

## ABSTRACT

ROGERS, BRIAN MICHAEL. Low-Overhead Designs for Secure Uniprocessor and Multiprocessor Architectures. (Under the direction of Dr. Yan Solihin).

The security of computer systems is becoming a growing concern as the increasing ability and motivation of attackers continues to expand the types of attacks that exist to exploit a vast amount of digital information. In particular, new types of hardware-based attacks have become widespread in addition to the more traditional software attack methods. For example, a hardware attack may consist of utilizing a device to physically observe or tamper with sensitive information in a system. Such attacks are able to subvert software-only security measures, and as a result, computer researchers and designers have investigated hardware security solutions to address these concerns. In particular, secure processor architectures have been proposed as a way to prevent hardware-based attacks by cryptographically protecting the data and code executed in a system to ensure its privacy and integrity. Through such a level of protection, many important security issues may be addressed such as the prevention of the theft or tampering of critical data, prevention of reverse engineering of code, and protection from software piracy. In this dissertation, we propose and evaluate novel secure processor architectures for two broad types of system designs.

First, for single processor chip systems, we propose a secure processor architecture based on the novel techniques of Address Independent Seed Encryption (AISE) and Bonsai Merkle Trees (BMT) for implementing memory encryption and integrity verification respectively. AISE is based on counter-mode encryption, and like prior counter-mode encryption schemes, it effectively hides cryptographic latencies from the critical path of off-chip data fetches. However, at the same time it eliminates significant security and system-level drawbacks associated with prior schemes such as the lack of support for virtual memory mechanisms and shared memory inter-process communication. BMT is a novel Merkle Tree memory integrity verification approach which retains the strong security properties of standard Merkle Tree protection, but with a significant reduction in execution time overheads and memory storage overheads. Experimental results on the SPEC 2000 benchmarks show that BMTs reduce the overhead of Merkle Tree integrity verification in a secure processor from 12% to 2% on average.

Second, we propose the first secure processor architectures designed specifically for protecting distributed shared memory (DSM) multiprocessors. DSM systems require not only protecting data communicated between a processor and its memory, but also data communicated between processors across the interconnection network. We present a security requirements analysis for protecting the privacy and integrity of code and data in a DSM system, and then propose three table-based hardware schemes to protect processor-processor data communication in a DSM, while leveraging uniprocessor-based approaches

for protecting processor-memory data communication. After evaluating these schemes, we identify several performance and complexity drawbacks that are inherent in two-level schemes such as this which protect the two types of DSM communication with separate mechanisms. Thus, we propose an alternative, single-level DSM data protection scheme which leverages a single mechanism for protecting all off-chip DSM data transfers. Our experimental results show that this single-level scheme has an average overhead of only 1.6% across all SPLASH-2 benchmarks compared to a similar but unprotected DSM system.

Low-Overhead Designs for Secure Uniprocessor and Multiprocessor Architectures

by  
Brian Michael Rogers

A dissertation submitted to the Graduate Faculty of  
North Carolina State University  
in partial fulfillment of the  
requirements for the Degree of  
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2009

APPROVED BY:

---

Dr. Yan Solihin  
Chair of Advisory Committee

---

Dr. Gregory Byrd

---

Dr. Thomas Conte

---

Dr. Peng Ning

---

Dr. Milos Prvulovic

## DEDICATION

*To my parents, Gary and Carol Rogers, for their endless love, guidance,  
encouragement, and support.*

## BIOGRAPHY

Brian Rogers was born in Greensboro, North Carolina in 1981 as the first of two sons of Gary and Carol Rogers. He is married to Lindsay Rogers and has a son, Hudson. In 1999, he attended the University of North Carolina at Chapel Hill as an undergraduate student, where he graduated in 2003 with a B.S. degree with Honors in Computer Science as well as a minor in Business Administration from the Kenan-Flagler Business School. He then began his graduate studies in Computer Engineering at North Carolina State University. Brian completed his M.S. degree in 2005, and has been a Ph.D. student in the ARPERS (Architecture Research for Performance, Reliability, and Security) research group led by Professor Yan Solihin. He has worked as an intern at IBM Corporation in the Austin Research Lab in the summer of 2007 and in Research Triangle Park from spring 2008 through the fall of 2009.

His research interests span a wide array of topics in the field of computer architecture and systems. Specifically, he is interested in secure processor architectures, cache and memory systems, performance modeling, and multi-core and many-core architectures along with the bottlenecks to future core scaling. He has authored a number of refereed conference papers and journal articles in the field of computer architecture.

## ACKNOWLEDGMENTS

First, I want to express my heartfelt appreciation to my graduate research advisor, Dr. Yan Solihin. His dedication to guiding both me and my work over the past six years has been outstanding. I thank him for sharing his knowledge and insights with me, not only on research and technical issues, but on many other levels as well. He has provided a perfect balance between someone to push me to be the best that I can be, while at the same time offering countless hours of assistance. He has shaped my abilities as a computer architecture researcher, and I will always be thankful to him for investing so much of himself into my future success.

I am also extremely grateful to the members of my graduate advisory committee, Dr. Gregory Byrd, Dr. Thomas Conte, Dr. Peng Ning, and Dr. Milos Prvulovic. They have offered excellent recommendations and guidance for my research work and dissertation throughout my time as a graduate student. They have also been more than willing and accommodating for any request that I have had of them, no matter how much extra work it created, and I cannot thank them enough for that. They have served as an outstanding model of what it takes to be a successful researcher, and it has been a privilege to work with them and learn from them. Additionally, I would like to thank Milos for serving as essentially a co-advisor, working closely with me on several research projects. His

consistently clever ideas and knack for deconstructing difficult problems made it a pleasure to learn from him.

The current students and alumni of the ARPERS research group have been wonderful to work with as fellow graduate students. I thank all of them for their dear friendship: Mazen Kharbutli, Rithin Shetty, Seongbeom Kim, Fei Guo, Radha Venkatagiri, Xiaowei Jiang, Abhik Sarkar, Aziz Eker, Mohit Gambhir, Fang Liu, Siddhartha Chhabra, Devesh Tiwari, Anil Krishna, and Ahmad Samih. Each of them has played a vital role in my graduate work and helped me to get the most out of my time as a graduate student. I also am very thankful for the many long discussions with Siddhartha on security-related research issues, which have helped me to identify many problems and solutions in this area, contributing to my dissertation.

To my parents, I cannot begin to express my gratitude for the countless hours that they have both invested into molding my abilities and giving me the opportunities to pursue my goals and dreams. I thank them for the sacrifices that they have made in order to always provide me with what I have needed to be the best that I can be. I also thank them for always believing in me and for helping me to believe that I can achieve more than I ever thought possible.

I also thank my brother Scott for always being there as a friend and someone to talk to and laugh with. He has always been an inspiration to me, and a major reason and



motivation for me to push myself to hopefully be a good role model. He is a great friend, and is someone who has had a major impact on who I am as a person.

To my wife Lindsay and my son Hudson, it is difficult to describe how much they have meant to me throughout my time as a graduate student. Lindsay has been a steadying voice on so many occasions, and she has endured many times where I have been busy with long hours of work without complaint. Her love and devotion have given me the strength to make it through the difficult times, and this dissertation certainly would not have been possible without her. Hudson has never failed to provide a constant source of love, fun, and enjoyment in my life.

Finally, I thank God for the talents and abilities that I have been blessed with. It is through His strength and the people that he has placed into my life that I have been able to achieve my goals.

## TABLE OF CONTENTS

<b>LIST OF TABLES .....</b>	<b>ix</b>
<b>LIST OF FIGURES.....</b>	<b>x</b>
<b>1 Introduction .....</b>	<b>1</b>
1.1 Secure Uniprocessor Architectures . . . . .	4
1.2 Secure DSM Multiprocessor Architectures . . . . .	13
1.3 Organization of the Dissertation . . . . .	19
<b>2 OS- and Performance-Friendly Secure Processors.....</b>	<b>20</b>
2.1 Background . . . . .	21
2.2 Attack Model and Assumptions . . . . .	24
2.3 Memory Encryption . . . . .	25
2.3.1 Overview of Counter-Mode Encryption . . . . .	25
2.3.2 Problems with Current Counter-Mode Memory Encryption . . . . .	28
2.3.3 Address Independent Seed Encryption . . . . .	32
2.3.4 Dealing with Swap Memory and Page Swapping . . . . .	35
2.3.5 Dealing with Page Sharing . . . . .	37
2.3.6 Virtualization support for Secure Processors . . . . .	37
2.3.7 Virtualizing the LPID . . . . .	40
2.3.8 Advantages of AISE . . . . .	43
2.4 Memory Integrity Verification . . . . .	46
2.4.1 Extended Merkle Tree Protection . . . . .	47
2.4.2 Bonsai Merkle Trees . . . . .	51
2.5 Experimental Setup . . . . .	56
2.5.1 Machine Models . . . . .	56
2.5.2 Benchmarks . . . . .	57
2.6 Evaluation . . . . .	59
2.6.1 AISE: Qualitative Evaluation . . . . .	59
2.6.2 Single-Core Processor Evaluation Results . . . . .	60
2.6.3 Multi-Core Processor Evaluation Results . . . . .	66
2.6.4 Sensitivity Studies . . . . .	70
2.6.5 Storage Overheads in Main Memory . . . . .	73
2.7 Conclusions . . . . .	76

<b>3</b>	<b>Data Protection for Distributed Shared Memory Multiprocessors.....</b>	<b>77</b>
3.1	Background . . . . .	79
3.2	Attack Model and Assumptions . . . . .	82
3.2.1	Architecture Assumptions . . . . .	82
3.2.2	Security Assumptions and Attack Model . . . . .	83
3.3	Security Analysis for DSM Protection . . . . .	87
3.4	Table-Based Processor-Processor DSM Data Protection . . . . .	90
3.4.1	Private Counter Stream Tables( <i>Private</i> ) . . . . .	95
3.4.2	Shared Counter Stream Tables( <i>Shared</i> ) . . . . .	97
3.4.3	Cached Counter Stream Tables( <i>Cached</i> ) . . . . .	99
3.4.4	Detecting Replay Attacks with Table-based Protection . . . . .	101
3.4.5	Implementation Issues . . . . .	103
3.5	DSM Experimental Setup . . . . .	104
3.6	Evaluation of Table-Based DSM Protection . . . . .	107
3.6.1	Table-Based DSM Data Protection Overhead . . . . .	108
3.6.2	Scalability of the <i>Cached</i> Scheme . . . . .	111
3.6.3	Sensitivity Analysis . . . . .	113
3.7	Drawbacks of Table-Based DSM Protection . . . . .	117
3.8	Single-Level Data Protection for DSM Systems . . . . .	122
3.8.1	Single Level Memory Encryption . . . . .	122
3.8.2	Single-Level Memory Authentication . . . . .	129
3.8.3	Security Analysis . . . . .	140
3.9	Evaluation of Single-Level DSM Protection . . . . .	142
3.9.1	Experimental Setup . . . . .	142
3.9.2	Overhead of Single-Level DSM Data Protection . . . . .	143
3.9.3	Sensitivity Analysis . . . . .	149
3.10	Conclusions . . . . .	153
<b>4</b>	<b>Conclusions.....</b>	<b>154</b>
	<b>Bibliography.....</b>	<b>159</b>

## LIST OF TABLES

Table 2.1 Qualitative comparison of AISE with other counter-mode encryption approaches .....	60
Table 2.2 MAC & Counter Memory Overheads .....	75
Table 3.1 Architectural Parameters .....	105
Table 3.2 Applications used in our evaluation. The L2 global miss rate is the number of L2 misses divided by the number of L1 cache accesses. The local L2 miss rate is the number of L2 misses divided by L2 accesses .....	106
Table 3.3 Fraction of requests served by the home node's local memory in SPLASH-2.	143

## LIST OF FIGURES

Figure 1.1 Abstract view of the secure processor architecture model. ....	5
Figure 2.1 Counter-mode based memory encryption. ....	26
Figure 2.2 Virtual Memory management allows virtual pages of different processes to map to a common physical page for sharing purpose (1), the same virtual pages in different processes to map to different physical pages (2), and some virtual pages to reside in the swap memory in the disk (3).....	29
Figure 2.3 Organization of logical page identifiers.....	34
Figure 2.4 VMM Models.....	38
Figure 2.5 Portion of Page Fault Handling Routine of an AISE-compliant OS.....	40
Figure 2.6 Merkle Tree organization for extending protection to the swap memory in disk.....	49
Figure 2.7 Reduction in size of Bonsai Merkle Trees compared to standard Merkle Trees.....	54
Figure 2.8 Performance overhead comparison of AISE with BMT vs. the Global counter scheme with a traditional Merkle Tree .....	61
Figure 2.9 Performance overhead comparison of AISE versus the global counter scheme	62
Figure 2.10 Performance overhead comparison of AISE with our Bonsai Merkle Tree vs. AISE with the Standard Merkle Tree .....	64
Figure 2.11 L2 cache pollution .....	65
Figure 2.12 L2 cache miss rate and bus utilization of an unprotected system, standard Merkle Tree, and our BMT scheme.....	67
Figure 2.13 Performance degradation comparison of AISE with our Bonsai Merkle Tree vs. 64-bit global counter scheme with the Standard Merkle Tree.....	68

Figure 2.14 Performance degradation comparison of AISE with our Bonsai Merkle Tree vs. AISE with the Standard Merkle Tree .....	69
Figure 2.15 L2 cache pollution .....	69
Figure 2.16 L2 cache miss rate and bus utilization of an unprotected system, standard Merkle Tree, and our BMT scheme .....	71
Figure 2.17 Performance overhead comparison across MAC size .....	72
Figure 2.18 Sensitivity to cache size .....	74
Figure 3.1 DSM processor-processor communication .....	83
Figure 3.2 Attack model, secure boundary, and location of our encryption and integrity verification HW. ....	85
Figure 3.3 Proc-proc communication latencies using various encryption and authenti- cation mechanisms. ....	91
Figure 3.4 Our mechanism for processor-processor secure data transfer in DSM sys- tems. ....	93
Figure 3.5 Execution time overheads for processor-memory data protection only, versus schemes with additional processor-processor data protection on 16 processors. ...	109
Figure 3.6 Send and receive table performance for our schemes on 16 processors. ....	110
Figure 3.7 Execution time overheads for our <i>Cached8</i> scheme across 16, 32, and 64 processors. ....	112
Figure 3.8 Pad miss rate at the receive tables for our <i>Cached8</i> scheme across 16, 32, and 64 processors .....	113
Figure 3.9 Execution time overheads for 1x, 2x, and 4x the base AES latency and occupancy. ....	114
Figure 3.10 Execution time overheads across 16, 32, and 64 processor DSM systems ...	115
Figure 3.11 Execution time overheads for 256KB, 512KB, and 1MB L2 cache sizes. ...	116

Figure 3.12 Critical Path of a remote data request for a Two-Level encryption scheme (a) and Single-Level encryption scheme (b). . . . .	118
Figure 3.13 Coherence protocol modifications for hiding cryptographic latencies of remote data request in our single-level counter mode memory encryption when data is supplied by the home node's local memory (a), home node's cache (b), remote owner (c), and remote owner with an owned block pad buffer (d). . . . .	125
Figure 3.14 Architecture modifications to a node. . . . .	129
Figure 3.15 High-Level illustration of an early (a) versus our delayed (b) authentication protocol. . . . .	131
Figure 3.16 Steps for authenticating data sent to a requesting processor across the interconnect. . . . .	136
Figure 3.17 The execution time overhead of single-level DSM data protection scheme versus two-level protection using the CACHED scheme with 4-entry communication counter table. . . . .	144
Figure 3.18 Fraction of off-chip data requests that experience pad hit (full latency-hiding), half-miss (partial latency-hiding), and miss (no latency-hiding). . . . .	146
Figure 3.19 Percentage of requests for which the counter value is correctly predicted. .	147
Figure 3.20 Local counter cache hit rate and portion of counter messages skipped due to correct prediction. . . . .	148
Figure 3.21 The normalized AES bandwidth usage of single-level vs. two-level DSM data protection. . . . .	149
Figure 3.22 Execution time overhead of our single-level DSM data protection scheme across system size. . . . .	150
Figure 3.23 Execution time overhead of our single-level DSM data protection scheme across L2 cache size. . . . .	151

# Chapter 1

## Introduction

With the tremendous amount of digital information stored on today's computer systems, and with the increasing motivation and ability of malicious attackers to target this wealth of information, computer security has become an increasingly important issue. An important research effort towards such computer security issues focuses on protecting the *privacy* and *integrity* of computations to prevent attackers from stealing or modifying critical information. This type of protection is important for enabling many important features of secure computing such as prevention of the theft or tampering of sensitive data, protection from reverse engineering and software piracy, enabling trusted distributed computing, and enforcement of Digital Rights Management.

An important, emerging security threat exploits the fact that most current computer systems communicate code and data in its plain form along exposed wires between



the processor chip and other chips such as the main memory. Also, the data is stored in its plain form in the main memory. By dumping the memory content and scanning it, attackers may gain a lot of valuable sensitive information such as passwords [25]. Another serious and feasible class of attacks that has emerged is *physical* or *hardware attacks* which involve physically observing or interfering with the operation of the system [18, 19, 24]. For example, an attacker could utilize a bus analyzer that snoops or alters data communicated between the processor chip and other chips. Although physical attacks may be more difficult to perform than software-based attacks, they are very powerful as they can bypass any software security protection employed in the system. The reason is that program variables, such as secret keys themselves, of the protection software are stored in main memory and may be subjected to hardware attacks. The proliferation of mod-chips [35] that bypass the Digital Rights Management protection in game systems has demonstrated that given sufficient financial payoffs, physical attacks are very realistic threats.

In order to more clearly illustrate the environment where hardware-based attacks may occur and how they can operate, we describe two example hardware attack scenarios. As a first example, we note that certain types of mod-chips can be attached to exposed storage or communication channels on the motherboard of game system consoles in order to perform various attacks such as the execution of unauthorized software on the system. This type of attack may proceed as follows. The original basic input/output system (BIOS) of the console contains code to verify that discs inserted into the console contain particular

digital signatures before the software on the disc is executed. A replica of this BIOS, minus this security-checking code, can be placed onto the mod-chip, which is then attached to the system. The mod-chip supplies this modified BIOS to the processor instead of the original BIOS so that unauthorized software can be executed on the system, as the security checks are not performed. As a second example, we note that large-scale server systems often have a significant amount of cable clutter in the back of the server racks due to the many interconnection links. In this type of environment, an attacker such as a malicious employee could attach a communication logger along one of the links to steal valuable information which is communicated across these links of the system.

To address these types of security concerns, researchers have proposed various types of secure processor architectures, which utilize hardware security mechanisms to protect the privacy and integrity of computations [14, 15, 28, 29, 30, 38, 39, 40, 41, 45, 46, 50, 52, 53]. In the secure processor model, it is assumed that off-chip communication and structures are vulnerable to attack and that the processor chip boundary provides a natural security boundary. This is because snooping or manipulating on-chip transistors/wires is much more difficult than for off-chip components/interconnections. Under these assumptions, secure processors seek to provide private and tamper-resistant execution environments [46] through *memory encryption* [15, 29, 30, 39, 40, 41, 45, 46, 50, 52, 53] and *memory integrity verification* [14, 30, 38, 39, 41, 45, 46, 50, 53]. The chip industry also

recognizes the need for secure processors as evident, for example, in the recent efforts by IBM in the SecureBlue project [21] and by Dallas Semiconductor [33].

To provide a better feel for this environment, Figure 1.1 shows an abstract representation of the secure processor model. All on-chip data (e.g. in caches or registers) is secure, but all data in exposed off-chip structures such as the memory bus and main memory is vulnerable to attack. Thus a secure processor utilizes on-chip cryptographic support to protect off-chip data through memory encryption and memory integrity verification (or authentication). Memory encryption protects computation privacy from *passive attacks*, where an adversary attempts to silently observe critical information, by encrypting code and data as it moves on and off the processor chip. Memory integrity verification protects computation integrity from *active attacks*, where an adversary attempts to modify values in off-chip storage or communication channels, by computing and verifying Message Authentication Codes (MACs) as code and data moves on and off the processor chip.

## 1.1 Secure Uniprocessor Architectures

Research on secure processor architectures includes a variety of proposed schemes for protecting the confidentiality and integrity of code and data on uniprocessor (or single processor chip) computer systems [14, 15, 29, 30, 38, 40, 41, 45, 46, 50, 52]. Unfortunately, these current memory encryption and integrity verification designs are not yet suitable for use in general purpose computing systems. In particular, we show that current secure

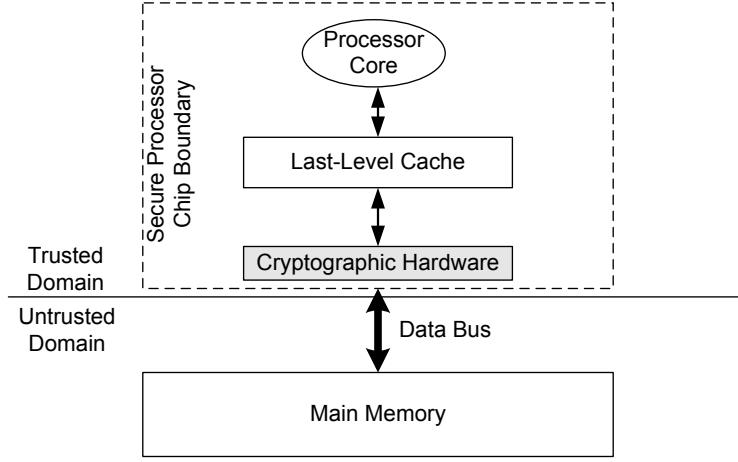


Figure 1.1: Abstract view of the secure processor architecture model.

processor designs are incompatible with important features such as virtual memory and Inter-Process Communication, in addition to having significant performance and storage overheads. The challenges are detailed as follows:

**Memory Encryption.** Recently proposed memory encryption schemes for secure processors have utilized *counter-mode encryption* due to its ability to hide cryptographic delays on the critical path of memory fetches. This is achieved by applying a block cipher to a *seed* to generate a cryptographic *pad*, which is then bit-wise XORed with the memory block to encrypt or decrypt it. A seed is selected to be independent from the data block value so that pad generation can be started while the data block is being fetched.

In counter-mode encryption, the choice of seed value is critical for both security

and performance. The security of counter-mode *requires* the uniqueness of each pad value, which implies that each seed must be unique. In prior studies [39, 40, 41, 46, 50, 52, 53], the memory block address is used as one of the seed’s components to ensure that pads are unique across different blocks in memory (*spatial uniqueness*). To ensure that pads are unique across different values of a particular block over time (*temporal uniqueness*), a counter value which is incremented on each write back of the block to off-chip memory is also used as a seed component. From the performance point of view, if most cache misses find the counters of the missed blocks available on-chip, either because they are cached or predicted, then seeds can be composed at the time of the cache miss, and pad generation can occur in parallel with fetching the blocks from memory.

However, using the address (virtual or physical) as a seed component causes a significant system-level dilemma in general purpose computing systems that must support virtual memory and Inter-Process Communication (IPC). A virtual memory mechanism typically involves managing pages to provide process isolation and sharing between processes. It often manages the main memory by extending the *physical memory* to *swap memory* located on the disk.

Using the physical address as a seed component creates re-encryption work on page swapping. When a page is swapped out to disk and then back into memory, it will likely reside at a new physical address. This requires the blocks of the page to be decrypted using their previous physical addresses and re-encrypted with their new physical

addresses. In addition, encrypted pages in memory cannot be simply swapped out to disk as this creates potential pad reuse between the swapped out page and the new page at that physical address in memory. This leaves an open problem as to how to protect pages on disk. We could entrust the OS to encrypt and decrypt swapped pages in software if the OS is assumed to be authentic, trusted, and executing on the secure processor. However this is likely not the most desirable solution because it makes the secure processor's hardware-based security mechanisms contingent on a secure and uncompromised OS. Alternatively, we could rely on hardware to re-encrypt swapped pages, but this solution has its own set of problems. First, this requires supporting two encryption methods in hardware. Second, there is the issue of who can request the page re-encryptions, and how these requests are made, which requires an extra authentication mechanism.

Using virtual address as a seed component can lead to vulnerable pad reuse because different processes use the same virtual addresses. While we can prevent this by adding process ID to the seed [50], this solution creates a new set of system-level problems. First, this renders process IDs non-reusable, and current OSes have a limited range of process IDs. Second, shared memory based inter-process communication (IPC) mechanisms are infeasible to use (e.g. mmap [31]). The reason is that different processes access a shared page in memory using different combinations of virtual address and process ID. This results in different encryptions and decryptions of the shared data. Third, other OS features that also utilize page sharing cannot be supported. For example, process forking

cannot utilize the copy-on-write optimization because the page in the parent and child are encrypted differently. This also holds true for shared libraries. This lack of IPC support is especially problematic in the era of Chip Multiprocessors (CMPs). Finally, storage is required for virtual addresses at the lowest level on-chip cache, which is typically physically indexed and tagged.

The root cause of the problems when using address in seed composition is that address is used as a *fundamental component of memory management*. Using address also as a basis for *security* intermingles security and memory management in undesirable ways.

**Memory Integrity Verification.** Recently proposed memory integrity verification schemes for secure processors have leveraged a variety of techniques [14, 21, 30, 38, 41, 45, 46, 50]. However, the security of Merkle Tree-based schemes [14] has been shown to be stronger than other schemes because every block read from memory is verified individually (as opposed to [46]), and *data replay* attacks can be detected in addition to spoofing and splicing attacks, which are detectable by simply associating a single Message Authentication Code (MAC) per data block [30]. In Merkle Tree memory integrity verification, a tree of MAC values is built over the memory. The root of this tree never goes off-chip, as a special on-chip register is used to hold its current value. When a memory block is fetched, its integrity can be checked by verifying its chain of MAC values up to the root MAC. When a cache block is written back to memory, the corresponding MAC values of the tree

are updated. Since the on-chip root MAC contains information about every block in the physical memory, an attacker cannot modify or replay any value in memory.

Despite its strong security, Merkle Tree integrity verification suffers from two significant issues. First, since a Merkle Tree built over the main memory computes MACs on memory events (cache misses and writebacks) generated by the processor, it covers the physical memory, but not swap memory which resides on disk. Hence, although Merkle Tree schemes can prevent attacks against values read from memory, there is no protection for data brought into memory from the disk. This is a significant security vulnerability since by tampering with swap memory on disk, attackers can indirectly tamper with main memory. One option would be to entrust the OS to protect pages swapped to and from the disk, but as with memory encryption it requires the assumption of a trusted OS. Another option, as discussed in [45], is to associate one Merkle Tree and on-chip secure root per process, and compute each Merkle Tree over the virtual address space of the process. However, managing multiple Merkle Trees results in extra on-chip storage and complexity.

Another significant problem is the storage overhead of internal Merkle Tree nodes in both the on-chip cache and main memory. To avoid repeated computation of internal Merkle Tree nodes as blocks are read from memory, a popular optimization lets recently accessed and verified internal Merkle Tree nodes be cached on-chip. Using this optimization, the verification of a memory block only needs to proceed up the tree until the first cached node is found. Thus, it is not necessary to fetch and verify all Merkle Tree nodes up to the



root on each memory access, significantly improving memory bandwidth consumption and verification performance. However, our results show that Merkle Tree nodes can occupy as much as 50% of the total L2 cache space, which causes the application to suffer from a large number of cache capacity misses.

**The Case for Secure CMPs.** Increasing chip densities and transistor counts have provided microarchitects with rich opportunities to improve single core performance through various microarchitectural innovations like exploiting instruction level parallelism via dynamic scheduling and issuing multiple instructions. However, the performance benefits achievable using a single processor will hit a ceiling due to fundamental circuit limitations and limited amounts of instruction level parallelism [37]. This motivates the need to better utilize the increasing transistor counts. Chip multiprocessors (CMPs) are the current design of choice to exploit the increasing transistor counts by placing multiple simple cores on a single die [37]. CMPs are particularly attractive for high-performance servers where commercial workloads having large amounts of thread level parallelism like web and database applications have become most popular [6].

Existing CMP designs [6, 26, 43] are typically organized with private L1 caches per core and some combination of shared and private lower-level caches, such as L2 and possibly L3 caches. All cores on the chip typically share a single, common memory bus and off-chip main memory. The memory encryption and integrity verification mechanisms proposed in this work, as well as those proposed in prior secure processor studies, can

be applied to such CMP architectures in the same manner as in uniprocessor systems. Since these mechanisms exist at the edge of the processor chip boundary, at the interface to the memory bus, they are independent of the on-chip cache hierarchy. This implies that the threat model and protection requirements are the same, namely to protect data communicated and stored between a processor’s last-level on-chip cache and main memory. As a result, we include the discussion and evaluation of our schemes on CMPs along with single-chip, uniprocessor systems. CMPs present secure processor architects with new challenges primarily due to the fact that the memory hierarchy design becomes even more critical. The limited cache space and off-chip bus bandwidth can prove to be an even more precious resource as multiple threads running simultaneously on multiple cores issue requests to the memory over the same bus. Despite the growing importance and application of CMPs, prior studies on secure processors have not evaluated their mechanisms on CMP architectures.

**Contributions.** In the uniprocessor-focused chapter of this dissertation, we investigate system-level issues in secure processors, and propose mechanisms to address these issues that are simple yet effective. The first contribution is *Address Independent Seed Encryption* (AISE), which decouples security and memory management by composing seeds using *logical identifiers* instead of virtual or physical addresses. The logical identifier of a block is the concatenation of a *logical page identifier* with the offset of the block within its page. Each virtual memory page has a logical page identifier which is distinct across the

entire memory and over the lifetime of the system. It is assigned to the page the first time the page is allocated or when it is loaded from disk. AISE provides better security since it provides complete seed/pad uniqueness for *every* block in the system (both in the physical and swap memory). At the same time, it also easily supports virtual memory and shared-memory based IPC mechanisms, and simplifies page swap mechanisms by not requiring decryption and re-encryption on a page swap. AISE also lends itself to easy support for *virtualization*. We show that using our mechanisms, virtualization can be easily supported without requiring modifications to the *guest* OSes running in the virtualized environment.

The second contribution is a novel and efficient extension to Merkle Tree based memory integrity verification that allows extending the Merkle Tree to protect off-chip data (i.e. both physical and swap memory) with a single Merkle Tree and secure root MAC over the physical memory. Essentially, our approach allows pages in the swap memory to be incorporated into the Merkle Tree so that they can be verified when they are reloaded into memory.

Next, we propose Bonsai Merkle Trees (BMTs), a novel organization of the Merkle Tree that naturally leverages counter-mode encryption to reduce its memory storage and performance overheads. We observe that if each data block has a MAC value computed over the data and its counter, a replay attack must attempt to replay an old data, MAC, and counter value together. A Merkle Tree built over the memory is able to detect any changes to the data MAC, which prevents any undetected changes to counter values or data.

Our key insight is that: (1) there are many more MACs of data than MACs of counters, since counters are much smaller than data blocks, (2) a Merkle Tree that protects counters prevents any undetected counter modification, (3) if counter modification is thus prevented, the Merkle Tree does not need to be built over data MACs, and (4) the Merkle Tree over counters is much smaller and significantly shallower than the one over data. As a result, we can build such a Bonsai Merkle Tree over the counters which prevents data replay attacks using a much smaller tree for less memory storage overhead, fewer MACs to cache, and a better worst-case scenario if we miss on all levels of the tree up to the root. As our results show, BMT memory integrity verification reduces the performance overhead significantly, from 12.1% to 1.8% across all SPEC 2000 benchmarks [44], along with reducing the storage overhead in memory from 33.5% to 21.5%.

Finally, an evaluation of the proposed schemes for a CMP architecture is provided. Results show the proposed mechanisms maintain a distinct advantage for secure CMPs, reducing the performance overhead from 15.1% to 4.1%.

## 1.2 Secure DSM Multiprocessor Architectures

Another important class of systems that will require tamper-resistant designs for data secrecy and integrity are *Distributed Shared Memory* (DSM) Multiprocessors. This is especially evident when one considers the settings in which many DSM systems will likely be used in the future. For example, a growing use of large-scale DSM systems is

in the context of *utility* or *on-demand computing* where a company owning large systems will “lease” computational and storage resources of the system to customers who want to outsource their IT operations or who need more computational resources to run their applications. For example, DSM systems such as the HP Superdome are already being used to offer on-demand computing services [36] to a variety of users. Because large DSMs are powerful but expensive, customers often run critical applications which access and store secret corporate data (e.g. financial data, product information, client records, etc.) on them. As the utility computing model grows in popularity, a more diverse array of companies will adopt this model, and DSMs will host a wider range of applications using many types of sensitive data. In addition, DSM will be the predominant architecture of these systems since Symmetric Multiprocessors (SMP) cannot scale to large systems easily. Since these DSM systems are in a physically remote location relative to the customers, customers are often very concerned about the privacy and integrity of their computations, especially against hardware attacks that may be very hard to detect or trace. IndustryWeek pointed out that data privacy is one of the major concerns that prevents fast adoption of the on-demand computing model [7]. This concern may prompt customers to require on-demand computing providers to utilize hardware-based privacy and tamper-resistance mechanisms in their DSMs.

We note that it is unlikely for the utility computing provider itself to be malicious, as this would create a poor business model. Instead, a large-scale DSM system owned by

a corporation will likely be protected with relatively tight physical security that restricts system access to select employees. However, lessons from history have taught us that it is unlikely that this single layer of security would be fail-proof. For example, despite the relatively good physical security protection and limited access for Automatic Teller Machines, there have been repeated cases of ATM fraud by some *supposedly trusted* employees [4]. In one case, an employee inserted a PC into an ATM to monitor and steal customer accounts and PINs. Another report by the Global ATM Security Alliance (GASA) found that more than 80% of computer-based bank-related frauds involve employees [27].

DSMs used for on-demand computing are in a similar situation in that the main ingredients that are conducive for physical tampering are there. First, DSMs (like ATMs) store highly valuable information belonging to many customers. For DSMs, this information may include financial data, product information, and client records. Second, the financial motivation to perform an attack can be large because stolen information is valuable to other corporations (corporate espionage) or criminals (identity theft). Finally, there exist some forms of attacks that hardly leave any traces. For example, physically inserting a snooping device in a DSM would be quite easy due to the exposed interconnection at the back of server racks. USB drive-sized devices with multi-GB storage can likely be attached and removed in a matter of seconds without shutting down the system, and without leaving visible traces. Thus, many corporations will likely wish to add another, difficult

to break, layer of protection for the security of their data in the form of tamper-resistant DSM systems.

While architectural support for data privacy and integrity has been studied extensively by researchers for uniprocessor systems [14, 15, 29, 30, 38, 40, 41, 45, 46, 50, 52], and more recently for Symmetric Multi Processor (SMP) systems [39, 53], such support specifically for DSM systems has unfortunately not yet been studied in detail. Uniprocessor secure processor architecture schemes provide data encryption and integrity verification only for *processor-memory* communication and the main memory but do not address data protection for *processor-processor* communication present in multiprocessor systems which are spread across multiple processor chips. Proposals for secure SMP systems include encryption and integrity checking mechanisms for processor-processor communication, but these mechanisms rely on the assumption that each processor can observe every coherence transaction in the system, which is satisfied due to the single shared bus in the system. This assumption cannot be made in DSMs, where communication is point-to-point rather than through broadcast mechanisms. As a result, new techniques for secure DSM architectures are needed.

**Contributions.** The first contribution of the secure DSM architecture chapter is an analysis of the security requirements for protecting DSM systems against hardware attacks. The findings of this analysis are that passive/eavesdropping attacks are more likely to be attempted because they are non-intrusive and leave very few (if any) traces.

Active attacks that modify coherence messages and alter the behavior of the DSMs are less likely to be attempted, especially if the system is augmented with the ability to detect them. Therefore, we seek to prevent passive attacks from succeeding, while we detect and report active attacks. To achieve this, we find that different coherence protocol messages and different parts of a message need to be protected differently: with both encryption and integrity verification, with integrity verification only, or with no protection.

One possible approach to create a secure DSM system is to provide *direct* encryption and integrity verification, in which direct encryption (or decryption) and Message Authentication Code (MAC) generation (or verification) are performed for each coherence message sent (or received). However, this approach would directly add cryptographic latencies to the already problematic communication latencies in DSM systems. Therefore, our second contribution is a new combined counter-mode encryption/integrity verification scheme that attempts to hide the cryptographic latencies due to protecting processor-processor communication. Our scheme relies on two essential techniques. First, we observe that if communicating processors share the same communication counter, they can pre-generate one-time pads used for message encryption and decryption. Hence, to hide encryption/decryption latencies, we use per-processor-pair communication counters that are incremented asynchronously after each message send/receive. Three separate hardware table organizations are proposed for storing such communication counters. Secondly, we also maintain data integrity through the use of Galois/Counter Mode (GCM), a MAC-based



integrity verification technique using a combined authenticated-encryption mode [34, 51] to reduce the MAC computation latency to only a few cycles after message ciphertext is available. We also show how our table-based mechanisms can be seamlessly combined with secure uniprocessor mechanisms for processor-memory data protection to provide system-wide data protection for DSMs. Our results show a moderate performance overhead for our techniques: ranging from 3.5% to 5.5% on average across all SPLASH-2 [49] benchmarks when compared to a DSM system with no support for data protection.

The next contribution of this chapter is an alternative approach for protecting the confidentiality and integrity of data in a DSM system, which requires only very small on-chip storage overheads and removes some of the inefficiencies of the table-based two-level protection schemes. As opposed to requiring separate mechanisms for protecting processor-memory and processor-processor communication, this approach uses a single mechanism for both. Thus we refer to this scheme as *single-level* memory encryption and authentication. The efficiency of the single-level approach comes from a significant reduction of cryptographic operations on remote requests. A single mechanism is used to encrypt and sign data when it is sent off-chip by a processor (either to memory or a remote processor), and to decrypt and authenticate data when it is brought on-chip for use by a requesting processor (either from memory or a remote processor). This approach not only reduces the amount of cryptographic work involved, but also reduces the possibility of contention-related latencies at the cryptographic engines. In addition, only one latency-hiding mechanism is involved,

which has a higher chance of succeeding compared to the simultaneous successes of multiple latency-hiding mechanisms of a two-level approach. Finally, we avoid transitions between two security mechanisms, allowing authentication of data on a remote request to be moved *off the critical path*.

Overall, we find that the single-level protection scheme can provide secure data encryption and authentication in a DSM system with very low overheads. Simulation results of the SPLASH-2 [49] benchmarks show that the average overhead of our mechanisms on a 16-processor DSM system is less than 1.6% with a maximum of 7%, relative to an identical but unprotected system.

### 1.3 Organization of the Dissertation

This dissertation is organized as follows. Chapter 2 describes our Address Independent Seed Encryption and Bonsai Merkle Tree schemes which alleviate system-level and performance problems of prior memory encryption and integrity verification schemes for secure uniprocessor systems. Chapter 3 takes a first look at the problem of protecting data confidentiality and integrity in DSM multiprocessor system, and proposes techniques for efficient encryption and integrity verification of data communicated between a processor and its local memory as well as data communicated between processors across the interconnect. Finally, Chapter 4 provides some concluding statements and thoughts on these studies of secure processor architectures.

## Chapter 2

# OS- and Performance-Friendly Secure Processors

This chapter is organized as follows. Section 2.1 discusses background and related work in the area of secure processor designs for uniprocessor systems. Section 2.2 outlines the attack model assumed in this study. Section 2.3 describes the Address Independent Seed Encryption scheme in detail. Section 2.4 describes an approach to extend Merkle Tree protection to the swap space on disk as well as the Bonsai Merkle Tree scheme to reduce performance and storage overheads. Section 2.5 outlines our experimental setup, and Section 2.6 presents our evaluation results and insights. Finally, Section 2.7 concludes this chapter of the dissertation.

## 2.1 Background

Research on secure uniprocessor architectures [14, 15, 29, 30, 38, 40, 41, 45, 46, 50, 52] consists of memory encryption for ensuring data *privacy* and memory integrity verification for ensuring data *integrity*. Early memory encryption schemes utilized *direct* encryption modes [15, 29, 30, 45], in which a block cipher such as AES [11] is applied directly on a memory block to generate the plaintext or ciphertext when the block is read from or written to memory. Since, on a cache miss for a block, the block must first be fetched on chip before it can be decrypted, the long latency of decryption is added directly to the memory fetch latency, resulting in execution time overheads of up to 35% (almost 17% on average) [52]. In addition, there is a security concern for using direct encryption because different blocks having the same data value would result in the same encrypted value (ciphertext). This property implies that the statistical distribution of plaintext values matches the statistical distribution of ciphertext values, and may be exploited by attackers.

As a result of these concerns, recent studies have leveraged *counter-mode* encryption techniques [40, 41, 46, 50, 52]. Counter-mode encryption overlaps decryption and memory fetch by decoupling them. This decoupling is achieved by applying a block cipher to a *seed* value to generate a cryptographic *pad*. The actual encryption or decryption is performed through an XOR of the plaintext or ciphertext with this pad. The security of counter-mode depends on the guarantee that each pad value (and thus each seed) is only used once. Consequently, a block’s seed is typically constructed by concatenating the

address of the block with a per-block counter value which is incremented each time the block is encrypted [40, 46, 50, 52]. If the seed components are available on chip at cache miss time, decryption can be started while the block is fetched from memory. Per-block counters can be cached on chip [46, 50, 52] or predicted [40].

Several different approaches have previously been studied for memory integrity verification in secure processors. These approaches include a MAC-based scheme where a MAC is computed and stored with each memory block when the processor writes to memory, and the MAC is verified when the processor reads from memory [30]. In [46], a Log Hash scheme was proposed where the overhead of memory integrity verification is reduced by checking the integrity of a series of values read from memory at periodic intervals during a program’s execution using incremental, multiset hash functions. Merkle Tree based schemes have also been proposed where a tree of MAC values is stored over the physical memory [14]. The root of the tree, which stores information about every block in memory, is kept in a secure register on-chip. Merkle Tree integrity verification is often preferable over other schemes because of its security strength. In addition to spoofing and splicing attacks, *replay* attacks can also be prevented. These attacks will be described in more detail in Section 2.4. We note that the Log Hash scheme can also prevent replay attacks, but as shown in [41], the long time intervals between integrity checks can leave the system open to attack.

The proposed scheme in this dissertation differs from prior studies in the following

ways. Our memory encryption avoids intermingling security with memory management by using *logical identifiers* (rather than address) as seed components. Our memory integrity verification scheme extends Merkle Tree protection to the disk in a novel way, and our BMT scheme significantly reduces the Merkle Tree size. The implications of this design will be discussed in detail in the following sections.

## 2.2 Attack Model and Assumptions

As in prior studies on hardware-based memory encryption and integrity verification, our attack model identifies two regions of a system. The secure region consists of the processor chip itself. Any code or data on-chip (e.g. in registers or caches) is considered safe and cannot be observed or manipulated by attackers. The non-secure region includes *all* off-chip resources, primarily including the memory bus, physical memory, and the swap memory in the disk. We do not constrain attackers' ability to attack code or data in these resources, so they can observe any values in the physical and swap memory and on all off-chip interconnects. Attackers can also act as a man-in-the-middle to modify values in the physical and swap memory and on all off-chip interconnects.

Note that memory encryption and integrity verification cover code and data stored in the main memory and communicated over the data bus. Information leakage through the address bus is not protected, but separate protection for the address bus such as proposed in [12, 54, 55] can be employed in conjunction with our mechanisms.

We assume that a proper infrastructure is in place for secure applications to be distributed to end users for use on secure processors. Finally, we also assume that the secure processor is executing applications in the *steady state*. More specifically, we assume that the secure processor already contains the cryptographic keys and code necessary to load a secure application, verify its digital signature, and compute the Merkle Tree over the application in memory.

## 2.3 Memory Encryption

### 2.3.1 Overview of Counter-Mode Encryption

The goal of memory encryption is to ensure that all data and code stored outside the secure processor boundary is in an unintelligible form, not revealing anything about the actual values stored. Figure 2.1 illustrates how this is achieved with counter-mode encryption. When a block is being written back to memory, a *seed* is encrypted using a block cipher (e.g. AES) and a *secret key*, known only to the processor. The encrypted seed is called a cryptographic *pad*, and this pad is combined with the plaintext block via a bitwise XOR operation to generate the ciphertext of the block before the block can be written to memory. Likewise, when a ciphertext block is fetched from memory, the same seed is encrypted to generate the same pad that was used to encrypt the block. When the block arrives on-chip, another bitwise XOR with the pad restores the block to its original plaintext form. Mathematically, if  $P$  is the plaintext,  $C$  is the ciphertext,  $E$  is the block cipher function, and  $K$  is the secret key, the encryption performs  $C = P \oplus E_K(Seed)$ . By XORing both sides with  $E_K(Seed)$ , the decryption yields the plaintext  $P = C \oplus E_K(Seed)$ .

The *security* of counter-mode encryption relies on ensuring that the cryptographic pad (and hence the seed) is unique each time a block is encrypted. Suppose two blocks having plaintexts  $P_1$  and  $P_2$ , and ciphertext  $C_1$  and  $C_2$ , have the same seeds, that is  $Seed_1 = Seed_2$ . Since the block cipher function has a one-to-one mapping, then their



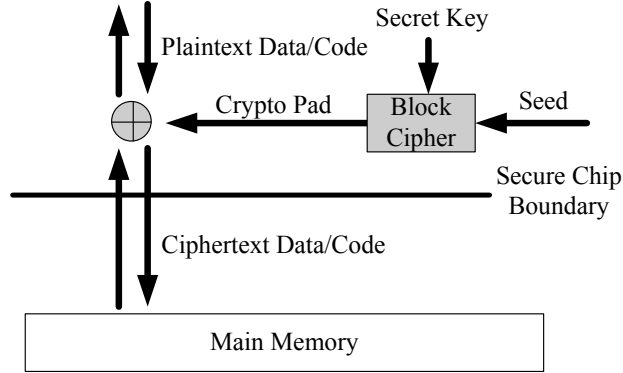


Figure 2.1: Counter-mode based memory encryption.

pads are also the same, i.e.  $E_K(Seed_1) = E_K(Seed_2)$ . By XORing both sides of  $C_1 = P_1 \oplus E_K(Seed_1)$  and  $C_2 = P_2 \oplus E_K(Seed_2)$ , we obtain the relationship of  $C_1 \oplus C_2 = P_1 \oplus P_2$ , which means that if any three variables are known, the other can be known, too. Since ciphertexts are known by the attacker, if one plaintext is known or can be guessed, then the other plaintext can be obtained. Therefore, the security requirement for seeds is that they must be *globally unique*, both spatially (across blocks) and temporally (versions of the same block over time).

The *performance* of counter-mode encryption depends on whether the seed of a code/data block that misses in the cache is available at the time the cache miss is determined. If the seed is known by the processor at the time of a cache miss, the pad for the code/data block can be generated in parallel with the off-chip data fetch, hiding the overhead of memory encryption.

Two methods to achieve the global uniqueness of seeds have been studied. The first is to use a *global counter* as the seed for all blocks in the physical memory. This global counter is incremented each time a block is written back to memory. The global counter approach avoids the use of address as a seed component. However, when the counter reaches its maximum value for its size, it will wrap around and start to reuse its old values. To provide seed uniqueness over time, counter values cannot be reused. Hence, when the counter reaches its maximum, the secret key must be changed, and the *entire physical memory* along with the *swap memory* must be decrypted with the old key and re-encrypted with the new secret key. This re-encryption is very costly and frequent for the global counter approach [50], and can only be avoided by using a large global counter, such as 64 bits. Unfortunately, large counters require a large on-chip *counter cache* storage in order to achieve a good hit rate and overlap decryption with code/data fetch. If the counter for a missed code/data cache block is not found in the counter cache, it must first be fetched from memory along with fetching the code/data cache block. Decryption cannot begin until the counter fetch is complete, which exposes decryption latency and results in poor performance.

To avoid the fast growth of global counters which leads to frequent memory re-encryption, prior studies use per-block counters [40, 46, 50, 52], which are incremented each time the corresponding block is written back to memory. Since each block has its own counter, the counter increases at an orders-of-magnitude slower rate compared to the

global counter approach. To provide seed uniqueness across different blocks, the seed is composed by concatenating the per-block counter, the block address, and chunk id <sup>1</sup>. This seed choice also meets the performance criterion since block addresses can be known at cache miss time, and studies have shown that frequently needed block counters can be effectively cached on-chip [46, 50, 52] or predicted [40] at cache miss time.

However, this choice for seed composition has several significant disadvantages due to the fact that the block address, which was designed as an underlying component of memory management, is now being used as a component of security. Because of this conflict between the intended use of addresses and their function in a memory encryption scheme, many problems arise for a secure processor when block address (virtual or physical) is used as a seed component. These problems are discussed in the following section.

### 2.3.2 Problems with Current Counter-Mode Memory Encryption

Most general purpose computer systems today employ virtual memory, illustrated in Figure 2.2. In a system with virtual memory, the system gives an abstraction that each process can potentially use all addresses in its virtual address space. A paging mechanism is used to translate virtual page addresses (that a process sees) to physical page addresses (that actually reside in the physical and swap memory). The paging mechanism provides *process isolation* by mapping the same page address of different processes to different

---

<sup>1</sup>A *chunk* refers to the unit of encryption/decryption in a block cipher, such as 128 bits (16 bytes). A cache or memory block of 64 bytes contains four chunks. Seed uniqueness must hold across chunks, hence the *chunk id*, referring to which chunk being encrypted in a block, is included as a component of the seed.

physical pages (circle (2)), and *sharing* by mapping virtual pages of different processes to the same physical page (circle (1)). The paging mechanism often extends the physical memory to the *swap memory* area in disks in order to manage more pages. The swap memory holds pages that are not expected to be used soon (circle (3)). When a page in the swap memory is needed, it is brought in to the physical memory, while an existing page in the physical memory is selected to be replaced into the swap memory.

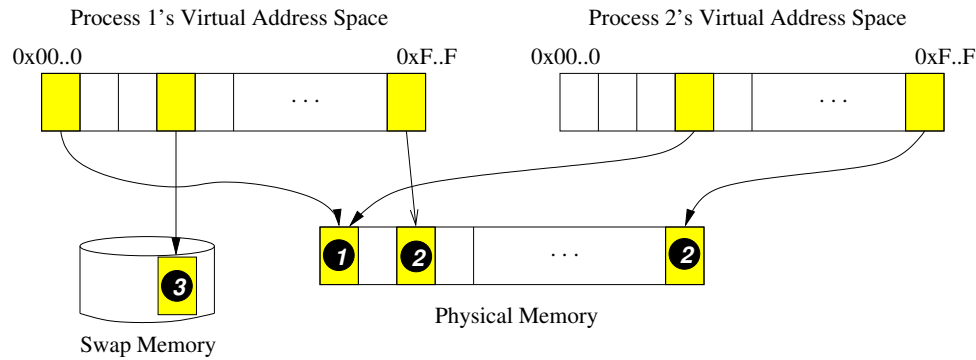


Figure 2.2: Virtual Memory management allows virtual pages of different processes to map to a common physical page for sharing purpose (1), the same virtual pages in different processes to map to different physical pages (2), and some virtual pages to reside in the swap memory in the disk (3).

The use of physical address in the seed causes the following *complexity* and possible *security* problems. The mapping of a virtual page of a process to a physical frame may change dynamically during execution due to page swaps. Since the physical address changes, the entire page must be first decrypted using the old physical addresses and then re-encrypted using the new physical addresses on a page swap. In addition, pages encrypted based on physical address cannot be simply swapped to disk, or pad reuse may

occur between blocks in the swapped out page and blocks located in the page's old location in physical memory. This leaves an open problem as to how to protect pages on disk.

The use of virtual address has its own set of critical problems. Seeds based on virtual address are vulnerable to pad reuse since different processes use the same virtual addresses and could easily use the same counter values. Adding process ID to the seed solves this problem, but creates a new set of system-level issues. First, process IDs can now no longer be reused by the OS, and current OSes have a limit on the range of possible process IDs. Second, shared-memory IPC mechanisms cannot be used. Consider that a single physical page may be mapped into multiple virtual pages in either a single process or in multiple processes. Since each virtual page will see its own process ID and virtual address combination, the seeds will be different and will produce different encryption and decryption results. Consequently, `mmap/munmap` (based on shared-memory) cannot be supported, and these are used extensively in `glibc` for file I/O and memory management, especially for implementing threads. This is a critical limitation for secure processors, especially in the age of CMPs. Third, other OS features that also utilize page sharing cannot be supported. For example, process forking cannot utilize the copy-on-write optimization because the page in the parent and child are encrypted differently. This also holds true for shared libraries. Finally, since virtual addresses are often not available beyond the L1 cache, extra storage may be required for virtual addresses at the lowest level on-chip cache.

One may attempt to augment counter-mode encryption with special mechanisms

to deal with paging or IPC. Unfortunately, they would likely result in great complexity. For example, when physical address is used, to avoid seed/pad reuse in the swap memory, an authentic, secure OS running on the secure processor could encrypt and decrypt swapped pages in software. However this solution is likely not desirable since it makes the secure processor's hardware-based security mechanisms contingent on a secure and uncompromised OS. OS vulnerabilities may be exploited in software by attackers to subvert the secure processor. Alternatively, we could rely on hardware to re-encrypt swapped pages; however this solution has its own set of problems. First, this requires supporting two encryption methods in hardware. A page that is swapped out must first be decrypted (using counter mode) and then encrypted (using direct mode) before it is placed in the swap memory, while the reverse must occur when a page is brought from the disk to the physical memory. Second, there is the issue of who can request the page re-encryptions, and how these requests are made, which requires an extra authentication mechanism. Another example, when virtual address is used, is that shared memory IPC and copy-on-write may be enabled by encrypting all shared pages with direct encryption, while encrypting everything else with counter-mode encryption. However, this also complicates OS handling of IPC and copy-on-write, and at the same time complicates the hardware since it must now support two modes of encryption. Therefore, it is arguably better to identify and deal with the root cause of the problem: address is used as a *fundamental component of memory*

*management*, and using the address also as a basis for *security* intermingles security and memory management in undesirable ways.

### 2.3.3 Address Independent Seed Encryption

In light of the problems caused by using address as a seed component, we propose a new seed composition mechanism which we call *Address-Independent Seed Encryption (AISE)*, that is free from the problems of address-based seeds. The key insight is that rather than using addresses as a seed component alongside a counter, we use *logical identifiers* instead. These logical identifiers are truly unique across the entire physical and swap memory and over time.

Conceptually, each block in memory must be assigned its own logical identifier. However, managing and storing logical identifiers for the entire memory would be quite complex and costly (similar to global counters). Fortunately, virtual memory management works on the granularity of pages (usually 4 Kbytes) rather than words or blocks. Any block in memory has two components: page address which is the unit of virtual memory management, and page offset. Hence, it is sufficient to assign logical identifiers to pages, rather than to blocks. Thus, for each chunk in the memory, its seed is the concatenation of a *Logical Page Identifier* (LPID), the page offset of the chunk's block, the block's counter value, and the chunk id.

To ensure complete uniqueness of seeds across the physical and swap memory

and over time, the LPID is chosen to be a *unique value* assigned to a page when it is *first allocated* by the system. The LPID is unique for that page across the system lifetime, and never changes over time. The unique value is obtained from an on-chip counter called the *Global Page Counter* (GPC). Once a value of the GPC is assigned to a new page, it is incremented. To provide true uniqueness over time, the GPC is stored in a *non-volatile* register on chip. Thus, even across system reboots, hibernation, or power optimizations that cut power off to the processor, the GPC retains its value. Rebooting the system does not cause the counter to reset and start reusing seeds that have been used in the past boot. The GPC is also chosen to be large (64 bits), so that it does not overflow for millenia, easily exceeding the lifetime of the system. Further details on the assignment of LPID to pages is presented in Section 2.3.6.

One may have concerns for how the LPID scheme can be used in systems that support multiple page sizes, such as when super pages (e.g. 16 MBs) are used. However, the number of page offset bits for a large page always exceeds the number of page offset bits for a smaller page. Hence, if we choose the LPID portion of the seed to have as many bits as needed for the smallest page size supported in the system, the LPID still covers the unit of virtual memory management (although sometimes unnecessarily covering some page offset bits) and provides seed uniqueness for the system.

The next issue we address is how to organize the storage of LPIDs of pages in the system. One alternative is to add a field for the LPID in page table entries and



TLB entries. However, this approach significantly increases the page table and TLB size, which is detrimental to system performance. Additionally, the LPID is only needed for accesses to off-chip memory, while TLBs are accessed on each memory reference. Another alternative would be to store the LPIDs in a dedicated portion of the physical memory. However this solution also impacts performance since a memory access now must fetch the block's counter and LPID in addition to the data, thus increasing bandwidth usage. Consequently, we choose to co-store LPIDs and counters, by taking an idea from the split counter organization [50]. We associate each counter block with a page in the system, and each counter block contains one LPID and all block counters for a page.

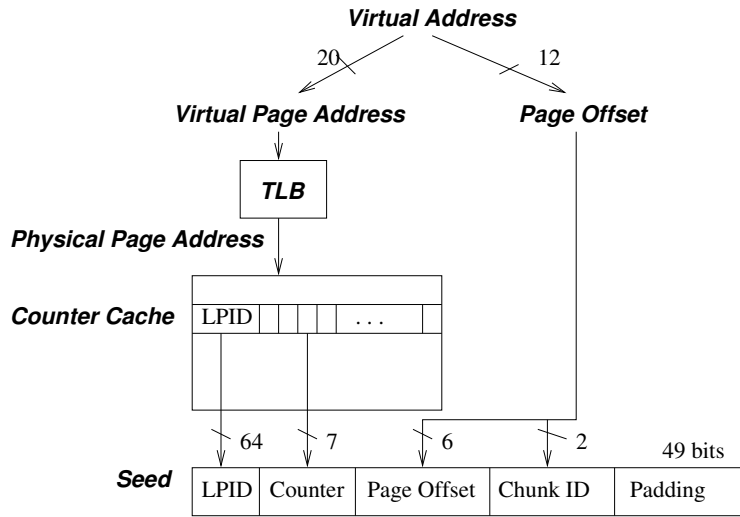


Figure 2.3: Organization of logical page identifiers.

Figure 2.3 illustrates the organization, assuming 32-bit virtual addresses, a 4-Kbyte page size, 64-byte blocks, 64-bit LPID, and a 7-bit counter per block. A virtual

address is split into the high 20-bit virtual page address and 12-bit page offset. The virtual page address is translated into the physical address, which is used to index a counter cache. Each counter block stores the 64-bit LPID of a page, and 64 7-bit counters where each counter corresponds to one of the 64 blocks in a page. If the counter block is found, the LPID and one counter are used for constructing the seed for the address, together with the 8 high order bits of the page offset (6-bit block offset and 2-bit chunk id). Padding is added to make the seed 128 bits, which corresponds to the chunk size in the block cipher. Note that the LPID and counter block can be found using simple indexing for a given physical address.

In contrast to using two levels of counters in [50], we only use small per-block (minor) counters. We eliminate the major counter and use the LPID instead. If one of the minor counter overflows, we need to avoid seed reuse. To achieve that, we assign a new LPID for that page by looking up the GPC, and re-encrypt only that page. Hence, the LPID of a page is no longer static. Rather, a new unique value is assigned to a page when a page is first allocated *and* when a page is re-encrypted.

### 2.3.4 Dealing with Swap Memory and Page Swapping

In our scheme, no two pages share the same LPID and hence seed uniqueness is guaranteed across the physical and swap memory. In addition, once a unique LPID value is assigned to a page, it does not change until the page needs to be re-encrypted. Hence,

when a page is swapped out to the disk, it retains a unique LPID and does not need to be re-encrypted or specially handled. The virtual memory manager can just move a page from the physical memory to the swap memory. The page's LPID and block of counters can be moved to the swap memory as well to free up as much memory as possible, or moved to a region in the kernel memory if one wants to reduce I/O activities.

When an application suffers a page fault, the virtual memory manager locates the page and its block of counters in the disk, then brings them into the physical memory. The block of counters (including LPID) is placed at the appropriate physical address in order for the block to be directly indexable and storable by the counter cache. Therefore, the only special mechanism that needs to be added to the page swapping mechanism is proper handling of the page's counter blocks. Since no re-encryption is needed, moving the page in and out of the disk can be accomplished with or without the involvement of the processor (e.g. we could use DMA).

Finally, while in AISE we rely on the OS to swap pages between the main memory and the disk, we do so in an indirect way. Specifically, we only rely on the OS to move already protected data, which has been previously encrypted and authenticated in hardware, around in memory and between memory and disk. Thus we avoid the more dangerous form of OS-dependence where the system relies on the OS to directly perform cryptographic functions in certain cases (i.e. directly encrypting, decrypting or authenticating data values). For example, we do not want to rely on the OS to encrypt and decrypt

data moved between memory and the disk because the OS has direct access to plaintext values in this case, and a compromised OS could access and/or tamper with plaintext values. In our scheme, the OS has no direct access to plaintext and hence, the security of the data afforded by our scheme is not contingent on an uncompromised OS.

### 2.3.5 Dealing with Page Sharing

Page sharing is problematic to support if virtual address is used as a seed component, since different processes may try to encrypt or decrypt the same page with different virtual addresses. With our LPID scheme, the LPID is unique for each page and can be directly looked up using the physical address. Therefore, all page sharing uses can naturally be facilitated without any special mechanisms.

### 2.3.6 Virtualization support for Secure Processors

Virtualization was first introduced in the 1960s to allow partitioning of expensive mainframe resources, particularly the main memory. Modern computers with increased processing power have led to a resurgence of interest in Virtualization Technology as a way to multiplex the real machine into multiple *Virtual Machines* (VMs) with each VM running a separate Operating System instance. The *virtualization layer* (*hypervisor* or *Virtual Machine Monitor*) is responsible for controlling and allocating hardware resources to the VMs.

Virtualization increasingly finds applications in supporting server and workload consolidation [32], distributed web servers [48] and secure computing platforms [13] among others. The growing importance and applications of virtualization make it imperative that secure processor designs continue to support virtualization in a manner similar to current systems. We show that AISE allows virtualization to be supported naturally by requiring minimal changes to the hypervisor or the Virtual Machine Monitor (VMM) for *virtualizing the LPID*. AISE allows support for all variants of virtualization, namely, *full virtualization*, *paravirtualization* and *hardware assisted virtualization*.

**Virtualization Model.** In a virtualized environment, the guest operating systems run on top of the VMM. The VMM itself can either run directly on top of the hardware (*Type 1* or *native VM*) or on top of a host operating system (*Type 2* or *hosted VM*). The two models are shown in Figure 2.4.

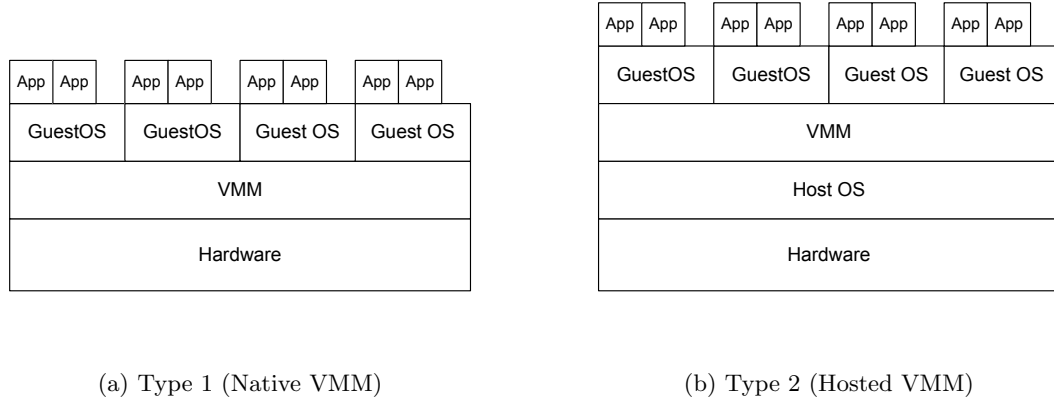


Figure 2.4: VMM Models

Type 2 VMM uses the existing host operating system’s abstractions to implement its services. The two levels of indirection in Type 2 VMM result in poor performance when compared to Type 1 VMM which runs directly on the hardware. The rest of this discussion assumes a Type 1 VMM similar to the one used by VMWare’s ESX server [47] and Xen [5].

**OS changes to support AISE.** Before we present the proposed virtualization of the LPID, we discuss the details of how LPIDs are assigned to pages. As discussed in section 2.3.3, a page is assigned an LPID when it is first loaded to the main memory. This assignment requires the OS to access the non-volatile, on-chip GPC register when a page is loaded from disk to main memory and to assign this counter value as the LPID for the page.

We propose a new *privileged* instruction for this purpose, which we call RDAGPC (Read and Assign GPC). This instruction loads (Read) the GPC register, associates (Assign) this GPC value as the LPID for the page being loaded from disk, and subsequently increments the GPC register value to prevent the reuse of LPIDs. RDAGPC is a privileged instruction so only the Operating System kernel running at the highest priority level has the permission to execute it. More specifically, the page fault handler of the OS will execute this instruction after determining that the required page was not in main memory and needs to be loaded from the disk (*major page fault*). Figure 2.5 shows a part of the page fault handling routine which is responsible for handling major page faults that needs to be modified in order to exploit AISE capabilities.

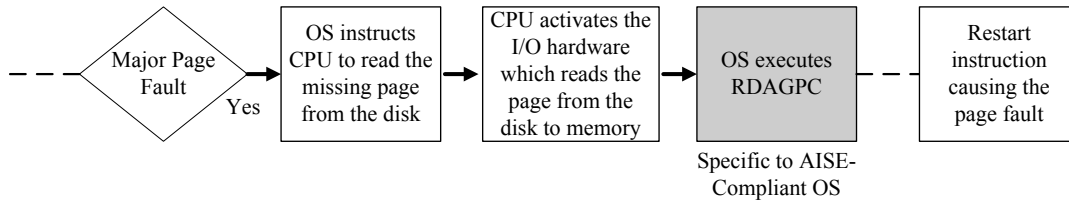


Figure 2.5: Portion of Page Fault Handling Routine of an AISE-compliant OS.

Based on this model, we now discuss how virtualization, in its three main varieties, can be supported on an AISE-enabled secure processor.

### 2.3.7 Virtualizing the LPID

**Full Virtualization.** Full virtualization using binary translation is the most widely used variant of virtualization today [1]. In Full virtualization, the guest OSes run unmodified except that they run at a lower privilege level (CPL 1). The VMM runs at the highest priority level (CPL 0) and is responsible for managing the hardware resources. When a guest OS attempts to execute a privileged instruction, it causes a trap into the VMM, and the VMM is responsible for emulating the instruction. The instruction we introduced for assigning LPIDs to pages loaded to memory from the disk, RDAGPC, is a privileged instruction and will therefore result in a trap to the VMM when executed by a guest OS.

The guest OS is not allowed to access the physical memory directly. The guest OS

assigns *virtual frames* for the virtual pages and the VMM is responsible for mapping these virtual frames to *real frames*. The guest OS maintains its own copy of virtual page tables which contain the mappings from virtual page numbers to virtual frame numbers. The VMM shadows the page tables of the guest OSes. The shadowed page structure maintains a mapping from the virtual page numbers to the real page numbers. Coherency of the shadow structures is maintained via *tracing*, wherein the guest OS can read its page table but any write to the page table results in a trap to the VMM.

On a page fault in the guest OS, the guest OS executes its page fault handling routine and constructs a mapping from the virtual page number to a virtual frame number. As part of the page fault routine, AISE compliant OSes execute the RDAGPC instruction to assign the LPID. When the guest OS executes this instruction, it causes a trap to the VMM. The VMM, knowing that it is responsible for assigning the LPIDs *ignores* the RDAGPC instruction and returns control to the guest OS. Next, the guest OS attempts to write the newly constructed mapping to its page table. This again results in a trap to the VMM. The VMM now constructs an appropriate shadow page table entry by mapping the virtual page to a real frame. As part of this process, the VMM may be required to load a page from the disk, if the page is not already present in the main memory. If the page is loaded from disk, the VMM executes the RDAGPC instruction to associate an LPID for this page. Once the shadow page table entry is created, the VMM resumes the guest OS execution. The guest OS can now write the mapping it constructed to its page table.



The same guest OS, when run in a non-virtualized environment will continue to assign the LPIDs to pages loaded from the disk.

Hence, by virtualizing the on-chip GPC register and making the VMM responsible for managing and assigning LPIDs on behalf of the guest OSes, full virtualization is easily supported with AISE, without requiring any changes to an AISE compliant OS and requiring minimal changes to the VMM.

**Paravirtualization.** In paravirtualization, guest OSes are modified to know that they are running inside a VM. The VMM provides a *hypercall* interface to provide services to the VMs. The guest OSes access all hardware state through hypercalls by voluntarily trapping to the VMM. Each guest OS knows that it does not have the entire physical memory. Instead, each guest OS requests physical memory from the VMM. Each guest OS is statically assigned its share of memory during initialization and is responsible for managing its own memory in a way similar to *self paging* [17].

Unlike full virtualization, there is no concept of virtual frame numbers as now the guest OS performs its own paging. When the guest OS requires a new page table, possibly due to a new process creation, it allocates the page table from its own memory store. Any writes to the page table must be validated by the hypervisor via hypercalls.

On a page fault, the guest OS executes its page fault handler. If the requested physical frame is loaded from disk, the RDAGPC instruction is executed. RDAGPC, being a privileged instruction, traps to the VMM via a hypercall. The VMM then emulates

this instruction by actually executing it on behalf of the guest OS. Once the RDAGPC instruction is completed, the physical frame loaded from the disk will have an associated LPID and the guest OS is restarted. Hence, similar to full virtualization, by virtualizing the on-chip GPC register, paravirtualization can be easily supported without requiring any changes to an AISE compliant OS and requiring minimal changes to the VMM.

**Hardware assisted virtualization.** Intel [9] and AMD [3] have recently introduced hardware virtualization where privileged instructions automatically trap to the VMM. The first generation of hardware assisted virtualization techniques do not provide support for memory virtualization. On present generation hardware, the VMM is responsible for virtualizing memory. Hence, AISE can support virtualization as described above. When hardware supports memory virtualization, it should be straightforward to extend the support with AISE. Current hardware allows the VMM to specify unconditional traps. Assuming that next generation hardware continues to allow the VMM to specify unconditional traps, virtualization can be supported with AISE in the environment by specifying accesses to GPC register to trap to the VMM.

### 2.3.8 Advantages of AISE

AISE satisfies the security and performance criteria for counter-mode encryption seeds, while naturally supporting virtual memory management features and IPC without much complexity. The LPID portion of the seed ensures that the blocks in every page, both

in the physical memory and on disk are encrypted with different pads. The page offset portion of the seed ensures that each block within a page is encrypted with a different pad. The block counter portion of the seed ensures that the pad is unique each time a single block is encrypted. Finally, since the global page counter is stored in non-volatile storage on chip, the pad uniqueness extends across system boots.

From a performance perspective, AISE does not impose any additional storage or runtime overheads over prior counter-mode encryption schemes. AISE allows seeds to be composed at cache miss time since both the LPID and counter of a block are co-stored in memory and cached together on-chip. Storage overhead is equivalent to the already-efficient split counter organization, since LPID replaces the major counter of the split counter organization and does not add extra storage. On average, a 4 Kbyte page only requires 64 bytes of storage for the LPID and counters, representing a 1.6% overhead. Similar to the split counter organization, AISE does not incur entire-memory re-encryption when a block counter overflows. Rather, it only incurs re-encryption of a page when overflow occurs.

From a complexity perspective, AISE allows pages to be swapped in and out of the physical memory without involving page re-encryption (unlike using physical address), while allowing all types of IPC and page sharing (unlike using virtual address). AISE can be naturally extended to provide support for all variants of virtualization without requiring any modifications to an AISE compliant OS and with minimal changes to the existing VMMs.

To summarize, memory encryption using our AISE technique retains all of the latency-hiding ability as proposed in prior schemes, while eliminating the significant problems that arise from including address as a component of the cryptographic seed.

## 2.4 Memory Integrity Verification

The goal of a memory integrity verification scheme is to ensure that a value loaded from some memory location by a processor is *equal to the most recent* value that the processor last wrote to that location. There are three types of attacks that may be attempted by an attacker on a value at a particular location. Attackers can replace the value directly (*spoofing*), exchange the value with another value from a different location (*splicing*), and replay an old value from the same location (*replay*). As discussed in XOM [15], if for each memory block a MAC is computed using the value and address as its input, spoofing and splicing attacks would be detectable. However, replay attacks can be successfully performed by rolling back both the value and its MAC to their older versions. To detect replay attacks, Merkle Tree verification has been proposed [14]. A Merkle Tree keeps hierarchical MACs organized as a tree, in which a parent MAC protects multiple child MACs. The root of the tree is stored on-chip at all times so that it cannot be tampered by attackers. When a memory block is fetched, its integrity can be verified by checking its chain of MAC values up to the root MAC. When a cache block is written back to memory, the corresponding MAC values of the tree are updated. Since the on-chip MAC root contains information about every block in the physical memory, an attacker cannot modify or replay any value in the physical memory.

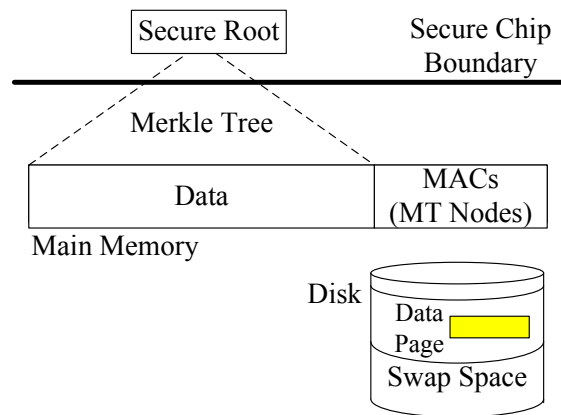
### 2.4.1 Extended Merkle Tree Protection

Previously proposed Merkle Tree schemes which only cover the physical memory, as shown in Figure 2.6(a), compute MACs on memory events (cache misses and write backs) generated by the processor. However, I/O transfer between the physical memory and swap memory is performed by an I/O device or DMA and is not visible to the processor. Consequently, the standard Merkle Tree protection only covers the physical memory but not the swap memory. This is a significant security vulnerability since by tampering with the swap memory in the disk, attackers can indirectly tamper with the main memory. We note that it would be possible to entrust a secure OS with the job of protecting pages swapped to and from the disk in software. However, this solution requires the assumption of a secure and untampered OS which may not be desirable. Also, as discussed in [45], it would be possible to compute the Merkle Tree over the virtual address space of each process to protect the process in both the memory and the disk. However this solution would require one Merkle Tree and on-chip secure root MAC per process, which results in extra on-chip storage for the root MACs and complexity in managing multiple Merkle Trees.

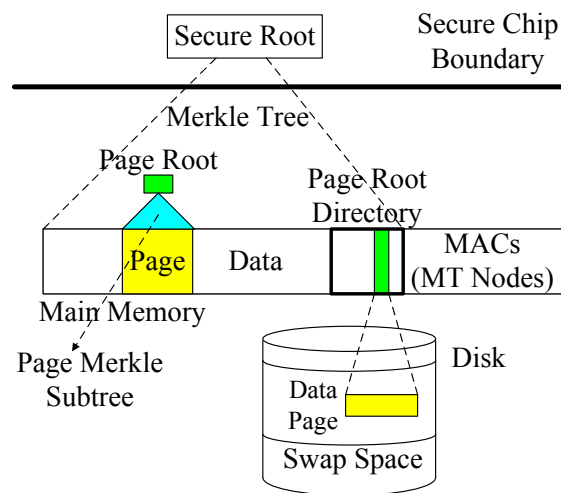
This security issue clearly motivates the need to extend the Merkle Tree protection to all off-chip data both in the physical and swap memory, as illustrated in Figure 2.6(b). To help explain our solution, we define two terms: *Page Merkle Subtree* and *page root*. A Page Merkle Subtree is simply the subset of all the MACs of the Merkle Tree which directly

cover a particular page in memory. A page root is the top-most MAC of the Page Merkle Subtree. Note that the Page Merkle Subtree and page root are simply MAC values which make up a portion of the larger Merkle Tree over the entire physical memory.

To extend Merkle Tree protection to the swap memory, we make two important observations. First, for each page, its page root is sufficient to verify the integrity of all values on the page. The internal nodes of the Page Merkle Subtree can be re-computed and verified as valid by comparing the computed page root with the stored, valid page root. Second, the physical memory is covered entirely by the Merkle Tree and hence it provides secure storage. From these two observations, we can conclude that as long as the page roots of all swap memory pages are stored in the physical memory, then the entire swap memory integrity can be guaranteed. To achieve this protection, we dedicate a small portion of the physical memory to store page root MACs for pages currently on disk, which we refer to as the *Page Root Directory*. Note that while our scheme requires a small amount of extra storage in main memory for the page root directory, the on-chip Merkle Tree operations remain the same and a single on-chip MAC root is still all we require to maintain the integrity of the entire tree. Furthermore, as shown in Figure 2.6(b), the page root directory itself is protected by the Merkle Tree. The implication of this design is that every page in memory is associated with a page root. If a page needs to be swapped out to the disk, then we can maintain the integrity of its page root by retaining it in memory (in the page root directory), which is secure since it is protected by the Merkle Tree over



(a) Standard Merkle Tree Organization



(b) Extended Merkle Tree Organization

Figure 2.6: Merkle Tree organization for extending protection to the swap memory in disk.



memory. The page roots are themselves stored on a memory page that also has its own page root. Thus we can continue this process of swapping pages out to disk, and retaining their page roots in the Merkle Tree protected memory to allow the later verification of any pages reloaded into memory from the swap space on disk.

To illustrate how our solution operates, consider the following example. Suppose that the system wants to load a page B from swap memory into physical memory currently occupied by a page A. The integrity verification proceeds as follows. First, the page root of B is looked up from the page root directory and brought on chip. Since this lookup is performed using a regular processor read, the integrity of the page root of B is automatically verified by the Merkle Tree. Second, page A is swapped out to the disk and its page root is installed at the page root directory. This installation updates the part of the Merkle Tree that covers the directory, protecting the page root of A from tampering. Third, the Page Merkle Subtree of A is invalidated from on-chip caches in order to force future integrity verification for the physical frame where A resided. Next, the page root of B is installed in the proper location as part of the Merkle Tree, and the Merkle Tree is updated accordingly. Finally, the data of page B can be loaded into the physical frame. When any value in B is loaded by the processor, the integrity checking will take place automatically by verifying data against the Merkle Tree nodes at least up to the already-verified page root of B.

### 2.4.2 Bonsai Merkle Trees

We introduce Bonsai Merkle Trees (BMTs), a novel Merkle Tree organization designed to significantly reduce their performance overhead for memory integrity verification. To motivate the need for our BMT approach, we note a common optimization that has been studied for Merkle Tree verification is to cache recently accessed and verified MAC values on chip [14]. This allows the integrity verification of a data block to complete as soon as a needed MAC value is found cached on-chip. Since this MAC value has previously been verified and is safe on-chip, it can be trusted as if it were the root of the tree. The resulting reduction in memory bandwidth consumption significantly improves performance compared to fetching MAC values up to the tree root on every data access. However, the sharing of on-chip cache between data blocks and MAC values can significantly reduce the amount of available cache space for data blocks. In fact, our experiments show that for memory-intensive applications, up to 50% of a 1MB L2 cache can be consumed by MAC values during application execution, severely degrading performance. It is likely that MACs occupy such a large percentage of cache space because MACs in upper levels of a Merkle Tree have high temporal locality when the verification is repeated due to accesses to the data blocks that the MAC covers.

Before we describe our BMT approach, we motivate it from a security perspective. BMTs exploit certain security properties that arise when Merkle Tree integrity verification is used in conjunction with counter-mode memory encryption. We make two observations.

First, the Merkle Tree is designed to prevent data replay attacks. Other types of attacks such as data spoofing and splicing can be detected simply by associating a single MAC value with each data block. Second, in most proposed memory encryption techniques using counter-mode, each memory block is associated with its own counter value in memory [39, 40, 46, 50, 52]. Since a block's counter value is incremented each time a block is written to memory, the counter can be thought of as a *version* number for the block. Based on these observations, we make the following claim:

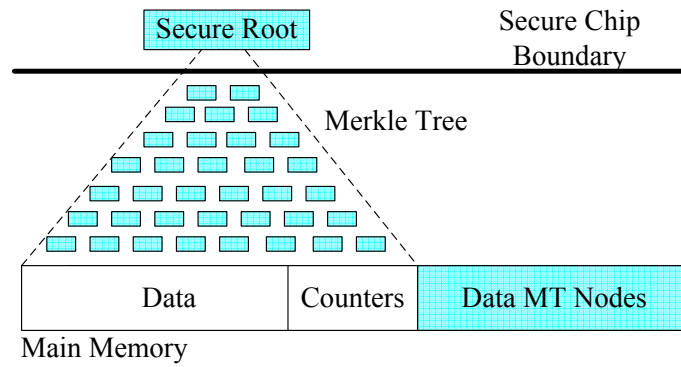
*In a system with counter-mode encryption and Merkle Tree memory integrity verification, data values do not need to be protected by the Merkle Tree as long as (1) each block is protected by its own MAC, computed using a keyed hashing function (e.g. HMAC based on SHA-1), (2) the block's MAC includes the counter value and address of the block, and (3) the integrity of all counter values is guaranteed.*

To support this claim, we provide the following argument. Let us denote the plaintext and ciphertext of a block of data as  $P$  and  $C$ , its counter value as  $ctr$ , the MAC for the block as  $M$ , and the secret key for the hash function as  $K$ . The MAC of a block is computed using a keyed cryptographic hash function  $H$  with the ciphertext and counter as its input, i.e.  $M = H_K(C, ctr)$ . Integrity verification computes the MAC and compares it against the MAC that was computed in the past and stored in the memory. If they do not match, integrity verification fails. Since the integrity of the counter value is guaranteed (a

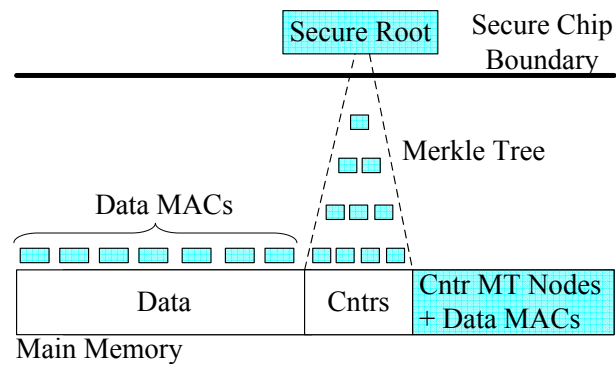
requirement in the claim), attackers cannot tamper with  $ctr$  without being detected. They can only tamper with  $C$  to produce  $C'$ , and/or the stored MAC to produce  $M'$ . However, since the attacker does not know the secret key of the hash function, they cannot produce a  $M'$  to match a chosen  $C'$ . In addition, due to the non-invertibility property of a cryptographic hash function, they cannot produce a  $C'$  to match a chosen  $M'$ . Hence,  $M' \neq H_K(C', ctr)$ . Since, during integrity verification, the computed MAC is  $H_K(C', ctr)$ , while the stored one is  $M'$ , integrity verification will fail and the attack will be detected. In addition, attackers cannot replay both  $C$  and  $M$  to their older version because the old version satisfies  $M^{old} = H_K(C^{old}, ctr^{old})$ , while the integrity verification will compute the MAC using the fresh counter value whose integrity is assumed to be guaranteed ( $H_K(C^{old}, ctr)$ ), which is not equal to  $H_K(C^{old}, ctr^{old})$ . Hence replay attacks are also detected.

The claim is significant because it implies that we only need the Merkle Tree to cover counter blocks, but not code or data blocks. Since counters are a lot smaller than data (a ratio of 1:64 for 8-bit counters and 64-byte blocks), the Merkle Tree to cover the block counters is *substantially* smaller than the Merkle Tree for data. Figure 2.7(a) shows the traditional Merkle Tree which covers all data blocks, while Figure 2.7(b) shows our BMT that only covers counters, while data blocks are now only covered by their MACs.

Since the size of the Merkle Tree is significantly reduced, and since each node of the Merkle Tree covers more data blocks, the amount of on-chip cache space required to



(a) Standard Merkle Tree



(b) Bonsai Merkle Tree

Figure 2.7: Reduction in size of Bonsai Merkle Trees compared to standard Merkle Trees.

store frequently accessed Bonsai Merkle Tree nodes is significantly reduced. To further reduce the cache footprint, we do not cache data block MACs. Since each data block MAC only covers four data blocks, it has a low degree of temporal reuse compared to upper level MACs in a standard Merkle Tree. Hence, it makes sense to only cache Bonsai Merkle Tree nodes but not data block MACs, as we will show in Section 2.6.

Overall, BMTs achieve the same security protection as in previous schemes where a Merkle Tree is used to cover the data in memory (i.e. data spoofing, splicing, and replay protection), but with much less overhead. Also note that BMTs easily combine with our technique to extend Merkle Tree based protection to the disk. When a page of data is swapped out to disk, the counters must always be swapped out and stored as well. Therefore we simply keep a portion of memory to store the page roots for *Bonsai Page Merkle Subtrees* on the disk as described in the previous section.

## 2.5 Experimental Setup

### 2.5.1 Machine Models

We use SESC [22], an open source, execution-driven, cycle-level simulator, to evaluate the performance of the proposed memory encryption and integrity verification approaches. We model a 2GHz, 3-issue, out-of-order processor with split L1 data and instruction caches. Both caches have a 32KB size, 2-way set associativity, and 2-cycle round-trip hit latency. The L2 cache is unified and has a 1MB size, 8-way set associativity, and 10-cycle round-trip hit latency. For counter mode encryption, the processor includes a 32KB, 16-way set-associative counter cache at the L2 cache level. All caches have 64B blocks and use LRU replacement. We assume a 1GB main memory with an access latency of 200 processor cycles. The encryption/decryption engine simulated is a 128-bit AES engine with a 16-stage pipeline and a total latency of 80 cycles as in the design described in [23], while the MAC computation models HMAC [16] based on SHA-1 [10] with 80-cycle latency [23]. Counters are composed of a 64-bit LPID concatenated with a 7-bit block counter. So a counter cache block contains one LPID value along with 64 block counters (enough for a 4KB memory page). The default authentication code size used is 128 bits.

For the CMP evaluation, we model a two-core CMP system where each core has private L1 data and instruction caches. The L2 cache and all lower levels of the memory hierarchy are shared by both cores. To better match current CMP configurations, we have

changed two of the main system parameters from the uniprocessor model. The L2 cache size is increased to 2 MB and the memory bus bandwidth is increased to 20GBytes/s. All other system parameters are the same as the uniprocessor case.

### 2.5.2 Benchmarks

We use 21 C/C++ SPEC2K benchmarks [44] for our single-core processor evaluations. We only omit Fortran 90 benchmarks, which are not supported on our simulator infrastructure. In each figure, we show the individual result for benchmarks that have an L2 cache miss rate higher than 20%, but the average is calculated across all 21 benchmarks that we simulate.

For our CMP evaluations, we have created 26 pairs of benchmarks using the SPEC2K benchmarks. Each pair consists of two SPEC2K benchmarks which are spawned as two separate threads, one on each of the two cores of the modeled CMP system. To capture different memory behaviors, we classify the benchmarks into two categories: those that when run alone have L2 cache miss rates of less than 20% and those that have L2 cache miss rates of 20% or higher. We select a few benchmarks from each group and match them so that all combinations are represented. In the first group of benchmark pairs, the benchmarks in a pair are both taken from the low miss rate set: *perlbnk\_twolf* and *twolf\_vpr*. In the second group of benchmark pairs, one benchmark in a pair is taken from the low miss rate set while another is taken from the high miss rate set: *apsi\_bzip2*, *gzip\_applu*, *gzip\_apsi*,



*gzip\_art*, *perlbmk\_art*, *perlbmk\_swim*, *swim\_gzip*, *swim\_twolf*, *twolf\_swim*, *vpr\_applu*, *vpr\_art*, *applu\_gzip*, and *swim\_perlbmk*). The last group of benchmark pairs takes both benchmarks in a pair from the high miss rate set: *apsi\_art*, *apsi\_eqquake*, *apsi\_mcf*, *art\_mcf*, *art\_swim*, *mcf\_art*, *mcf\_swim*, *swim\_art*, *swim\_mcf*, *equake\_apsi*, and *mcf\_apsi*.

For each simulation, we use the reference input set and simulate for 1 billion instructions after fast forwarding for 5 billion. For CMP simulations, the simulation ends when the combined number of instructions simulated for the benchmark pair reaches 1 billion. In our experiments, we ignore the effect of page swaps as the overhead due to page swaps with our techniques is negligible. Finally, for evaluation purpose, we use timely but non-precise integrity verification, i.e. each block is immediately verified as soon as it is brought on chip, but we do not delay the retirement of the instruction that brings the block on chip if verification is not completed yet. Note that all of our schemes (AISE and BMT) are compatible with both non-precise and precise integrity verification.

## 2.6 Evaluation

To evaluate our approach, we first present a qualitative comparison of AISE against other counter-mode encryption approaches. Then we present simulation results for the performance of our AISE and BMT schemes on a single-core uniprocessor system. Next we show the results of our schemes on multi-core systems. Then we present several sensitivity studies, and finally we compare storage overheads.

### 2.6.1 AISE: Qualitative Evaluation

Table 2.1 qualitatively compares AISE with other counter-mode encryption approaches, in terms of IPC support, cryptographic latency hiding capability, storage overheads, and other miscellaneous overheads. The first scheme, *Global Counter*, was discussed in Section 2.3.1. Like AISE, this scheme supports all forms of IPC and requires no special mechanisms to protect swap memory. However, global counters need to be large (64 bits) to avoid overflow and frequent entire-memory re-encryption (32-bit counters cause entire memory re-encryption every few minutes [50]). Thus, these large counters cache poorly, and predicting counter values is difficult because the values are likely non-contiguous for a particular block over time, resulting in little latency-hiding opportunity (we evaluate this in Section 2.6.2). In addition, the memory storage overhead for using 64-bit counters per-block is high at 12.5%.

The next two configurations represent counter-mode encryption using either phys-

Table 2.1: Qualitative comparison of AISE with other counter-mode encryption approaches

Encryption Type	Global Counter	Counter (Phys Addr)	Counter (Virt Addr)	AISE
IPC Support	Yes	Yes	No shared-memory IPC	Yes
Latency Hiding	Caching: Poor Prediction: Difficult	Depends on ctr size	Depends on ctr size	Good
Storage Overhead	High (64-bit: 12.5%)	Depends on ctr size	Depends on ctr size	Low (1.6%)
Other Issues	None	Re-enc on page swap	VA storage in L2	None

ical (*Counter (Phys Addr)*) or virtual address (*Counter (Virt Addr)*) plus per-block counters to compose seeds. As shown in the table, while AISE is amenable to all forms of IPC, including shared-memory, virtual address based schemes cannot support this popular type of communication. In addition, virtual address schemes require this address to be stored in the lowest level cache so that it can be readily accessed, and physical address schemes require page re-encryption on page swaps. Finally, while AISE will work well with proposed counter caching and prediction schemes and require only small storage overheads, virtual and physical address schemes depend on the chosen counter size.

### 2.6.2 Single-Core Processor Evaluation Results

The first experiment compares AISE+BMT to another memory encryption and integrity verification scheme which can provide the same type of system-level support as our approach (e.g. shared memory IPC, virtual memory support, etc.). Figure 2.8 shows these results of AISE+BMT compared to the 64-bit global counter scheme plus standard Merkle Tree protection (global64+MT), where the execution time overhead is shown normalized

to a system with no protection. While the two schemes offer similar system level benefits, the performance benefit of our AISE+BMT scheme is tremendous. The average execution time overhead of global64+MT is 25.9% with a maximum of 151%, while the average for AISE+BMT is a mere 1.8% with a maximum of only 13%. This figure shows that our AISE+BMT approach overwhelmingly provides the best of both worlds in terms of support of system-level issues and performance overhead reduction, making it more suitable for use in real systems.

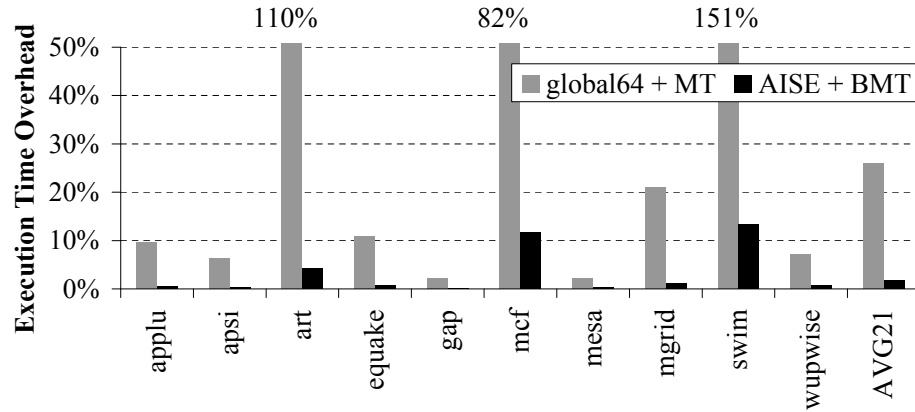


Figure 2.8: Performance overhead comparison of AISE with BMT vs. the Global counter scheme with a traditional Merkle Tree

To better understand the results from the previous figure, the next figures break the overhead into encryption vs. integrity verification components. Figure 2.9 shows the normalized execution time overhead of AISE compared to the global counter scheme with 32-bit and 64-bit counters (note that only encryption is being performed for this figure). As the figure shows, AISE by itself is significantly better from a performance perspective than

the global counter scheme (1.6% average overhead vs. around 4% and 6% for 32 and 64-bit global counters). Recall also that 64-bit counters, which should be used to prevent frequent entire-memory re-encryptions [50], require a 12.5% memory storage overhead. Note that we do not show results for counter-mode encryption using address plus block counter seeds since the performance will be essentially equal to AISE if same-sized block counters are used. Since AISE supports important system level mechanisms not supported by address-based counter-mode schemes, and since the performance and storage overheads of AISE are superior to the global counter scheme, AISE is an attractive memory encryption option for secure processors

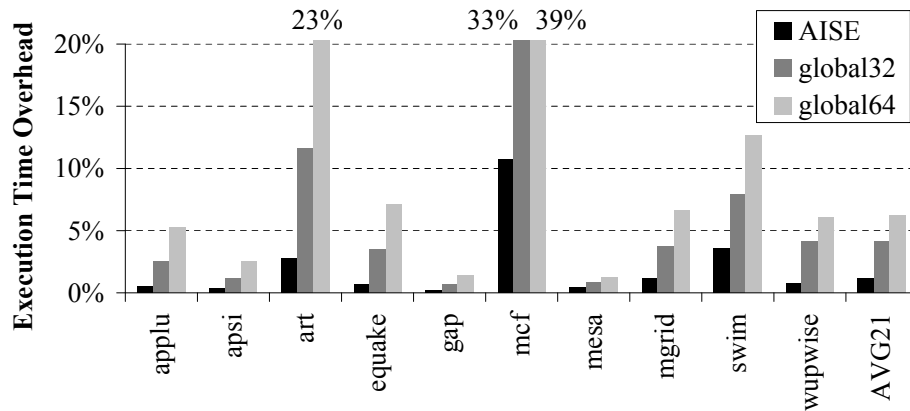


Figure 2.9: Performance overhead comparison of AISE versus the global counter scheme

To see the overhead due to integrity verification, Figure 2.10 shows the overhead of AISE only (the same as the AISE bar on the previous figure), AISE plus a standard Merkle Tree (AISE+MT), and AISE plus our BMT scheme (AISE+BMT). Note that we

use AISE as the encryption scheme for all cases so that the extra overhead due to the different integrity verification schemes is evident. Our first observation is that integrity verification due to maintaining and verifying Merkle Tree nodes is the dominant source of performance overhead, which agrees with prior work [50]. From this figure, it is also clear that our BMT approach outperforms the standard Merkle Tree scheme, reducing the overhead from 12.1% in AISE+MT to only 1.8% in AISE+BMT. Even for memory intensive applications such as *art*, *mcf*, and *swim*, the overhead using our BMT approach is less than 15% while it can be above 60% with the standard Merkle Tree scheme. Also, for every application except for *swim*, the extra overhead of AISE+BMT compared to AISE is negligible, indicating that our BMT approach removes almost all of the performance overhead of Merkle Tree-based memory integrity verification. We note that [50] also obtained low average overheads with their memory encryption and integrity verification approach, but for more memory-intensive workloads such as *art*, *mcf*, and *swim*, their performance overheads still approached 20% and they assumed a smaller, 64-bit MAC size. Since our BMT scheme retains the security strength of standard Merkle Tree schemes, the improved performance of BMTs is a significant advantage.

To understand why the BMT scheme can outperform the standard Merkle Tree scheme by such a significant amount, we next present some important supporting statistics. Figure 2.11 measures the amount of “cache pollution” in the L2 cache due to storing frequently accessed Merkle Tree nodes along with data. The bars in this figure show the

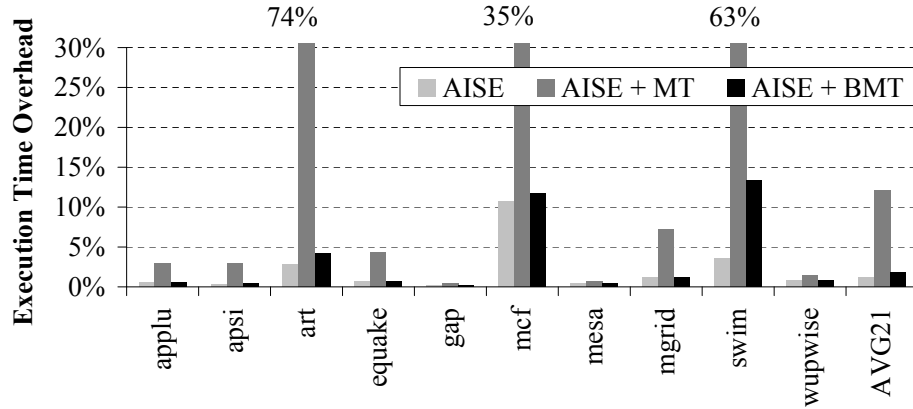


Figure 2.10: Performance overhead comparison of AISE with our Bonsai Merkle Tree vs. AISE with the Standard Merkle Tree

*average* portion of L2 cache space that is occupied by data blocks during execution. For the standard Merkle Tree, we found that on average data occupies only 68% of the L2 cache, while the remaining 32% is occupied by Merkle Tree nodes. In extreme cases (e.g. art and swim), almost 50% of the cache space is occupied by Merkle Tree nodes. Note that for 128-bit MACs, the main memory storage overhead incurred by Merkle Tree nodes stands at 25%; so if the degree of temporal locality of Merkle Tree nodes is equal to data, then only 25% of the L2 cache should be occupied by Merkle Tree nodes. Thus it appears that Merkle Tree nodes have a higher degree of temporal locality than data. Intuitively, this observation makes sense because for each data block that is brought into the L2 cache, one or more Merkle Tree nodes will be touched for the purpose of verifying the integrity of the block. With our BMT approach, on the other hand, data occupies 98% of the L2 cache,

which means that the remaining 2% of the L2 cache is occupied by Bonsai Merkle Tree nodes. This explains the small performance overheads of AISE+BMT. Since the ratio of the size of a counter to a data block is 1:64, the footprint of the BMT is very small, so as expected it occupies an almost negligible space in the L2 cache. Furthermore, since data block MACs are not cached, they do not take up L2 cache space.

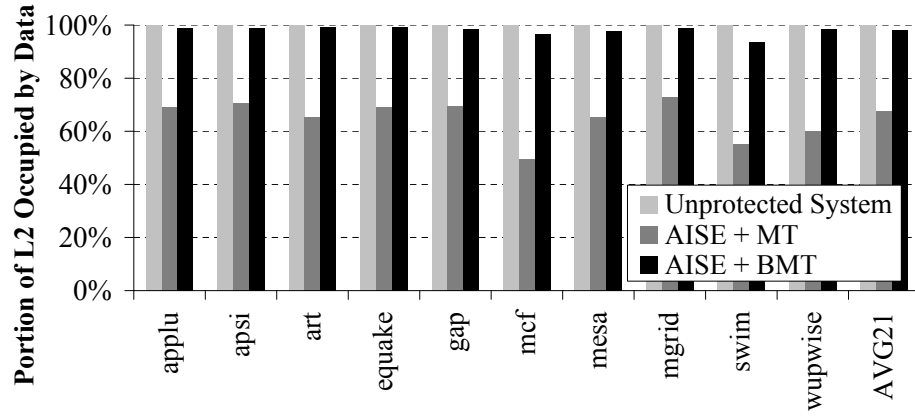


Figure 2.11: L2 cache pollution

Next, the (local) L2 cache miss rate and bus utilization of the base unprotected system, the standard Merkle Tree, and our BMT scheme, are shown in Figure 2.12. The figure shows that while the L2 cache miss rates and bus utilization increase significantly when the standard Merkle Tree scheme is used (average L2 miss rate from 37.8% to 47.5%, bus utilization from 14% to 24%), the BMT scheme only increases L2 miss rates and bus utilization slightly (average L2 miss rate from 37.8% to 38.5% and bus utilization from 14% to 16%). These results show that the impact of reduced cache pollution from Merkle

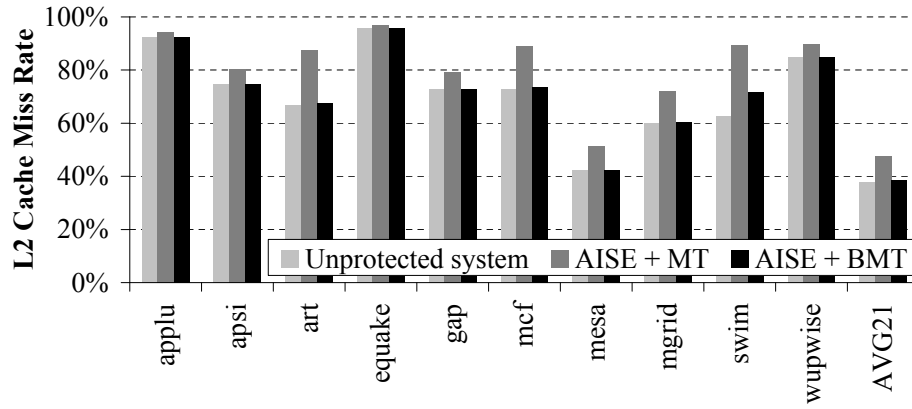


Tree nodes results in a sizable reduction in L2 cache miss rates and bus utilization and thus the significant reduction of performance overheads seen in Figure 2.10.

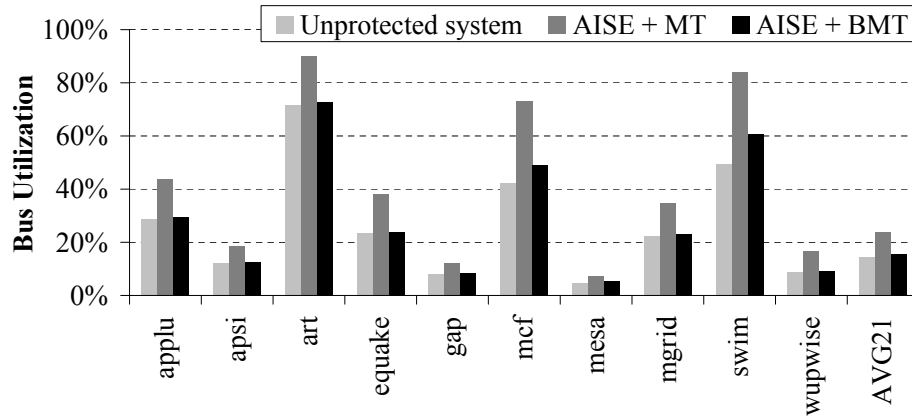
### 2.6.3 Multi-Core Processor Evaluation Results

In the first experiment on a Chip Multi-Processor system (CMP), we compare the overheads of our scheme (AISE+BMT) to the 64-bit global counter scheme with standard Merkle Tree protection (global64+MT). Figure 2.13 shows the percentage degradation in the combined IPC of both benchmarks that run on different cores, for global64+MT and AISE+BMT, relative to a base system with no protection. The combined IPC for the benchmark pair is calculated as the harmonic mean of the IPCs of the individual benchmarks when run together as separate threads on each core of our modeled CMP system. While the two schemes offer similar system level benefits, the performance overheads of our scheme are significantly lower. The average execution time overhead of global64+MT is 24.3%, while the average for AISE+BMT is 4.1%. Hence, as in the uniprocessor case, our scheme provides the best of both worlds in terms of supporting critical system features and low performance overheads.

Next, we compare the overheads of our AISE scheme in conjunction with a standard Merkle Tree (AISE+MT) to AISE with our Bonsai Merkle Tree scheme (AISE+BMT). Figure 2.14 shows the degradation in combined IPC for AISE+MT and AISE+BMT relative to a base system with no protection. As can be seen, AISE+BMT maintains a large



(a) L2 cache miss rate



(b) Bus utilization

Figure 2.12: L2 cache miss rate and bus utilization of an unprotected system, standard Merkle Tree, and our BMT scheme

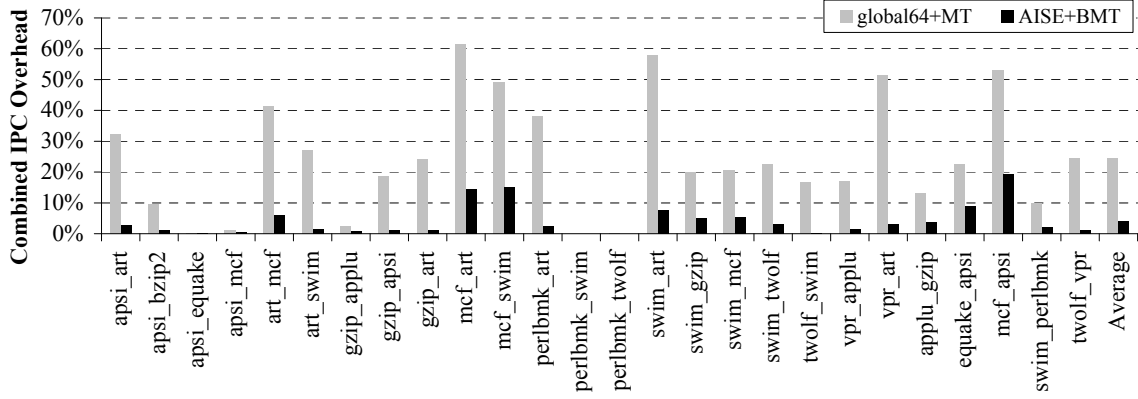


Figure 2.13: Performance degradation comparison of AISE with our Bonsai Merkle Tree vs. 64-bit global counter scheme with the Standard Merkle Tree

advantage over AISE+MT for CMPs, reducing the average IPC degradation from 15.1% to 4.1%. In addition, several benchmark pairs, such as *mcf\_art*, *swim\_art* and *mcf\_apsi*, suffer from large IPC degradations of 35% or more, while no benchmark pair suffers an IPC degradation of more than 20% with our AISE+BMT scheme, and only three benchmark pairs suffer more than 10% degradation.

We now present supporting statistics, similar to those shown in our evaluation for uniprocessor systems, to understand why BMT outperforms the standard Merkle tree scheme in CMP systems. Figure 2.15 shows the average portion of L2 cache space that is occupied by data blocks during execution. For the standard Merkle Tree scheme, on an average, data occupies 63% of the L2 cache space and the remaining 37% is occupied by

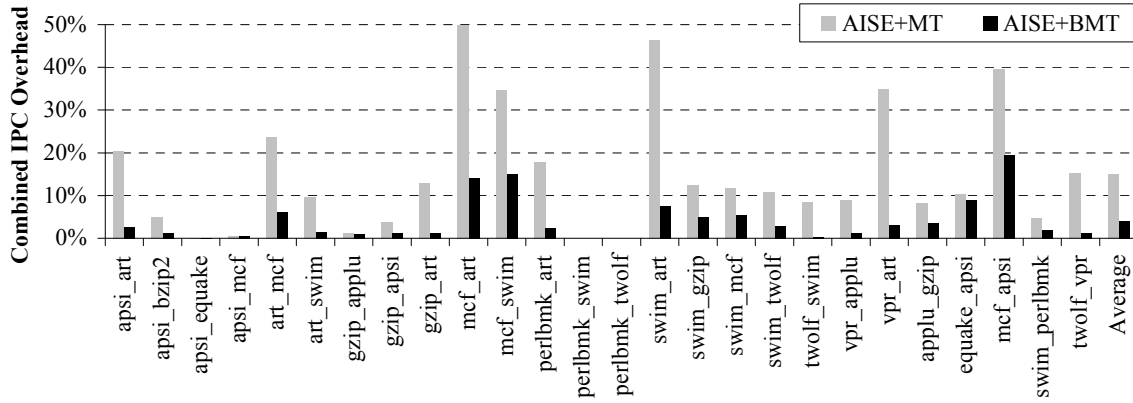


Figure 2.14: Performance degradation comparison of AISE with our Bonsai Merkle Tree vs. AISE with the Standard Merkle Tree

Merkle Tree nodes. On the other hand, for our BMT scheme data occupies 98.2% of the L2 cache space. This motivates the small overheads of AISE+BMT even in a CMP system.

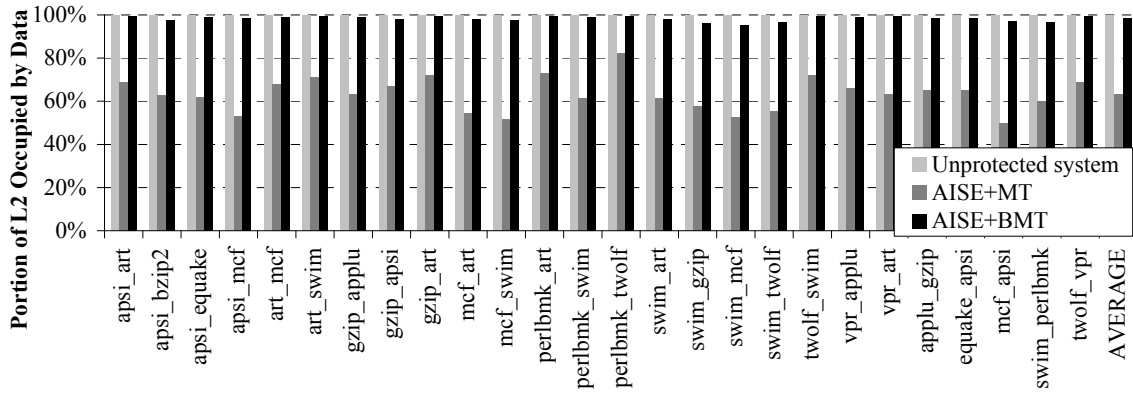


Figure 2.15: L2 cache pollution

Finally, Figure 2.16 shows the L2 cache miss rate (local) and the off-chip bus utilization of AISE+MT and AISE+BMT compared to a base system with no protection.

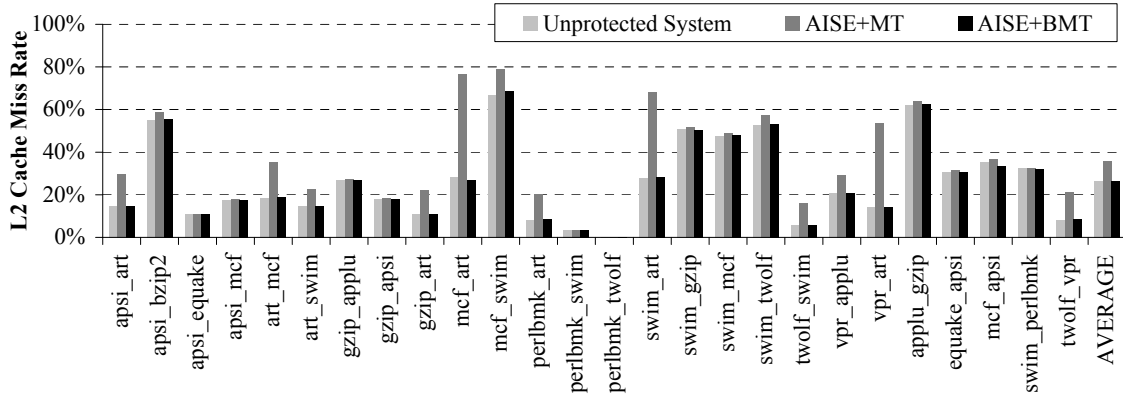
The figure shows that L2 miss rate and bus utilization increases significantly for AISE+MT (average L2 miss rate from 26% to 36% and bus utilization from 20% to 41%). On the other hand, for AISE+BMT, the L2 cache miss rate and bus utilization increase only slightly (average L2 miss rate virtually unchanged at 26.3% and bus utilization from 20% to 21.4%). These results are attributed primarily to the reduced L2 cache pollution from the Merkle Tree nodes and explain the significant reduction in performance degradation.

#### 2.6.4 Sensitivity Studies

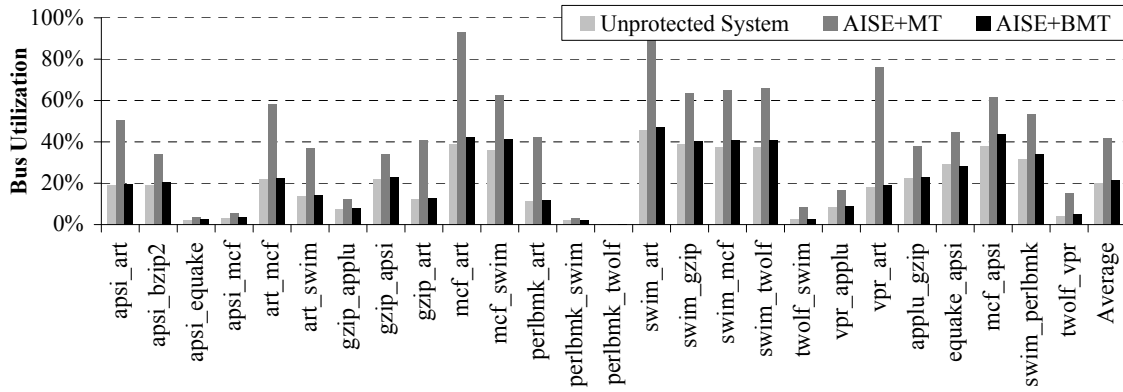
In this section, we present two sensitivity studies for the uniprocessor environment. In the first case study, we examine the sensitivity of the standard Merkle Tree (MT) and our BMT schemes to MAC size variations. In the second case study, we examine the sensitivity of the MT and BMT schemes to cache size variations.

##### Sensitivity to MAC Size

The level of security of memory integrity verification increases as the MAC size increases since collision rates decrease exponentially with every one-bit increase in the MAC size. Security consortia such as NIST, NESSIE, and CRYPTREC have started to recommend the use of longer MACs such as SHA-256 (256-bit) and SHA-384/512 (512 bits). However, it is possible that some uses of secure processors may not require a very high cryptographic strength, relieving some of the performance burden. Hence, Figure 2.17



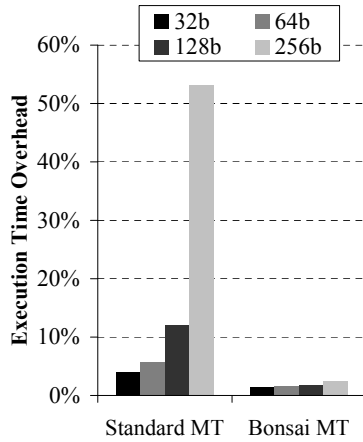
(a) L2 cache miss rate



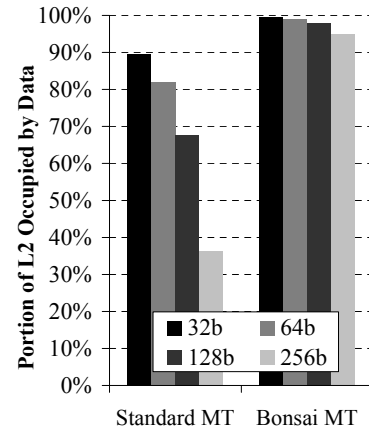
(b) Bus utilization

Figure 2.16: L2 cache miss rate and bus utilization of an unprotected system, standard Merkle Tree, and our BMT scheme

shows both the average execution time overhead and fraction of L2 cache space occupied by data across MAC sizes, ranging from 32 bits to 256 bits. The figure shows that as the MAC size increases, the execution time overhead for MT increases almost exponentially from 3.9% (32-bit) to 53.2% (256-bit). In contrast, for BMT, the overhead remains low, ranging from 1.4% (32-bit) to 2.4% (256-bit). The overheads are related to the amount of L2 cache available to data, which is reduced from 89.4% (32-bit) to 36.3% (256-bit) for MT, but is only reduced from 99.5% (32-bit) to 94.9% (256-bit) for our BMT. Overall, it is clear that while large MAC sizes cause severe performance degradation in standard Merkle Trees, they do not cause significant performance degradation for BMTs.



(a) Average Performance Overhead



(b) Average Cache Pollution

Figure 2.17: Performance overhead comparison across MAC size

### Sensitivity to Cache Size

Figure 2.18 shows the average performance overheads for L2 cache sizes of 512KB, 1MB and 2MB. The figure shows that as the cache size increases, the average overhead for both the schemes decreases, with standard Merkle Tree benefiting more than BMT. This is expected as in the standard Merkle tree scheme, the Merkle Tree nodes cause thrashing of data blocks and an increased cache size helps reduce this thrashing, whereas our BMT scheme has very little thrashing to begin with. Two other important observations are evident from this figure. First, BMT overheads are stable across cache sizes (from almost negligible for a cache size of 2MB to 2.4% for a cache size of 512KB). On the other hand, standard Merkle Tree overheads vary significantly (from 2.3% for a cache size of 2MB to 17.1% for a cache size of 512KB). Second, the overheads of the standard Merkle Tree with a cache size of 2MB (2.3%) are the same as BMT overheads with a much smaller cache size of 512KB (2.4%). To summarize, BMT offers performance stability across cache sizes, even in memory constrained environments. With the growth of number of cores on a chip in future CMPs, it is important for a security scheme to achieve small overheads even when the cache size per core is relatively small.

#### 2.6.5 Storage Overheads in Main Memory

An important metric to consider for practical implementation is the required total storage overhead in memory for implementing a memory encryption and integrity



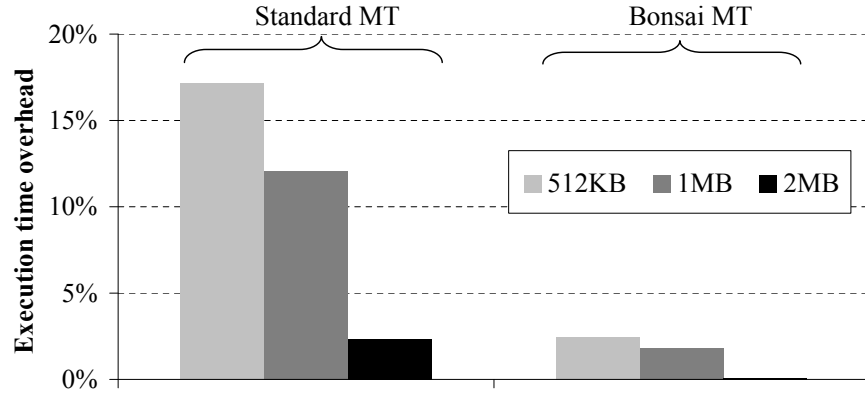


Figure 2.18: Sensitivity to cache size

verification scheme. For our approach, this includes the storage for counters, the page root directory, and MAC values (Merkle Tree nodes and per-block MACs). The percentage of total memory required to store each of these security components for the two schemes: global64+MT and AISE+BMT across MAC sizes varying from 32-bits to 256-bits is shown in Table 2.2.

Since each data block (64B) requires effectively 8-bits of counter storage (one 7 bit block counter plus 1 bit of the LPID), the ratio of counter to data storage is only 1:64 (1.6%) versus 1:8 (12.5%) if 64-bit global counters are used. This counter storage would occupy 1.23% of the main memory of the secure processor with 128-bit MACs. The page root directory is also small, occupying 0.31% of main memory with 128-bit MACs. The most significant storage overhead comes from Merkle Tree nodes, which grow as the MAC

Table 2.2: MAC &amp; Counter Memory Overheads

		MT	Page Root	Counters	Total
256b	global64+MT	49.83%	0.35%	5.54%	55.71%
MAC	AISE+BMT	33.50%	0.51%	1.02%	35.03%
128b	global64+MT	24.94%	0.26%	8.31%	33.51%
MAC	AISE+BMT	20.02%	0.31%	1.23%	21.55%
64b	global64+MT	12.48%	0.15%	9.71%	22.34%
MAC	AISE+BMT	11.11%	0.17%	1.36%	12.65%
32b	global64+MT	6.24%	0.08%	10.41%	16.73%
MAC	AISE+BMT	5.88%	0.09%	1.45%	7.42%

size increases. The traditional Merkle Tree suffers the most, with overhead as high as 25% of the main memory with 128-bit MACs and 50% for 256-bit MACs. The overhead for our BMT is both smaller and increases at a much slower rate as the MAC size increases (i.e. 20% overhead for 128-bit MACs and 33% for 256-bit MACs). The reason our BMT still has significant storage overheads is because of the per-block MACs. BMT nodes themselves require a very small storage. These overheads are still significant; however our scheme is compatible with several techniques proposed in [14] that can reduce this overhead, such as using a single MAC to cover not one block but several blocks. However, the key point here is that AISE+BMT is more storage-efficient than global64+MT irrespective of the MAC size used. AISE+BMT uses  $1.6\times$  less memory compared to global64+MT with 256-bit MACs with the gap widening to  $2.3\times$  with 32-bit MACs. Hence AISE+BMT maintains a distinct storage advantage over global64+MT across varying levels of security.

## 2.7 Conclusions

We have proposed and presented a new counter-mode encryption scheme which uses address-independent seeds (AISE), and a new Bonsai Merkle Tree integrity verification scheme (BMT). AISE is compatible with general computing systems that use virtual memory and inter-process communication, and it is free from other issues that hamper schemes associated with counter-based seeds. AISE can easily be extended to support the different variants of virtualization without requiring any changes to an AISE compliant OS and requiring minimal changes to existing VMMs. Despite the improved system-level support, with careful organization, AISE performs as efficiently as prior counter-mode encryption.

We have proposed a novel technique to extend Merkle Tree integrity protection to the swap memory on disk. We also found that the Merkle Tree does not need to cover the entire physical memory, but only the part of the memory that holds counter values. This discovery allows us to construct BMTs which take less space in the main memory, but more importantly much less space in the L2 cache, resulting in a significant reduction in the execution time overhead from 12.1% to 1.8% for single threaded SPEC 2000 benchmarks and from 15% to 4% on multi-programmed workloads, along with a reduction in storage overhead in memory from 33.5% to 21.5%.

## Chapter 3

# Data Protection for Distributed Shared Memory Multiprocessors

This chapter is organized as follows. Section 3.1 discusses background of secure processor designs as they relate to DSM multiprocessor systems. Section 3.2 discusses our architectural assumptions and assumed attack model. Section 3.3 describes our security analysis for the requirements of data protection in DSM systems. Section 3.4 overviews our proposed table-based mechanisms to protect data communicated between processors across the interconnect network in DSM systems. Section 3.5 details our DSM evaluation setup, and Section 3.6 presents evaluation results and insights for the table-based protection schemes. Section 3.7 discusses some of the drawbacks of the table-based DSM protection schemes, and motivates the need to address these drawbacks. Section 3.8 then overviews

the challenges to addressing the drawbacks, and discusses a new single-level DSM data protection scheme that alleviates them. Then Section 3.9 evaluates this single-level DSM data protection scheme, and compares the results to the table-based approach. Finally, Section 3.10 concludes the chapter on secure architectures for DSM multiprocessors.

### 3.1 Background

Constructing a secure multiprocessor system by piecing together secure processors alone does not give sufficient protection because communication between processors is not automatically protected. The main challenge is that communicating processors must share necessary encryption information, so that a data block encrypted by one processor can be decrypted by another processor. In a bus-based Symmetric Multi-Processor (SMP) system, the shared bus provides an ideal medium for sharing encryption information because all processors can observe the same bus. For example, a global bus counter can be used to encrypt data transfers between processors [39], or data transmitted on the bus itself can be used to encrypt new data blocks through Cipher Block Chaining [53]. Additionally, [8] proposes a technique to authenticate a shared bus. Finally, [28] proposes an approach which claims to provide data protection in a multiprocessor with an arbitrary interconnection network. However, due to its general nature, this scheme suffers from significant on-chip storage overhead requirements for security-specific structures, and some security vulnerabilities if attackers can drop messages in the system.

Because uniprocessor protection mechanisms only apply to processor-memory communication, researchers have proposed protection schemes for *processor-processor* communication in bus-based Symmetric Multi-Processor (SMP) systems [39, 53]. The fundamental assumption used for such protection is that each processor can observe every coherence transaction in the system provided naturally through snooping the shared bus. In

these schemes, each processor maintains a global encryption counter or global encryption pad used to protect processor-processor communication. On each bus transaction, each processor updates its counter and encrypts it to generate an encryption pad [39], or uses the snooped data to generate a new Cipher Block Chaining (CBC) encryption pad [53]. The pad is used for both encrypting and integrity checking processor-processor communication.

Unfortunately, neither uniprocessor nor SMP protection schemes can be extended directly to protect DSM systems. Extending direct encryption and integrity verification for processor-processor communication would incur large performance overheads due to the added latencies at the sender side for encrypting data and generating MACs, and at the receiver side for decrypting data and verifying the MACs. With a recent hardware implementation showing an AES latency of 37ns and MD5 or SHA-1 over 300ns [23], this approach is either too costly or not feasible.

Alternatively, one may envision an approach in which uniprocessor counter-mode encryption is directly extended in a straightforward manner to protect processor-processor communication by treating processor-to-processor data transfer similarly to a processor-to-memory writeback. However, this approach is problematic to support due to the need to keep the counters in both the sending and receiving processor coherent. For example, in response to an intervention to a dirty line, a processor flushes the line to the requester, and the flushed line would be encrypted by XORing it with a pad obtained by incrementing the current counter for the block. This increment would trigger invalidation of other cached

copies of the same counter. In order for the receiving processor to decrypt the flushed line, it needs to obtain the new counter value for the block. It would do so by sending a read request for the cache block that contains the counter, which eventually appears as an intervention to the sender processor. Hence, the latency for processor-processor communication is effectively doubled (obtain data, then its counter). Similar difficulties exist with maintaining coherency among nodes in the Merkle Tree if it were distributed across all processors.

It is also clear that SMP protection cannot be extended easily for protecting DSM systems. The requirement that each processor observes all coherence transactions would be costly to support in terms of ensuring a global ordering of all transactions as well as the large bandwidth requirement needed for broadcasting each transaction to all processors.

The schemes proposed in this dissertation differ from previous approaches in that we propose the first architectural solutions for data privacy and integrity in *DSM* multiprocessors that are also scalable and low in performance and storage overheads. We describe both our table-based DSM protection scheme and our single-level DSM data protection scheme in the following sections.



## 3.2 Attack Model and Assumptions

### 3.2.1 Architecture Assumptions

Our first assumption is that the DSM system uses a home-based directory protocol for maintaining cache coherence, and we specifically illustrate our mechanisms with the MESI protocol in mind. Figure 3.1 illustrates processor-processor communication in such a DSM system. Suppose a processor  $P_R$  sends a read request to the home node  $P_H$  which keeps the directory for the requested address (Step 1).  $P_H$  finds that the line is currently owned (in a modified/dirty state) by another processor  $P_O$ . So it sends an intervention request to  $P_O$  (Step 2), which will respond by downgrading its line to a clean shared state, flushing its line to update the copy at home (Step 3a), and replying with the data to the requester (Step 3b). The figure shows that some processor-processor coherence messages only carry cache coherence protocol information (command, address, message source and destination), which we will refer to as *non-data messages*, while others also carry data of the application program, which we will refer to as *data messages*. In a DSM system, data messages are either responses to non-data coherence messages, such as intervention or invalidation requests, or self-initiated writebacks to the home memory.

In addition, we assume a DSM system in which a node's directory controller is integrated in the processor chip, similar to the configuration used in the IBM Power4 system [20] and AMD Opteron [2]. In this case, any accesses to the home memory consist of

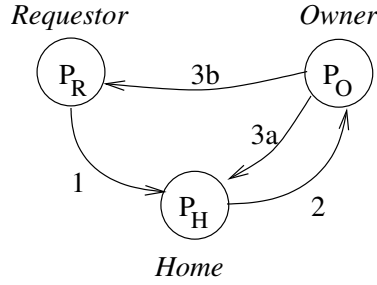


Figure 3.1: DSM processor-processor communication.

two steps: communication between the home node's directory controller and the requesting processor, and between the home node's directory controller and its local memory. For example, a reply to a remote read (to a clean line) first causes the line to be fetched from the home's local memory into the home processor chip, and then forwarded to other processor chips through an interconnection network. This assumption enables us to employ two separate protection mechanisms in our table-based DSM protection scheme: processor-memory communication protection can be handled using well-studied uniprocessor memory protection techniques such as those described Chapter 2, while processor-processor communication needs new protection mechanisms.

### 3.2.2 Security Assumptions and Attack Model

As mentioned earlier, our goal is to protect DSM systems against hardware attacks in environments such as on-demand computing. We assume that the system has relatively

strong physical security, but is not immune to attacks by a select few employees or other parties who have physical access to the machine. Since it is likely that only a few people have physical access to the machine, any attacks that leave *traces* may easily provide sufficient information that can lead to the attacker. We define a trace as a detectable anomaly of the system behavior. Hence, the fundamental assumption is that *the goal of an attacker is to perform traceless attacks* in order to steal sensitive data that belongs to the application.

We broadly categorize hardware attacks into three categories. The first category is *sabotage* attacks in which the attacker’s goal is to crash the application or even damage the system. Our scheme does not seek to protect against sabotage attacks, including application or system crashes, since it is extremely difficult to protect the system against such sabotage when the attacker has physical access to the machine. On the other hand, the attacker lacks the incentive to do so because the attack can be easily traced back to him/her, and there is probably little financial reward for sabotage attacks.

Another category is *passive* attacks in which the attacker’s goal is to eavesdrop on processor-processor or processor-memory communication along major off-chip structures such as the local memory bus, DRAM, and the interconnection network, as illustrated in Figure 3.2. An example of this attack is the physical insertion of a snooping device onto the exposed interconnect at the back of server racks. A small USB drive-sized device with multi-GB storage can likely be attached and removed in a matter of seconds without

shutting down the system if the system can recover from temporary link failures. Cable clutter may also hide the device from cursory visual inspections.

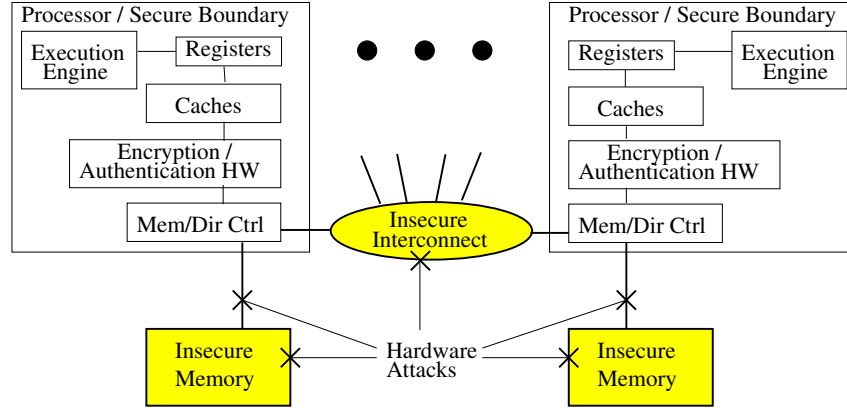


Figure 3.2: Attack model, secure boundary, and location of our encryption and integrity verification HW.

Finally, in *active* attacks, the goal of the attacker is to steal sensitive information by modifying coherence messages communicated between processors, or data in a node's local memory or on the memory bus. Although active attacks are certainly more difficult to perform than passive attacks, we cannot rule out the possibility of an attacker attempting them, especially if passive attacks are no longer fruitful due to the system encrypting all off-chip communication, and if the attack does not result in any traces. A coherence message typically contains message type, memory block address, routing information (source and destination processors), and, for data messages, user data. We do not make any assumptions as to the specific abilities of attackers to modify signals, so we assume the worst case in which the attacker is able to modify any parts of the message. We distinguish be-

tween attacks that modify application data as *data spoofing* versus ones that modify other information as *non-data spoofing*. The attacker may also be able to *replay* an old message. Finally, the attacker may also modify the coherence protocol directory information stored at each node.

We also assume that *precise authentication*, which refers to delaying load instruction retirement (or even load data use) until data authentication is completed [39, 50], is not needed. The rationale for this assumption is that a hardware attack takes time to perform, so detection within a reasonable time window (say, thousands to millions of cycles) serves as a sufficiently powerful deterrent to attackers. In our design, detection of an attack occurs much more quickly than this (at least within the latency of a message round trip, which is on the order of hundreds to thousands of cycles).

A secure DSM system with memory encryption support must have mechanisms in place to provide secure booting of the machine and secure key setup. This is an important issue that is beyond the scope of this paper. In this paper, we simply assume that there is a mechanism in place to guarantee secure booting and key set up on the machine.

### 3.3 Security Analysis for DSM Protection

This section analyzes the requirements for secure data protection in DSM systems. We note that the security requirement for protecting against passive attacks on data confidentiality is straightforward: application data in data messages must be securely encrypted during communication. Thus we focus our discussion on the requirements for protection against active attacks. At first glance, it may seem that this requirement is also obvious: simply verify the integrity of all parts of *every* coherence message through the use of a MAC. However, we note that this requirement would introduce excessive performance overhead, and it is likely overly strong for DSM systems and can be relaxed.

Specifically, we can derive a new security requirement for integrity verification based on the assumption that the goal of an attacker is to perform traceless attacks (Section 3.2.2). This assumption implies that the protection requirement can be met by two components: (1) Ensuring that every attempt at an active attack results in a *trace* of the attack, and (2) Traces prompt analysis and preventative/corrective actions to block the active attack attempts from succeeding. We define a trace of an attack as a detectable anomaly of system behavior. More specifically, in the context of DSMs, we can define a trace as *incorrect/anomalous coherence protocol behavior* or *cryptographic errors* such as a failed message integrity check. Interestingly, many active attack attempts will naturally result in detectable coherence protocol anomalies. We assume that mechanisms are in place to detect such anomalies, and the discussion of these mechanisms is outside the scope of

this dissertation. Hence, we only need to ensure that attacks that do not result in coherence protocol anomalies can still be detected as cryptographic errors.

We first analyze this requirement for non-data messages. These messages contain coherence message type, address, and routing information (source and destination processor IDs). If a message's type is modified by an attacker, a protocol anomaly will result because the receiving processor can check whether the message is allowed based on the stored coherence state for that address. Examples of anomalies would be receiving an invalidation acknowledgment without a matching invalidation request, or not receiving an acknowledgment or negative acknowledgment as expected, etc. Similarly, if a message's address or source/destination processor IDs are changed, a protocol anomaly will likely result. There may be special cases in which no anomaly results because the cache state of a line allows such an operation. However, we note that the attacker in this case can only force a data write back, force data in the cache to be sent across the interconnect (and data secrecy is not compromised as long as data messages are protected), or cause denial of service (which we do not seek to protect against because its success is itself an easily detected trace). An attacker may also attempt to delete messages rather than tamper with them. This can achieve an effect similar to a replay attack because the attacker can, by not delivering invalidation messages, force a processor to use stale data from its cache. In this case, we assume that the protocol logs an anomaly if, after a certain period of time, it has not received a response to its invalidation or intervention message. Finally, some protocol

anomalies can be expected to occur naturally due to protocol races. For example, an invalidation may be received while a line is being written back to the home, so an invalidation message to a line that is not in the cache may not signal an active attack attempt. In this case, we rely on the protocol to detect unusual patterns such as an excessive number of anomaly cases.

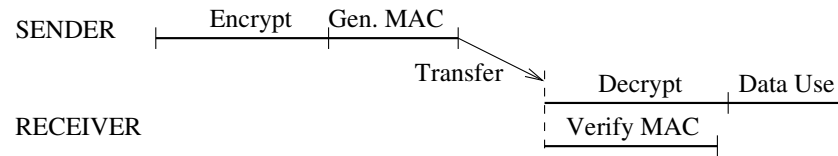
Now we consider data messages which carry sensitive data that belongs to the application program. Since application data is not used directly in the coherence protocol, it may be modified by an attacker without raising any protocol anomalies. Therefore, application data needs to be protected by authentication codes so that tampering with this data is detected. Finally, attacks such as routing application data to another processor, or faking the address of the data may achieve a similar effect to data tampering (i.e. causing a processor to use the wrong data value). Therefore, for data messages, we also need to authenticate all parts of the message including the message header information. In addition, an attacker may attempt to *replay* an old message together with its authentication code, so a mechanism must be used to detect such replays.



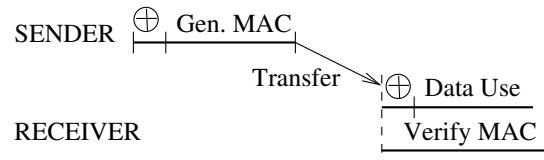
### 3.4 Table-Based Processor-Processor DSM Data Protection

As we discussed in Section 3.3, our goal is to provide protection against hardware attacks on data messages in DSM systems. To accomplish this goal it is necessary to have mechanisms to encrypt and verify the integrity of data during processor-processor data transfers across the interconnection network. Section 3.1 has discussed that direct encryption and integrity verification of data messages incurs a high overhead, while direct extension of uniprocessor or SMP protection is either costly or infeasible. To make DSM protection practical, our mechanism should significantly reduce or hide encryption, decryption, and MAC generation/verification latencies, while meeting the security requirements discussed in Section 3.3.

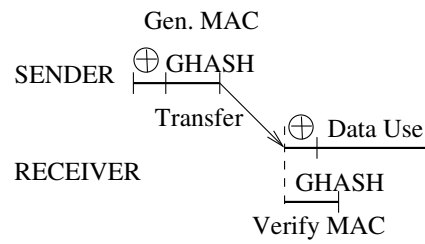
We achieve this goal through *counter-mode encryption* and *GCM integrity verification*. Let us discuss encryption issues first. First, it is well known that if two communicating processors agree on a common stream of counter values used for encryption, they can pre-generate encryption pads before data is ready to be sent or received. To encrypt or decrypt data, it only needs to be XORed with the pre-generated pad. Figure 3.3(b) shows the new reduced encryption/decryption latencies compared to a direct encryption approach (Figure 3.3(a)). However, the latency of integrity verification is still not hidden. To hide integrity verification latency, we use a combined counter-mode encryption/authentication scheme, Galois Counter Mode (GCM) [34, 51]. GCM offers many benefits. First, its security strength has been thoroughly studied and proven to be as strong as the underly-



(a) Direct encryption and ciphertext-based MAC.



(b) Pre-generated pads for encryption.



(c) Pre-generated pads for encryption and integrity verification.

Figure 3.3: Proc-proc communication latencies using various encryption and authentication mechanisms.

ing AES encryption algorithm [34, 51]. Second, GCM utilizes the existing AES hardware already used for encryption. Finally, since GCM relies on the use of counters, if a counter value is known, the authentication pad can be pre-generated, hiding most of the integrity verification latency. The only exposed part of the latency is GHASH computation, which is a short chain of Galois Field Multiplications and XORs, each of which can be performed in one cycle [51]. Figure 3.3(c) illustrates this reduced integrity verification delay and compares it to direct MAC generation (Figures 3.3(a) and 3.3(b)).

Figure 3.4 shows our processor-processor encryption mechanism in the upper part and integrity verification in the lower part. The encryption pad is obtained through AES encryption of the *encryption seed*. To ensure that pads are not reused, which is the security requirement of counter-mode schemes, the seed must be unique for each message in the system. Thus, we choose the concatenation of the communication counter ( $Ctr$ ), processor ID of the sender ( $ID(S)$ ), processor ID of the receiver ( $ID(R)$ ), and an arbitrary Encryption Initialization Vector ( $EIV$ ). Once a data transfer needs to be made for a certain cache line, the plaintext of the data  $Ptext$  is XORed with the pre-generated pad to produce the ciphertext  $Ctext$ . The seed selection is such that a pad is always unique for any communicating processor pair, and even for a processor pair it is unique if the role of the sender and receiver are switched. The counter is incremented after each message is sent or received; hence the pad is also unique across messages communicated by a processor pair. Therefore, each pad is globally unique across all messages, so the attacker cannot

discover application data through passive/eavesdropping attacks. Finally, note that the seed we choose still allows pads to be pre-generated because the seed input is independent of application data.

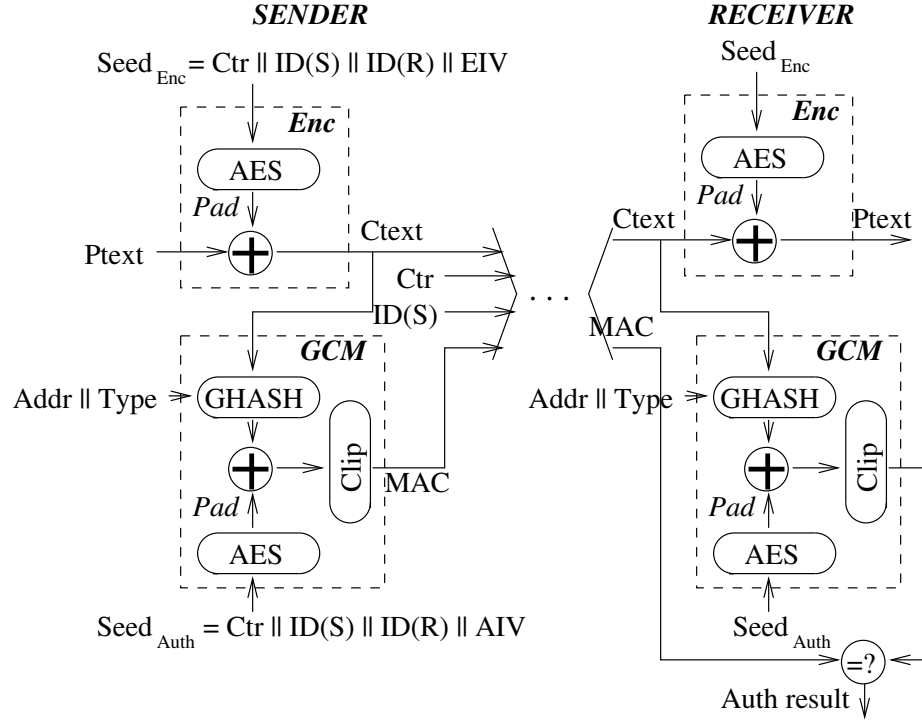


Figure 3.4: Our mechanism for processor-processor secure data transfer in DSM systems.

For integrity verification, recall that the security requirement requires us to check the integrity of all parts of a *data* message. Therefore, the MAC must encompass the counter, processor IDs of the sender and receiver, data address, and type of the coherence message. This ensures that tampering with any part of a data message will be detected as a cryptographic error (Section 3.3). The authentication seed for GCM is chosen as the

concatenation of  $Ctr$ ,  $ID(S)$ ,  $ID(R)$ , and an arbitrary Authentication Initialization Vector ( $AIV$ ). Such a seed ensures that each authentication pad is unique across all messages in the system, and is different from encryption pads. We note that in GCM, *additionally authenticated data* (i.e. data that is authenticated but not encrypted) can be supplied along with the data ciphertext into the GHASH function to generate the MAC. We leverage this feature in order to protect the message header information such as address and message type. The counter along with sender and receiver processor IDs are accounted for in the MAC since they are part of the seed used to pre-generate the authentication pad. When encryption has produced the ciphertext of a cache line to be transferred, the ciphertext and additionally authenticated data are input into the GHASH function. The output of the GHASH function is XORed with the authentication pad, and the result may be clipped to the desired MAC size [34]. Therefore, the GHASH and XOR latencies are the only part of the MAC generation latency that cannot be hidden. Fortunately, with GCM the GHASH latency is only a few cycles [51].

Finally, the coherence message sent must now contain the ciphertext of the application data and the MAC. Additional fields need to be sent also, including the counter and the processor ID of the sender. The counter is needed because the counters kept by the sender and by the receiver may occasionally be out of sync. The counter sent along with the message allows the receiver to verify that its pre-generated pad corresponds to

the correct counter value. The need for sending the processor ID of the sender along with the message will be described in the following subsections.

In order for this mechanism to operate efficiently, it is clear that the key issue is how to manage the communication counters and their pre-generated pads so that the counters are synchronized at the sender and receiver side most of the time. If counters and pads are not available, or the sender and receiver get out of sync, then we will suffer the full pad generation latency before data can be encrypted or decrypted and its integrity verified. Thus, a counter management scheme should ideally incur a low *performance overhead*, low *storage overhead*, and at the same time have *no scalability restrictions* which prevent the DSM from scaling to a large number of processors. We propose three table-based counter management techniques described in the following subsections. They differ in how they meet the criteria outlined above.

### 3.4.1 Private Counter Stream Tables(*Private*)

The first scheme is a straightforward approach in which a processor uses a separate counter stream for sending data to each of the other processors. In this scheme, each processor pair maps one-to-one to a counter for communication in each direction. If  $P$  denotes the total number of processors in the system, each processor contains a *send table* with  $P - 1$  entries for encrypting and sending data messages, and a *receive table* with  $P - 1$  entries for receiving and decrypting data messages. Separate send and receive tables are

necessary because when the two processors switch roles as sender and receiver, the order of concatenation of their processor IDs is different, and thus the pre-generated pads are different. Between a specific sender and a receiver, counters are kept synchronized by having each processor increment the corresponding counter each time that counter's pad is used to encrypt or decrypt a data transfer. Each send or receive table entry contains a 64-bit *counter value*, a 512-bit *pre-generated encryption pad* (assuming 64-byte cache block size), a 128-bit *pre-generated authentication pad* (assuming 128-bit MACs), and a *valid bit* indicating that the pad has been pre-generated but has not been *consumed* (used for encryption/decryption and integrity verification). The total storage per entry is  $1 + 64 + 512 + 128 = 705$  bits. The total table overhead is somewhat small except for very large DSMs. For example, for a 64-processor DSM, the table overhead is  $(2 \times 64 \times 705) / 8 = 11,280$  bytes per processor.

To send a data message, a processor first looks up its send table to find the entry that corresponds to the receiving processor. If it finds the entry with the valid bit set (i.e., a *pad hit*), the pads can be immediately used for encrypting and generating a MAC of the data. After the pads are consumed, the counter value is incremented, the valid bit is cleared, and new pads based on the new counter value are generated. If the sender finds the entry with the valid bit cleared (i.e., a *pad half-miss*), it waits until the pads that are currently being generated become available. Note that a pad half-miss is relatively rare: it

only occurs when two data requests to/from the same processor are separated in time by less than the latency of the AES unit.

Upon receiving the message, the receiver locates the sender's entry in its receive table, and decrypts and verifies the integrity of the message immediately if it has valid pads and the counter value of the entry matches the counter value in the message. Otherwise, it waits until it finishes computing the correct pads. If messages are delivered in-order through the interconnection, then decryption and integrity verification latencies will always be fully or partially hidden. However, if messages from a single sender can arrive out-of-order, performance of the receiver could be degraded because the receiver must generate a pad that corresponds to the counter value in the message. Since, in general, out-of-order message delivery may be rare, the extra pad generation latency also occurs rarely, so we choose to tolerate it. An alternative solution would keep a few counter values per processor and their pre-generated pads in the receive table, at the expense of having a larger receive table.

### 3.4.2 Shared Counter Stream Tables(*Shared*)

The *Private* scheme is simple and has almost perfect pad hit rates, but its counter storage overhead can be non-trivial for very large DSMs (e.g. 180KB for 1024-processor DSM). The second table organization scheme, which we refer to as *Shared*, seeks to reduce the storage overhead by a factor of two. To achieve this, we replace the send table of a



processor with a single counter and pad for sending data messages to any processor. This shared counter is incremented after each sent message, so pad uniqueness is still guaranteed. In order to pre-generate a sender's pad that is usable for sending a message to any receiver, the seeds used for encryption and authentication pad computation do not include the receiver processor ID (i.e.,  $ID(R)$ ). Pads are still unique because a processor updates its sending counter and pads after each sent message. To meet the security requirement for integrity verification,  $ID(R)$  is now concatenated with the block address and message type as input to the GHASH function.

As in *Private*, upon receiving a data message, the receiving processor accesses the entry corresponding to the sending processor and checks whether a pre-generated pad is available for that sender. However, since the sending processor uses the same counter to send messages to all processors, it is less likely for a receiving processor to see back-to-back messages with contiguous counter values. Non-contiguous counters occur when a processor receives a message from a particular sender, while the sender's previous message was sent to different processor. As a result, the receiver will suffer from a higher pad miss rate and more frequent full decryption and MAC generation latencies. The ability of the receive tables to pre-generate the correct pad more often could possibly be enhanced through prediction of sharing patterns.

### 3.4.3 Cached Counter Stream Tables(*Cached*)

Recall that *Private* should achieve good performance due to a low pad miss rate, but requires larger storage overheads, while *Shared* sacrifices performance for lower storage overheads. However, both *Private* and *Shared* are not scalable in the sense that the tables are designed to support only a fixed number of processors. Unless the tables are very large, they prevent DSMs from scaling to larger numbers of processors. In this section we introduce the *Cached* table configuration to address the scalability drawback of previous schemes.

Our *Cached* scheme can scale to an arbitrary number of processors with fixed send and receive table sizes, while still providing good performance. The intuition behind its design is that processors in a DSM system may often communicate with a set of neighbors that is much smaller than the total number of processors in the system. Therefore, we can limit the size of each processor’s send and receive table to some number of entries that is a fraction of the number of processors in the system. This table can operate similarly to a cache, where a send/receive to/from a processor that does not have an entry in the table will create a new entry for this processor in the table, replacing the entry that has been unused for the longest time. However, unlike a regular cache, displaced entries are simply discarded, instead of written back to other storage. By simply discarding displaced entries, we avoid the need to allocate off-chip storage for them, which would need to be protected against attacks with additional security mechanisms.

This *Cached* scheme raises the question of which counter value should be used to generate the encryption/decryption pad if no table entry is found. For receiving, a straightforward solution is to use the counter that is sent with the data. For sending, we must select a counter that has not been used before in order to prevent pad reuse. To achieve this, we keep track of the *maximum* counter value that has been used by a given sender to generate a pad across all receivers ( $maxCtr$ ). Furthermore, as an optimization, we always keep a pre-generated pad for a counter  $maxCtr + 1$ . If the sender finds a table entry corresponding to a receiver, it simply uses the pads in that entry. If it does not find an entry, it creates a new entry by replacing the LRU entry in the send table, then immediately uses  $maxCtr + 1$  and its pre-generated pads to encrypt and generate the MAC of the message, increments  $maxCtr$ , and generates its next pads. This optimization avoids pad-miss stalls at the sender in most cases. Compared to *Shared*, *Cached* will tend to have more consecutive counter values at the receiver, so it will likely have fewer pad misses and better performance than *Shared*. However, it is still expected to perform more poorly than *Private*.

Finally, in order for the  $maxCtr + 1$ 's pads to be usable for any receiver, the encryption/authentication seed used for the  $maxCtr + 1$ 's pad computation does not include the receiver processor ID ( $ID(R)$ ). However, pads stored in the table still include the receiver processor ID in the seed. For the receiver to handle this correctly, each message

is augmented with a bit to tell the receiving processor which seed type to use for pad generation.

### 3.4.4 Detecting Replay Attacks with Table-based Protection

Section 3.3 states that data messages need to be fully integrity checked, and a mechanism is needed to detect replay attacks. While we have satisfied the former requirement (Figure 3.4), this section presents the mechanism to deal with the latter. Note that due to the full protection of data messages, the attacker cannot modify any part of a message without causing failed integrity verification. Therefore, the only replay attack an attacker can do is to send an old copy of a message together with its valid but old MAC.

One way to detect such replay attempts is to note that a counter's value is monotonically increasing. A replayed old message is detected when a received message's counter is smaller than the current counter stored in the receive table. An alarm can be raised when this situation is detected. However, we note that out-of-order message delivery can also cause out-of-order counter values. Thus, false positives due to out-of-order delivery of messages could occur.

We may like to distinguish between a replay attack and out-of-order message delivery more definitively. To achieve that, first we observe that data messages are either responses to data request/intervention/invalidation messages, or a natural write back of modified lines. In the former case, we can send an authenticated counter value along

with the request/intervention/invalidation message. We will refer to this counter value as the *originator counter*. The receiving processor then sends the data message reply augmented with the authenticated originator counter value. The originator of the request/intervention/invalidation message (the receiver of the data message) keeps a list of outstanding transactions and their corresponding originator counter values. If the received data message has an originator counter not matching any in its outstanding transactions, it has detected a real replay attack. For the latter case of natural write backs, the sender of the written-back cache line can keep track of outstanding write back transactions and their originator counter values. Upon receiving the written back cache line, the home node is now required to send a fully authenticated write-back acknowledgment that contains the originator counter from the write back message. Upon receiving the acknowledgment, the sender detects a replay attack if the originator counter in the acknowledgment message does not match any one from its outstanding write back transactions. This protection works because, even though an attacker knows the value of the originator counter, he could not have produced a reply containing the encrypted and authenticated originator counter unless the attacker has the encryption/integrity verification key. So the message reply received by the originator must have been produced by a legitimate sender. In addition, the reply message is accepted only once by the originator because it keeps a list of outstanding transactions, and a transaction is removed from the list if its legitimate reply has been received.

With the ability to detect replay attacks, combined with full protection of data messages, we have satisfied all the encryption and integrity verification security requirements discussed in Section 3.3.

### 3.4.5 Implementation Issues

The size of the counters for encrypting processor-processor messages can be chosen to be large enough so that they will not overflow for the expected life of the system, but not too large to cause an excessive increase in bandwidth. In this work, we use 64-bit counters to avoid counter overflows for many years, and all of our evaluation results take into account the extra bandwidth necessary to transmit these counters with the encrypted data and MAC. 64-bit counters do not present a storage problem in our schemes because we only deal with processor-processor communication, so our counters are only needed in the send and receive tables.

Finally, we apply uniprocessor counter-mode encryption and Merkle Tree integrity verification to each node's main memory, including data and directory information, using an efficient GCM-based scheme as in [50]. To avoid coherence problems for uniprocessor counters and Merkle Trees, each node only protects its own local memory. A reply to a remote request is protected with uniprocessor processor-memory protection when it is brought from the main memory to the local node's processor, and with our processor-processor protection when it is sent from the local node to the remote requester.

### 3.5 DSM Experimental Setup

In order to evaluate our approaches for data protection in DSM systems, we added DSM support and implemented our proposed mechanisms again using SESC [22]. Table 3.1 shows the relevant architectural features of our simulated system. Some important features of our simulated DSM system to note are the use of the MESI coherence protocol with a hypercube network and fixed-path routing protocol. Also note that the on-chip counter cache is used to store counters for use in processor-memory data encryption, not for storing counters for our schemes. In addition, we model an 80 cycle latency for the AES engine, which is similar to the 37ns implementation described in [23] on a 2 GHz processor.

We assume round-robin page allocation. More optimized systems may employ a first touch policy which minimizes accesses to remote memory. Note that round-robin allocation stresses our schemes more compared to an unprotected system because of the high number of remote memory accesses, and each remote memory access results in processor-processor communication. Our simulations also take into account the extra bandwidth usage due to sending counters, MACs, and other information along with data messages in our DSM system data protection schemes.

We use all 12 SPLASH-2 benchmark suite [49] applications to evaluate our DSM protection techniques. The relevant application parameters are shown in Table 3.2, along with the global L2 cache miss rate (number of L2 misses divided by all memory reference instructions) and local L2 miss rate (number of L2 misses divided by L2 accesses) of each

Table 3.1: Architectural Parameters.

Processor	16 Processors, 2 GHz, 3-way out-of-order issue
Memory	L1-Inst: 16KB, 2-way, 64B line, WT, 2 cycles L1-Data: 16KB, 2-way, 64B line, WT, 2 cycles L2-Unif:256KB, 8way, 64B line, WB, 10 cycles Round robin page allocation, 4KB pages Memory bus: 1 GHz, 4-Byte wide, split-trans. RT memory latency: 200 cycles (uncont.)
Proc-Mem Encryption	Counter-mode, 8-bit split counters [50] SN Cache: 32KB, 4-way, 64B line, WB GCM-based Merkle Tree authentication
Proc-Proc Encryption	Counter-mode encryption, 64-bit counters Send and receive table size depend on scheme. Authenticated-Encryption [34]
Encryption HW	1 AES unit for all encr./decr. and auth. 16 stage AES pipe: 80 cycle lat., 5 cycle occp. [23]
Network	Hypercube, fixed-path routing Link BW 6GB/s, 50ns hop delay (as in [42])
Coherence Prot.	MESI, full bit vector, home-based directory, reply forwarding

application. Applications are simulated from start to completion without fast forwarding or sampling.



Table 3.2: Applications used in our evaluation. The L2 global miss rate is the number of L2 misses divided by the number of L1 cache accesses. The local L2 miss rate is the number of L2 misses divided by L2 accesses.

Application	Input	Global L2 Miss Rate	Local L2 Miss Rate
Barnes	16K particles	0.05%	3.3%
Cholesky	tk25.O	0.24%	45.9%
FFT	1M points	0.83%	63.0%
FMM	8K particles	0.04%	18.9%
LU	512x512 matrix, 16x16 blocks	0.05%	24.4%
Ocean	258 x 258 ocean	1.05%	27.4%
Radiosity	-room	0.07%	28.4%
Radix	4M keys, radix 1024	0.65%	22.3%
Raytrace	car	0.39%	22.3%
Volrend	head	0.26%	34.6%
Water-n2	512 molecules	0.04%	6.4%
Water-sp	512 molecules	0.03%	12.1%

### 3.6 Evaluation of Table-Based DSM Protection

In this section we use several different DSM configurations to evaluate the data protection and counter management schemes proposed in this paper. The *P2M Only* configuration refers to protection for data in main memory and for processor-memory communication within each DSM node, using previously described uniprocessor counter-mode memory encryption and Merkle Tree integrity verification. Note that this configuration has no protection for data communicated between nodes. The *Direct* configuration uses the AES block cipher and SHA-1 MAC generation algorithm to encrypt and verify the integrity of processor-processor data communication. In order to not penalize the *Direct* configuration too greatly, an 80 cycle latency is assumed for both AES and SHA-1, which is optimistic compared to over 300ns in a recent hardware implementation of SHA-1 [23]. Lastly, *Private*, *Shared*, and *CachedX*, where  $X$  denotes the number of send and receive table entries, represent our table-based DSM protection schemes described in Section 3.4 for processor-processor data protection. All schemes also include the mechanisms of *P2M Only* for protecting processor-memory data communications. All figures showing execution time overheads are relative to a baseline DSM system with no cryptographic support for data protection.

### 3.6.1 Table-Based DSM Data Protection Overhead

In order to evaluate the performance of our processor-processor communication data privacy and integrity protection, Figure 3.5 compares the execution time overheads of the *P2M Only* and *Direct* configurations with our approaches, *Private*, *Shared*, and *Cached4* on a 16-processor DSM system. With these configurations, the total send and receive table size of *Shared* and *Cached4* are a half and a quarter of that of *Private*, respectively. This figure shows that adding just processor-memory data protection (*P2M Only*) results in a very modest overhead of less than 4% for most benchmarks and on average. However, the straightforward approach to processor-processor data protection (*Direct*) results in an average slowdown of more than 10% across all applications, and more than 15% in *cholesky*, *ocean*, *radiosity*, and *volrend*, even with a very optimistic assumption on the MAC generation latency. This execution time overhead is not likely to be tolerable for real-world DSM systems, motivating the need for more efficient techniques.

As shown in the figure, each of our proposed techniques is able to reduce the overhead of processor-processor data protection significantly over *Direct*, with *Private* being the best in terms of execution time overhead at only 3.5%, and with *Shared* and *Cached4* performing just slightly worse despite much smaller storage overheads. The figure also shows that *Cached4* gives better performance than *Shared* in most applications, although it has only half the table storage overhead of *Shared*. Finally, we note that the majority

of the overhead in most applications comes from processor-memory protection rather than from our processor-processor protection.

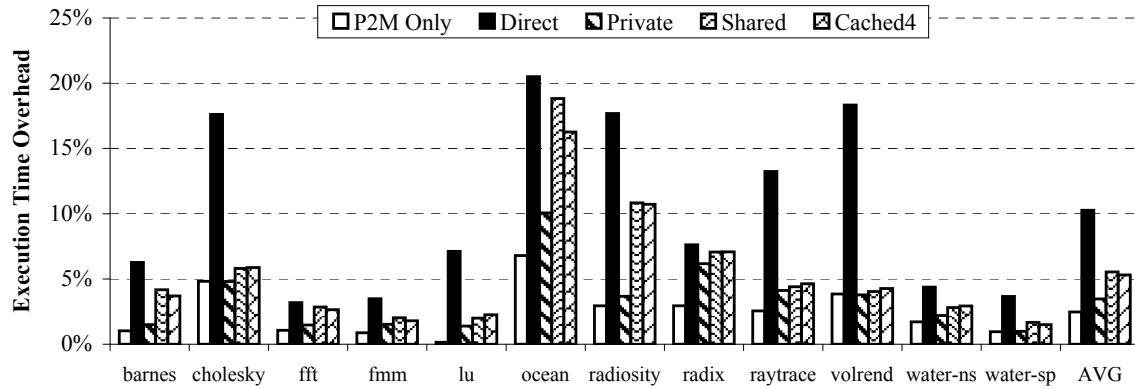


Figure 3.5: Execution time overheads for processor-memory data protection only, versus schemes with additional processor-processor data protection on 16 processors.

In order to gain more insight into the performance of our techniques from the previous figure, we introduce the terms *pad hit*, *pad half-miss*, and *pad miss*. A pad hit refers to the case in which the correct encryption and authentication pads have been fully pre-generated and only an XOR is needed to encrypt, decrypt, or verify the integrity of the data. A pad half-miss refers to the case in which one or both of the correct encryption and authentication pads are in the process of being generated, so only part of the pad generation latency is hidden. Lastly, a pad miss refers to the case in which the correct pads are not currently in the table or in the AES pipeline, so they must be generated directly and none of the pad generation latency is hidden.

Figure 3.6 shows the average (across all benchmarks) pad hit, pad half-miss, and

pad miss rates. In *Private*, the pad miss rate is nearly zero, so this scheme has the lowest execution time overheads. This is expected, because a dedicated counter and its pre-generated set of pads are kept for each processor pair in *Private*. In *Shared*, we reduce the storage overhead relative to *Private* by using one counter and pad in place of the send table. However, this results in an increased pad half-miss rate for sending data, and a relatively large miss rate in the processors' receive tables for receiving and decrypting data. This is due to the fact that a processor's decryption pads are only kept synchronized with a particular sender's encryption pad if consecutive messages from a particular sending processor go to the same receiving processor. Finally, we see that the table performance of *Cached4* is between *Private* and *Shared*. The pad half-miss rate for sending data is almost as low as in the *Private* scheme, and the pad miss rate for receiving data is significantly lower than that observed for *Shared*.

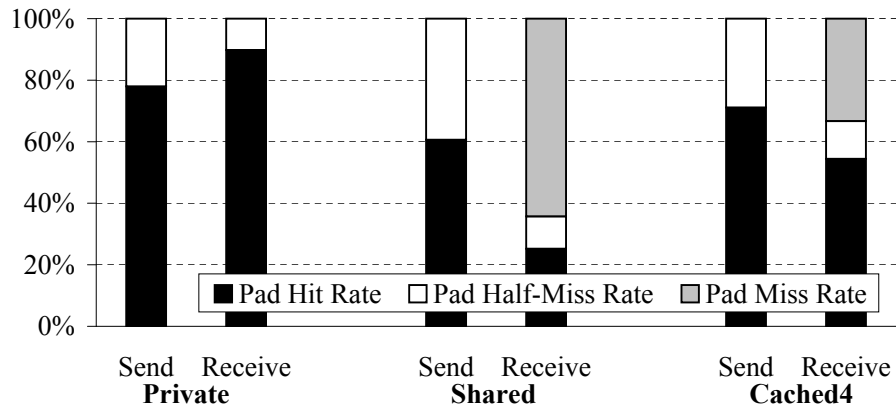


Figure 3.6: Send and receive table performance for our schemes on 16 processors.

### 3.6.2 Scalability of the *Cached* Scheme

Since the motivation for our *Cached* scheme was to have an approach with a small storage overhead that could scale to work well on an arbitrary number of processors, we now evaluate its effectiveness in achieving these goals. Figure 3.7 shows the execution time overhead of *Cached8* as the system size varies from 16 to 32 to 64 processors. As before, note that the overhead shown for each system size is relative to a DSM system of the same size, but without any data protection mechanisms. This figure shows that for several benchmarks, and on the average, the execution time overhead actually decreases as the number of processors increases, but in some cases the reverse is true. This can be explained by two opposing performance factors we observe as the system size increases. First, the size of the interconnection network increases as the number of processors increases, and therefore data sent from one processor to another travels more link hops on average to reach its destination. This amortizes the encryption/decryption and integrity verification latency for sending and receiving data over a greater total network delay, and reduces the execution time overhead relative to the baseline system. The second factor is the pad miss rate at the processors' receive tables. As the system size increases, the 8 table entries cover a smaller percentage of the total number of processors. Therefore we observe increasing pad miss rates for our applications as the DSM size grows. Overall, no matter which factor dominates the total execution time overhead, this figure shows that our *Cached8* scheme

indeed scales reasonably well to relatively large DSM systems, even with a small, fixed, table storage overhead of  $\frac{2 \times 8 \times 705}{8}$  bytes, or slightly less than 1.5 KB.

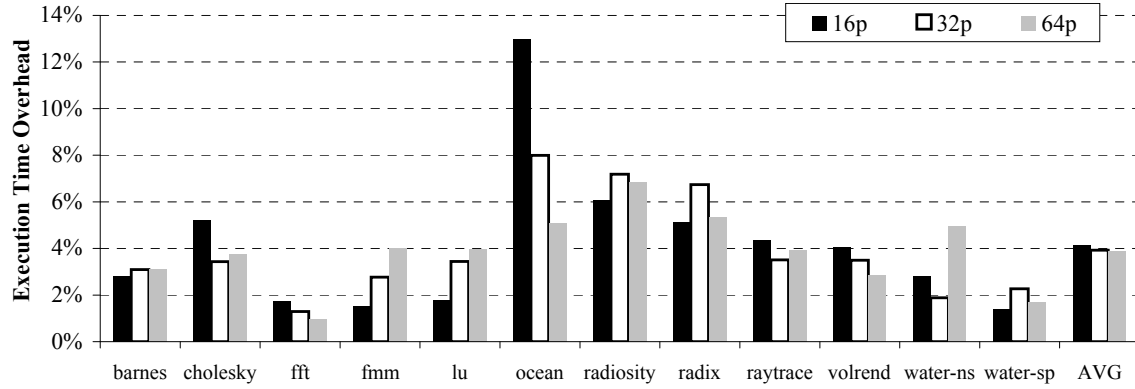


Figure 3.7: Execution time overheads for our *Cached8* scheme across 16, 32, and 64 processors.

To further evaluate our *Cached* scheme, we also determine how well this scheme performs at correctly caching counters and pre-generated pads of frequently communicating processor pairs, even with small table sizes and a large number of processors. Figure 3.8 shows the pad miss rate at the receive tables for *Cached8* with 16, 32, and 64 processors. This figure shows that our *Cached8* scheme is effective at storing the correct pre-generated pads most of the time. Even with small, 8-entry tables, the pad miss rate only increases from 17% to 30% to 36% on average for 16, 32, and 64 processor DSM systems respectively. This is promising because it indicates that even as the DSM size doubles, the pad miss rate for our tables only suffers slightly.

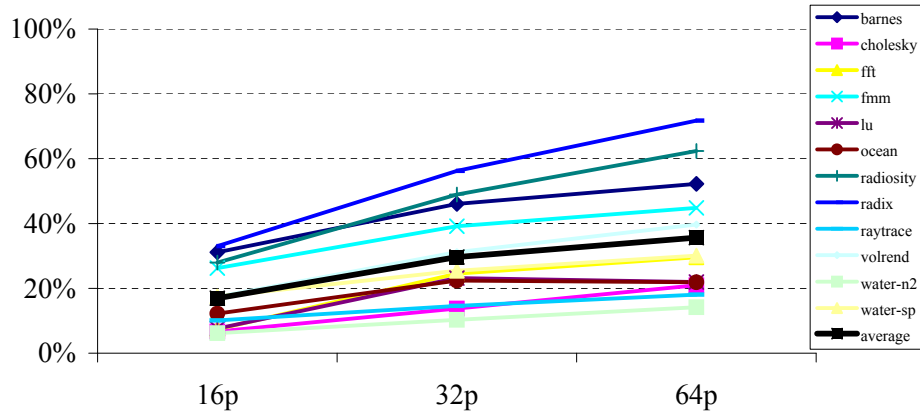


Figure 3.8: Pad miss rate at the receive tables for our *Cached8* scheme across 16, 32, and 64 processors.

### 3.6.3 Sensitivity Analysis

In this section, we present several sensitivity studies to confirm that our schemes perform well for a variety of system parameters. We first evaluate our schemes under various AES latencies and occupancies. Then we evaluate the performance of our schemes with various DSM system sizes. Lastly we examine the performance of our schemes as the size of the L2 cache is varied.

Figure 3.9 shows the performance of *Direct*, *Private*, *Shared*, and *Cached4* averaged across all benchmarks as the AES latency and occupancy is increased from  $1\times$  (the default value) to  $2\times$  and  $4\times$ . The  $4\times$  latency corresponds to a 320-cycle AES latency and 20-cycle occupancy, which is very pessimistic. This figure shows that as the AES latency and occupancy increase, the overhead of all schemes increases, especially for the  $4\times$  con-



figuration. We investigate this further and found that most of the increase in overheads is due to the increase in occupancy of the AES unit which cannot keep up with the arrival rate of encryption and integrity verification requests for both processor-memory and processor-processor communication. Therefore, these additional overheads can be reduced by providing more than one AES unit. Also note that with one AES unit the performance gap between the performance of *Direct* and our schemes becomes larger (a percentage-point difference of 5% between *Direct* and *Cached4* for  $1\times$  vs. 13% for  $4\times$ ). This indicates that our latency-hiding techniques work even better with longer AES unit latencies and AES queueing latencies.

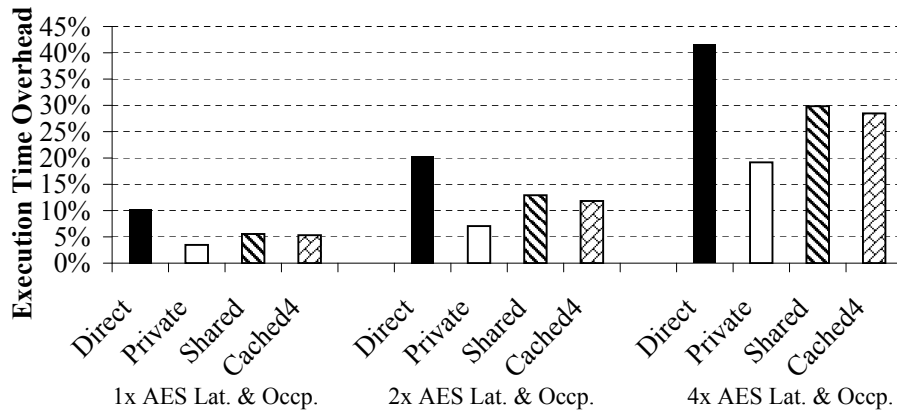


Figure 3.9: Execution time overheads for 1x, 2x, and 4x the base AES latency and occupancy.

Figure 3.10 examines the execution time overhead of *Direct*, *Private*, *Shared*, and *Cached( $p/4$ )* (where  $p$  = number of processors) as size of the DSM system varies from 16 (the default value) to 32 to 64 processors. Again, because of the amortization of the

encryption/decryption and integrity verification delays over more network hops in a large system, we observe that our schemes perform well on large systems. We note that even on a 64 processor system, however, the overhead of *Direct* is still almost 8%, and each of our schemes reduces this overhead by almost 50% or better.

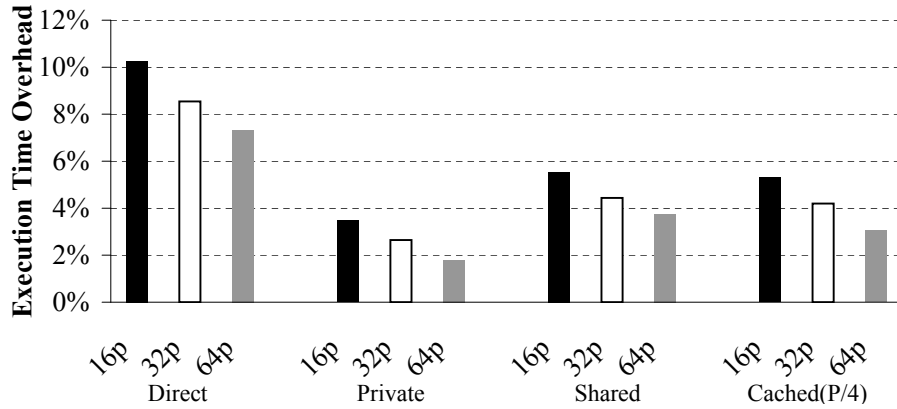


Figure 3.10: Execution time overheads across 16, 32, and 64 processor DSM systems.

Figure 3.11 validates our schemes against a number of L2 cache sizes, ranging from 128KB to 256KB (the default value) to 512KB. While we evaluate the scientific, SPLASH-2 benchmarks against our schemes, it is possible that large commercial workloads such as database and transaction processing workloads would place more stress on our schemes through more frequent processor-memory and processor-processor communication. To approximate this impact, reducing the L2 cache size to 128KB causes more frequent off-chip data requests to more heavily stress our schemes. This figure shows that in general the execution time overhead decreases as the L2 cache size increases because the L2 cache

miss rate decreases and less data must be communicated between processors. This figure validates that our schemes perform well across a variety of L2 cache sizes in a DSM system. Even with a 128KB L2 cache, the overhead of our Private table scheme remains below 4% while a naive direct protection approach hurts execution time by more than 11%.

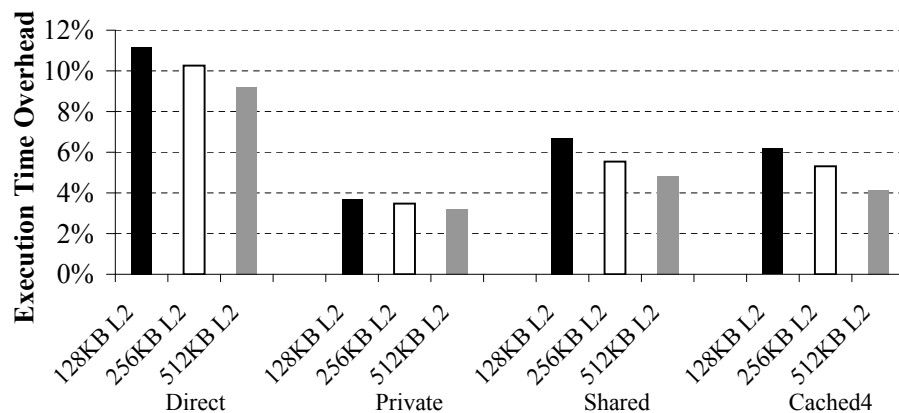


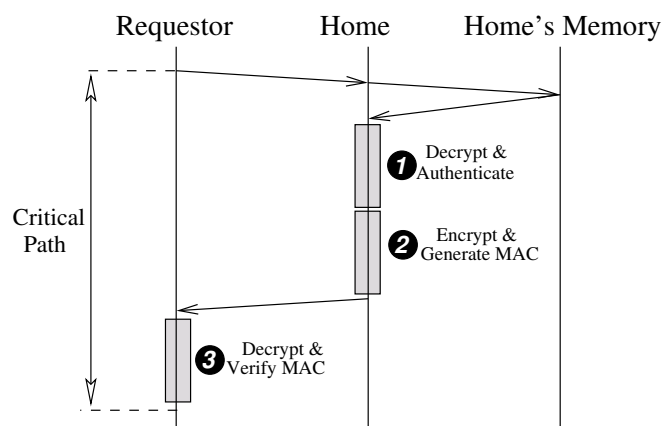
Figure 3.11: Execution time overheads for 256KB, 512KB, and 1MB L2 cache sizes.

### 3.7 Drawbacks of Table-Based DSM Protection

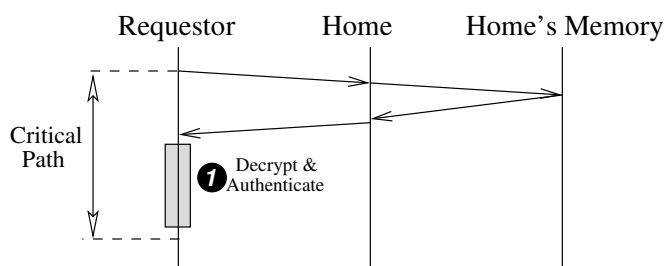
As we have described, the table-based DSM protection scheme differentiates between processor-memory communication and processor-processor communication across the interconnect, protecting each with a separate security mechanism. For example, a remote data request may result in one processor fetching, decrypting, and authenticating data from its local memory using processor-memory protection mechanisms. Then this processor will encrypt, sign, and communicate the data to the requesting processor which will decrypt and authenticate the data again, all using processor-processor protection mechanisms.

Because inter-node communication involves two separate security mechanisms, we hereafter refer to this approach as a *two-level* approach. There are several drawbacks to this two-level approach which can result in several performance inefficiencies. As shown in Section 3.6, the Cached table-based approach (which has the fewest scalability and implementation restrictions) resulted in overheads of more than 5% on average and more than 10% for some benchmarks even though it significantly improves upon the naive direct encryption and authentication approach. The first drawback is that in order for the full cryptographic latencies of a remote read request to be hidden, all latency-hiding techniques associated with each cryptographic operation must simultaneously succeed. This is illustrated in Figure 3.12(a).

The critical path of the two-level approach includes decryption and Merkle Tree



(a) Two-level encryption.



(b) Single-level encryption.

Figure 3.12: Critical Path of a remote data request for a Two-Level encryption scheme (a) and Single-Level encryption scheme (b).

authentication after local memory fetch (Circle 1), re-encryption of the requested block and its MAC generation using the processor-processor mechanism (Circle 2), and finally decryption and MAC verification at the requestor side (Circle 3). Each of these cryptographic operations incurs a latency that needs to be hidden by latency-hiding techniques such as counter caching and pad pre-generation. Each technique is imperfect (the counter cache may miss, while pads may be pre-generated too late), and this contributes to the inability to fully hide cryptographic latencies in some cases. For example, if individual techniques operate independently and the success rate of each technique ranges from 70% to 80%, full latency-hiding only occurs between 34% to 51% of the time (i.e. because  $0.7^3 = 0.343$  and  $0.8^3 = 0.512$ ), which means that a significant fraction of the time only parts of cryptographic latencies are hidden. Note that this is in addition to non-hidable latency such as the GHASH function in GCM.

Another drawback is the large amount of cryptographic work which may increase occupancy and contention at the cryptographic engines due to excessive utilization, which could further exacerbate cryptographic latencies. Overall, a two-level approach is bound to suffer from moderate to high execution time overheads as we saw in Section 3.6 and high complexity due to employing two security mechanisms.

We note that the drawbacks of the two-level approach can be avoided if we employ a unified encryption and MAC generation scheme for both processor-memory and processor-processor communication, as illustrated in Figure 3.12(b). In other words a sin-

gle mechanism is used to encrypt and sign data when it is sent off-chip by a processor (either to memory or a remote processor), and to decrypt and authenticate data when it is brought on-chip for use by a requesting processor (either from memory or a remote processor). As shown in the figure, on a remote request, the home node fetches the ciphertext of a data block, and immediately sends it off to the requestor. Only the requestor performs cryptographic operations to decrypt the block and verify its MAC. This cuts down on the total cryptographic latencies in the critical path by *two thirds*. More importantly, because only one latency-hiding mechanism must succeed in order to hide the full cryptographic latency, a single-level approach achieves lower and more stable overheads than a two-level approach, as we will show in our evaluation. However, designing a solution for such a single-level DSM data protection scheme is challenging. In the next section of the dissertation, these challenges are discussed and our single-level DSM secure architecture is described.

Finally, we note that the single-level approach provides several benefits in addition to lower and more stable performance overheads compared to the two-level approach. First, we do not require any specific assumptions on the configuration of the interconnect network such as message delivery ordering or routing strategies. Second, the single-level scheme makes no assumptions about the number of processors in the DSM system, and thus it can cleanly scale from small to very large systems. Lastly, our approach requires relatively

modest on-chip hardware resources, and in fact can even operate by only using the on-chip hardware proposed for uniprocessor secure architectures.



## 3.8 Single-Level Data Protection for DSM Systems

In this section, we first introduce the challenges and our solution for providing single-level data encryption mechanism in DSM systems. Then we overview how data integrity verification is performed with this approach. Finally, a security analysis of these mechanisms is presented.

### 3.8.1 Single Level Memory Encryption

In a single-level encryption scheme based on counter-mode encryption for cryptographic latency-hiding, the per-block encryption counter that was used to encrypt a data block in one node (e.g., the home) is needed to decrypt it in another node (the requestor). Logically, one may think that this counter can be cached at both nodes and can automatically be kept coherent by the coherence protocol. However, it is problematic to keep counters coherent when coherence messages that ensure counter coherence are themselves subject to security attacks, e.g. attackers can replay an old counter message to force pad reuse. We can protect counter coherence messages using a separate security mechanism, but this increases complexity and the difficulty of hiding latencies even more.

To avoid the complexity of relying on the cache coherence protocol to keep counters securely coherent, we adopt an alternative approach in which a *counter of a block can only be cached at the home node and at the owner node (if any)*. For a data block in a clean state, only the home can cache the counter value. For a data block in the modified

state cached at an owner, the owner eventually needs to encrypt the block and send it off to another processor (due to intervention or write-back), and hence it is allowed to cache the counter value temporarily until the block is replaced or becomes clean. When a processor asks for an exclusive state for a data block through an upgrade or read-exclusive request, the home processor increments the counter value of the block, and replies with the new counter value to the requestor. Certainly, this counter value must be protected from tampering, so we protect it using the same mechanism used to protect data communication (to be discussed in Section 3.8.2). We will now discuss the latency-hiding aspect of this approach, and leave the security aspect until later in Section 3.8.2.

**Hiding Decryption Latency at the Requestor.** In our single-level memory encryption, typically the only node that performs a cryptographic operation is the requestor of a data block, which must decrypt the block before it can use it. The key to successful latency-hiding at the requestor is that the requestor must have the counter to pre-generate the appropriate decryption pad before the data arrives. Since the requestor is not allowed to cache the counter, the home node that caches the counter must supply the counter early. Fortunately, with simple coherence protocol modifications, this is achievable. Figure 3.13 shows how various scenarios are handled. The first scenario is when the requested data is in the home node's local memory (Figure 3.13(a)). In this case, the home looks up its local counter cache to obtain the block's counter (CTR). If it finds the counter, it forwards the counter immediately to the requestor, and in parallel begins to fetch the data block

ciphertext (CTEXT) from its local memory. The counter value arrives at the requestor one memory-latency before the data, allowing the requestor to generate the decryption pad ahead of receiving the data ciphertext.

Figure 3.13(b) shows another scenario in which the data block requested is also in the home processor's cache (in plaintext form because it is on chip). Before forwarding the data block to the requestor, the home processor must first encrypt the data block. In parallel, it sends the counter value to the requestor. The requestor, upon receiving the counter, immediately pre-generates the pad needed to decrypt the ciphertext of the data block that is still yet to arrive. In this case, the one pad generation latency at the home node is exposed, but the pad generation latency at the requestor is hidden. This is still better than the two-level approach which in the worst case can suffer up to three pad generation latencies. Furthermore, the scenario in Figure 3.13(b) may be rare because typically different processors work on different data, so a remote read rarely finds the data in the *clean* state at the home processor's cache (if the state of the data is dirty, it is handled in the next scenario).

Figure 3.13(c) shows a scenario in which the data is owned by another processor. In this case, the home node will forward the request to the owner, and in parallel send the counter value to the requestor. When the data block owner receives the request, it will first encrypt the data block and send it to both the home processor and the requestor. The requestor still receives the counter before it receives the data, and can overlap pad

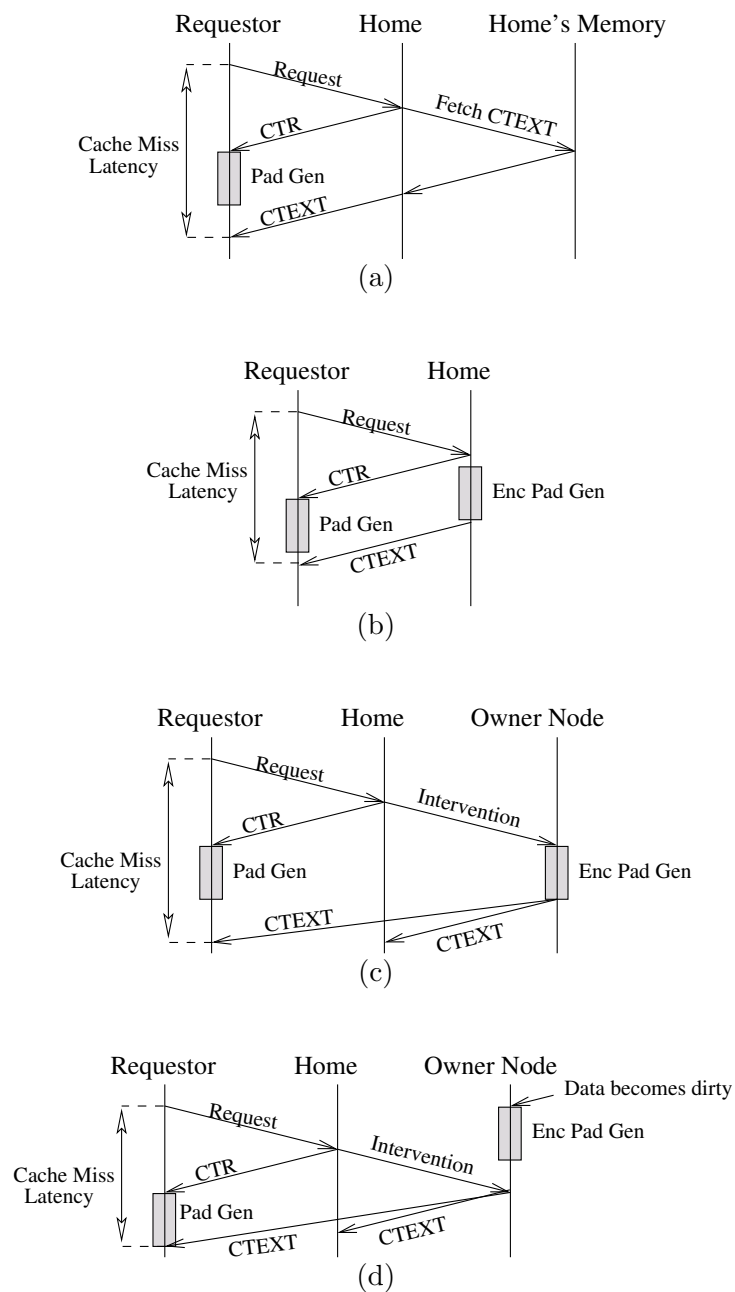


Figure 3.13: Coherence protocol modifications for hiding cryptographic latencies of remote data request in our single-level counter mode memory encryption when data is supplied by the home node's local memory (a), home node's cache (b), remote owner (c), and remote owner with an owned block pad buffer (d).

generation with the communication latency. However, one pad generation latency at the owner is exposed. Unfortunately, compared to the scenario in Figure 3.13(b), this case quite frequently occurs when different processors share data that is frequently written (both true and false sharing). Hence, we seek ways to optimize this scenario next.

**Hiding Encryption Latency at the Sender/Owner.** In Figure 3.13(c), the encryption pad generation at the owner’s side is in the critical path. In Figure 3.13(d), we illustrate an optimization to move the latency off the critical path. To achieve that, when a data block becomes dirty or modified, immediately we trigger its encryption (and authentication) pad generation. These pads are then stored in a small buffer called the *Owned-Block Pad Buffer*. We only pre-generate pads for modified blocks because these are the only blocks that have a possibility to be intervened by another processor. When an owner receives an intervention for a data block, if it finds pads for the block in the owned-block pad buffer, it can immediately encrypt and generate the MAC for the block in a few cycles, and send the block off to the requestor.

One important question is how large the owned-block pad buffer needs to be. We observe that frequently communicated blocks typically reside in any one cache for only a short time since they receive interventions quite frequently (e.g. shared locks), and that blocks that have been in the modified state for a long time are unlikely to be intervened frequently. Therefore, the owned-block pad buffer only needs to store pads for the blocks which are most recently upgraded to a modified state. We find that a 32-entry owned-block

pad buffer is sufficiently large to ensure that in most cases pre-generated pads are available for intervened data blocks.

**Local Counter Management and Counter Prediction.** Note that Figure 3.13 assumes that the home finds the counter in its local counter cache for each request. If this is not the case, the home must first fetch the counter from the local memory before the counter can be sent to the requestor. The frequency of this case can be made very rare with a good counter caching policy at the home. Since a home node only caches counters of blocks that are in its local memory, the counter cache size can be kept constant regardless of the number of processors in the DSM. In fact, when a cache block is shared by multiple processors, the first processor that requests the data block from its home node will cause the home node to cache the counter of the block, effectively prefetching the counter for subsequent processors that want to fetch the block. In addition, since block counters can be small (e.g. 8 bits), a counter cache block can hold many counters from many cache blocks. Hence, the common case is that the home finds a counter of a requested block in its local counter cache.

Furthermore, the requestor can employ a *counter prediction* mechanism such that even if a home node cannot find the counter in its local counter cache, the requestor can still pre-generate pads ahead of time. For example, each processor can keep track of the last counter value of remote blocks that it has recently evicted. If the last value is zero, it is highly likely that it corresponds to a read-only block for which the counter value will

remain zero; hence we can start pad generation assuming that the counter value is zero. In addition, tracking such zero-valued counters is very space efficient since we only need a single bit per counter. We use a *mask buffer* to store these bits, with a bit value of “1” indicating a particular counter was last seen as zero. We find that a 32 entry mask buffer is able to provide effective counter prediction of zero-valued counters. When a node sends a data request to a remote node, it can send along the predicted counter value that it used for pregenerating its pad. If the predicted value of the counter is correct, the home node can skip sending the counter value to the requestor; hence we have an additional benefit of conserving network bandwidth. We note that there are many other techniques for predicting counter values, but we find that even this simple counter prediction scheme can achieve high accuracy of 76% (Section 3.9).

**Summary.** Overall, we have shown that cryptographic latency-hiding can be achieved through simple coherence protocol modifications that allow the home node to forward counter values, pad pre-generation at owner nodes for modified blocks that they cache, and relatively simple counter prediction at the requestor. With these mechanisms in place, pad generation latency is exposed in only a few cases, and even in these cases, only one pad generation latency is exposed.

Small architecture components are added to each processor as shown in Figure 3.14. A secure processor architecture for uniprocessor systems already contains a cryptographic engine and local counter cache. Over them, we add a 32-entry owned-block

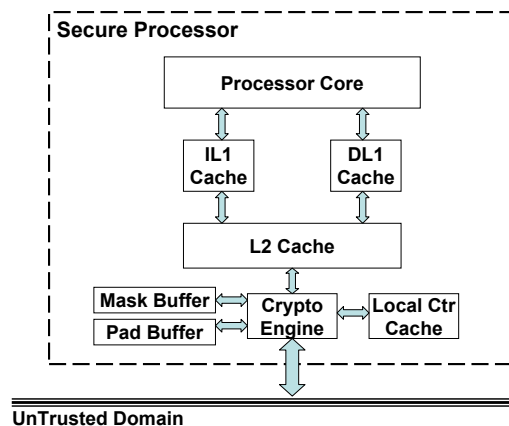


Figure 3.14: Architecture modifications to a node.

pad buffer to hold pre-generated encryption and authentication pads, and a 32-entry mask buffer to store the bit masks indicating zero-valued counters for counter prediction.

### 3.8.2 Single-Level Memory Authentication

Next, we will discuss the security aspects of our scheme. Since data values are always communicated as ciphertext, passive attacks are protected against. We now discuss our mechanisms to ensure that active attacks are detected. Memory authentication in DSM systems requires protection not only from attacks on data loaded from a processor's local memory, but also from attacks on data communicated between processors across the system interconnect. As we have discussed, protection of data loaded from off-chip memory can be achieved through a Merkle Tree mechanism. However, as with counter-mode encryption, it



is not an easy task to extend Merkle Tree based protection to protect data communicated between processors. It may seem possible to cover the entire shared memory with a single Merkle Tree which is distributed across all processors. However, keeping the Merkle Tree nodes coherent is similar to the problem of keeping block counters coherent, as discussed in Section 3.8.1.

To avoid the significant complexity of supporting a coherent global Merkle Tree, we adopt an alternative approach. Each node maintains its own *local Merkle Tree* that covers only the node's local memory, and a lightweight message authentication protocol provides authentication for all data communicated in a DSM system. In a traditional *early* authentication protocol (e.g. the authentication approach previously described for the table-based DSM protection schemes) which is illustrated in Figure 3.15(a), protection is provided as follows. To send a message  $X$  over the interconnect from node A to node B, the message authentication code (MAC) of the message is first computed at the sender A as  $MAC = H_K(X)$ , where  $H_K(.)$  is a cryptographic keyed hash function with private key  $K$ . Upon receiving the message, B recomputes  $H_K(X)$  and compares it against the received MAC. If either  $X$  or the MAC has been altered, the recomputed MAC will mismatch at B. B then sends an acknowledgement message back to A to indicate the receipt of the message. Attackers cannot generate a correct MAC for a given value of  $X$  or for an arbitrarily chosen value since they do not know the secret key  $K$ . However, this approach may be vulnerable to replay attacks where the value of  $X$  and its MAC are replaced by old, stale values that

an attacker has recorded, unless another protection mechanism is used to prevent such attacks. In addition, the MAC computation is in the critical path of communicating  $X$  from A to B.

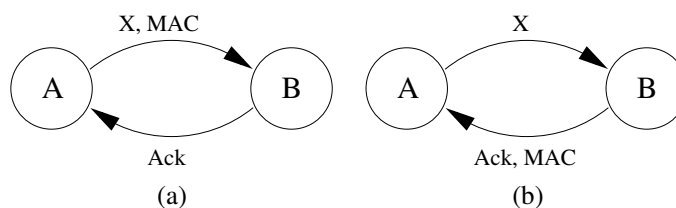


Figure 3.15: High-Level illustration of an early (a) versus our delayed (b) authentication protocol.

Our approach uses a different *delayed* protocol as illustrated in Figure 3.15(b). First, node A sends  $X$  to node B (we refer to it as the *forward edge*). Node B then computes the MAC for the message value it receives and sends it back to node A piggybacked to the standard acknowledgement message (we refer to it as the *backward edge*). Node A knows the original value of  $X$  so it can recompute  $H_K(X)$  and compare it against the received MAC. If either  $X$  or the MAC have been tampered with, the recomputed MAC will mismatch at A. Note that both protocols incur the same cryptographic work, but our approach incurs no delay in the latency-critical path of communicating  $X$  from node A to node B. If the communication already involves such a *loop* (both forward and backward edges are naturally present), no extra bandwidth is consumed over the traditional early protocol. In addition, this approach can support protection against replay attacks as we will describe below.

While attackers cannot inject a valid message or compute a valid MAC given a message, attackers may attempt to reroute a message destined for one node to another node. To detect such attempt, the MAC computed by node B should include its own processor ID, i.e.  $MAC = H_K(X, ID_B)$ . Meanwhile, node A can remember the message value  $X$  and its intended destination  $ID_B$  in its structure which tracks outstanding transaction, so it can compute the MAC that it expects to receive from node B. If the received MAC mismatches, then it has discovered a message tampering or rerouting attempt by the attacker. Similarly, attackers may attempt to replay an old message. As a general mechanism to detect such attacks, the sender node A may piggyback a unique timestamp  $ID_A||TS$  to each message.  $TS$  can be a monotonically increasing *message counter* that each node increments when it sends a forward-edge message. Thus the combination of a processor's ID and current message counter forms a unique timestamp for each message. Node A can also store the timestamp value in the outstanding transaction table. Node B must compute the MAC that includes the timestamp it receives, i.e.  $MAC = H_K(X, ID_B, ID_A||TS)$ , and send the MAC to A in the backward edge. Since no two messages share the same timestamp value, the computed MAC is always unique. Overall, our protocol is able to detect communication attacks such as message tampering, message reroute, and message replay without adding delay to the critical path of communication.

In our secure DSM coherence protocol, we can identify five main types of communication: *request* (sent by a requestor to the home), *data response* (data sent by the

home or owner as a response to a request), *non-data response* (acknowledgement or negative acknowledgment as a response to a request), *write-back* (data sent to the home by an owner), and *counter* (per-block counters sent to requestors to enable decryption latency hiding). In principle, it is straightforward to protect them all by (1) requiring all of them to have loops (adding backward edges when necessary), and (2) applying our delayed authentication protocol to protect them.

Based on this classification of messages, we identify two protection levels that seem to be reasonable. The first is simply *full* protection in which all types of communication are protected using delayed authentication. Full protection's security level is the strongest since all messages in the system are cryptographically protected. Another option is to provide *data-counter-only* protection in which only communications that include data and counter values are protected (i.e. data response, write-back, and counter messages), but others are not (i.e. request, and non-data response messages). We note that requests, invalidations, and other non-data messages are directly involved in maintaining the correctness of coherence protocol implementations and hence their tampering will result in anomalous coherence protocol operations. For example, a non-home node could receive a read request or an invalidation could be received by a node that does not cache the block. So it may be an attractive option to use data-counter-only protection and rely on coherence protocol anomaly detection to further deter attackers as discussed in Section 3.2.

While the above discussion of the delayed authentication protocol provides a

framework for us to provide protection, another interesting issue is with regard to optimizations of different cases in terms of efficiency or flexibility. For example, a node requesting data will receive both a counter value and clean data from the data provider. As such, for both communication types, the requestor can send a single backward edge message for validation, rather than two separate backward edge messages. In the following subsections, we provide a more concrete discussion of how to apply the delayed authentication protocol to protect DSM systems.

### **Authenticating Data Response and Counter Messages**

The first type of protection we discuss is that for data response and counter messages. As we mentioned, since these messages are always paired, we can merge the protection of each with a single MAC on the backward edge message. Figure 3.16 shows the steps of our process to authenticate data communicated to a requesting processor, where in this case the data is supplied by the data's home node. This figure also notes the roles of various messages and message components as they relate to our delayed authentication approach described in the previous section. This scenario matches that in Figure 3.13(a), except that now we augment its security through message authentication. When the home node receives the data request, it first sends the block counter as the forward message of counter communication (Circle 1b). At the same time it fetches both the data ciphertext (CTEXT) and also the block's MAC – which is the lowest level MAC in the node's local

Merkle Tree and is generated as described previously (Circle 1a). Then the home sends the forward message for the data response which consists of the ciphertext, its MAC, and a *message counter* (MCTR) (Circle 2). The MCTR is the per-processor, on-chip message counter value as described in the previous section. The MCTR and processor ID will form the timestamp used to prevent replay attacks. Each processor's message counter is stored in an on-chip, non-volatile register so that its value remains even across system resets. Also, each processor will need to buffer the MCTR values for outstanding messages so that they can be used to verify the response MAC that is included in the backward edge acknowledgement message. This is needed so that processors can have more than one outstanding transaction at a time. Also, note that we choose to send the block's MAC value in this message. We describe the reason for this below, but note that this MAC is not in the critical path of sending the data response.

Upon receiving the data response message, the requester decrypts the data for use by the processor. Then it verifies the MAC of the ciphertext, which ensures that the ciphertext, block counter, and MAC combination are valid, since the counter is a component of MAC generation using GCM (Circle 3). Thus, up to this point, the only option for an attacker is to attempt a replay attack by providing stale, but corresponding block counter, ciphertext, and MAC values. However, our delayed authentication protocol still has one more step, the backward edge MAC, which ensures that both the counter and data response messages are free from tampering, including replay. The MAC computed

(Circle 4) for the backward edge message (MSGMAC as shown in the figure) is computed based on the components shown in Equation 3.1.

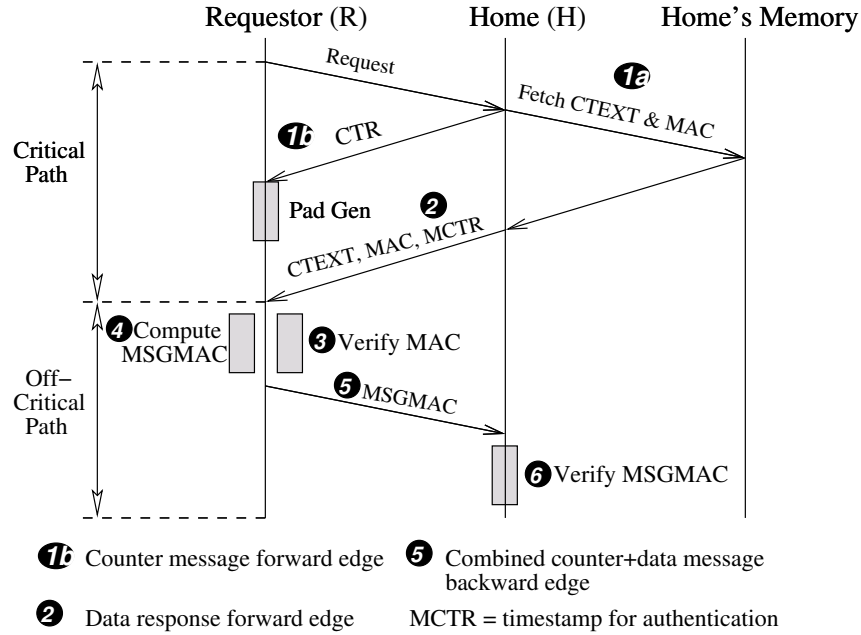


Figure 3.16: Steps for authenticating data sent to a requesting processor across the interconnect.

$$MSGMAC \Leftarrow H_K(MAC || PID_R || PID_H || MCTR) \quad (3.1)$$

The MSGMAC is computed by the requestor following the general process described in the previous section for backward edge MACs. Again, the combination of the home's PID ( $PID_H$ ) and the home's message counter ( $MCTR$ ) form the timestamp to make each MSGMAC computation unique, and the requestor's PID ( $PID_R$ ) ensures that messages cannot be rerouted. The final component is the data block's MAC. Since a replay

attack on data response message must roll-back the block ciphertext, block counter, and block MAC in unison, verifying that the correct block MAC was received by the requestor allows the verifying processor to ensure that such a replay attack did not occur.

Finally, the MSGMAC is piggybacked to the traditional acknowledgement message (Circle 5), and the home node can then verify MSGMAC (Circle 5) by recomputing it based on the components that it *knows* the requestor should have used. If any of the components of the counter or data response messages have been modified, the MSGMAC verification will fail. Additionally, an attacker cannot generate a correct MSGMAC because: (1) they do not have the secret key of the keyed-MAC function, (2) the timestamp included in the MSGMAC ensures that MSGMAC values do not repeat, and (3) the home node always knows the components that it is verifying in the MSGMAC.

We would like to point out that we have made the design choice to include the block's MAC value in the data response message and have this verified by the requestor for the following reason. This prevents an attacker from being able to forge any arbitrary value that the requesting processor will receive and use before the tampering can be detected at the home node through the MSGMAC. However, if this issue is not a concern, it would be straightforward to eliminate the block's MAC from the data response message and have the MSGMAC computation include both the block ciphertext and block counter instead of the MAC value. Finally, we note that the procedure of authenticating a counter and data response message is similar if the data is provided by a remote processor which currently



owns the data. The main difference is that in this case, the backward edge message and MSGMAC are communicated between the requesting and owning processors. Additionally, the owning processor may also be required to flush the data to the home node using the writeback protocol discussed in the next subsection.

### **Authenticating Data Writeback Messages**

For authenticating writeback messages from a node owning a data block to the home node we apply another optimization compared to the basic delayed protocol described in Section 3.8.2, which allows us to eliminate the backward edge message. The key observation is that due to the way our encryption scheme manages the per-block counters, both a node owning a data block and the data's home node know the correct, current block counter value. Therefore, the block counter value can be used as the timestamp in the MAC computation for writeback messages. The owner of the data block can first generate the block ciphertext and MAC value using GCM and the per-block counter, and then send the ciphertext and MAC in the writeback message to the home node. Then the home node can use its known block counter value to recompute and verify the MAC of the received ciphertext.

Attackers cannot alter the ciphertext or MAC without triggering a failed MAC verification at the home node. Additionally they cannot replay an old ciphertext and MAC pair. The reason is that this would require also replaying the block counter value since

it is included in the MAC computation, but this is infeasible since both the owning node and the home node know the correct counter value. The advantage of including the MAC value on the forward edge for writeback messages is that it still prevents replay attacks, and writeback messages are typically not latency-critical. This also relieves the owning processor from having to send its current message counter value to the home node with the writeback message and from having to buffer the MAC value that it would expect to see on the backward acknowledgement edge from the home node.

### **Authenticating Request and Other Non-Data Coherence Messages**

The last types of messages that may require authentication are coherence messages which do not contain data or counters such as request, invalidation, and intervention messages. For these types of messages, we can apply the delayed authentication protocol in a straightforward manner. In full protection mode, along with each forward edge request, invalidation, or intervention message, the initiating node will also send its message counter value. Then the node receiving the message can compute the response MAC based on the timestamp of the message counter and initiator processor ID as well as the content of the received message. This MAC is sent along with the backward edge reply, ACK, NACK, etc. message to be verified by the initiating node.

Protection of these messages may be important in protecting certain types of attacks. For example, if an invalidation message is dropped by an attacker, then the node

meant to receive that message may continue to use stale data from its cache, resulting in an *indirect replay attack*. By cryptographically protecting these messages, we can prevent such attacks because the initiator of the message will expect to see a valid MAC of the message returned to it, and the attacker cannot generate this valid MAC.

### 3.8.3 Security Analysis

In the previous sections we have discussed how our authentication mechanisms protect the various types of messages in our secure DSM system, and how this protection can prevent various attacks. Now we will reiterate the security of our delayed authentication protocol for DSM systems by identifying some of the major attacks that may be attempted and how our schemes can prevent these attacks. First and foremost, attackers may attempt to tamper with ciphertext, block counter, or MAC values in the system in hope of discovering some plaintext information or altering the execution of an application. However, since all data and block counter communication is protected by the data's cryptographically secure MAC value, attackers cannot tamper with any of these components without triggering a failed MAC verification. In addition if an attacker tries to replay all three components in unison to a requesting processor, this will be detected when the node that supplied the data verifies the response MAC included in the backward edge message. Since the response MAC includes information from the original message plus a unique timestamp, attackers also cannot forge this response MAC.

Attackers may also attempt to manipulate the system in other ways than just by modifying data or counter values. For example, attackers may attempt to drop messages. However, doing this will simply result in a timeout and retry if the initiating node does not receive a backward edge acknowledgement message. Additionally, attackers cannot forge such messages since they cannot generate the valid MAC that should accompany backward edge acknowledgement messages. Attackers may also attempt to reroute a message destined for one processor to another, but this is prevented because the response MAC to authenticate the message contains the destination node's processor ID.

## 3.9 Evaluation of Single-Level DSM Protection

### 3.9.1 Experimental Setup

In order to evaluate the single-level DSM data protection scheme, we again implement the required features in a DSM multiprocessor model on SESC [22]. The simulated DSM system has the same baseline parameters as described in Section 3.5. The hardware simulated specifically for our single-level protection scheme includes a default 32KB, 8-way counter cache to store the block counters of frequently accessed blocks in a processor’s local memory. This cache is the same as in uniprocessor memory encryption schemes. For the owned-block pad buffer, we use a 32-entry FIFO buffer having a total size of 1 KB. For counter prediction, we add a small 32-entry mask buffer for storing a bit vector of which data blocks last had a counter value of zero. The total size of the bit vector is approximately 512 Bytes because each entry is 16-bytes in size. We assume a 2-cycle latency for accessing the pad buffer and mask buffer.

Again we evaluate against the SPLASH-2 benchmark suite, using the same application parameters shown in Table 3.2. Table 3.3 shows the percentage of L2 misses in each application which are satisfied by the home node’s local memory. This data serves as a reference for the frequency that the various cryptographic-latency hiding mechanisms of the single-level scheme are utilized for protecting data in the DSM system.

Table 3.3: Fraction of requests served by the home node’s local memory in SPLASH-2.

App	% Home Reqs
Barnes	64%
Cholesky	88%
FFT	99%
FMM	60%
LU	80%
Ocean	75%
Radiosity	66%
Radix	96%
Raytrace	91%
Volrend	89%
Water-n2	50%
Water-sp	88%

### 3.9.2 Overhead of Single-Level DSM Data Protection

In Figure 3.17 we show the execution time overhead our single-level DSM data protection scheme, normalized to a DSM system with no support for data encryption or authentication. For comparison, we again show the corresponding overhead of the two-level, Cached scheme. We compare against this scheme since it is the only one which is similar to the single-level scheme in that it meets the design criteria of small on-chip storage overheads and the ability to scale to arbitrarily large DSM system sizes (in terms of number of processors in the DSM).

From this figure, it is clear that while both schemes are similar in terms of small hardware support, the ability to support large systems, and security, our single-level scheme provides significantly better performance than the Cached two-level scheme. The average overhead across all applications is 1.6% while the overhead of Cached is 5.3%, representing

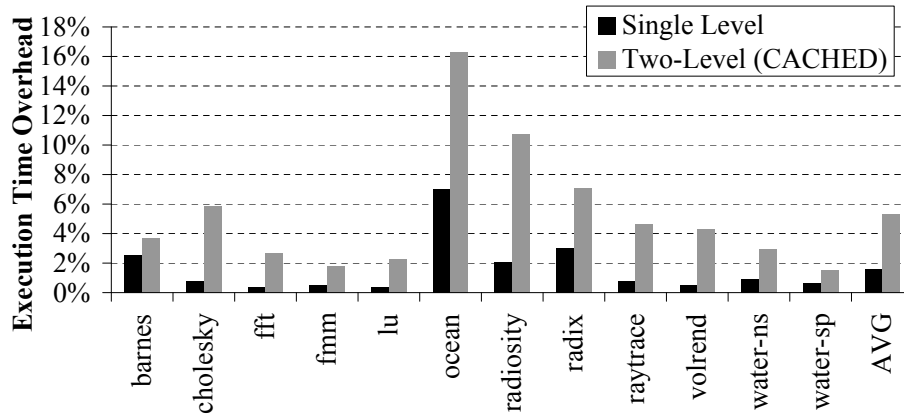


Figure 3.17: The execution time overhead of single-level DSM data protection scheme versus two-level protection using the CACHED scheme with 4-entry communication counter table.

a reduction of overheads by a factor of  $3.3\times$ . In addition, there are several applications which suffer from fairly significant overheads under the Cached scheme, for example *ocean* and *radiosity* at 16.3% and 10.7% respectively. Our scheme reduces these overheads to 7.0% for *ocean* and 2.1% for *radiosity*. Equally important is the number of cases in which the execution time overheads are practically negligible. With Cached, there are nine benchmarks that are slowed down by more than 2%, while for our proposed single-level scheme there are only three benchmarks slowed down by more than 2%. Since DSM systems are typically very pricey and they are often used to run critical applications, it is likely that performance overheads such as those seen for the worst-case applications with a two-level scheme are not tolerable. Additionally, the performance of the single-level scheme is more stable than that of the two-level scheme. With a standard deviation of 1.9% in execution

time overhead, the single-level scheme provides more confidence to users that their applications will perform acceptably well than the two-level scheme with a standard deviation of 4.3%.

The central reason for the larger performance overheads of the two-level scheme shown in Figure 3.17 is that cryptographic latencies may be exposed at multiple points in the critical path of a data fetch from a remote processor. More specifically, there are three points in this critical path where cryptographic delays may occur as shown in Figure 3.12: (1) when a memory block requested by a remote processor is fetched on-chip by the block’s home processor and decrypted, (2) when the block is encrypted again to be sent to the requesting processor, and (3) when the requesting processor receives and decrypts the block on-chip. Our data confirms this observation: on average, cryptographic latencies are at least partially exposed at point (1) 9% of the time, at point (2) 29% of the time, and at point (3) 46% of the time. This means that roughly only  $(1 - 0.09) \times (1 - 0.29) \times (1 - 0.46) = 35\%$  of the time full cryptographic latencies are hidden in the Cached scheme (versus 82% of the time for our scheme – we will discuss the result in Figure 3.18 later). This shows that two-level schemes are inherently less efficient because there are too many points on the critical path where delays can be introduced.

Now that we have examined the performance of our single-level DSM data protection scheme, we will take a closer look at the reasons for its low performance overhead. Figure 3.18 shows the percentage of off-chip data requests for which the decryption pads



are fully generated (pad hit), partially generated (pad half-miss), or not generated (pad miss) when the requested data arrives on-chip. If a pad is fully generated, the decryption latency is totally hidden, while if it is partially generated then the latency is partially hidden. From this figure, we can see that for most applications and on average the full, critical-path decryption latency is hidden 80% of the time or more. Additionally, the percentage of requests which suffer from the full decryption latency is less than 1% in almost all applications. On average, 82.4% of the time pad generation latency is fully hidden, 17.1% of the time it is partially hidden, while only 0.5% of the time the latency is fully exposed.

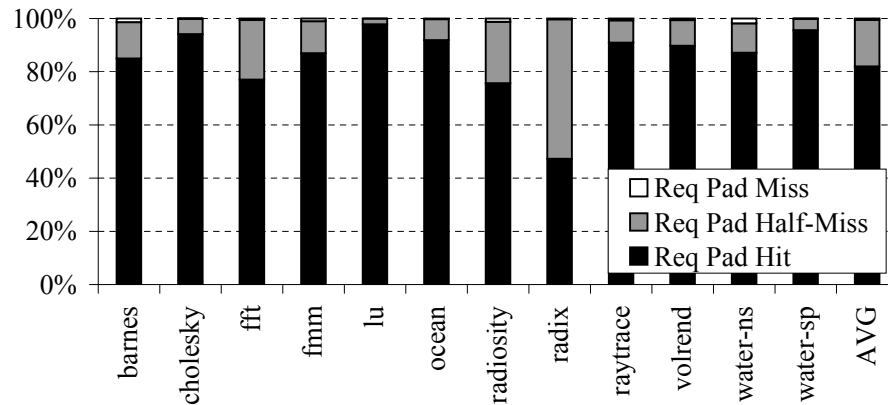


Figure 3.18: Fraction of off-chip data requests that experience pad hit (full latency- hiding), half-miss (partial latency-hiding), and miss (no latency-hiding).

To further explain the performance of the single-level scheme, we present Figures 3.19 and 3.20. In Figure 3.19, we show the percentage of off-chip data requests for which we correctly predict the counter value for the data block. In Figure 3.20 we show

how frequently the home node of a data block finds the requested block's counter cached in its local counter cache. This percentage is the total height of the two bars, and it corresponds to the percentage of data requests in which the home node can reply with the data's counter early to hide the decryption delay at the requestor. However, with our counter prediction scheme, if the counter has been predicted correctly by the requestor, the home node does not need to reply with a separate counter message. The gray portion of the bars shows how frequently this event occurs.

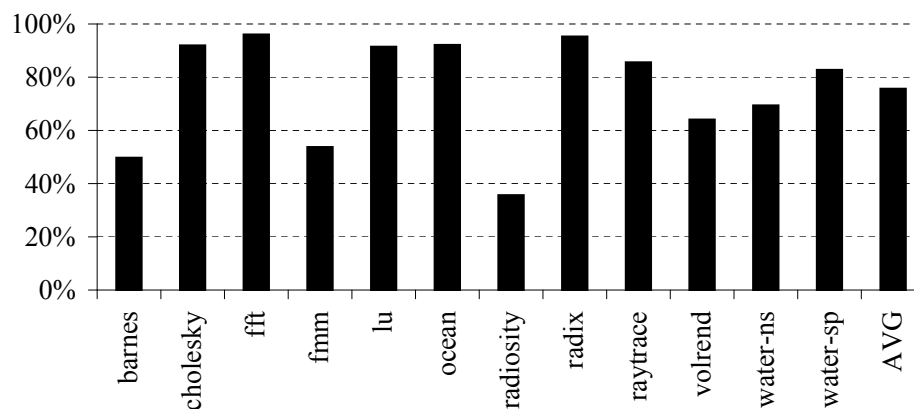


Figure 3.19: Percentage of requests for which the counter value is correctly predicted.

Figure 3.19 shows that, despite its simplicity, our counter prediction scheme is very successful at correctly predicting counter values. The correct prediction rate is 76% on average, and over 90% for 5 applications. This high prediction rate benefits our scheme in two ways. First, if counters are predicted correctly, then it is *very* likely that the decryption pad is fully pre-generated before it is needed, thus fully hiding the decryption

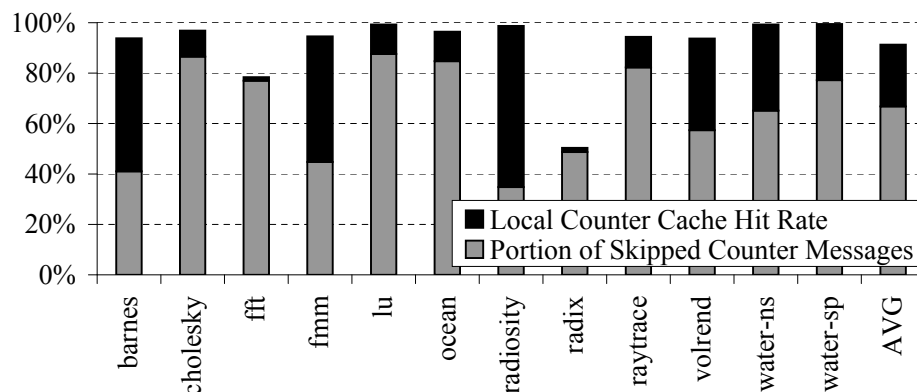


Figure 3.20: Local counter cache hit rate and portion of counter messages skipped due to correct prediction.

latency. Also, as shown in Figure 3.20, we can eliminate a large number of separate counter messages from the home processor to the requestor. This reduces the pressure placed on interconnect bandwidth, because a separate counter message requires more overhead than simply including the counter value with the data of the reply message. Figure 3.20 also shows that, when counter prediction fails, most of the time we can still hide the decryption latency by forwarding the correct counter. This figure shows that a block's counter value can either be predicted or sent early over 90% of the time in most cases, and 91% of the time on average.

The final comparison we make between the single-level scheme and the two-level scheme is on the AES unit bandwidth utilization, shown in Figure 3.21. Due to the reduced amount of cryptographic work, for most applications we observe a large decrease in AES utilization with the single-level scheme. For all but two of the applications, we use the

AES unit 30% less than in the two-level scheme, and for some applications this savings is closer to 40%. This result shows that the improved single-level scheme provides secure data encryption and authentication in a DSM system with fewer cryptographic operations required compared to a two-level scheme.

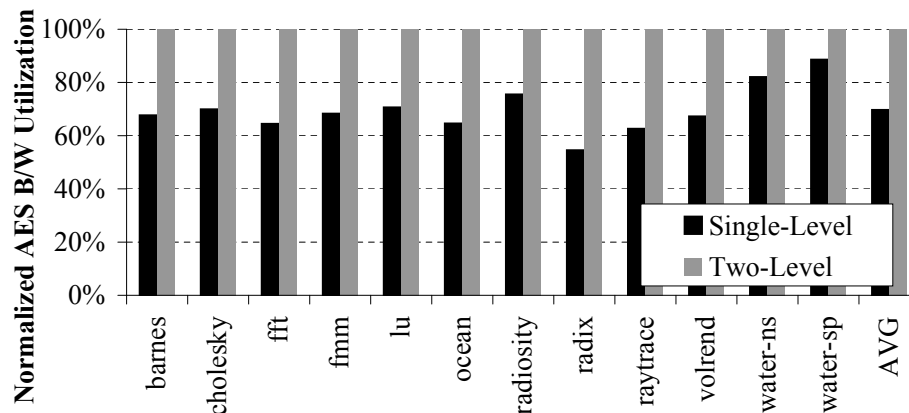


Figure 3.21: The normalized AES bandwidth usage of single-level vs. two-level DSM data protection.

### 3.9.3 Sensitivity Analysis

In Figure 3.22, we show how the single-level DSM data protection scheme performs as the number of processors in the DSM system increases. Again, the performance is shown as execution time normalized to a DSM system with equivalent configuration but with no support for data protection.

It is clear from this figure that the overheads of our protection scheme remain low as the number of processors increases. In fact, in most cases and on average the overhead decreases with respect to the system size. On average, the overhead goes from 1.6% on

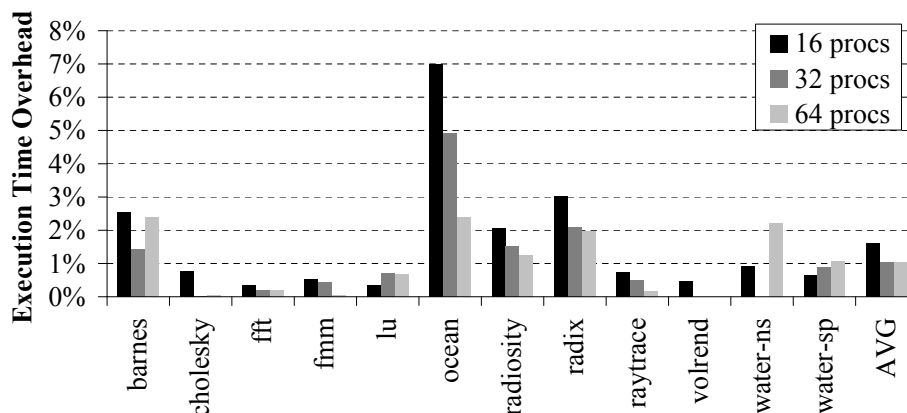


Figure 3.22: Execution time overhead of our single-level DSM data protection scheme across system size.

16-processor DSM to 1.0% for both 32-processor and 64-processor DSM system. The worst-case overhead also improves significantly from 7.0% on 16 processors down to 4.9% on 32 processors and 2.4% on 64 processors. With a larger DSM size, there are more remote data requests because data is scattered around more nodes. However, at the same time communication latencies increase with the number of processors since data must travel more hops across the interconnection network on average, making the impact of cryptographic latencies less significant relative to the total remote data fetch latencies. The figure shows that, in general, the increase in inter-node communication is the more important factor, resulting in execution time overheads of just 1% as the DSM size increases.

Figure 3.23 shows the overheads of the single-level DSM data protection scheme as the size of the L2 cache varies from 128KB to 256KB (our baseline size) to 512KB.

The performance is shown as execution time normalized to a DSM system with equivalent configuration but with no support for data protection.

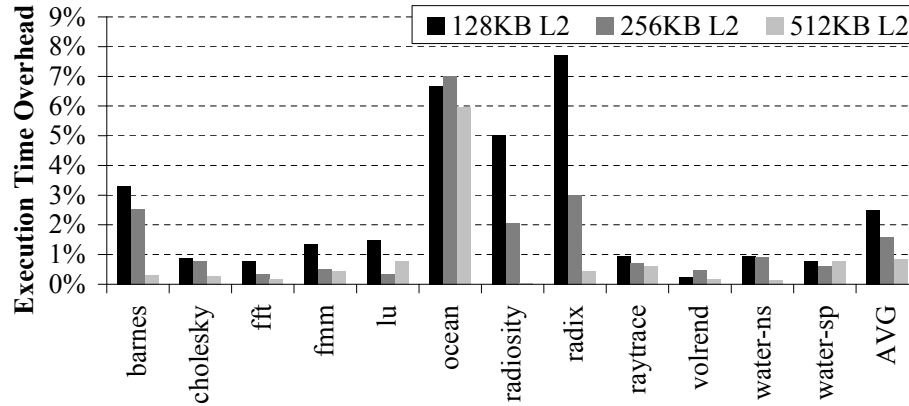


Figure 3.23: Execution time overhead of our single-level DSM data protection scheme across L2 cache size.

As shown in this figure, generally the overheads are reduced as the cache size increases; for example the average overhead is 2.5% with 128KB L2, 1.6% with 256KB L2, and only 0.8% with 512KB L2. The reason is that larger caches in general reduce the amount of traffic to memory, and thus the amount of data encryptions, decryptions, and authentications that our single-level DSM protection scheme must perform. Note that even with a 128KB L2 cache size, where our scheme is stressed more heavily, the overheads remain low, with an average of only 2.5% and a maximum of only 7.7%. This indicates that our scheme will perform well even when it is heavily stressed in scenarios where the amount of off-chip data communication is large. For example, this may be the case when large-scale

commercial workloads are executed on the system as opposed to scientific applications such as those studied.

### 3.10 Conclusions

While hardware memory encryption and integrity verification schemes have been studied in detail for uniprocessor and SMP systems, no such work has been done explicitly for DSM systems, which provide a unique set of challenges. This chapter presented the first study of data protection with hardware memory encryption and integrity verification mechanisms for protecting processor-processor communication in DSM systems. Through a security analysis, we find that if coherence protocol anomalies are detected, only data messages need to be fully protected by encryption, integrity verification, and a mechanism against message replays. We propose three table-based DSM protection schemes to hide the cryptographic latencies of protecting processor-processor DSM data communication. We find that each scheme has advantages in various aspects of security, performance, flexibility, and scalability. However, these table-based, two-level protection schemes have some drawbacks in terms of performance inefficiencies and implementation issues. Therefore, we also developed a single-level DSM data protection scheme which addresses these drawbacks. We find that the single-level approach to data encryption and authentication in DSM systems is secure, while reducing the performance overheads of the two-level approach from 5.3% to 1.6% on average by eliminating the number of exposed cryptographic latencies and by reducing utilization pressure on the hardware cryptographic units.



## Chapter 4

# Conclusions

With increasing concerns for the security of today's computer systems, security has joined performance and power as primary considerations in the design of many systems. As the amount of valuable digital data that exists continues to grow, the variety of attacks has also increased along with the motivation and ability of attackers to target this information. Making matters worse, not only are traditional software-based attacks still prevalent, but the threat of sophisticated hardware-based attacks has emerged. Hardware attacks can subvert software protection of computer systems as they can operate at a lower level than software can control. As a result, researchers have investigated secure processor architectures as a means to address these security concerns. Through hardware-based security mechanisms, secure processors can ensure the confidentiality and integrity of computations, even in the face of such hardware attacks. This is done by encrypting and

authenticating all code and data in the system which leaves the security boundary of the processor chip. In this dissertation, we have proposed novel secure processor architectures for both single-chip, uniprocessor systems and distributed shared memory (DSM) multiprocessors. These designs address many important questions regarding performance and storage overhead, implementability, and complexity issues in the design of secure processors. The specific contributions and concluding findings of this dissertation are described as follows.

First, we proposed Address Independent Seed Encryption (AISE) and Bonsai Merkle Trees (BMT) for securing the data communicated between a processor and its main memory across the system bus. AISE eliminates critical system-level problems associated with prior counter-mode memory encryption schemes which use either virtual or physical address as a component of the encryption seed. AISE eliminates the use address as a fundamental security component, which allows AISE-enabled systems to support virtual memory mechanisms and shared memory Inter-Process Communication seamlessly. At the same time it retains the same low performance overheads as prior counter-mode memory encryption schemes. BMTs leverage counter-mode memory encryption to create a Merkle Tree in memory which covers only the counter values, while still providing the same level of protection as standard Merkle Trees. This results in a significantly smaller, shallower Bonsai Merkle Tree and significantly reduces the memory storage and execution time overheads associated with Merkle Tree based authentication. Additionally, we eval-

uate our AISE and BMT schemes on a Chip Multiprocessor simulation model, which has not been done in previous studies on secure processor architectures. We find that the BMT scheme reduces the performance overheads compared to standard Merkle Tree protection from 12% to 2% on average across the SPEC 2000 benchmarks. On multi-programmed workloads composed from these benchmarks, this overhead reduction is from 15% to 4%. This increase in memory encryption and authentication efficiency has a drastic impact on the feasibility of leveraging such protection in real systems.

The second major contribution of this dissertation is the development of the first secure processor architectures designed for providing private and tamper-resistant execution environments for DSM multiprocessors. Such a system requires not only protecting data communicated between a processor and its local memory, but also between processors across the interconnection network. We analyzed the security requirements for protecting DSM systems against hardware-based attacks, and determined what types of cryptographic protection are required for various types of messages. We also proposed initial table-based hardware mechanisms for ensuring the confidentiality and integrity of data communicated between processors. These schemes proved to be efficient in many cases, and to scale reasonably well to large system sizes. However, we also identified the ways in which such two-level protection mechanisms for multiprocessor systems have inherent inefficiencies. As a result, we proposed an alternative, single-level DSM protection scheme. This scheme alleviates the two-level scheme inefficiencies by reducing the number of cryptographic op-

erations on the critical path of a remote data request in a DSM system. This increases the chance for cryptographic latency hiding, and reduces the contention for the cryptographic units. Our experimental results show that the single-level scheme has an average overhead of only 1.6% across all SPLASH-2 benchmarks when compared to a similar but unprotected DSM system. Additionally, the low overheads are relatively constant across all individual benchmarks. This is crucial for a large-scale DSM multiprocessor which is likely to be used in a performance-critical environment where unnecessary performance overheads are not tolerable.

Secure processor architectures indeed appear to be a promising solution to address many important security concerns in our increasingly digital society. For example, by protecting the privacy and integrity of computations, secure processors can enable the prevention of the theft or tampering of confidential or sensitive data, the enforcement of Digital Rights Management, reverse engineering and software piracy prevention, and trusted distributed computing. Additionally, features of secure processor architectures are starting to appear in particular segments of computer systems, most notably in gaming consoles and some embedded devices. In this dissertation, we have investigated the design, security, complexity, and performance of secure processor architectures for general-purpose computing systems. Our findings indicate that the associated overheads of these designs are promising for enabling such techniques to be implemented in a wider range of systems to help attain the security goals mentioned above. This would have a significant social

and economic impact. Future research work will likely focus on continuing to reduce the complexity of secure processor architectures. Additionally, more precisely identifying the security requirements of various types of systems and tailoring secure solutions which meet these requirements with as little overhead as possible will be an important aspect of this effort as well. With such research work, and using the secure architectures introduced in dissertation, secure processor architectures may be realized in a wide range of processor systems.

# Bibliography

- [1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, New York, NY, 2006. ACM.
- [2] AMD. AMD Opteron Processor for Servers and Workstations. <http://www.amd.com/us-en/Processors/ProductInformation/0,,30-8796-8804,00.html>, 2005.
- [3] AMD. AMD64 Virtualization Codenamed Pacifica Technology: Secure Virtual Machine Architecture Reference Manual, 2005.
- [4] R. Anderson. Why cryptosystems fail. In *Proceedings of the 1st Conf. Computer and Communications Security (CCS '93)*, pages 215–227, 1993.
- [5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebar, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, New York, NY, 2003. ACM.
- [6] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzky, Shaz Qadeer, Barton Sano, Scott Smith, Roert Stets, and Ben Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proc. of the 27th International Symposium on Computer Architecture*, pages 282–293, New York, NY, 2000. ACM.
- [7] Doug Bartholomew. On Demand Computing – IT On Tap? <http://www.industryweek.com/ReadArticle.aspx?ArticleID=10303&SectionID=4>, June 2005.
- [8] Dwaine Clarke, G. Edward Suh, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Checking the Integrity of Memory in a Snooping-Based Symmetric Multiprocessor (SMP) System. In *MIT CSAIL CSG-TR-470*, July 2004.

- [9] Intel Corporation. Intel Virtualization Technology Specifications for the IA-32 Intel Architecture, 2005.
- [10] FIPS Publication 180-1. Secure Hash Standard. National Institute of Standards and Technology, Federal Information Processing Standards, 1995.
- [11] FIPS Publication 197. Specification for the Advanced Encryption Standard (AES). National Institute of Standards and Technology, Federal Information Processing Standards, 2001.
- [12] L. Gao, J. Yang, M. Chrobak, Y. Zhang, S. Nguyen, and H-H. Lee. A Low-cost Memory Remapping Scheme for Address Bus Protection. In *Proc. of the 15th International Conference on Parallel Architectures and Compilation Techniques*, 2006.
- [13] Tal Garfinkel, Mendel Rosenblum, and Dan Boneh. Flexible os support and applications for trusted computing, 2003.
- [14] B. Gassend, G. Suh, D. Clarke, M. Dijk, and S. Devadas. Caches and Hash Trees for Efficient Memory Integrity Verification. In *Proc. of the 9th International Symposium on High Performance Computer Architecture*, 2003.
- [15] T. Gilmont, J-D. Legat, and J-J. Quisquater. Enhancing the Security in the Memory Management Unit. In *Proc. of the 25th EuroMicro Conference*, 1999.
- [16] H. Krawczyk and M. Bellare and R. Canetti. HMAC: Keyed-hashing for message authentication. <http://www.ietf.org/rfc/rfc2104.txt>, 1997.
- [17] Steven M. Hand. Self-paging in the Nemesis operating system. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation*, pages 73–86, Berkeley, CA, 1999. USENIX Association.
- [18] A.B. Huang. *Hacking the Xbox: An Introduction to Reverse Engineering*. No Starch Press, San Francisco, CA, 2003.
- [19] Andrew "Bunnie" Huang. The Trusted PC: Skin-Deep Security. *IEEE Computer*, 35(10):103–105, 2002.
- [20] IBM. IBM Power4 System Architecture White Paper. <http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html>, 2002.
- [21] IBM. IBM Extends Enhanced Data Security to Consumer Electronics Products. [http://domino.research.ibm.com/comm/pr.nsf/pages/news.20060410\\_security.html](http://domino.research.ibm.com/comm/pr.nsf/pages/news.20060410_security.html), April 2006.
- [22] J. Renau et al. SESC. <http://sesc.sourceforge.net>, 2004.

- [23] T. Kgil, L. Falk, and T. Mudge. ChipLock: Support for Secure Microarchitectures. In *Proc. of the Workshop on Architectural Support for Security and Anti-Virus*, October 2004.
- [24] Markus G. Kuhn. Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP. *IEEE Transactions on Computers*, Oct(10), 1998.
- [25] A. Kumar. Discovering Passwords in Memory. [http://www.infosec-writers.com/text\\_resources/](http://www.infosec-writers.com/text_resources/), 2004.
- [26] James Laudon and Lawrence Spracklen. The coming wave of multithreaded chip multiprocessors. *International Journal of Parallel Programming*, 35(3):299–330, 2007.
- [27] M. Lee. Global ATM Security Alliance focuses on insider fraud. *ATMMarketplace*, <http://www.atmmarketplace.com/article.php?id=7154>, May 2006.
- [28] M. Lee, M. Ahn, and E.J. Kim. I2SEMS: Interconnects-Independent Security Enhanced Shared Memory Multiprocessor Systems. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 2007.
- [29] D. Lie, J. Mitchell, C.A. Thekkath, and M. Horowitz. Specifying and Verifying Hardware for Tamper-Resistant Software. In *Proc. of the 2003 IEEE Symposium on Security and Privacy*, 2003.
- [30] D. Lie, C.A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.
- [31] Linux Programmer’s Manual. <http://www.kernel.org/doc/man-pages/online/pages/man2/mmap.2.html>.
- [32] Michael R. Marty and Mark D. Hill. Virtual hierarchies to support server consolidation. In *Proc. of the 34th International Symposium on Computer Architecture*, pages 46–56, New York, NY, 2007. ACM.
- [33] Maxim/Dallas Semiconductor. DS5002FP Secure Microprocessor Chip. [http://www.maxim-ic.com/quick\\_view2.cfm/qv\\_pk/2949](http://www.maxim-ic.com/quick_view2.cfm/qv_pk/2949), 2007 (last modification).
- [34] David A. McGrew and John Viega. The Galois/Counter Mode of Operation (GCM). [http://csrc.nist.gov/Crypto\\_Toolkit/modes/proposedmodes/gcm/](http://csrc.nist.gov/Crypto_Toolkit/modes/proposedmodes/gcm/), 2004.
- [35] mod-chip.com. <http://www.mod-chip.com>.



- [36] T. Olavsrud. HP Issues Battle Cry in High-End Unix Server Market. *ServerWatch*, <http://www.server-watch.com/news/article.php/1399451>, 2000.
- [37] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kun-Yung Chang. The case for a single-chip multiprocessor. *SIGOPS Operating Systems Review*, 30(5):2–11, 1996.
- [38] W. Shi and H-H. Lee. Authentication Control Point and Its Implications for Secure Processor Design. In *Proc. of the 39th Annual International Symposium on Microarchitecture*, 2006.
- [39] W. Shi, H-H. Lee, M. Ghosh, and C. Lu. Architectural Support for High Speed Protection of Memory Integrity and Confidentiality in Multiprocessor Systems. In *Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques*, 2004.
- [40] W. Shi, H-H. Lee, M. Ghosh, C. Lu, and Alexandra Boldyreva. High Efficiency Counter Mode Security Architecture via Prediction and Precomputation. In *Proc. of the 32nd International Symposium on Computer Architecture*, 2005.
- [41] W. Shi, H-H. Lee, C. Lu, and M. Ghosh. Towards the Issues in Architectural Support for Protection of Software Execution. In *Proc. of the Workshop on Architectural Support for Security and Anti-virus*, 2004.
- [42] Silicon Graphics, Inc. SGI Altix 3000 Data Sheet. <http://www.sgi.com/products/servers/altix>, 2004.
- [43] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, 2005.
- [44] Standard Performance Evaluation Corporation. <http://www.spec.org>, 2004.
- [45] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proc. of the 17th International Conference on Supercomputing*, 2003.
- [46] G.E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processor. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, 2003.
- [47] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

- [48] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications., 2002.
- [49] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, 1995.
- [50] C. Yan, B. Rogers, D. Englander, Y. Solihin, and M. Prvulovic. Improving Cost, Performance, and Security of Memory Encryption and Authentication. In *Proc. of the International Symposium on Computer Architecture*, 2006.
- [51] B. Yang, S. Mishra, and R. Karri. A high speed architecture for galois/counter mode of operation (gcm). In *Cryptology ePrint Archive: Report 2005/146*, 2005.
- [52] J. Yang, Y. Zhang, and L. Gao. Fast Secure Processor for Inhibiting Software Piracy and Tampering. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, 2003.
- [53] Youtao Zhang, Lan Gao, Jun Yang, Xiangyu Zhang, and Rajiv Gupta. SENSS: Security Enhancement to Symmetric Shared Memory Multiprocessors. In *Proc. of the 11th International Symposium on High-Performance Computer Architecture*, 2005.
- [54] X. Zhuang, T. Zhang, H-H. Lee, and S. Pande. Hardware Assisted Control Flow Obfuscation for Embedded Processors. In *Proc. of the 2004 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2004.
- [55] X. Zhuang, T. Zhang, and S. Pande. HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. In *Proc. of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.