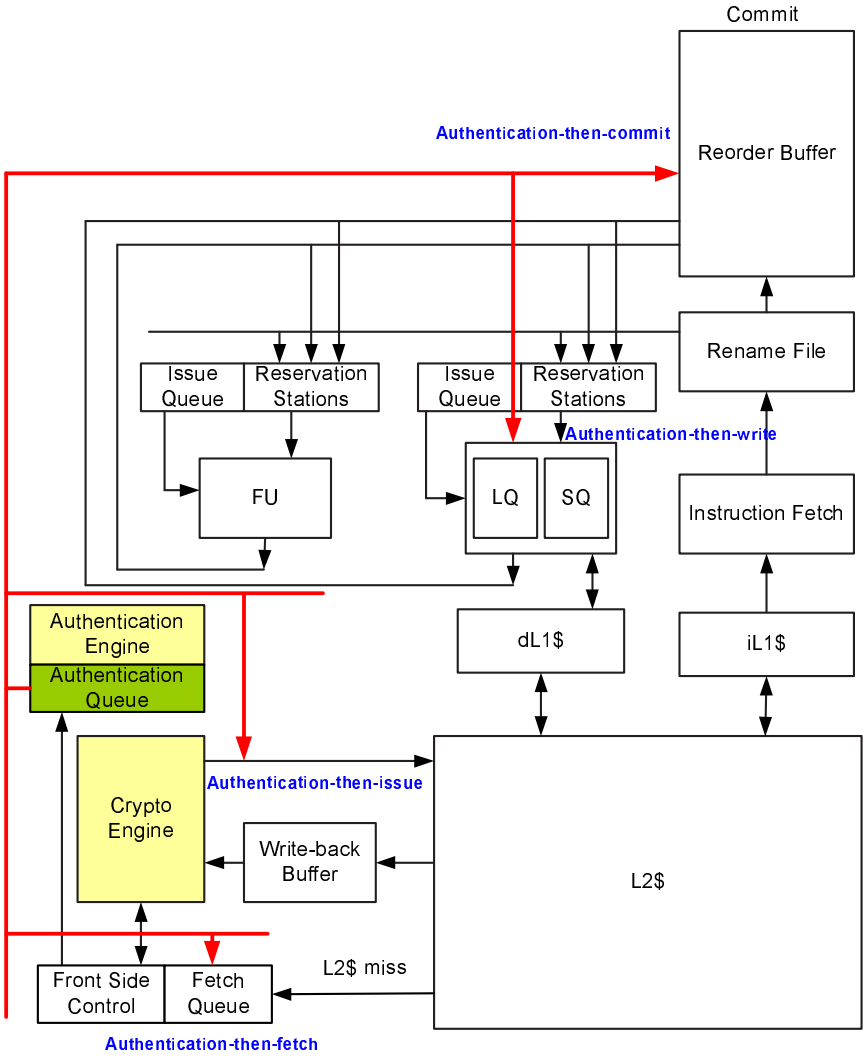


Figure 47: Secure Processor Block Diagram

Register" that points to the index of the most recent authentication request.

7.2.2 Authentication Architecture

In this subsection, we will explore four designs that connect integrity verification result with instruction execution.

7.2.2.1 Authentication-then-issue

Authentication-then-issue is a conservative approach. According to *authentication-then-issue* only after integrity verification, fetched instructions can be issued and fetched data can be used as operand. This approach is simple to implement and it prevents all the runtime

Table 7: Comparison of Security Strength of Different Schemes Against Side-channel Disclose

	pointer conversion	binary search	disclose kernel	data-to-code/ code-to-code conversion
Authentication-then-issue	√	√	√	√
Authentication-then-write				
Authentication-then-commit				
Authentication-then-fetch	√	√	√	√

exploits of the aforementioned side-channel. However, it buys security at a significant cost on performance because in this case integrity verification is on the critical path.

7.2.2.2 Authentication-then-write

Authentication-then-write is the most optimistic of the four approaches. According to *authentication-then-write*, integrity of memory state is guaranteed if for every piece of data written to the memory, the secure processor is certain that the data is generated based on the verified code and data. Detailed implementation of this approach may vary.

In one implementation, for every store instruction, when the instruction is ready for issue, the secure processor will read the "LastRequest Register" and associate the index value as a tag of the store instruction, called "authentication tag". The store value that should be written to either L1 cache or memory will remain in the store queue until it receives a broadcast result that indicates that the authentication request referenced by the store value's authentication tag has been successfully verified. This ensures that the secure processor only updates cache and memory with values produced by the verified code and data. Note that the authentication engine broadcasts verification result in the natural request order. Upon receipt of a broadcasted matching authentication tag, the secure processor can be certain that all the codes and data before the waiting store are also authenticated.

Under *authentication-then-write*, the secure processor guarantees that at any moment, the information stored in the memory can be trusted in the sense that they are produced based on the authenticated code and data. However, this approach does not prevent any of the aforementioned active exploits of memory fetch side-channels. It does not guarantee confidential software and data from being disclosed through the side-channel exploits.

Table 8: Characteristic Comparison of Different Schemes

	prevent active fetch address side-channel disclose	**precise interrupt**	**authenticated [1] memory state**	**authenticated [2] processor state**
Authentication-then-issue	√	√	√	√
Authentication-then-write			√	
Authentication-then-commit		√	√	√
Authentication-then-fetch plus authentication-then-commit	√	√	√	√

7.2.2.3 Authentication-then-commit

In an out-of-order processor design, instructions wait in a structure called reorder buffer before they are committed. According to *authentication-then-commit*, the secure processor will not commit an instruction until both the instruction itself and its operands are authenticated. In some aspects, *authentication-then-commit* is similar to *authentication-then-write*. For example, *Authentication-then-commit* also guarantees that the secure processor only updates the memory states using authenticated code and data. However, there are many sometimes fundamental differences between these two approaches. First, *authentication-then-write* deals with only memory state, sometimes only the external memory state while *authentication-then-commit* handles both memory state and processor state. Second, *Authentication-then-commit* provides precise interrupt for authentication exceptions while *authentication-then-write* does not. Though seemingly a very promising design, *authentication-then-commit* does not prevent any of the aforementioned memory fetch side-channel exploits because it allows unverified memory fetches to be speculatively issued.

7.2.2.4 Authentication-then-fetch

According to *authentication-then-fetch*, a secure processor must not grant bus cycles to an external memory fetch until both the fetch instruction or branch instruction and the fetch address are authenticated. In addition, strong *authentication-then-fetch* requires not only the fetch/branch instruction or the fetch address itself be authenticated but also all the codes and data that the fetch/branch instruction and the fetch address depend on. This includes

[1] Memory state updated based on authenticated code and data
[2] Processor state updated based on authenticated code and data

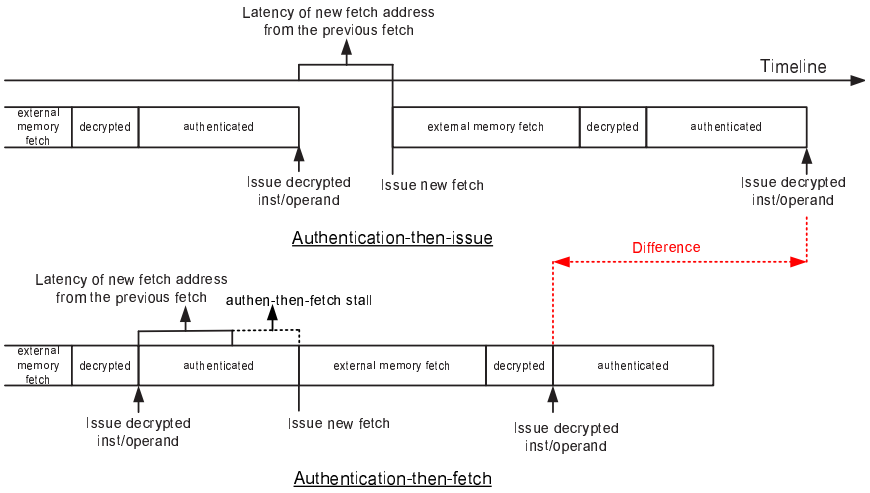


Figure 48: Timeline of Authentication-then-fetch vs. Authentication-then-issue

both control and data dependency. To reach a particular data fetch or branch instruction, there is an execution path, also called program slice, that includes all the previous dependent instructions. According to strong *authentication-then-fetch*, a secure processor is allowed to issue fetch address on the front side bus only if 1) the data fetch/branch instruction itself is authenticated; 2) all the previous instructions it has control and data dependency are authenticated; 3) the fetch address itself or all the data for deriving the fetch address if the fetch address is computed are authenticated. In another word, all the instructions and data involved in the path or slice have to be authenticated before the fetch is issued to the front side bus.

A precise implementation of this approach requires dynamic tracking of control and data dependency that may be too complex. Fortunately, there are many alternative implementations that sufficiently satisfy all the requirements of *authentication-then-fetch* without resorting to dependency tracking. In one variation, memory fetch is not issued until the secure processor drains the authentication queue. This is called *drain-authentication-then-fetch*. For a new memory fetch, the secure processor stops sending more authentication requests to the authentication queue, waits for the current authentication to be drained, issues the memory fetch afterward, and then resumes sending more authentication requests. Alternatively, a secure processor may associate the current value of the "LastRequest Register" with each issued instruction. If the instruction triggers a memory fetch, the fetch

is stalled until the authentication queue has completed authentication of the associated request. Figure 48 uses an example to illustrate the difference between *authentication-then-issue* and *authentication-then-fetch*. The example shows two external memory fetches where the second fetch is based on the first fetch. There is an assumed fixed latency between when the data/instruction based on the first fetch result is issued to the processor pipeline and when the second fetch address is ready. The dotted line highlights the time difference between the two schemes and shows the latency advantage of *authentication-then-fetch* over *authentication-then-issue*. Note that under *authentication-then-fetch*, new external memory fetches only stall on already issued recently decrypted instructions or operands. Instructions or data that are decrypted after the new memory fetch is created, or outstanding external memory fetches will have no latency impact on the new memory fetch.

Table 7 summarizes and compares five authentication architectures in term of their security strength against the four mentioned runtime exploits of fetch address side-channel. According to the comparison, *authentication-then-issue* and *authentication-then-fetch* are the most secure authentication architectures. Table 8 provides additional comparison of different authentication architecture in more aspects. It is recommended that *authentication-then-fetch* be used together with *authentication-then-commit* to quarantine authenticated memory/ processor state and achieve precise interrupt on security faults.

7.3 Performance Analysis

7.3.1 Simulation Framework

Our simulation work is based on SimpleScalar [6] running Alpha binaries compiled with -03 option. Each benchmark is fast-forwarded according to SimPoint's suggestion [56] and then simulated for 400 million instructions in performance mode. During fast-forwarding, L1 and L2 caches were warmed up. Table 7.3.1 summaries the architectural and microarchitectural parameters.

Eighteen SPEC2000 INT and FP benchmarks with high L2 misses and memory throughput requirements were used for evaluation.

Table 9: Processor model parameters

Parameters	Values
Frequency	1.0 GHz
Fetch/Decode width	8
Issue/Commit width	8
L1 I-Cache	DM, 16KB, 32B line
L1 D-Cache	DM, 16KB, 32B line
L2 Cache	4way, Unified, 64B line, write back cache 256KB and 1M
L1/L2 Latency	1 cycle / 4 cycles (256KB), 8 cycles (1M)
RUU	128, 64 entries
Memory Bus	200MHz, 8B wide
Memory Latency	X-5-5-5 (core clocks) X depends on page status
CAS latency	20 mem bus clocks
Precharge latency (RP)	7 mem bus clocks
RAS-to-CAS (RCD) latency	7 mem bus clocks
Triple-DES latency	96ns, 48 stages
AES latency	80ns

7.3.2 Implementation

The latency of decryption and integrity verification varies substantially depending on many factors such as encryption mode, cipher, authentication scheme, process technology, architecture design and etc. To best justify our performance conclusions, we use reference implementations. In simulation study, we conduct sensitivity study to capture different variations and design scenarios.

Integrity verification based on MAC is often standard operation. But variation of different MAC approaches can have significant impact on verification latency. Though it is plausible to carry out integrity verification and decryption concurrently, unlike counter mode, integrity verification must wait until the data is fetched. Furthermore, integrity verification has to be conducted on the entire data block. As a result, latency of integrity verification is often longer than decryption. In the reference implementation, we use standard HMAC [34] for protecting integrity of data blocks stored in the external RAM. The default size of MAC is 64 bits. The reference HMAC uses standard SHA-256 algorithm [50]. Simulation study is based on Verilog implementation of SHA-256 [64], synthesized using Synopsys. This design is totally asynchronous and has a gate count of 19,000 gates. The latency for this design is 74ns for 512 bits of padded input (padding with the required padding in SHA-256).

In addition to per-data block based integrity verification, secure processor sometimes

also applies a hash tree or MAC tree for preserving the overall memory space integrity and preventing replay attacks of data blocks. This causes substantially additional amount of latency overhead. The CHTree scheme in [62] constructs an m-ary hash tree where m is the number of child nodes per parent node has and is equal to the size of the cache line divided by the size of hash values. A typical value of m is 4. To verify a data block, it takes $log_m(L)$ hashing computations, assuming the entire memory space comprising L total data blocks. The reference implementation uses a m-ary MAC tree instead of hash tree for improved performance and security. A default value of m is either 4 or 8. This means that a new MAC is calculated over every 4 or 8 MACs. Recursively, each node of the MAC tree is a MAC of 4 or 8 child MACs. On top of the MAC tree is a root MAC of 64 bits. The root MAC is a signature of the entire memory space of a software application.

7.3.3 Performance Result

In this section we summarize performance results. The results are collected on two L2 cache settings, 256K and 1M.

7.3.3.1 Performance of Different Authentication Architectures

Figure 49 shows normalized IPC of the five ways of using authentication result under 256K L2 for eighteen SPEC2000 integer and floating benchmarks. The IPC is normalized against IPC of a baseline situation of decryption only without integrity verification. The results suggest that *authentication-then-issue* have the worst performance where the average IPC is about 87% of the baseline IPC. For some benchmarks such as ammp, bzips, mgrid, twolf, and vpr, their IPCs are below 80% of the their respective baseline IPCs. In contrast, *authentication-then-write* has the best performance. On average, IPC under *authentication-then-write* is more than 98% of the baseline IPC, which means less than 2% performance penalty. The next one is *authentication-then-commit*, its average IPC is more than 96% of the baseline IPC. For most benchmarks, IPC under this scheme is over 90% of the baseline IPC except mgrid whose normalized IPC is 86%. Under *authentication-then-fetch*, the average IPC is 92% of the baseline IPC. Combination of *authentication-then-commit* and *authentication-then-fetch* yields average IPC performance of 90% of the baseline.

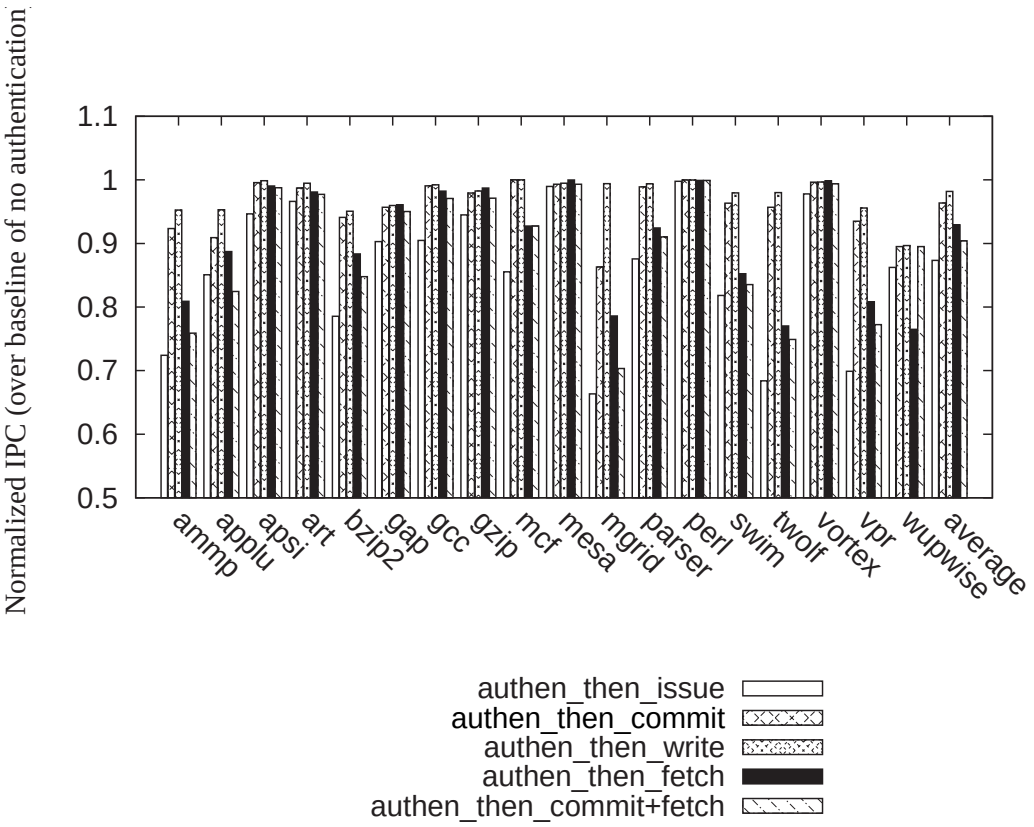


Figure 49: Normalized IPC Under Different Authentication Schemes, 256K L2

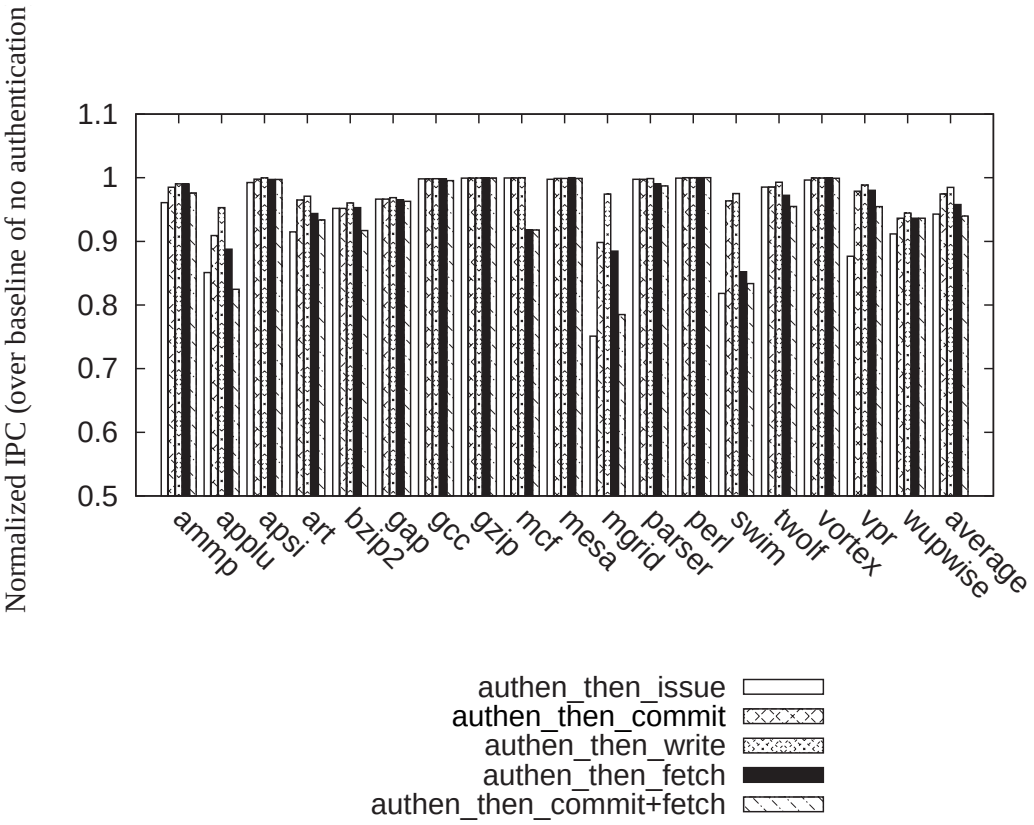


Figure 50: Normalized IPC Under Different Authentication Schemes, 1M L2

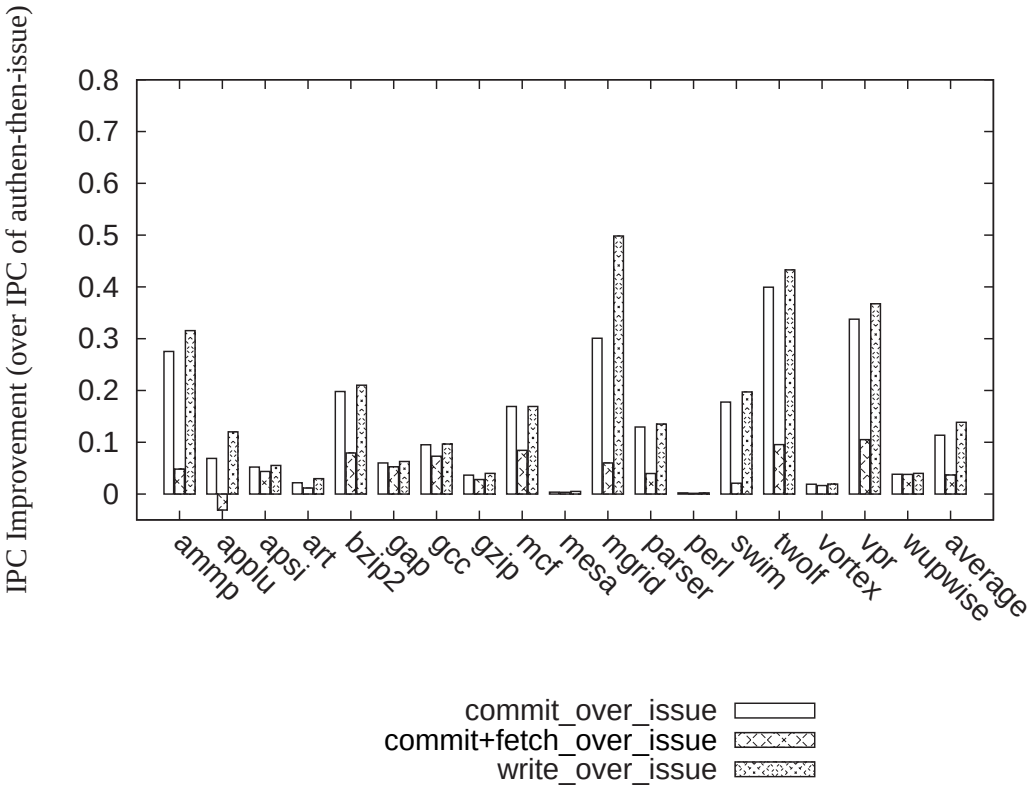


Figure 51: Comparison of IPC Speedup Over Authentication-then-issue Using Three Other Schemes, 256K L2

Figure 50 shows normalized IPC performance under 1M L2 cache. Since there are less amount of memory accesses when the L2 size quadruples, the performance impact of different schemes is less than the scenario of 256K L2. However, the performance ranking of the five schemes is the same where *authentication-then-issue* has the lowest performance and *authentication-then-write* has the highest performance.

As aforementioned, *authentication-then-issue* is the most secure scheme. Figure 51 compares *authentication-then-commit*, *authentication-then-write*, and *authentication-then-commit plus authentication-then-fetch* with *authentication-then-issue* by showing the IPC speedup of the three schemes over IPC of *authentication-then-issue*. The results indicate that on average, *authentication-then-commit* improves IPC by 12%. For four benchmarks, the improvement is over 20% and for other six benchmarks, the improvement is in the range from 10% to 20%. Three benchmarks have performance speedup over 30%. For *authentication-then-write*, the performance improvement on average is about 14%. However, as discussed before, both *authentication-then-commit* and *authentication-then-write*

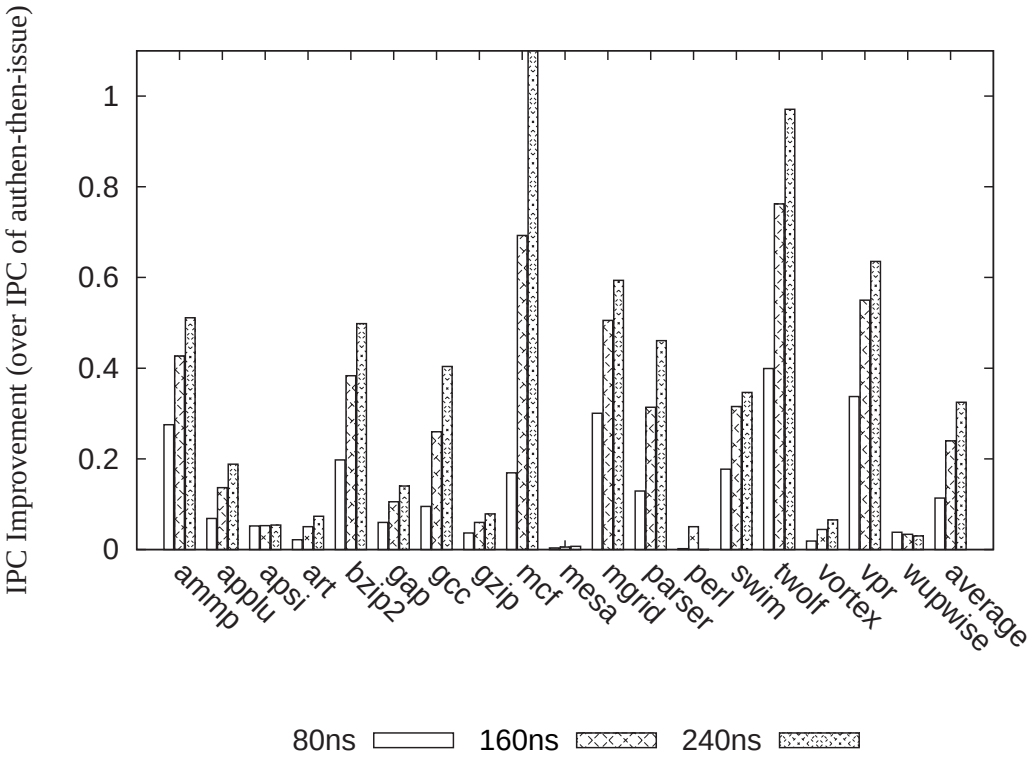


Figure 52: IPC Speedup of Authentication-then-commit Over Authentication-then-issue Under Three Authentication Latencies, 256K L2

are less secure than *authentication-then-issue*. In contrast, combination of *authentication-then-commit* and *authentication-then-fetch* provides much better security and it does not suffer from the aforementioned exploits just like *authentication-then-issue*. For five benchmarks, *authentication-then-commit plus authentication-then-fetch* provides about 10% performance improvement over *authentication-then-issue*.

Figure 52 shows that when the latency overhead of authentication increases, the performance improvement of *authentication-then-commit* over *authentication-then-issue*. As expected, when the authentication latency goes up, IPC speedup of *authentication-then-commit* over *authentication-then-issue* also increases.

7.3.3.2 RUU Size

Under *authentication-then-commit*, completed instructions will wait for authentication result before they are committed. Size of RUU may have some impact on performance of the studied schemes. To conduct sensitivity study, we reduce the number of RUU entries by half. Figure 53 shows the results. The results indicate the same performance pattern.

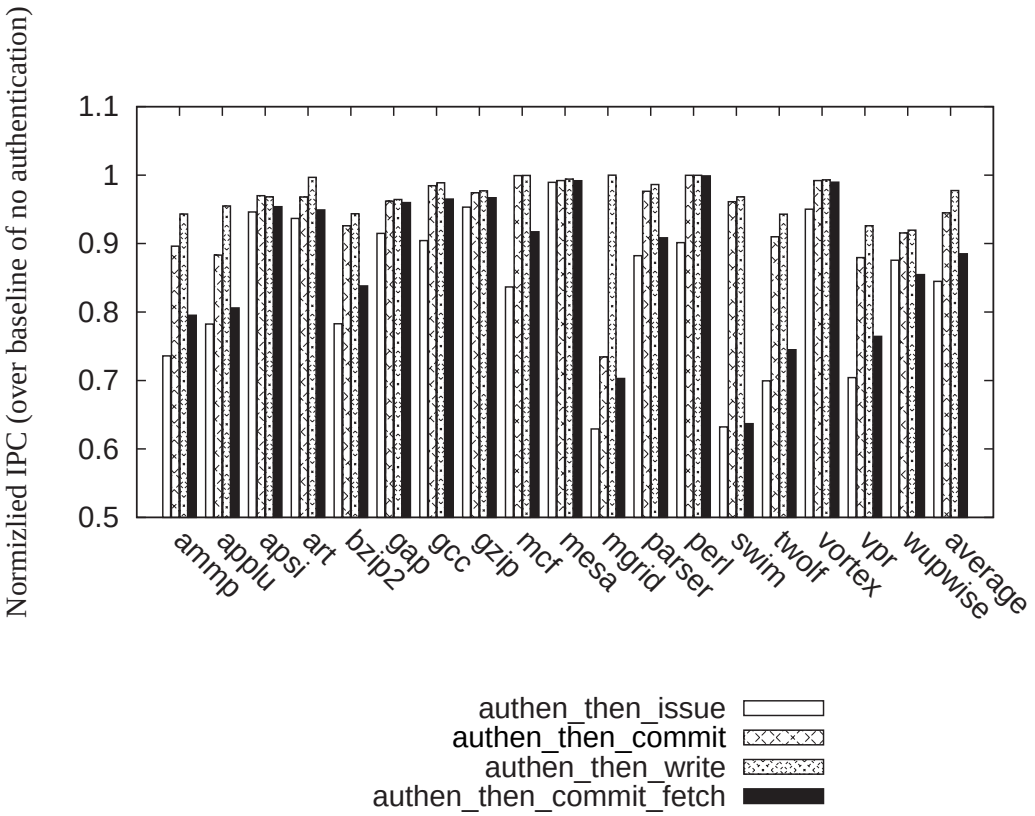


Figure 53: Normalized IPC Under Different Authentication Schemes, 256K L2, 64-Entry RUU

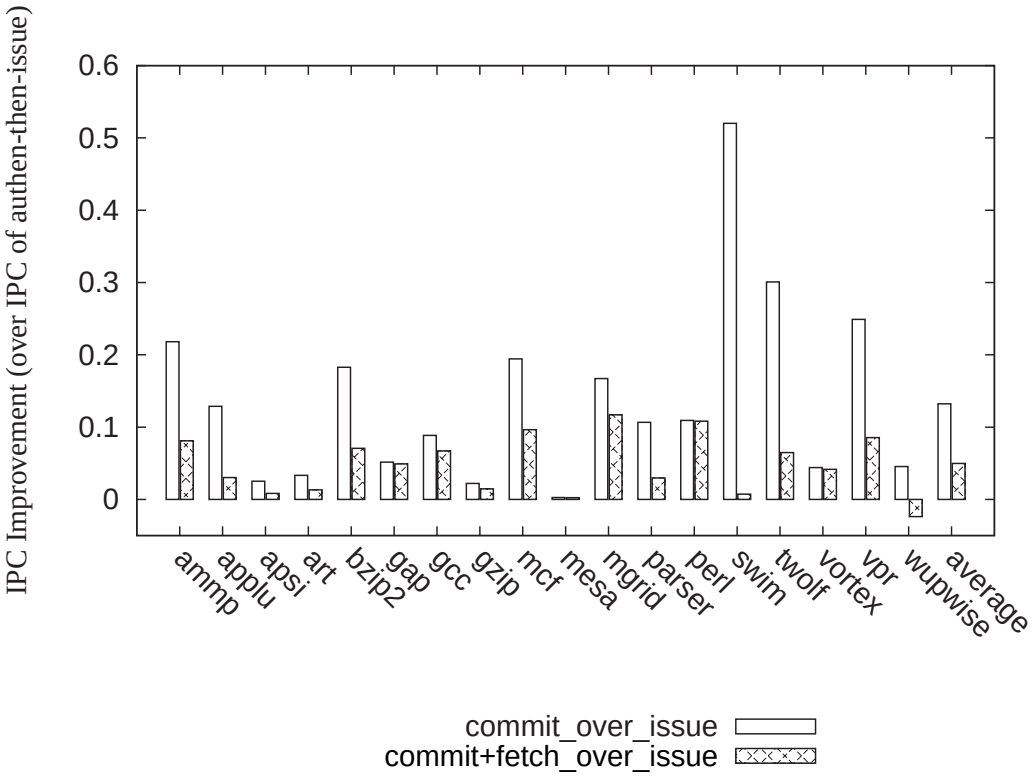


Figure 54: Comparison of IPC Speedup Over Authentication-then-issue Under Different Authentication Schemes, 256K L2

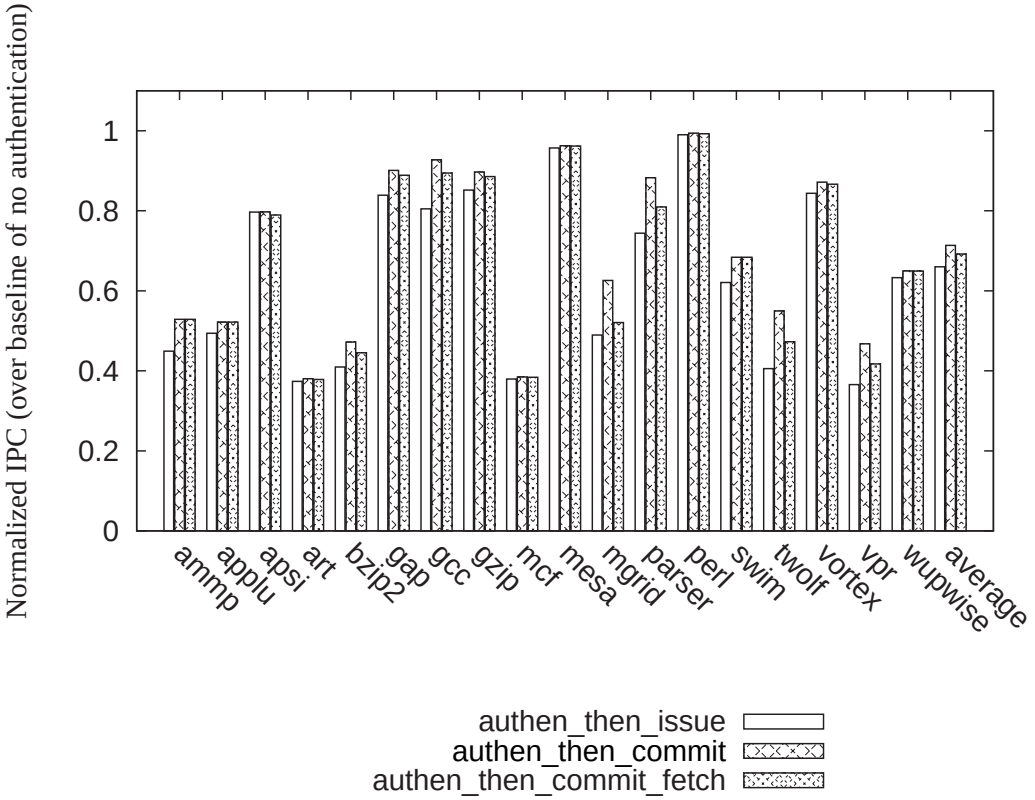


Figure 55: Normalized IPC Under Different Authentication Schemes, Memory Authentication Tree

The performance rank of four schemes from the lowest to the highest are, *authentication-then-issue*, *authentication-then-commit plus authentication-then-fetch*, *authentication-then-commit*, and *authentication-then-write*. Figure 54 shows IPC speedup of *authentication-then-commit* and *authentication-then-commit plus authentication-then-fetch* over *authentication-then-issue*. For five benchmarks, *authentication-then-commit plus authentication-then-fetch* has performance improvement about 10%. For ten benchmarks, *authentication-then-commit* improves IPC in the range from 10% to 50%.

7.3.3.3 Impact of Hash Tree Authentication

Hash or MAC tree can prevent replay attacks. One side-effect of using hash or MAC tree such as the CHTree approach is the additional latency overhead for integrity verification. We evaluate the five schemes under hash tree authentication. The hash tree implementation is based on [62]. Our implementation conducts verification of the internal hash tree nodes concurrently when it is allowed. Authenticated and verified tree nodes are cached inside the processor using a dedicated hash tree cache. The size of the tree cache

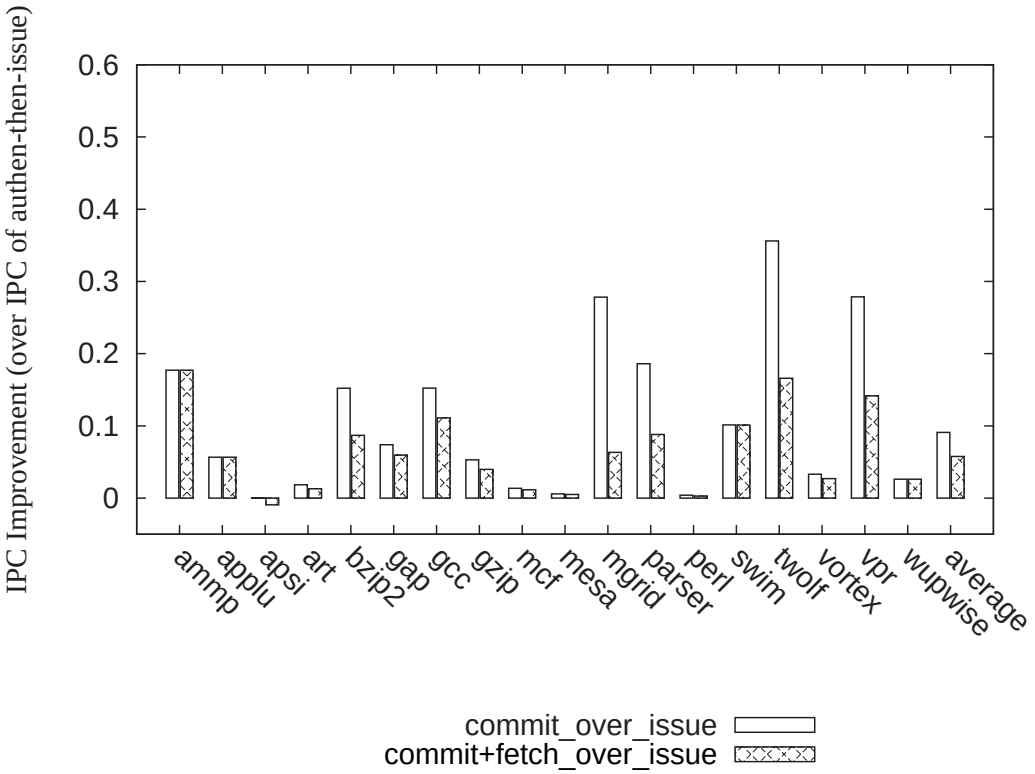


Figure 56: Comparison of IPC Speedup Over Authentication-then-issue, Memory Authentication Tree

is 8KB. Figure 55 shows the normalized IPC performance of *authentication-then-issue*, *authentication-then-commit* and *authentication-then-commit plus authentication-then-fetch*. Again, the performance results indicate the same ranking of performance among the three schemes. Figure 56 shows performance improvement of *authentication-then-commit* and *authentication-then-commit plus authentication-then-fetch* over *authentication-then-issue*. For *authentication-then-commit*, seven benchmarks have performance improvement from 10% to 35%. For *authentication-then-commit plus authentication-then-fetch*, five benchmarks have performance improvement more than 10%.

7.4 Conclusions

In this chapter, we propose and explore design spectrum of integrating memory integrity verification and decryption into an out-or-order high performance processor pipeline to provide a secure computing environment. The chapter provides in-depth analysis of the risk associated with memory fetch side-channel in the context of secure processor design. It analyzes

and evaluates the security implications of several design choices including *authentication-then-issue*, *authentication-then-commit*, *authentication-then-write*, *authentication-then-fetch*. Based on both security analysis and performance evaluation, we show that *authentication-then-issue* provides the best protection against the side-channel exploits but has he most performance overhead. *Authentication-then-write* guarantees the integrity of processing results stored in un-trusted external memory. In addition, *authentication-then-commit* ensures the integrity of both memory and processor state at any moment and achieves precise interrupt for security exceptions. But neither approach prevents violation of software and data confidentiality through runtime exploit of memory fetch side-channel. Among all the evaluated techniques, combination of *authentication-then-fetch* and *authentication-then-commit* attains best trade-off between security and performance. *Authentication-then-fetch* protects against runtime exploits of the memory fetch side-channel and at the same time incurs only moderate performance degradation. In conclusion, analysis and result of this chapter provides valuable risk assessment and design trade-off information for guiding design of tamper-proof secure processor.

CHAPTER VIII

PROTECTION OF MEMORY FOR SYMMETRIC MULTI-PROCESSORS

Most secure processor solutions proposed so far assume that the system memory is exclusively "owned" by one processing element (often the main processor). Inside the processor, memory is authenticated on per process basis with a memory integrity signature computed for each process's virtual space. This signature is generally the root of an authentication tree such as MACTree or CHTree discussed in chapter 7.2. This kind of strong process isolation prevents the signature from being shared by multiple processors. When inter-processor memory sharing is inevitable, a copy from one processor's authenticated domain to another's is required. Such copying operations often require duplicated integrity checking of the shared memory by the destination processor. For symmetric multiprocessor (SMP) systems, it is not a trivial task to synchronize and maintain integrity signatures for frequently shared data without significantly degrading system performance. To make things even harder, integrity protection of memory shared among multiple processors has to prevent eavesdrop or replay attacks on the shared bus or the system memory interface.

In this chapter, we describe a fast and low overhead solution to authenticate the shared memory of a SMP system. Through securing every component along the path from a computing device to another computing device or the commonly shared memory, a chain of integrity protection is constructed. The chained authentication scheme is capable of preventing most software based and hardware based exploits on the memory system and the shared data path among processors. The scheme also optionally provides high performance protection on information confidentiality for shared data.

8.1 Challenges in Shared Memory Protection

In this section, we discuss many basic issues associated with shared memory protection at high level. It presents the basic platform architecture our solution is targeted for. It also shows the types of attacks our solution is aimed to prevent and detect.

Efficiency and Security: Almost all the recently proposed security computing platforms with hardware-based memory protections assume that everything in such a system is insecure except the main processor with built-in security support [73, 37, 61]. Under such assumptions, many proposed protection solutions often have all the hardware security features such as memory decryption and integrity verification implemented in the main processor. Gassend et al. [22], the CHTree authentication scheme constructs a m-ary hash tree for protecting the memory integrity of an application process under a uni-processor environment. As shown, CHTree slows down the execution by around 20% with a 2MB L2 and incurs 33% memory space overhead. The LHash scheme in [60] was proposed to improve the authentication speed. This scheme logs memory operations and performs integrity checks only when a large number of memory operations is accumulated. As discussed in chapter 7, aggressive disassociation between integrity verification and memory decryption may cause significant security risk. How to protect confidentiality and integrity of SMP shared memory remains a great challenge.

Multiprocessor Domain: Furthermore, most existing secure processor architectures are designed for uni-processor memory protection and assume that the security boundary lies at the interface between a secure processor and its system bus. Such a centralized view fits with a uni-processor platform but does not apply to a SMP system. A simple way to extend the existing solutions to the SMP scenario is to have a separate copy of the memory for each processing element[1] (PE) and have the secure OS to copy the data from one trusted domain to another using protected message passing mechanism whenever needed. This will however significantly increase the delay of inter-processor communication and substantially undermines performance of SMP applications.

[1] A processing element (PE) can be a core processor, a co-processor, or a peripheral in a uni-processor system or a processor in a SMP system. We generalize such a device as a PE.

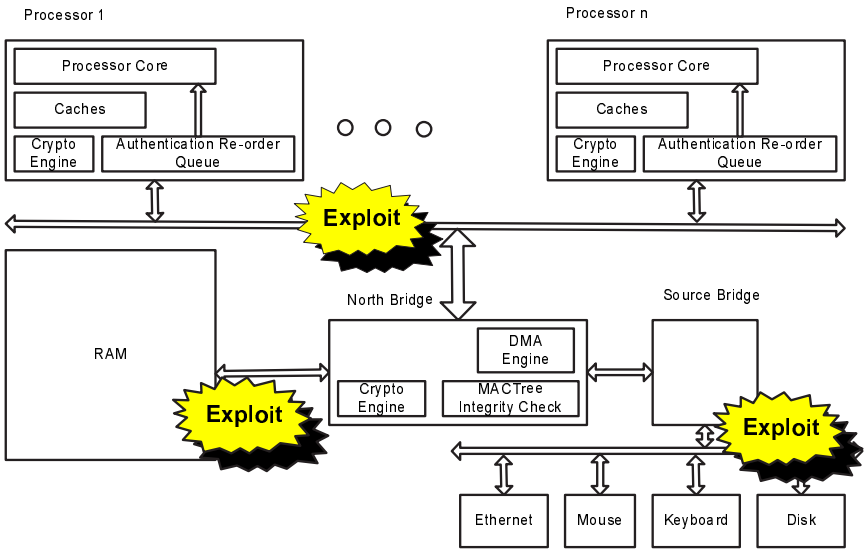


Figure 57: MP platform

Replay Attacks: One of the major challenges of designing a secure SMP platform is how to prevent and detect replay attacks. There are two types of replay attacks: 1) replay logged bus transactions, including both the cache-to-cache and the memory-to-cache bus transactions; 2) replay information stored in the physical RAM. Under the system where the memory is exclusively owned by a single PE (i.e. core processor), replay attack can be prevented using MACTree or CHTree inside the PE. But such solutions do not apply when the shared memory can be updated by multiple PEs. If each PE maintains its own integrity verification tree, to verify and synchronize the root signatures of these authentication trees across multiple processors could be cumbersome and lead to significant performance impact on frequent inter-processor communication.

Distributing Secrets: Another challenge of designing a tamper-proof SMP system is how to distribute and share secret information among processors and devices. Such shared information may include symmetric cryptographic keys, shared sequence number, etc. Distributing and sharing secrets is a unique problem to SMP shared memory protection.

In summary, shared memory authentication is a unique problem. To enable a fast, secure, and unified solution for authenticating shared memory, we propose a distributed scheme where both the main processor(s) and the chipset contribute to create a secure environment for trusted software execution. Our shared memory authentication scheme is

universal for both SMP shared memory systems and uni-processor systems with a memory shared by a core processor and other peripherals/agents.

Instead of having each PE to use its own MACTree, our solution enables shared memory protection through a centralized MACTree implemented in the memory system with a secure SMP coherent bus protocol. The shared memory protection is achieved by securing the data path from each PE to the shared system memory. With the MACTree embedded in the North Bridge (i.e. the memory controller), the data path between the memory controller and the physical RAM is secured. Moreover, the secure SMP bus protocol provides a trusted and authenticated environment for both cache-to-cache and memory-to-cache bus transactions. Untrusted devices cannot commit any bus transaction to the protected shared memory and any attempt to replay past authenticated bus transactions can also be detected. Since both the data path from each PE to the system memory and the data paths among PEs are protected, secure and authenticated sharing of the system memory is provided. Different from the previous approaches that put all the hardware resources for memory protection into one single secure processor, our solution provides a trusted environment for SMP shared memory through securing the platform.

8.2 Security Model for Shared Memory

In this section, we present the detailed security model and architectural support for shared memory authentication. We only focus on symmetric multiprocessor (SMP) systems where a coherent snoopy bus and a large physical RAM are shared by a number of processors. It is straightforward to extend the solution to situations where each processor maintains a local memory. The proposed SMP shared memory protection scheme can be used to ensure both integrity and confidentiality for SMP shared memory. We first present a SMP platform oriented security model, then, architectural supports for the security model.

8.2.1 Multiprocessor Security Model

Figure 57 shows target architecture of SMP systems. A split-transaction, cache coherent system bus connects each processor to the shared memory. The security model holds no specific assumptions about the coherent bus protocol, neither is it tied to any particular

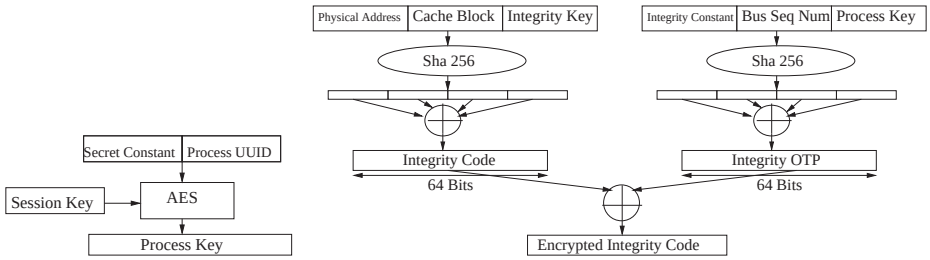
SMP system. Examples of applicable SMP systems include SGI's POWERpath-2, Alpha's SMP protocol, and Intel Xeon based SMP systems. To simplify the discussion, the scheme to be presented uses a split-transaction, SGI Powerpath-2 like cache-coherency protocol [21]. Applying the scheme to other SMP system should be straightforward with little changes. The shaded blocks in the system are trusted and protected components. The dark stars denote points of potential attacks.

The proposed security model provides a system level specification for constructing a trusted environment for SMP software execution. It addresses four issues — management of protected SMP processes, distribution and sharing of secrets, protection on integrity and secrecy, and software distribution. The security model assumes the existence of a secure OS kernel [22]. A secure OS kernel is a set of trusted core OS services. These services are executed in a trusted domain. The secure kernel is verified by a secure BIOS during system boot [5].

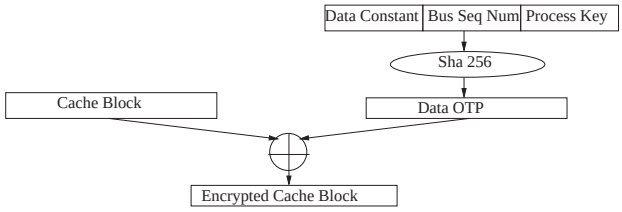
8.2.1.1 Process Control

One basic and essential protection on a multi-tasking system is *process or task isolation*. Different process should not be allowed to access other processes' protected domain. To achieve *process isolation*, two conditions must be satisfied. First, each process must be uniquely identified. Second, unique per-process cryptographic information must be used for protecting integrity and confidentiality of each process's memory. The traditional *process id* (pid) is not a good choice because the likelihood of reusing a pid is very high. Here we define a unique 128-bit number, *process uuid*, a universal unique identifier to uniquely identify a process. The *process uuid* is obtained from a random number generator. The process uuid itself is not considered as secret and can be securely shared among multiple processors during initialization by the secure kernel. Process uuid is treated as process context and is protected against tampering during context switch.

Secret padding and keys used for authenticating or encrypting memory information of each process is derived from each process' uuid. A new privileged instruction which is used only by the secure kernel is introduced to set up process uuid, called *set_uuid*.



(a) Process Key Generation (b) Integrity Code Generation (*Integrity Key* and *Bus Sequence Number* generation are described in figure 60)



(c) Data Encryption

Figure 58: Security Operations on Each Cache Block Evicted from Processor or Transmitted as Coherence Miss

Execution of the instruction includes several steps. One step involves that the processor assigns the *process uuid* to an internal uuid register and computes a process key as described in figure 58(a). The session key shown in the figure is created at boot time and described in detail in Section 8.2.1.3. Secret Constant is an initial value, same for all Processing Elements (PE's) and known only to each PE and the secure memory controller. Other steps of the instruction *set_uuid* are described later in section Section 8.2.2.2. relate to how the *process uuid* is shared by other devices attached to the shared bus.

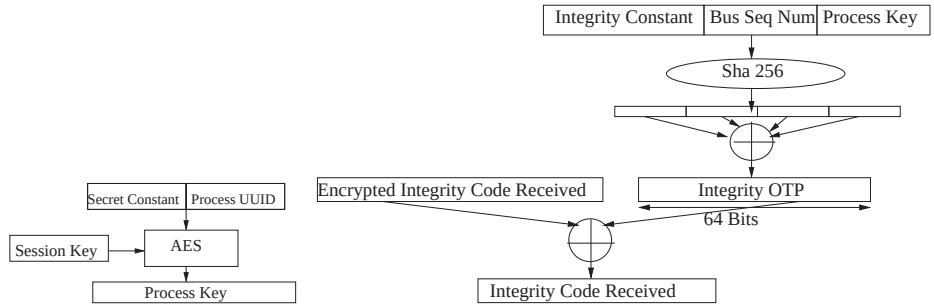
8.2.1.2 Integrity and Confidentiality Protection

Message authentication code (MAC) is a well established technique for guaranteeing data integrity by verifying whether a piece of received or retrieved data was tampered during transmission or storage. For the purpose of memory integrity protection and authentication, before a PE stores a chunk of data to the insecure memory, it will compute an integrity code using a MAC generation algorithm and keep it alongside the data. In the case of digital rights protection or secure software execution, the data itself may or may not be encrypted depending on the security requirement. Later, when the same PE or any other PE attempts to access the data, the integrity of the data will be verified by re-generating the integrity

code of the retrieved data using the same MAC algorithm and comparing it against the stored one. Any tampering to the stored data will be detected when a mismatch occurs.

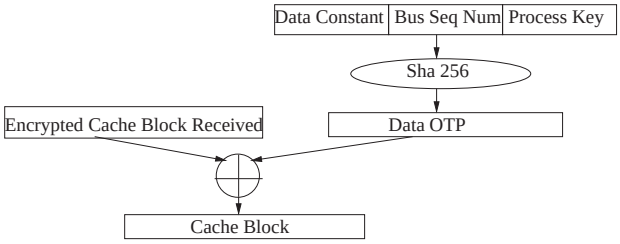
In our shared memory authentication scheme, each PE (including the North Bridge) is responsible for computing the integrity code for data to be stored to the memory or requested by other PEs. When the data is shared by multiple PEs, the *key* along with other necessary information for generating/verifying the integrity code must be shared among all the involved PEs. The integrity code is encrypted when it is transmitted through the shared system bus. A counter mode class encryption pad is uniquely computed using a confidential shared bus sequence number which is tracked by all the PEs on the system bus. The sequence number is incremented by all the devices attached to the system bus after each bus transaction. For protecting confidentiality of either information stored to the memory and coherence response to other processor's request, the data is also encrypted using encryption pads computed based on the shared bus sequence number.

Figure 58 shows needed operations for a cache block that is either dirty-evicted from the protected domain or requested by other processors. The operations illustrated are conducted by a PE on each protected cache block to be written to the system bus. They involve, generation of the process key using the process uuid (figure 58(a)), generation of the encrypted integrity code using the generated process key, bus sequence number, the data to be written and the integrity key (figure 58(b)) and finally encryption of the data using the bus sequence number and the process key (figure 58(c)). SHA256 [50] and AES128 [20] are hash and encryption standards. Integrity key is a 256-bit secret shared by the units attached to the shared system bus. Session key is an AES key uniquely initialized every time after the system is started. Distribution of the shared secrets such as the sequence number and the session key is addressed in Section 8.2.1.3. For two blocks next to each other in the figures implies they are concatenated. The $\oplus$ stands for XOR. Both the integrity key and the sequence number are hidden from software access and can not be accessed externally. Similarly, computed data such as *integrity_code* and the *process_key* are also hidden from software access. Only encrypted integrity codes and encrypted cache blocks are observable as they are transmitted over the shared bus. All the PEs share the same constant values

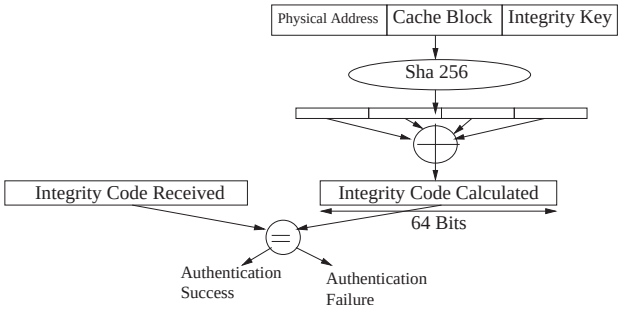


(a) Process Key Generation

(b) Reading Received Integrity Code with Process Key



(c) Decrypting Data with Process Key



(d) Authentication of Data

Figure 59: Security Operations on Each Cache Block Received

which are burnt inside each PE. Most of the shared secrets, such as the session key, the integrity key, and the sequence number are not fixed constants. They are uniquely assigned each time after the system is booted using approaches described in Section 8.2.1.3.

Integrity verification and decryption of received cache block (coherence reply and memory read) are shown in figure 59. Reading involves, computation of the process key (figure 59(a)), decrypting the received encrypted integrity code and encrypted data using the process key (figure 59(b) and figure 59(c)) and finally recomputing the integrity code from the decrypted data to compare with the received integrity code for authentication (figure 59(d)).

The security model shown in figure 58 and figure 59 minimizes the performance critical

path between encryption and decryption. The encryption_OTP is pre-computed. In the best scenario, the interval of transferring an encrypted cache block consists of only time of a XOR operation on the sender, transmission delay, and another XOR operation on the receiver. Integrity check requires much more time because integrity code has to be computed before transmission and verified after the cache block is received.

8.2.1.3 Secrets Distribution and Sharing

How to securely distribute and share secret such as the keys, the padding, the sequence number, and etc., is a major challenge for designing a secure distributed system. Obviously, the secret can not be broadcast as plaintext over the system bus. Integrity of the shared secret also has to be maintained so that it can not be forged. Furthermore, the shared secret such as the session keys, the sequence number must be different each time the machine is rebooted to prevent replay attack.

Similar to a regular symmetric multiprocessor system, one processor has to be designated as the boot processor to bring up the system. This processor will execute its secure BIOS and boot into a secure OS. The uniqueness of our solution is that the shared secrets themselves are not transmitted, instead they are computed by each involved processor in a secure way based on information that can be openly shared.

During boot time, the bootstrap processor broadcasts the range of physical memory to be protected to all the processing units and the memory controller. It could be a portion of the entire physical address space or all the physical RAM space. After that, it starts key generation. During key generation, each unit attached to the shared bus is granted bus cycles in turn to broadcast a random 64-bit number. Then each unit concatenates all the random numbers it collects from the bus including the one it broadcasts and computes a hash value using some hash function. The hash result is truncated into a 128-bit AES session key. Then, all the shared secrets including the shared bus sequence number, the process key, the integrity key are all synchronously computed based on the session key as shown in figure 60.

Both the *secret_hash_key* and the *secret_constant* are constants permanently burnt into

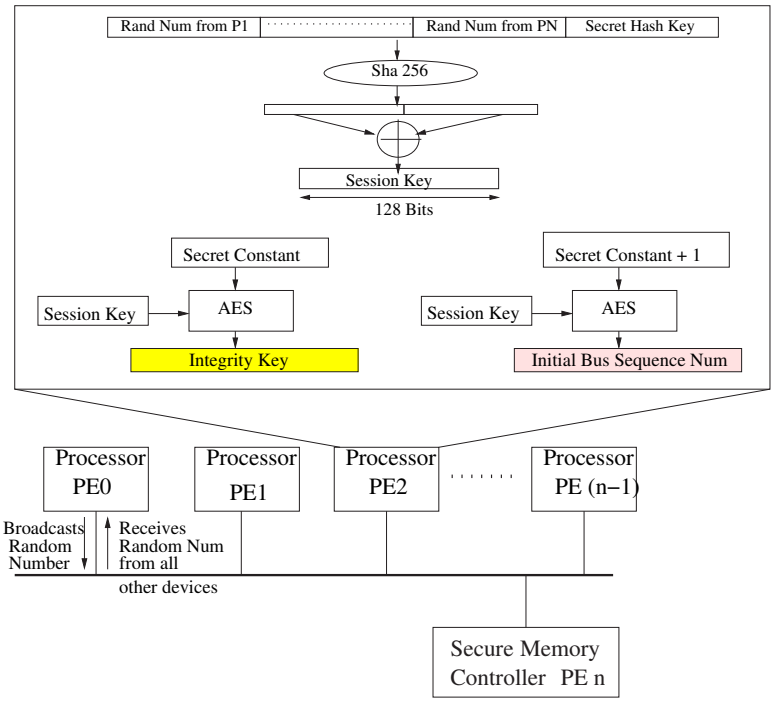


Figure 60: Processor Initialization and Distribution of Shared Secrets

the processor chip, and the memory controller during manufacturing. They are secrets stored in the chip inaccessible by either software running inside or any device from outside. After a device creates all the required keys and numbers, it will enter a synchronization barrier. After all the devices on the shared bus complete key generation and enter the synchronization barrier, regular bus and memory transaction are resumed with the appropriate protection specified in this chapter.

The session key and bus sequence number are not tied to a particular process and are not considered part of a process context. But a process is free to specify whether segments/pages of its virtual memory should be mapped to the protected physical memory. Note that a different session key is generated each time after the system is freshly rebooted.

8.2.1.4 Software Distribution and Platform Key

The cryptographic protection proposed is "self-contained" because the session key, the root of all the keys, the sequence number, etc, are not constants and will be modified each time

after reboot. Software vendors are not able to generate the integrity code or encryption pads used in the protection because they do not have the session key. To execute software either encrypted or authenticated by vendors in the mode with the proposed protection, conversion from the vendor protected domain to the SMP platform protected domain is required. This is achieved through a platform key. A platform key is a pair of public-private keys with private key permanently burnt into the SMP's chipset. Vendors encrypt the symmetric cryptographic key used to encrypt/authenticate a software with the public platform keys. When the software is copied to memory from its disk image, it will be decrypted, authenticated using the keys set by the software vendors, and then re-encrypted using the methods described in figure 58 and figure 59. As we will describe next, this conversion does not necessarily require processor involvement and can be performed in high speed with security enabled DMA engines.

8.2.2 Architectural Support of SMP Security

In this section, we present a detailed architecture model for implementing the SMP security model described in the previous section. There are three security enabled platform architecture components, the shared system bus, the memory controller, and the secure processors. Extra security related functionality has to be added to these components to support the proposed SMP security model. Furthermore, new techniques must be invented to minimize the performance impact of security verification. For SMP systems and benchmarks, integrity verification is a significant performance influencing factor because integrity code of coherent response of cache-to-cache communication has to be computed, transmitted, re-computed, and verified. To tolerate the latency of integrity checking, we propose two new techniques. First, we propose a split transaction bus model for data and its integrity code. Second, we use secure decryption and authentication disassociation described in 7 to further hide the latency of integrity checking.

8.2.2.1 Secure Symmetric Coherent Bus Protocol

The purpose of the secure multiprocessor bus protocol is to prevent spoof and replay attack on the shared coherent SMP bus. It plays an essential role for providing a chain of authentications for both cache-to-cache and memory-to-cache accesses. Although the principle of how we secure the SMP bus is in fact not tied to any SMP coherence bus protocol, but for the sake of discussion, we restrict the design to a four-state coherence protocol similar to SGI POWERpath-2 with cache-to-cache transfer triggering a write-back to memory. Each cache has four states; invalid, exclusive, dirty exclusive, and shared. Each transition between states is either initiated by the processor or by a coherent transaction. A duplicate set of cache tags [21] is maintained by each processor interface ASIC and bus arbitration is done in distributed manner. Similar to POWERpath-2, every bus transaction requires five clock cycles. A system wide bus controller logic executes the same five-state machine synchronously: arbitration, resolution, address, decode, and acknowledge.

All the devices on the shared multiprocessor bus including the memory controller share the same secret 64-bit bus transaction sequence number described in figure 58 and figure 59. Since all the bus transactions are visible to all the units attached to the snoopy SMP bus, it is straightforward for a unit on the bus to update and keep track of the sequence number. After a bus transaction completes, every unit on the bus *increments* its copy of the sequence number internally. The sequence number is initialized during system boot as shown in figure 60. The number is kept as secret by all the involved devices and never transmitted in either plaintext or ciphertext on the bus.

One unique performance feature of our secure bus is the split transaction of data and its integrity code. We may infer from figure 58 and figure 59, that integrity code generation and verification is the critical path of the SMP security model. To minimize the impact of authentication on performance, our secure bus model allows data block and its integrity code transmitted separately. For coherent response, a cache block can be transmitted first followed by the encrypted integrity code after it is computed. The unverified data will be used by the processor pipeline of the destination processor speculatively under the secure decryption-authentication decoupling constraints. After the integrity code is finally

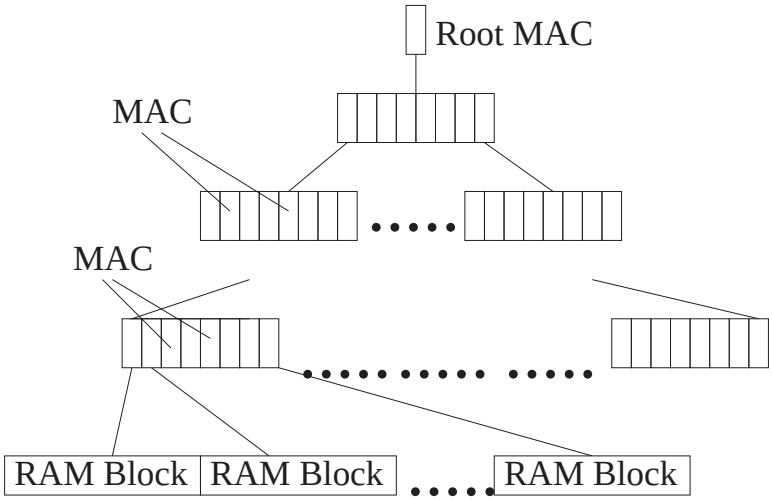


Figure 61: MACTree

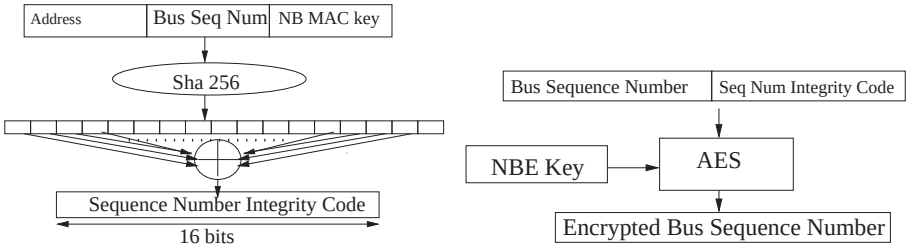
received and verified, completed instructions using the unverified data can be retired and stalled memory fetches can be issued.

Note that the 64-bit bus sequence is good enough for security protection. This is because for a bus running at speed of hundreds of MHz or a few GHz, it would take hundreds of years if not thousands for a 64-bit sequence number to wrap-around.

8.2.2.2 Secure Memory System

This section describes the architecture of the secure memory system consisting of a memory controller with an integrated security engine, and a number of physical RAM chips. It is worth mentioning that integrity code alone is not sufficient for memory integrity protection because an adversary can replace a piece of current data and its MAC with some old stale data and MAC. Such replay exploit cannot be detected without using techniques such as MACTree.

The primary goal of the security engine embedded in the North Bridge memory controller is to detect tampering or replay of data stored in the system memory. It is another critical component in the chain of shared memory authentications. A simple solution is to have either Merkle hash tree or MACTree implemented in the memory controller. Note that the integrity code itself is not transmitted in plaintext over the SMP bus and is unknown to the hackers. Organization of the MACTree is shown in figure 61. A leaf node represents an



(a) Sequence Number Integrity Code Generation

(b) Encrypting Bus Sequence Number with Sequence Number Integrity Code

Figure 62: Bus Sequence Number Encryption

individual integrity code and each internal node denotes a MAC of all the children nodes.

The detailed operation of a memory write is as follows. After the integrity of received data is verified, the memory controller will update the MACTree by substituting the new integrity code (after it is XORed with the integrity encryption pad) into the tree. Then it will send the data with the encrypted integrity code to the memory. To be able to verify the integrity code later, the memory controller will also store the encrypted bus sequence number to the memory. Each bus sequence number can be encrypted using AES as shown in figure 62. Both the *NB_MAC_key* and the *NB_E_key* (where NB stands for North Bridge and E stands for encryption) used during encryption are secret information maintained by the North Bridge itself.

To improve performance, the bus sequence numbers of frequent data blocks can be cached inside the North Bridge or speculated using the counter prediction technique described in chapter 4.

Upon receipt of a read request, the memory controller will fetch both the data and the associated encrypted integrity code from the physical RAM. The corresponding encrypted bus sequence number will also be retrieved if it is not cached in the North Bridge. The authentication mechanism will extract the original integrity code using the approach detailed in figure 59. To verify whether the integrity code and the data is a replay, it is inserted into the MACTree. Starting from the bottom of the tree, recursively, a new MAC is computed and compared with the cached internal MACTree node. If a match is found, the integrity code is verified valid. Since it is impractical to cache all the internal nodes of the MACTree, some internal MACTree nodes will be stored in the insecure system memory and brought

into the memory controller when needed. To prevent from leaking sensitive information and jeopardizing security, confidentiality of the internal MACTree node has to be maintained. This is achieved by encrypting the internal MACTree node using 128-bit AES encryption scheme.

The aforementioned security operations introduces additional latency overhead to the critical path of fetching data from the shared memory. A number of design and architectural optimizations are proposed to address the latency overhead of security protection.

- The integrity encryption pad and data encryption pad are pre-computed. Since the bus sequence number always increments, each PE can speculatively pre-compute a number of encryption pads ahead of the current bus sequence number and store the result pads in a small buffer.
- To speed up the process of verifying retrieved integrity code from memory, the North Bridge can pre-fetch bus sequence number or pre-compute encryption pads if addresses of future memory accesses can be predicted or speculated.
- The North Bridge can use sequence number speculation as described in chapter 4 to overlap decryption operations with memory fetch.
- Both the MACTree node or the bus sequence number of frequent data blocks can be cached to improve memory access speed.

The secure memory controller (North Bridge) is the center of the proposed SMP security model. In addition to the MACTree, it also shares secrets with other processors on the shared bus, maintains platform key pairs described in the security model, and transforms protected information from the software vendor's domain to the platform's domain.

The memory controller holds several security oriented registers. Fixed addresses are assigned to these registers and access to these registers must be performed through protected bus transactions. First, there is a *process uuid register* in each processor. During execution of a *set_uuid* instruction by a secure processor, the processor also issues a secure write access to the corresponding uuid register in the North Bridge so that a process key can be derived by the memory controller. Upon receipt of a new uuid value, the memory

controller computes its version of the current process key based on figure 58. Similar to the uuid register, there are North Bridge registers assigned for holding software vendor keys encrypted by the platform's public key. The North Bridge can extract the vendor keys using the private platform key. When the vendor keys are enabled, transactions from peripheral devices such as disks, network devices are first verified or decrypted using the vendor keys and then converted using the process key and the integrity key understandable by the secure processors based on figure 58. The platform key pair is permanently burnt in the North Bridge.

Note that the secure OS kernel always resides in a separately protected memory space. Access to the secure kernel uses a different process key and *set_uuid* is not needed when application switches to secure kernel mode. The memory range of the secure kernel is also maintained by the North Bridge. The uuid registers and encrypted vendor key registers in the North Bridge reside in the secure kernel memory space. Another important security role served by the North Bridge is the conversion of memory protection when data is transmitted from peripherals such as disk to the physical memory. The conversion mechanism supports both DMA based and processor based memory operations. For DMA, the involved secure processor initializes both the related uuid register, and the software vendor key register first, then starts the DMA engine. The memory controller will automatically verify and convert protections from vendor's domain to the platform's domain for every chunk of data written to the memory. Similarly, when results in the memory are DMAed to the peripherals, they can be optionally converted back to the software vendor's domain. Both operations can be achieved with support of the security DMA engine without increasing workload on the secure processor.

8.2.3 Security Analysis

This section provides a security analysis for both the SMP bus protocol and the secure memory system. The objective of SMP shared memory authentication is to prevent unauthorized tampering and replay of coherent response and data stored in the shared memory. There are a number of potential exploits adversaries can try and we will show that none

of them will succeed in breaking the proposed protection mechanism. First, an adversary may try to forge the integrity code. This clearly does not work since each PE can verify the integrity code and the integrity code is generated using a strong cipher. The integrity key is a secret and it is re-generated after the machine is rebooted. Second, an adversary may try to replay both data and the associated integrity code on the SMP bus, this will fail because every bus transaction is protected by a sequence number that does not stay the same. Third, an adversary may try to do replay attack on the memory. This would not succeed neither because of the MACTree protection. Moving memory block and its encrypted MAC around does not work neither because the encrypted MAC is generated using address as part of the input. Replay data written by a different process will also fail because the MAC is encrypted by encryption pads generated using a process key that is unique to each process. Finally, security privileged instructions such as *setup_uuid* can be only used inside the secure kernel. User program is not allowed to use these instructions thus preventing spoofing of a process uuid.

8.3 Performance Analysis

8.3.1 Memory overhead

The sequence numbers associated with each cache line size RAM block and the intermediate nodes of the MACTree are stored in the shared system RAM. The space needed is approximately 1/(m-1) of the RAM size with an m-ary balanced MACTree. For example, for a 256-bit cache line with a 64-bit sequence number and MAC, the RAM overhead is about 25% of the protected RAM space. Note that the scheme allows only portion of the whole RAM protected. It is up to the system on how the protected physical memory is allocated. The caches implemented in the memory controller are MACTree caches for the MACTree nodes. They are typically small from 32KB to no more than 64KB.

8.3.2 Simulation Environment

For characterizing and evaluating our proposed MP system, we use RSIM [51] as our infrastructure to simulate a 4-node MP system. Each node includes a MIPS R10000 like out-of-order processor, L1, and L2 cache. We modified the simulator to support the SGI

Table 10: Applications and input parameters

Application	Parameters
`lu`	256 by 256 matrix, block 8
`radix`	512K keys
`water`	343 molecules
`quiksort`	32768
`mp3d`	5000
`fft`	65536

Table 11: Processor model parameters

Parameters	**Values**
CPU	4-issue per cycle
reorder buffer	64 instructions
load/store queue	64 instructions
L1 cache	8-Kbyte, directly mapped
L2 cache	4-way, Unified, 32B line, 256KB
L1 Latency	1 cycle
L2 Lat (256KB)	3 cycles
Memory Latency	X-5-5-5 (cpu cycles) X depends on mem page status
Memory Bus	200 MHz, 8B wide
SHA-256 Latency	80ns
AES Latency	80ns
MACTree	2Way; 8KB,32KB; 64B line

POWERpath-2 MP coherent bus protocol and a shared main memory. Secure snoopy bus protocol, memory authentication, and authentication speculative execution are all implemented into the RSIM simulator. To characterize the memory transactions more accurately, we integrated an accurate DRAM model [25] based on the PC SDRAM specification. SPLASH-2 benchmark suite [71] was used. The SPLASH-2 benchmark applications [71] and its input parameters use in this study are listed in table 10. table 11 lists the basic processor configuration parameters used throughout the experiments unless otherwise specified. The latencies of MACTree caches were obtained using CACTI [69].

8.3.3 Performance

8.3.3.1 Impact of Authentication Approach

Figure 63 compares the performance between authentication-then-issue with that of secure-speculative -authentication. The figure shows IPC normalized to the baseline[2] under two scenarios, *authentication-then-issue*, and *secure speculative authentication.* In authentication-then-issue, data brought into each processor from the external is not issued for use until its

[2]Baseline has zero security protection of any kind.

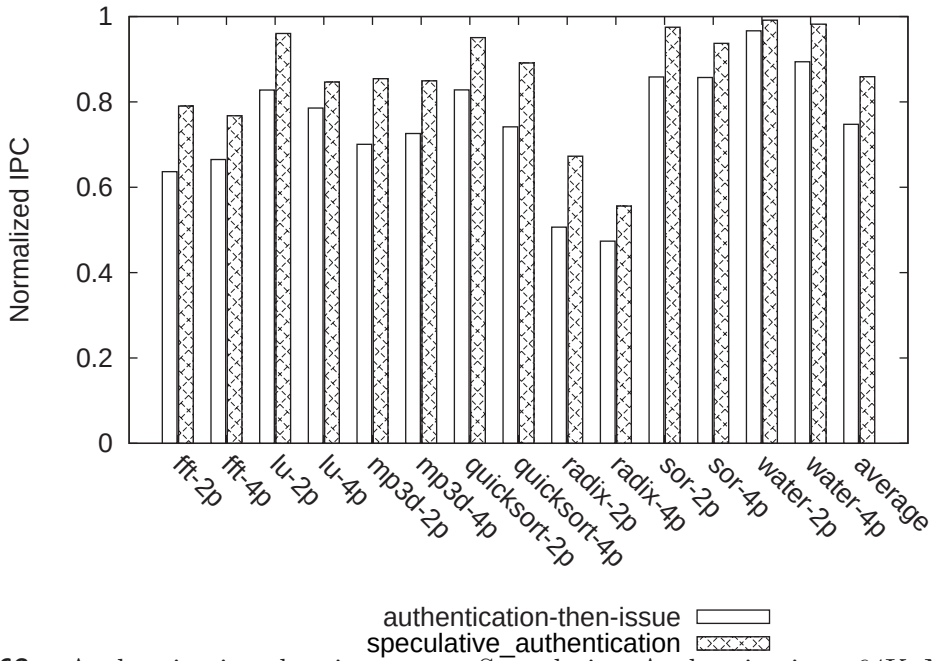


Figure 63: Authentication-then-issue vs. Speculative Authentication, 64K MACTree Cache, Counter Prediction

integrity is verified, while in *secure speculative authentication*, unverified data is speculatively processed under certain security constraints and restrictions - see chapter 7. We experimented two MP settings, 2P and 4P systems because dual and quad processor platforms are the most popular choices for today's commercial workstations. The results were collected under the simulation configurations that all the shared memory data were encrypted with MACTree authentication enabled. The size of the MACTree cache is 64KB. The results indicate that speculative-authentication delivers faster performance than *authentication-then-issue*. The average IPC speedup is 15%. For some benchmarks, such as fft, mp3d, quicksort, the speedup is more than 20%. In general, the speedup under 2P setting is more significant than the speedup under 4P.

Figure 64 and Figure 65 show two important profiling results of the benchmarks, combined L1/L2 cache misses and proportion of memory references with respect to the total number of instructions executed. The profile data show that some benchmarks such as quicksort, radix, and mp3d. are memory bounded applications because they have both high memory access ratio and relatively high cache miss rates. Results of Figure 64 and Figure 65 will be used later for explaining some applications' performance results.

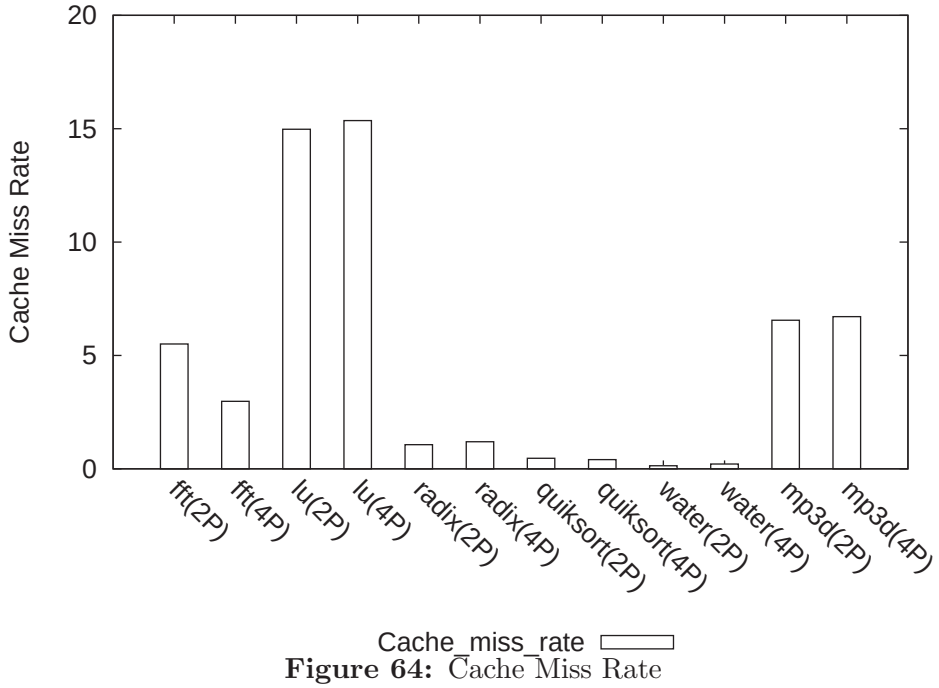


Figure 64: Cache Miss Rate

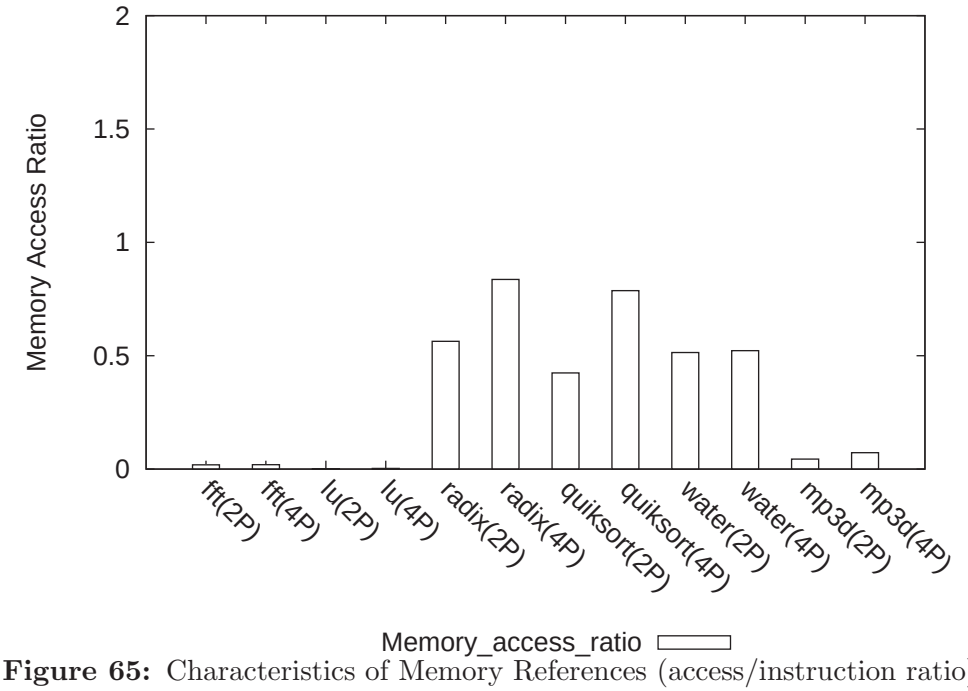


Figure 65: Characteristics of Memory References (access/instruction ratio)

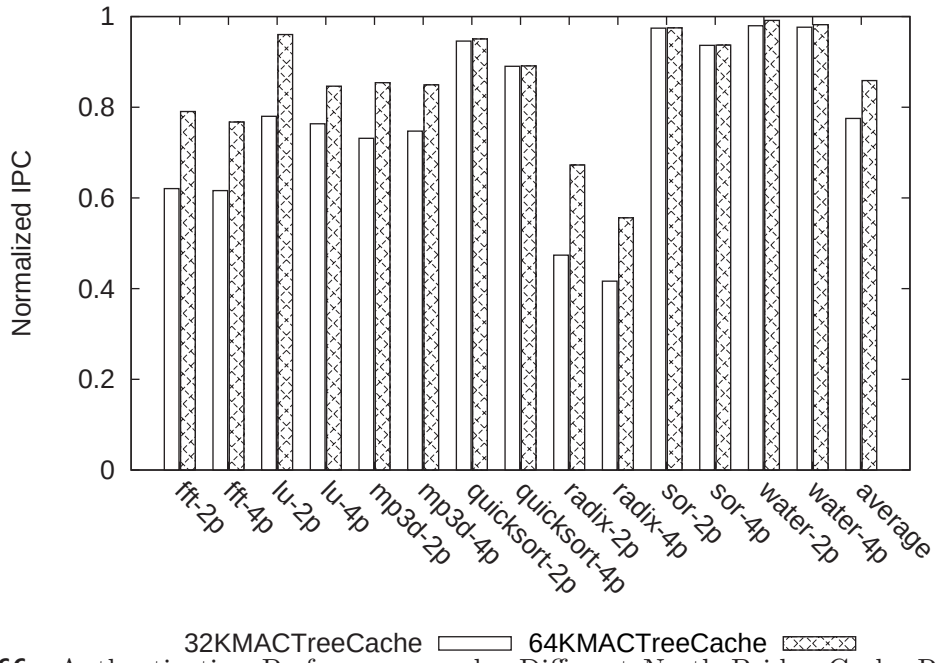


Figure 66: Authentication Performance under Different North Bridge Cache Resources, Processors=4

8.3.3.2 MACTree Cache

When the share memory is protected with the MACTree integrity verification. The overall performance would be sensitive to the amount of cache resources in the North Bridge. Results in figure 66 show the effects of MACTree cache size using IPC normalized to IPC of an ideal MACTree situation. It compares two MACTree cache settings, 32KB vs. 64KB. On average, a 64KB MACTree cache can deliver higher normalized IPC than that of 32KB MACTree cache. The average speedup is about 10%. For certain benchmarks, such as fft, mp3d, and radix, the speedup is higher than 10%. Comparing with the baseline of no security, the average normalized IPC under 64K MACTree cache is about 86% of the baseline condition. For some benchmarks, such as water, sor, the performance degradation caused by shared memory security protection is less than 5%. For benchmarks fft and radix, the performance loss is more than 15%. As a general trend, for almost all the benchmarks, the normalized IPC under 2P setting is often higher than that of 4P setting, which means that comparing 2P with 4P, performance degradation is smaller.

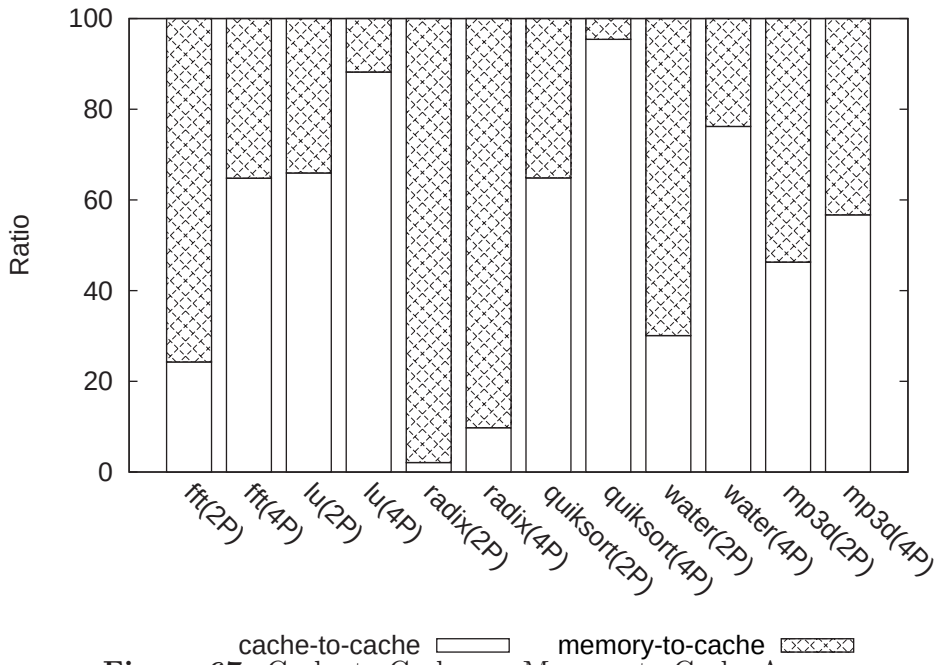


Figure 67: Cache-to-Cache vs. Memory-to-Cache Access

The latency decryption overhead is un-balanced for the cache-to-cache and the memory-to-cache accesses. For the cache-to-cache access, the overhead is very small because the data is encrypted and decrypted by two XOR operations given that the decryption pad is pre-computed. For the memory-to-cache access, the overhead is bigger because the decryption pad can not be pre-computed. However, the memory controller can start the decryption process as soon as the request is received using sequence number speculation. Figure 67 gives results showing categorizations of the L2 misses, in which, most of the benchmarks except for a few like "radix" show more cache-to-cache accesses than memory accesses. The behavior of the "radix" benchmark may be explained by the fact that it actually does a radix sort on a huge array of non-negative numbers. The huge number of memory accesses in "radix" may be explained as capacity misses for the large array. We may also observe from the figure that cache-to-cache accesses for a four processor system is larger than for a two processor system. This is because of that for a four processor system there is more data in the caches which causes more communication among them.

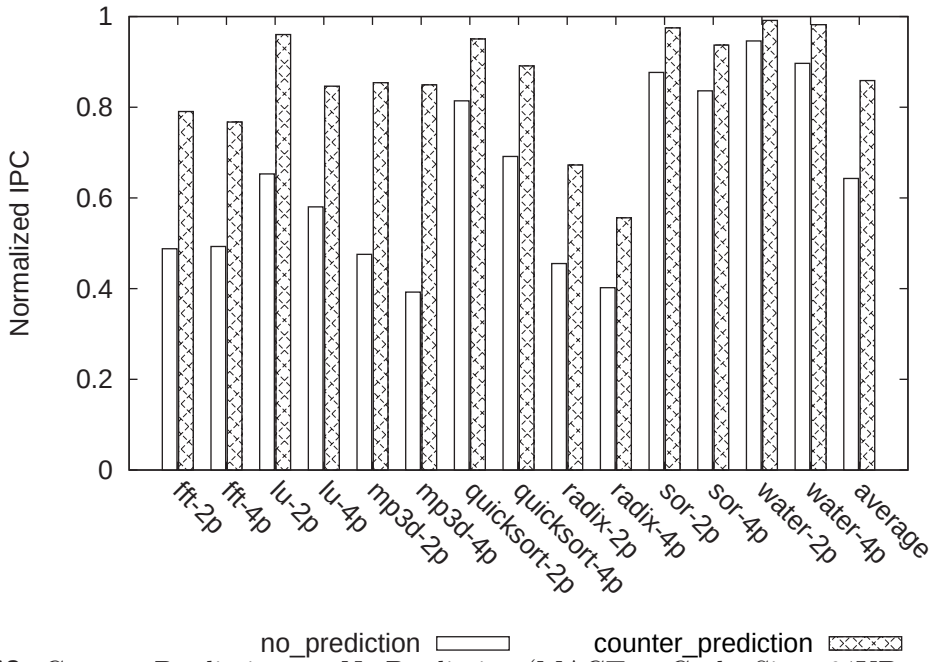


Figure 68: Counter Prediction vs. No Prediction (MACTree Cache Size=64KB, no counter cache)

8.3.3.3 Counter Prediction

Counter prediction is implemented in the North Bridge to reduce the overall RAM fetch latency. Figure 68 compares normalized IPC results between with counter prediction with without prediction. The non-prediction case does not use any counter cache. As the results indicated, for all the benchmarks, the counter prediction achieves higher IPC than non-prediction. The average IPC speedup is close to 30%.

8.3.3.4 MACTree Integrity Verification Disabled

The main purpose of MACTree integrity verification is to detect replay attack. If such an attack does not pose a major security threat, it can be disabled, which will certainly bring down the overhead of shared memory protection. Figure 69 shows the performance results as normalized IPC. With MACTree disabled, the average normalized IPC is about 95% of the baseline, which means very small amount of performance degradation. For all the benchmarks, the normalized IPC is over 90%. Again, as discussed before, in most cases, the normalized IPC under 2P setting is often higher than the normalized IPC under 4P setting.

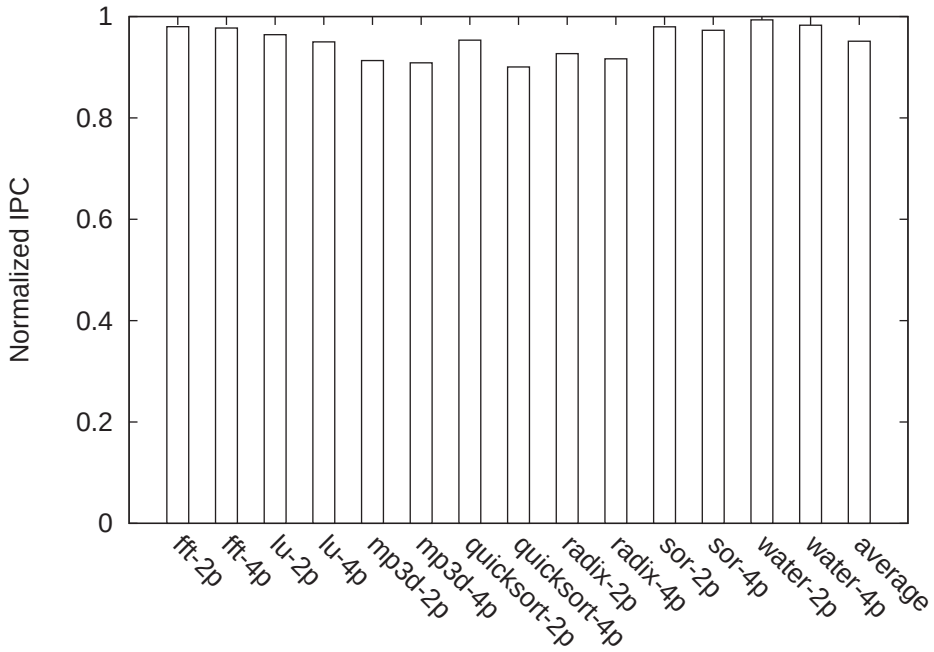


Figure 69: Normalized IPC With MACTree Integrity Verification Disabled

8.4 Conclusions

In this chapter, we propose a unified hardware-based memory protection scheme for both uni-processor and SMP platforms. Different from the previous endeavors on uni-processor memory protection, the design achieves memory security in a platform distributed manner and relies on a light-weighted secure processor implementation.

CHAPTER IX

CONCLUSION

In this thesis, we propose and evaluate a set of silicon based memory security primitives and services that allow system designers to innovate and experiment with new types of secure computing systems that take advantages of the improved tamper-proof protection and enhanced performance over many existing approaches. The proposed memory security model and secure processor architecture protects digital content and software stored in un-trusted system memory from both software and simple physical tamper.

The thesis introduces a set of architecture innovations that aim for secure and high performance implementation of the proposed security model and secure processor architecture. These include,

- High performance and low area overhead design of hardware memory encryption/decryption units. The thesis compares and evaluates the architecture and performance implications of several standard encryption modes on memory protection. It addresses the issue of finding the proper encryption mode that is both secure and efficient. In addition to evaluating different encryption modes, we optimize memory decryption process by introducing a couple of unique prediction techniques that can significantly reduce the latency overhead of fetching encrypted digital information from memory. First, the thesis describes a prediction technique called *OTP prediction/sequence number prediction* that removes much of the latency penalty of fetching "counter mode" encrypted data from memory. Based on adaptive prediction of memory update frequency, the *OTP prediction* allows speculative pre-computation of decryption pads and pushes the advantages of using counter mode for protecting randomly accessed memory to its limit. Second, the thesis combines "value" prediction and hardware memory cryptography model for hiding latency overhead of direct encrypted memory without sacrifice of security.
- High performance tamper proof memory integrity verification and authentication. The

thesis uses a MAC (message authentication code) tree based approach to prevent tamper on integrity of digital data or code stored in the physical memory. Comparing with other similar techniques, our memory authentication approach and implementation have the following benefits. It requires less area, has faster performance, and at the same time is secure. Furthermore, the thesis describes in detail the different approaches of integrating integrity verification with memory decryption under the modern out-of-order processor architecture. The thesis compares those approaches in terms of security strength against various attacks, support for precise interrupt, implementation complexity, and performance.

- A fast and secure means for protecting memory shared in a SMP (symmetric multiprocessor) system. In this thesis, we propose a unique distributed approach that uses both security enabled processors and a secure memory controller (also called *North Bridge*) to provide integrity and confidentiality protection of SMP shared memory. It includes: 1) an innovative secure multiprocessor bus protocol for authenticating coherent bus transactions; 2)a fast memory authentication approach; and 3) a speculative integrity verification mechanism to tolerate the latency of memory authentication for both processor-to-processor and memory-to-processor accesses. The secure authentication mechanism is not only fast, but also secure, and supports precise interrupts for security exceptions.

Building a secure system is a never ending challenge. It is impractical to build a maximum secure computing system without considering power, area, cost, backward compatibility, programmability, usability, etc. The end result is often a compromise among many diversified sometimes contradictory requirements. The thesis represents a small step toward solving those issues and paves the road for future research instead of trying to solve all the problems once-for-all. We envision that real computing systems with strong tamper-proof support will emerge in future in many areas such as military embedded systems.

REFERENCES

[1] "Secure Digital Music Initiative." http://www.sdmi.org/. Oct/2005.

[2] "Secure Video Processing Alliance." http://www.svpalliance.org/. Oct/2005.

[3] "Hashes and Message Digests." http://www.cc.gatech.edu/classes/AY2004 /cs6262_spring/hashes.ppt. Dec/2004.

[4] Alliance, T. T. C. P., "https://www.trustedcomputinggroup.org/home,". Dec/2003.

[5] Arbaugh, W. A., Farber, D. J., and Smith, J. M., "A secure and reliable bootstrap architecture," in *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, p. 65, IEEE Computer Society, 1997.

[6] Austin, T. M., "Simplescalar 4.0 Release Note." http://www.simplescalar.com/. Jan/2004.

[7] Baer, J.-L. and Chen, T.-F., "Effective hardware-based data prefetching for high-performance processors," *IEEE Trans. Comput.*, vol. 44, no. 5, pp. 609–623, 1995.

[8] Base, N.-G. S. C., "http://www.microsoft.com/resources/ngscb/default.mspx,". Aug/2004.

[9] Bellare, M., Desai, A., Jokipii, E., and Rogaway, P., "A concrete security treatment of symmetric encryption," in *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97)*, p. 394, IEEE Computer Society, 1997.

[10] Bellare, M. and Namprempre, C., "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm," *In Advances in Cryptology — Asiacrypt 2000 Proceedings, Lecture Notes in Computer Science*, vol. 1976, 2000.

[11] Biham, E., "Cryptanalysis of multiple modes of operation," in *Proceedings of the 4th International Conference on the Theory and Applications of Cryptology*, pp. 278–292, Springer-Verlag, 1995.

[12] Black, J. and Rogaway, P., "CBC MACs for arbitrary-length messages: The three-key constructions," *J. Cryptol.*, vol. 18, no. 2, pp. 111–131, 2005.

[13] Burger, D. and Austin, T., "The simplescalar toolset, version 2.0," Tech. Rep. 1342, University of Wisconsin, June 1997.

[14] Chen, T.-F. and Baer, J.-L., "Reducing memory latency via non-blocking and prefetching caches," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, vol. 27, (New York, NY), pp. 51–61, ACM Press, 1992.

[15] Cowan, C., Beattie, S., Johansen, J., and Wagle, P., "PointGuard^{TM}: Protecting pointers from buffer overflow vulnerabilities," in *Proc. of the 12th Usenix Security Symposium*, Aug 2003.

[16] Denning, D. E. and Denning, P. J., "Certification of programs for secure information flow," *Commun. ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[17] Diffie, W. and Hellman, M., "Privacy and Authentication: An Introduction to Cryptography," in *Proceedings of the IEEE, 67*, 1979.

[18] Dolev, D., Dwork, C., and Naor, M., "Non-malleable cryptography," in *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, (New York, NY, USA), pp. 542–552, ACM Press, 1991.

[19] Eberle, H., "A high-speed DES implementation for network applications," *Lecture Notes in Computer Science*, vol. 740, pp. 521–539, 1993.

[20] Federal Information Processing Standard Draft, "Advanced Encryption Standard (AES). National Institute of Standards and Technology," 2001.

[21] Galles, M. and Williams, E., "Performance optimizations, implementation, and verification of the sgi challenge multiprocessor," in *Technical report, Silicon Graphics Computer Systems,*, (Mountain View), 1994.

[22] Gassend, B., Suh, G. E., Clarke, D., van Dijk, M., and Devadas, S., "Caches and Hash Trees for Efficient Memory Integrity Verification," in *Proceedings of the Ninth Annual Symposium on High Performance Computer Architecture*, 2003.

[23] Gassend, B., Suh, G. E., Clarke, D., van Dijk, M., and Devadas, S., "Caches And Merkle Trees For Efficient Memory Integrity Verification," in *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, 2003.

[24] Gligor, V. D. and Donescu, P., "Fast encryption and authentication: XCBC encryption and XECB authentication modes," in *FSE '01: Revised Papers from the 8th International Workshop on Fast Software Encryption*, (London, UK), pp. 92–108, Springer-Verlag, 2002.

[25] Gries, M. and Romer., A., "Performance Evaluation of Recent DRAM Architectures for Embedded Systems," in *TIK Report Nr. 82, Computing Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich*, November 1999.

[26] Hodjat, A. and Verbauwhede, I., "Minimum area cost for a 30 to 70 Gbits/s AES processor," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI '04), pp. 498-502.*

[27] Hodjat, A. and Verbauwhede, I., "Speed-area trade-off for 10 to 100 gbits/s," in *37th Asilomar Conference on Signals, Systems, and Computer, Nov. 2003.*

[28] Huang, A., "Keeping secrets in hardware the microsoft xbox case study," *MIT AI Memo*, 2002.

[29] Intel Corporation, *Intel 82802 Firmware Hub: Random Nummber Generator*, December 1999.

[30] Interposer, "http://www.arium.com,". May/2003.

[31] Jutla, C. S., "Encryption modes with almost free message integrity," in *EUROCRYPT '01: Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques*, (London, UK), pp. 529–544, Springer-Verlag, 2001.

[32] Kocher, P., Lee, R., McGraw, G., and Raghunathan, A., "Security as a new dimension in embedded system design," in *DAC '04: Proceedings of the 41st annual conference on Design automation*, (New York, NY, USA), pp. 753–760, ACM Press, 2004. Moderator-Srivaths Ravi.

[33] Kocher, P., "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology - CRYPTO'96. Springer-Verlag Lecture Notes in Computer Science, vol. 1109*, pp. 104–113, 1996.

[34] Krawczyk, H., Bellare, M., and Canetti, R., "Hmac: Keyed-hashing for message authentication," 1997.

[35] Kumar, A., "Discovering passwords in the memory," *http://www.infosecwriters.com/text_resources/*. Nov/2004.

[36] Lee, H., Beckett, P., and Appelbe, B., "High-performance extendable instruction set computing," in *ACSAC '01: Proceedings of the 6th Australasian conference on Computer systems architecture*, (Washington, DC, USA), pp. 89–94, IEEE Computer Society, 2001.

[37] Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., and Horowitz, M., "Architectural Support For Copy and Tamper Resistant Software," in *Proceedings of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems*, 2000.

[38] Lie, D., Thekkath, C. A., and Horowitz, M., "Implementing an untrusted operating system on trusted hardware," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp. 178–192, ACM Press, October, 2003.

[39] Lipasti, M. H., Wilkerson, C. B., and Shen, J. P., "Value locality and load value prediction," in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pp. 138–147, ACM Press, 1996.

[40] Lipmaa, H., Rogaway, P., and Wagner, D., "Comments to NIST Concerning AES-modes of Operations: CTR-mode Encryption," in *In Symmetric Key Block Cipher Modes of Operation Workshop, Baltimore, Maryland, US*, 2000.

[41] L.Knudsen, "Block ciphers - analysis, design and applications," in *PhD thesis, Aarhus University, Denmark*, 1994.

[42] McLoone, M. and McCanny, J. V., "High performance single-chip fpga rijndael algorithm implementations," in *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 65–76, Springer-Verlag, 2001.

[43] Menezes, A. J., Vanstone, S. A., and Oorschot, P. C. V., *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, Inc., 1996.

[44] Merkle, R. C., "A digital signature based on a conventional encryption function," in *CRYPTO '87: A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, (London, UK), pp. 369–378, Springer-Verlag, 1988.

[45] Merkle, R. C. and Hellman, M. E., "On the security of multiple encryption," *Commun. ACM*, vol. 24, no. 7, pp. 465–467, 1981.

[46] Messerges, T. S., Dabbish, E. A., and Sloan, R. H., "Examining smart-card security under the threat of power analysis attacks," *IEEE Trans. Comput.*, vol. 51, no. 5, pp. 541–552, 2002.

[47] Myers, A. C., "Jflow: practical mostly-static information flow control," in *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (New York, NY, USA), pp. 228–241, ACM Press, 1999.

[48] National Bureau of Standards, "Data encryption standard," *FIPS-Pub.46, National Bureau of Standards, U.S. Department of Commerce, Washington D.C.*, January 1977.

[49] National Institude of Standards and Technology, "Recommendation for the triple data encryption algorithm (TDEA) block cipher," *SP-800-67, NIST.*

[50] National Institute of Science and Technology, "SHA256 hashing algorithm," *FIPS PUB 180-2,NIST.*

[51] Pai, V. S., Ranganathan, P., and Adve, S. V., "RSIM Reference Manual. Version 1.0," in *Department of Electrical and Computer Engineering, Rice University. Technical Report 9705*, 1997.

[52] Pritchard, M., "How to Hurt the Hackers: The Scoop on Internet Cheating and How You Can Combat It." http://www.gamasutra.com/features/20000724/pritchard01.htm. Nov/2004.

[53] Rogaway, P., Bellare, M., and Black, J., "OCB: A block-cipher mode of operation for efficient authenticated encryption," *ACM Trans. Inf. Syst. Secur.*, vol. 6, no. 3, pp. 365–403, 2003.

[54] Rogaway, P., Bellare, M., Black, J., and Krovetz, T., "OCB: a block-cipher mode of operation for efficient authenticated encryption," in *ACM Conference on Computer and Communications Security*, pp. 196–205, 2001.

[55] S.Flogel, F.Grassert, M.Grothmann, M.Haase, P.Nimsch, H.Ploog, D.Timmermann, and A.Wassatsch, "A design flow for 1.28 Gbit/s triple DES using dynamic logic and standard synthesis tools," *Synopsys User Group (SNUG) Europe, S. E3.2.*, pp. 1–8, 2001.

[56] Sherwood, T., Perelman, E., Hamerly, G., and Calder, B., "Automatically characterizing large scale program behavior," in *Proceedings of the 10th Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 45–57, Oct. 2002.

[57] Shi, W., Lee, H.-H. S., Ghosh, M., Lu, C., and Boldyreva, A., "High efficiency counter mode security architecture via prediction and precomputation.," in *ISCA*, pp. 14–24, 2005.

[58] Shi, W., Lee, H.-H. S., Lu, C., and Ghosh, M., "Towards the Issues in Architectural Support for Protection of Software Execution," Report GIT-CERCS-04-29, Geogia Institute of Technology, Atlanta, GA, Aug. 2004.

[59] Shi, W., Lee, H.-H. S., Lu, C., and Ghosh, M., "Towards the issues in architectural support for protection of software execution," *SIGARCH Comput. Archit. News*, vol. 33, no. 1, pp. 6–15, 2005.

[60] Suh, E. G., Clarke, D., Gassend, B., van Dijk, M., and Devadas, S., "Efficient Memory Integrity Verification and Encryption for Secure Processors," in *Proceedings 0f the 36th Annual International Symposium on Microarchitecture*, December, 2003.

[61] Suh, E. G., Clarke, D., van Dijk, M., Gassend, B., and S.Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing ," in *Proceedings of The Int'l Conference on Supercomputing*, 2003.

[62] Suh, G. E., Clarke, D., Gassend, B., van Dijk, M., and Devadas, S., "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," 2003.

[63] Suh, G. E., Clarke, D., Gassend, B., van Dijk, M., and Devadas, S., "Efficient Memory Integrity Verification and Encryption for Secure Processors," December 2003.

[64] Tandon, P., "High-Performance Advanced encryption Standard (AES) Security Co-Processor Design ," Master's thesis, School of Electrical and Computer Engineering, Georgia Institute of Technology, December 2003.

[65] T.Sander and Tschudin, C., "Protecting mobile agents against malicious hosts," *Mobile Agents and Security. LNCS*, Feb, 1998.

[66] van Eck, W., "Electromagnetic radiation from video display units: an eavesdropping risk?," *Comput. Secur.*, vol. 4, no. 4, pp. 269–286, 1985.

[67] Vanderwiel, S. P. and Lilja, D. J., "Data prefetch mechanisms," *ACM Comput. Surv.*, vol. 32, no. 2, pp. 174–199, 2000.

[68] Wang, Z., Burger, D., McKinley, K. S., Reinhardt, S. K., and Weems, C. C., "Guided region prefetching: A cooperative hardware/software approach," 2003.

[69] Wilton, S. and Jouppi, N., "Cacti: An enhanced cache Access and Cycle Time Model ," in *IEEE Journal of Solid-State Circuits,vol. 31, no. 5*, pp. 677–688, May 1996.

[70] Witteman, M., "Advances in smartcard security," in *Information Security Bulletin*, July, 2002 2002.

[71] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A., "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd International Symposium on Computer Architecture*, (Italy), pp. pages 24–36, 1995.

[72] Yang, J. and Gupta, R., "Frequent value locality and its applications," *Trans. on Embedded Computing Sys.*, vol. 1, no. 1, pp. 79–105, 2002.

[73] Yang, J., Zhang, Y., and Gao, L., "Fast Secure Processor for Inhibiting Software Piracty and Tampering," in *36th Annual IEEE/ACM International Symposium on Microarchitecture*, December, 2003.

[74] Zhang, X. and Gupta, R., "Hiding program slices for software security," in *Proceedings of the 2003 Internal Conference on Code Genration and Optimization*, pp. 325–336, 2003.

[75] Zhuang, X., Zhang, T., Pande, S., and Lee, H.-H. S., "HIDE: Hardware-support for Leakage-Immune Dynamic Execution," Report GIT-CERCS-03-21, Geogia Institute of Technology, Atlanta, GA, Nov. 2003.

[76] Zhuang, X., Zhang, T., Lee, H.-H. S., and Pande, S., "Hardware Assisted Control Flow Obfuscation for Embedded Processors," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 292–302, 2004.

[77] Zhuang, X., Zhang, T., and Pande, S., "HIDE: an Infrastructure for Efficiently Protecting Information Leakage on the Address Bus," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 72–84, 2004.