ABSTRACT


With the growth of digital information stored on modern computer systems, and the increased ability of attackers to target this wealth of information, security has taken a front seat in the design of systems today. One of the design goals of security systems is to protect the integrity and privacy of code executing on the system and prevent attackers from injecting and executing arbitrary code. This type of protection becomes particularly important for enforcing Digital Rights Management (DRM) (for example, in gaming systems), preventing reverse engineering and software piracy. Recently, in addition to software attacks, hardware attacks have emerged to the horizon and present a new set of security challenges for security architects. For example, the plaintext data stored in the main memory presents the attackers with a situation where they can easily scan and dump the contents of main memory, thereby, getting hold of potentially sensitive information such as user passwords and cryptographic keys. Despite the increased complexity of hardware attacks, they are very powerful as they can bypass any software security protection employed in the system. Since software security cannot prevent against hardware attacks, researchers have proposed secure processor architectures which employ hardware mechanisms for memory encryption and authentication to protect the privacy and integrity of applications’ code.
and data against these attacks. In this dissertation, we attempt to address the performance, system, and security issues with secure processor architectures making them practical for adoption in real systems as a powerful solution to system security.

First, we propose a secure processor architecture based on 1) Address Independent Seed Encryption (AISE), a counter-mode based memory encryption scheme using a novel seed composition, and 2) Bonsai Merkle Trees (BMT), a novel Merkle Tree-based memory integrity verification technique. AISE, being a counter-mode memory encryption scheme, effectively hides cryptographic latencies by overlapping the cryptographic work with off-chip data fetches. However, at the same time it eliminates critical system-level issues associated with prior mechanisms such as the lack of virtual memory support and inter-process communication based on shared memory. BMT is a Merkle Tree organization that retains the strong security properties of standard Merkle Tree protection, but provides this protection at significantly reduced execution time and memory storage overheads. Our results show that AISE+BMT reduces the overhead of prior memory encryption and integrity verification schemes from 12% to 2% on average for single-threaded benchmarks on uniprocessor systems, and from 15% to 4% for co-scheduled benchmarks on multicore systems, while eliminating critical system-level problems.

Second, we propose SHIELDSTRAP, a security architecture capable of booting a system securely in the face of hardware and software attacks targeting the boot phase. While secure processor architectures protect the steady state execution of
applications, many attacks (for e.g., modchips) target a system during booting before any employed security measure can take effect. SHIELDSTRAP bridges the gap between the vulnerable initialization of the system and the secure steady state execution environment provided by the secure processor. We present an analysis of the security of SHIELDSTRAP against several common boot time attacks and show that SHIELDSTRAP requires an on-chip area overhead of only 0.012% and incurs negligible boot time overhead of 0.37 seconds.

Third, we propose SecureME, a hardware-software mechanism that provides defense against hardware attacks and attacks against the Operating system (OS). Previously proposed secure processor architectures provide defense against hardware attacks but inherently assume a trusted OS. However, a complete security solution must protect applications against vulnerabilities in both the hardware and the OS. SecureME protects an application from hardware attacks by using a secure processor substrate, and also from the Operating System (OS) through memory cloaking, permission paging (a novel secure paging mechanism), and system call protection. Based on our performance evaluation using microbenchmarks, single-program workloads, and multiprogrammed workloads, we found that SecureME only adds a small execution time overhead, averaging 5.2% for SPEC2006 applications compared to a fully-unprotected system. Roughly half of the overheads are contributed by the secure processor substrate. SecureME also incurs a negligible additional storage overheads of 0.58% over the secure processor substrate.
Towards Performance, System and Security Issues in Secure Processor Architectures

by
Siddhartha Chhabra

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
2011

APPROVED BY:

[Signatures]
Dr. Yan Solihin
Chair of Advisory Committee

Dr. Gregory Byrd

Dr. Douglas Reeves

Dr. Eric Rotenberg
DEDICATION

To my parents
BIOGRAPHY

Siddhartha Chhabra was born in 1982 in Mumbai, India. He received the Bachelor of Technology (B.Tech) degree in Information Technology from Amity School of Engineering and Technology, G.G.S.I.P University, in 2003. He then received a Master of Technology (M.Tech) degree in Information Technology from Indian Institute of Information Technology (IIIT), Bangalore, in 2005. From 2005 to 2006, he worked as a Graphics Software Engineer at Intel Corporation, where he worked on development and maintenance of the user-mode component of Intel graphics device driver. Siddhartha joined the Computer Engineering department at North Carolina State University as a Ph.D. student in Fall 2006. He has worked as an Intern at Intel Corporation in the Circuits and Systems Research Lab in the summer of 2008, 2009 and 2010.

His research interests include providing architecture level solutions for system security and dealing with the design issues for such solutions. Specifically he is interested in processor only solutions for security and the interaction of processor architecture, Operating systems and the virtualization layer to provide system security. He is also interested in the security aspects of Non-volatile memories such as Phase-Change Memory (PCM) and Magnetoresistive Random Access Memory (MRAM). He has authored/co-authored several conference papers and journal articles in the field of Computer Architecture.
ACKNOWLEDGMENTS

First and foremost, I would like to thank my parents, Puran and Aruna Chhabra, for their unconditional and selfless love throughout my life. Without their support and guidance, I would not have achieved whatever I have. There were times when I did not believe in myself but they always believed in me and my abilities and made me believe in myself again and helped me carry on. This might sound like something straight from a bollywood movie, but there are no words that can express my gratitude for all that they have done for me. I dedicate this dissertation to them. I would also like to thank my brother Kapil for always being there as a friend, a guide, and a source of inspiration. I would also like to thank my uncle and aunt, Sanjeev and Suwarna Agarwal, my sister-in-law Radhika, my niece Sara and my younger brother Sidhant for their encouragement over the years. I would also like to thank my grandparents who could unfortunately not be here to see this day, but I am sure their blessings were always with me.

I sincerely thank my advisor Dr. Yan Solihin for his constant support and guidance throughout my Ph.D. study. He set very high standards for research which despite being challenging to meet made my study a satisfying experience. I have learned a lot from his experience and attitude towards research and thank him for providing me with an opportunity to work with him.

I would also like to thank my graduate advisory committee members, Dr. Gregory Byrd, Dr. Douglas Reeves and Dr. Eric Rotenberg for their invaluable and insightful
feedback and recommendations on my research work and dissertation.

It has been a pleasure to have known and worked with ARPERS members as fellow graduate students. I thank them all for their friendship: Seongbeom Kim, Fei Guo, Mohit Gambhir, Brian Rogers, Xiaowei Jiang, Fang Liu, Devesh Tiwari, Ahmad Samih, Anil Krishna and Ganesh Balakrishnan. I would also like to thank my colleagues at CESR: Rajeshwar Vanka, Niket Choudhary, Sandeep Navada, Muawya Al-Otoom and Salil Pant. The lunch/dinner sessions with Ahmad, Devesh, Rajesh, Niket and Muawya were really helpful even though they had nothing to do with computer architecture.

I am grateful to all my friends back in Bangalore and Delhi for always being there and for their constant encouragement and true friendship over the years. I intentionally did not mention names here as I did not want to miss any of them, but all I can say is, some people talk the talk, but we have walked the walk. Despite being over 8000 miles away, they have keenly followed my progress and were never short of an inspiring talk. Thank you all. Also to all my numerous friends in and out of North Carolina, thank you for your friendship and companionship through these years.

Finally, I would like to thank God for making this Ph.D. a part of my life and all that he has given to me. I am thankful to Him for giving me the strength and showering his love through a wonderful family and amazing friends.
# TABLE OF CONTENTS

LIST OF TABLES ................................................. x

LIST OF FIGURES ................................................. xi

Chapter 1 Introduction ............................................ 1

1.1 System-level and Performance issues .......................... 3

   1.1.1 Memory Encryption .................................... 4

   1.1.2 Memory Integrity Verification .......................... 7

   1.1.3 Secure CMPs ........................................... 8

   1.1.4 Energy-Efficient Secure Processor Architectures .......... 10

1.2 Security Issues .............................................. 13

   1.2.1 Secure Booting ....................................... 13

   1.2.2 SecureME: A Hardware-Software Approach to Full System Security ........................................ 18

Chapter 2 Making Secure Processors OS- and Performance- Friendly 25

2.1 Background and Related work .................................. 26

2.2 Attack Model and Assumptions ................................ 28

2.3 Memory Encryption ........................................... 29

   2.3.1 Overview of Counter-Mode Encryption .................. 29

   2.3.2 Problems with Current Counter-Mode Memory Encryption .. 33
3.3 SHIELDSTRAP architecture ........................................ 89
  3.3.1 SHIELDSTRAP .................................................. 89

3.4 Evaluation ............................................................. 102
  3.4.1 SHIELDSTRAP Security Analysis ............................ 102
  3.4.2 SHIELDSTRAP Area and Performance Evaluation ...... 110

3.5 Conclusions ............................................................. 115

Chapter 4 SecureME: A Hardware-Software Approach to Full System Security ................................. 116

  4.1 Background and Related Work ................................. 117

  4.2 Attack Model, Security Model, and Assumptions ............. 121
    4.2.1 Attack Model .................................................. 121
    4.2.2 Security Model ............................................... 122
    4.2.3 Assumptions ................................................... 123

  4.3 SecureME Design ................................................... 124
    4.3.1 Secure Processor Substrate ................................. 124
    4.3.2 Overview of SecureME ....................................... 126
    4.3.3 Protecting an Application from OS Vulnerabilities ...... 131
    4.3.4 Protecting An Application from Other Applications ..... 141
    4.3.5 Additional Protection for System Calls .................. 144
    4.3.6 Other Design Issues ......................................... 147
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4 SecureME Implementation</td>
<td>148</td>
</tr>
<tr>
<td>4.5 Experimental Setup</td>
<td>152</td>
</tr>
<tr>
<td>4.6 Evaluation</td>
<td>155</td>
</tr>
<tr>
<td>4.6.1 Microbenchmark Results</td>
<td>155</td>
</tr>
<tr>
<td>4.6.2 Real Benchmark Evaluation Results</td>
<td>156</td>
</tr>
<tr>
<td>4.6.3 Sensitivity Studies</td>
<td>162</td>
</tr>
<tr>
<td>4.6.4 Other Evaluation Results</td>
<td>162</td>
</tr>
<tr>
<td>4.7 Conclusions</td>
<td>164</td>
</tr>
<tr>
<td>Chapter 5 Conclusion</td>
<td>166</td>
</tr>
<tr>
<td>5.1 Performance and System-level Issues</td>
<td>167</td>
</tr>
<tr>
<td>5.2 Security Issues</td>
<td>168</td>
</tr>
<tr>
<td>Bibliography</td>
<td>172</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 2.1  MAC & Counter Memory Overheads  . . . . . . . . . . . . . . .  81
LIST OF FIGURES

Figure 2.1 Counter-mode based memory encryption. .......................... 30

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). ........................................ 34

Figure 2.3 Organization of logical page identifiers. ......................... 39

Figure 2.4 VMM Models .......................................................... 43

Figure 2.5 Portion of Page Fault Handling Routine of an AISE-compliant
OS. ................................................................. 44

Figure 2.6 Merkle Tree organization for extending protection to the swap
memory in disk. .......................................................... 52

Figure 2.7 Reduction in size of Bonsai Merkle Trees compared to standard
Merkle Trees. ............................................................. 58

Figure 2.8 Performance overhead comparison of AISE with BMT vs. the
Global counter scheme with a traditional Merkle Tree ................. 63

Figure 2.9 Performance overhead comparison of AISE versus the global
counter scheme .......................................................... 64
Figure 2.10 Performance overhead comparison of AISE with our Bonsai Merkle Tree vs. AISE with the Standard Merkle Tree .............. 65
Figure 2.11 L2 cache pollution ..................................................... 67
Figure 2.12 L2 cache miss rate and bus utilization of an unprotected system, standard Merkle Tree, and our BMT scheme ................. 68
Figure 2.13 Energy Overhead of AISE with standard Merkle Tree and our BMT scheme over an unprotected system .................... 69
Figure 2.14 Normalized Energy-Delay Product (EDP) for Standard vs Bonsai Merkle tree ....................................................... 70
Figure 2.15 Performance degradation comparison of AISE with our Bonsai Merkle Tree vs. 64-bit global counter scheme with the Standard Merkle Tree ....................................................... 71
Figure 2.16 Performance degradation comparison of AISE with our Bonsai Merkle Tree vs. AISE with the Standard Merkle Tree ....... 73
Figure 2.17 L2 cache pollution ..................................................... 73
Figure 2.18 L2 cache miss rate and bus utilization of an unprotected system, standard Merkle Tree, and our BMT scheme ............. 75
Figure 2.19 Energy Overhead of AISE with standard Merkle Tree and BMT over an unprotected CMP system ............................. 76
Figure 2.20 Performance overhead comparison across MAC size ........ 78
Figure 2.21 Sensitivity to cache size ............................................. 79
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Comparison of Secure Booting mechanisms with SHIELDSTRAP</td>
<td>87</td>
</tr>
<tr>
<td>3.2</td>
<td>SHIELDSTRAP: Secure bootstrap mechanism</td>
<td>91</td>
</tr>
<tr>
<td>3.3</td>
<td>SHIELDSTRAP: Architecture changes</td>
<td>100</td>
</tr>
<tr>
<td>3.4</td>
<td>Performance Evaluation of SHIELDSTRAP</td>
<td>114</td>
</tr>
<tr>
<td>4.1</td>
<td>Classification of related security works</td>
<td>118</td>
</tr>
<tr>
<td>4.2</td>
<td>Address Independent Seed Encryption (a), and Merkle Tree</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>used for integrity verification (b).</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>SecureME Architecture: hardware components (a), new instructions (b), and</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>the interaction of applications, the OS, hypervisor, and secure processor (c).</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Page fault handling mechanism for permission paging.</td>
<td>143</td>
</tr>
<tr>
<td>4.5</td>
<td>Temporary access mechanism for System Calls with read-only</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td>arguments (a), and read-write arguments (b).</td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td>Execution time overheads of SecureME for system-call intensive</td>
<td>156</td>
</tr>
<tr>
<td></td>
<td>Microbenchmarks(left), and real benchmarks(right).</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td>Execution time overheads of secure processor and SecureME</td>
<td>158</td>
</tr>
<tr>
<td></td>
<td>(a), and the breakdown of kernel and user-mode instructions (b).</td>
<td></td>
</tr>
<tr>
<td>4.8</td>
<td>L2 cache miss rate (a), and L2 misses per thousand instructions</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>(b), for secure processor vs. SecureME.</td>
<td></td>
</tr>
<tr>
<td>4.9</td>
<td>Execution time overhead breakdown for SecureME</td>
<td>161</td>
</tr>
</tbody>
</table>
Figure 4.10 Evaluation results for multiprogrammed workloads on a two-core CMP ............................. 161

Figure 4.11 Sensitivity to Cache size (a), and Sensitivity to AES latency (b). 163
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 topic. An important research effort towards such computer security issues focuses on protecting the privacy and integrity of computation 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 enforcement of Digital Rights Management, reverse engineering and software piracy prevention, and trusted distributed computing.

One important emerging security threat exploits the fact that most current computer systems communicate data in its plaintext form along wires between the processor chip and other chips such as the main memory. Also, the data is stored in its
plaintext form in the main memory. This presents a situation where, by dumping the memory content and scanning it, attackers may gain a lot of valuable sensitive information such as passwords [39]. Another serious and feasible threat is physical or hardware attacks which involve placing a bus analyzer that snoops data communicated between the processor chip and other chips [29, 30]. 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 proliferation of mod-chips that bypass Digital Rights Management protection in game systems has demonstrated that given sufficient financial payoffs, physical attacks are very realistic threats.

In addition, computing is increasingly becoming more dispersed, relying on various computer systems that are distributed physically, yet working together to solve a common problem. Examples include mobile devices (laptops, PDAs, cellphones), distributed computing, cloud computing, remote data center, etc. The increasing reliance on physically remote computer systems for solving a computing problem carries some security risks in that adversaries may obtain physical access to some of the computer systems through theft or security breaches. Example scenarios include an adversary stealing a company laptop in order to obtain secret company information, combat robotic vehicles confiscated by enemies who try to reverse engineer code or steal data stored in the vehicles, or even owners of gaming consoles who try to bypass security protection of the consoles.
Recognizing these threats, computer architecture researchers have recently proposed various types of secure processor architectures [24, 25, 40, 41, 60, 62, 63, 64, 65, 69, 70, 79, 80, 82]. Secure processors assume that off-chip communication is vulnerable to attack and that the chip boundary provides a natural security boundary. Under these assumptions, secure processors seek to provide private and tamper-resistant execution environments [70] through memory encryption [25, 40, 41, 60, 63, 64, 65, 69, 70, 79, 80, 82] and memory integrity verification [24, 41, 60, 62, 63, 65, 69, 70, 79, 82].

The chip industry also recognizes the need for secure processors, as evident, for example, in the recent effort by IBM in the SecureBlue project [31] and Dallas Semiconductor [45]. Memory encryption protects computation privacy from passive attacks, where an adversary attempts to silently observe critical information, by encrypting and decrypting 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 System-level and Performance issues

Current research on secure processor architectures [24, 25, 40, 41, 60, 62, 63, 64, 65, 69, 70, 79, 80, 82] has focussed primarily on providing various memory encryption and
integrity mechanisms, but, without considering system-level issues. Unfortunately, current memory encryption and integrity verification designs are not yet suitable for use in general purpose computing systems. In particular, we show in this work that current secure processor designs are incompatible with important features such as virtual memory, Inter-Process Communication (IPC), in addition to having large performance and storage overheads. The challenges are detailed as follows:

1.1.1 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 [60, 63, 64, 65, 70, 79, 80, 82], to ensure that pads are unique across different blocks in memory (spatial uniqueness), the block address is used as one of the seed’s components. 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 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 cache miss time, 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, however 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 [79], this solution creates a new set of serious 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). 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 Multi-Processor systems (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 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.

1.1.2 Memory Integrity Verification

Recently proposed memory integrity verification schemes for secure processors have leveraged a variety of techniques [24, 31, 41, 62, 65, 69, 70, 79]. However, the security of Merkle Tree-based schemes [24] has been shown to be stronger than other schemes because every block read from memory is verified individually (as opposed to [70]), and data replay attacks can be detected in addition to spoofing and splicing attacks, which are detectable by simply associating a single MAC per data block [41]. 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. 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, however as with memory encryption it requires the assumption of a trusted OS. Another option, as discussed in [69], is to associate one Merkle Tree and on-chip secure root per 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 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.

1.1.3 Secure CMPs

Increasing chip densities and transistor counts have provided microarchitects 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 [54]. 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 [54]. 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 [8].

Existing CMP designs [8, 53, 66] 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. However, CMPs present secure processor architects with new challenges primarily due the fact that the memory hierarchy design becomes even more critical. The limited 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 works on secure processors have not evaluated their mechanisms on CMP architectures.

1.1.4 Energy-Efficient Secure Processor Architectures

The hardware security mechanisms not only result in performance and storage overheads, but also increase the overall power consumption and based on the actual mechanisms used, this increase in power can be very significant. There has been prior work in designing low power security cryptographic algorithms, however, power issues have not been considered at the architecture level. If secure processors are to be adopted in future systems, it is critically important that power issues are considered in addition to performance and system-level issues.

Contributions. In this work, we investigate system-level issues in secure processors, and propose mechanisms to address these issues that are simple yet effective. Our 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 page offset of the block. Each 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 of this work 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 [67], along with reducing the storage overhead in memory from 33.5% to 21.5%.

Next, we present a complete energy evaluation of our schemes and show that, BMT memory integrity verification, in addition to reducing the performance and storage overheads, results in significant energy savings compared to prior schemes decreasing the overall energy overhead from 17.5% to 3.15% across all SPEC 2000 benchmarks.

Finally, we provide a complete evaluation of the proposed schemes for a CMP architecture. As our results show, the proposed mechanisms maintain a distinct advantage for secure CMPs, reducing the performance overhead from 15.1% to 4.1% and the energy overhead from 23.53% to 2%. 
1.2 Security Issues

1.2.1 Secure Booting

As discussed earlier, recognizing both the increased opportunity and motivation for attackers to compromise the privacy and integrity of data and computation, researchers have proposed secure processor architectures [24, 25, 31, 40, 41, 59, 63, 64, 65, 69, 70, 79, 80]. These architectures assume that the processor die provides a reasonable security boundary. Thus they protect any data stored off-die (including the main memory) with encryption and integrity verification. Unfortunately, these studies describe such mechanisms for steady state execution after the system is up and running. They do not address the threat of physical attacks during system boot. In this work, we address the problem of how to securely boot a secure processor system.

The goal of secure bootstrapping is to ensure that the system has booted using valid boot components including the basic input output system (BIOS) and operating system (OS). Recognizing the vulnerability of the bootstrapping process to the overall security of systems, prior work has proposed various secure bootstrapping mechanisms, such as AEGIS by Arbaugh et al. [5], Trusted Computing Platform [72], and others [33]. These approaches all follow a chained integrity verification approach where each layer in the boot process verifies the integrity of the next layer before passing control to it. However for each approach the root of trust is the BIOS code, which in most computer systems is located off-chip. Hence, they cannot ensure the
integrity of the bootstrapping process if physical tampering of the BIOS code is a possibility. Thus, they are inappropriate for implementation on a secure processor.

There have also been proposed solutions which do assume that physical attacks to off-chip components are possible. For example, in ARM TrustZone [6], security extensions to the ARM architecture, the entire boot code is moved on chip, eliminating the possibility of physical tampering to alter the boot code. Additionally, any application that needs to be secured must be kept on-chip in a secure RAM. While this solution is secure against the physical attacks we consider, it is too heavyweight for use in many systems. Especially for more general-purpose systems such as PCs and gaming systems, the cost in terms of die area of keeping the entire BIOS on-chip (which can be as large as 8MB) as well as the code and data of applications (possibly several MBs) is prohibitive. For this wide range of systems, a more lightweight and robust solution is needed.

It is clear that the root of trust for a secure bootstrapping solution should be located on-chip to provide strong security. However, an on-chip root of trust does not alone guarantee secure bootstrapping. We find that a class of attacks which we classify as a type of *Time-Of-Check To Time-Of-Use (TOCTTOU)* attacks are still possible. The main idea is that after a boot component has been verified, an attacker with physical access to the system can tamper with the component before it is actually fetched and executed by the processor. Hence, the processor will now execute an image that is different from the image that was originally verified. Therefore, as
another requirement, a secure bootstrapping mechanism should enforce that a boot component cannot be modified (without detection) after it is initially verified during secure boot and before it is executed by the processor. AEGIS by Arbaugh et al. and TCG do not meet this requirement and hence are susceptible to such TOCTTOU attacks. ARM TrustZone, on the other hand, meets this requirement since verified components never leave the processor chip, however again this is an expensive solution that is infeasible for many systems. As a result, the currently existing solutions to secure booting are inappropriate for implementation on secure processors.

Contributions. In this work, we propose SHIELDSTRAP, an architecture for protecting the integrity of the bootstrapping process of secure processors. SHIELDSTRAP is designed with three principle design goals: security, complexity and flexibility

Security: SHIELDSTRAP is designed to provide protection from sophisticated hardware-based attacks during system booting. Our solution to secure booting is based on the observation that any component off of the processor chip is relatively easy to compromise with hardware-based attacks. Whereas on-chip components are significantly more difficult to attack and can be made more difficult by various manufacturing techniques such as special coating [45]. Hence, the root of trust for the booting process should be located on the processor chip. SHIELDSTRAP uses a two-phased approach to boot the system to a trusted state: a verification phase and a booting phase. In the verification phase, SHIELDSTRAP verifies the integrity of
the BIOS using on-chip components (our root of trust) before passing control to the BIOS to start the actual booting phase. In the booting phase, we also follow a chained integrity verification approach where each layer verifies the integrity of the next layer in the boot chain before passing control to it. A failure at any point will prevent the system from booting. SHIELDSTRAP offers distinct security advantages over the previously proposed approaches for secure booting. First, SHIELDSTRAP protects systems from even sophisticated hardware attacks during booting. Hardware attacks against a boot component will be detected since that component’s integrity verification during our secure boot process will fail. This is accomplished with only lightweight additions to the processor and small changes to the boot procedure. On a reset (hard/soft), the processor instead of directly executing the BIOS, jumps to code stored in an on-chip memory which we call the SHIELDSTRAP ROM (ST-ROM). The ST-ROM is responsible for carrying out the verification phase to establish the integrity of the BIOS before handing over control to the BIOS to start the booting phase. This is the first work to bridge the gap between the vulnerable initialization of a system to the secure steady state execution environment provided by secure processors.

Secondly, SHIELDSTRAP leverages secure processor support to prevent TOCTTOU-style attacks during booting. After the integrity of a boot component is verified, integrity information about that component is retained on-chip such that any subsequent tampering with the component will be detected when the component is executed
by the processor at a later time. Lastly, similar to current solutions, SHIELDSTRAP continues to provide protection against the most widespread software attacks that target the boot components. We provide a security analysis of SHIELDSTRAP to motivate its security against several known boot-time attacks.

**Complexity:** We show how our SHIELDSTRAP mechanisms can be combined with typical memory encryption and authentication mechanisms provided by a secure processor to protect a system and the programs it executes all the way from power-on to steady-state application execution. In addition to enhancing the overall system security, this novel use of memory encryption and authentication for implementing a secure boot mechanism significantly reduces its complexity with respect to the on-chip storage overheads. The boot components can now be safely evicted to off-chip memory where they automatically come under the protection of the memory encryption and authentication mechanisms. This enables SHIELDSTRAP to have a very small on-chip area overhead making it suitable for use in both embedded and general-purpose systems. Our work explores the interaction of secure memory with a secure booting mechanism. We present a detailed analysis of the complexity and overheads of a secure booting mechanism with and without secure memory support. We also provide an area estimation for the amount of on-chip hardware that is required for SHIELDSTRAP and we find that this overhead is reasonable at less than 0.1% of the chip area. We also provide an evaluation to show that SHIELDSTRAP adds a negligible overhead of 0.37 seconds during system booting (which is not typically a
latency-critical process), compared to a base system with no secure booting.

**Flexibility:** Finally, SHIELDSTRAP is flexible, allowing for hardware and software reconfiguration without requiring changes for the end-user. For example, the user can install a new operating system (software reconfiguration) or install a new expansion card, such as a graphics card (hardware reconfiguration), without causing boot failures. We also explore the use of a group signature scheme, Direct Anonymous Attestation [9], to allow the system to use the BIOS from various vendors while ensuring the integrity of the secure booting mechanism, thereby preventing tying the processor manufacturer to specific BIOS manufacturers. However, our scheme places some requirements on the boot components designed to run on a secure system using the SHIELDSTRAP architecture. In particular, SHIELDSTRAP requires the boot components to be signed by their respective manufacturers before installation on the system. We discuss the particular requirements of each component in Chapter 3.

### 1.2.2 SecureME: A Hardware-Software Approach to Full System Security

If we assume that attackers can gain complete access to computer systems for which we want to provide secure computing environment, we must assume that not only all the off-chip hardware components but also all software components are vulnerable to both passive (eavesdropping) attacks as well as active attacks. Therefore,
the goal of a secure computing environment on untrusted computing node is to provide *privacy* (a guarantee that plaintext of code or data will not be leaked to the adversary) and *integrity* (a guarantee that program behavior or data cannot be modified by the adversary). Vulnerable components include many hardware components (all components outside of the processor die including the main memory, the system bus, I/O, etc.) as well as software components especially the Operating System (OS). Hence, a complete security solution *must* protect applications against vulnerabilities in *both the hardware and the OS*.

Most research in secure processor technology assumes untrusted hardware, where only the processor chip is assumed to be secure [24, 40, 41, 59, 60, 62, 63, 64, 70, 79, 80, 82]. Data in main memory or anywhere off chip is stored in encrypted form and protected by an integrity verification scheme. However, the system software, more specifically the Operating system (OS), is intrinsically trusted. For example, in most previously proposed secure processor architectures, an on-chip key is used to decrypt/encrypt data moving on/off the processor chip. Now, when the OS runs, it can access all the application pages in plaintext. A compromised OS can leak sensitive information such as user passwords, credit card numbers etc. directly or through a rogue application by mapping its pages to the physical pages of a security sensitive application, thereby rendering the security of the system worthless.

Providing complete security over untrusted computing nodes is a very challenging problem. At the heart of the problem is that in traditional systems, the OS is *im-
explicitly entrusted to manage the memory resources of the application. OS performs context switching, memory protection, memory allocation, initialization, sharing, etc. on behalf of applications. This automatically exposes an application to security vulnerabilities of the OS. One can envision an approach where core OS functionality like virtual memory management, context switching etc. is delegated to a security kernel. However, a security kernel can be quite large (e.g. 74K lines of code just for virtual management alone in AEGIS [69]) but simultaneously must be free from bugs and security vulnerabilities. As another option, one can envisage to restructure the OS such that key memory management functions are either disallowed, or relegated to special secure processor mechanisms, hence removing the implicit trust assumption completely. Such approach has been pursued in XOM OS [42] and AEGIS [69]. For example, regular OS mechanisms to allow sharing between processes (e.g. inter-process communication), copy on write, and dynamic linked library are not allowed under AEGIS, or special mechanisms are in place to bypass regular OS mechanisms in XOM OS. Such an approach requires significant modifications or restrictions to the OS, making it impractical to use in general purpose systems.

A promising alternative approach is to adapt the hypervisor-based approach in the untrusted-OS but trusted-hardware studies, such as Overshadow [11] and SP³ [81]. The approach is to avoid OS modifications, but before the OS performs any vital management functions, a hypervisor intercepts such events and hides the application data plaintext from the OS, using a technique called cloaking [11]. In this approach,
the OS is allowed to do its vital management functions, but on data that is already en-
crypted and hashed. Upon the completion of such functions, the hypervisor decrypts
the data and verifies its integrity, before returning control to the application.

Unfortunately, existing cloaking mechanisms do not provide protection against un-
trusted hardware and are incompatible with secure processor technology [24, 40, 41,
59, 60, 62, 63, 64, 70, 79, 80, 82] that protects against hardware attacks, for several
reasons. First, they keep each application data page in both encrypted and non-
encrypted forms in the main memory, making the non-encrypted pages vulnerable to
hardware eavesdropping. Secondly, they require direct encryption, where the encryp-
tion function is address-independent. That is, two ciphertext pages corresponding
to the same plaintext data must be identical, regardless of where the pages are lo-
cated in the main memory. This allows OS to operate solely on a ciphertext basis,
for example it can create a perfect copy of data by copying the ciphertext of a page
from one location to another. However, as pointed out in prior studies [63, 79, 80],
counter mode encryption has been shown to be superior to direct encryption in terms
of security protection and performance. Direct encryption provides weaker security
protection than counter mode encryption because the statistical distribution of the
ciphertext and plaintext are identical. In addition, direct encryption has been shown
to impose significant performance overheads of up to 35% [80] even when the decryp-
tion is implemented in hardware, due to the inability of the processor to overlap the
decryption delay with cache miss delay. These security and performance drawbacks
are eliminated when we use counter mode encryption [70, 80], which is unfortunately incompatible with current cloaking mechanisms.

The goal of this work is to investigate how to provide privacy and integrity to applications that run on untrusted computing nodes, where both hardware and the OS are vulnerable to security attacks. To achieve that, we propose an integrated security protection framework which we refer to as Secure My Execution (SecureME). SecureME consists of a secure processor technology controlled by small trusted software which can be integrated either through OS kernel dynamic instrumentation or as a hypervisor. In SecureME, an application can explicitly request the processor to protect its address space by invoking a special instruction that traps to the hardware. Unlike SP$^3$ and Overshadow, the entire address space of the application is cloaked (encrypted and hashed) at all time, avoiding hardware eavesdropping from discovering plaintext data. Only when the application accesses its own data, data is decrypted and its integrity checked on chip. SecureME is compatible with the state-of-the-art secure processor technology, which employs counter-mode encryption and Merkle Tree for integrity protection. We show mechanisms to ensure this compatibility. SecureME allows a commodity OS to be employed nearly unmodified, requiring only the addition of two virtual instructions to indicate the intent of the OS for page copying and initialization (discussed in Section 4.3). The addition of these virtual instructions can be automated, for example, through a compiler pass, and does not in any way change the OS functionality. Hence the OS is free to implement vital management functions
SecureME also provides an additional protection mechanism for system calls. By making system calls, an application explicitly trusts the OS to perform service with data it supplies or will receive. Thus, system call parameters must be uncloaked, and there is no guarantee that a compromised OS will perform the service correctly. However, SecureME ensures that the OS only accesses data it is supposed to (spatial protection), and only within the window of time of the system call invocation (temporal protection). Again, we demonstrate that such spatio-temporal protection can be achieved by leveraging secure processor mechanisms, without much additional complexities.

To evaluate the performance overheads of SecureME, we implement SecureME’s software component through dynamic instrumentation of Linux OS, and SecureME’s architecture component on detailed processor simulator based on Simics [43]. Our results show that SecureME adds only a small runtime overhead, averaging 5.2% for SPEC2006 applications over an unprotected system; of that, 2.2% is caused by the use of secure processor itself, while only the remaining (3%) is caused by SecureME mechanisms. Even on a multicore system consisting of two cores and two applications running on different cores concurrently, SecureME runtime overheads remain negli-
gible. Furthermore, SecureME incurs trivial storage overheads of 0.58% apart from the meta-data needed by the secure processors (counters and Merkle Tree nodes), in contrast to cloaking in software, which can incur storage overheads of up to 100%. Finally, SecureME mechanisms require only approximately 700 lines of code (vs >74,000 lines of code in secure kernel approach), making it much easier to guarantee a secure bug-free implementation.

The rest of the dissertation is organized as follows. Chapter 2 presents Address Independent Seed Encryption and Bonsai Merkle Trees as memory encryption and authentication mechanisms designed specifically to resolve the system-level and performance issues with currently proposed secure processor architectures. Chapter 3 presents SHIELDSTRAP, a secure bootstrap architecture, that can be used to boot secure uniprocessors securely. Chapter 4 presents SecureME, a hardware-software approach to defend against vulnerabilities in both the hardware and the OS. Chapter 5 concludes this dissertation with a summary of the work presented in this dissertation.
Chapter 2

Making Secure Processors OS- and Performance- Friendly

This chapter is organized as follows. We discuss related work in section 2.1. Section 2.2 describes our assumed attack model. Section 2.3 describes our proposed encryption technique while section 2.4 describes our proposed integrity verification techniques in detail. Section 2.5 shows our experimental setup, and section 2.6 discusses our results and findings. Finally, section 2.7 summarizes our main contributions and results for this study.
2.1 Background and Related work

Research on secure processor architectures [24, 25, 31, 40, 41, 60, 62, 63, 64, 65, 69, 70, 79, 80, 82] consists of memory encryption for ensuring data privacy and memory integrity verification for ensuring data integrity. Early memory encryption schemes utilized direct encryption modes [25, 40, 41, 69], in which a block cipher such as AES [21] 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) [80]. 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 [60, 63, 64, 65, 70, 79, 80, 82]. 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 [64, 70, 79, 80]. 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 [70, 79, 80] or predicted [64].

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 [41]. In [70], 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 [24]. 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. We note that the Log Hash scheme can also prevent replay attacks, but as shown in [65], the long time intervals between integrity checks can leave the system open to attack.
The proposed scheme in this study 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.

Most parts of Section 2.3 and Section 2.4 were done in joint collaboration with Brian Rogers. The discussion on support for virtualization, CMP evaluations, and the energy implications of using hardware mechanisms for security were added as further contributions to this chapter.

### 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 [22, 83, 84] can be employed in conjunction with our scheme.

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 in 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 \( P = C \oplus E_K(Seed) \).

![Counter-mode based memory encryption](image_url)

Figure 2.1: Counter-mode based memory encryption.

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. The reason for this is that suppose two blocks having plaintexts \( P_1 \) and \( P_2 \), and ciphertexts \( C_1 \) 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 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 [79], 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 [64, 70, 79, 80], 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. 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 [70, 79, 80] or predicted [64] at cache miss time.

However, this choice for seed composition has several significant disadvantages due to the fact that 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

---

1A 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.
encryption scheme, many problems arise for a secure processor when block address (virtual or physical) is used as a seed component. We discuss these problems in the next 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 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.

The use of physical address in the seed causes the following complexity and pos-
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).

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. We
present further details on the assignment of LPID to pages 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 [79]. 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 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 [79], 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 it into the physical memory. The block of counters (including LPID) are 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 [44], distributed web servers [74] and secure computing platforms [23] 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.

![Diagram showing Type 1 and Type 2 VMM models](image)

(a) Type 1 (Native VMM)  (b) Type 2 (Hosted VMM)

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 our discussion assumes a Type 1 VMM similar to the one used by VMWare’s ESX server [73] and Xen [7].

**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 reads (Load) 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 (CPL 0) 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.

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

**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), and 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* [28].

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 instructions 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 [14] and AMD [2] 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. We believe that when hardware supports memory virtualization, it
should be straightforward to extend the support with AISE. The current hardware allows the VMM to specify unconditional traps. Assuming that the 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.

2.3.7 Advantages of AISE

Our AISE scheme 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 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 [25], 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 [24]. 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 [69], 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 memory page, its page root is sufficient to verify the integrity of all values on the page. The internal nodes of the Page Merkle Subtree
(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.
can be re-computed and verified as valid by comparing the computed page root with the stored, valid page root. Secondly, 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 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 [24]. This allows the integrity verification of a data block to complete as soon as a needed MAC value is found cached on-chip. The reason be-
ing, 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 [63, 64, 70, 79, 80]. 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 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 would also be 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 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 3.4.2.

Overall, BMTs achieve the same security protection as in previous schemes where
(a) Standard Merkle Tree

(b) Bonsai Merkle Tree

Figure 2.7: Reduction in size of Bonsai Merkle Trees compared to standard Merkle Trees.
a Merkle Tree is used to cover the data in memory (i.e. data spoofing, splicing, and replay protection), but with much less overhead.

2.5 Experimental Setup

2.5.1 Machine Models

We use SESC [34], an open source execution driven simulator, to evaluate the performance of our proposed memory encryption and integrity verification approaches. For uniprocessor evaluations, 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 a 64B block size and use LRU replacement. We assume a 1GB main memory with an access latency of 200 processor cycles. We model a memory bus with a bandwidth of 10GBytes/s. The encryption/decryption engine simulated is a 128-bit AES engine with a 16-stage pipeline and a total latency of 80 cycles, while the MAC computation models HMAC [27] based on SHA-1 [18] with 80-cycle latency [36]. Counters are composed of a 64-bit LPID concatenated with a 7-bit block counter. So a 64B 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.

We use Wattch power models for our energy evaluations with an assumed feature size of 70nm. We only consider the increase in energy consumption of the pipeline and cache structures on the processor. In essence, we account for the increased miss rate of on-chip caches due to the sharing of cache space between application data and security metadata (MACs) and the energy consumption of the on-chip counter cache added for caching counters. We however do not account for the increase in energy consumption due to the operations of the on-chip cryptographic engine itself.

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 [67] for our uniprocessor 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 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 type of benchmark pairs, the benchmarks in a pair are both taken from the low miss rate group: `perlbmk_twolf` and `twolf_vpr`. In the second type of benchmark pairs, one benchmark in a pair is taken from the low miss rate group while another is taken from the high miss rate group: `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 type of benchmark pairs are ones in which both benchmarks in a pair are taken from high miss rate group: `apsi_art`, `apsi_equake`, `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, instructions are skipped only for the first benchmark in the benchmark pair and 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 purposes, we
use timely but non-precise integrity verification, i.e. each cache 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 simulations results for the performance and energy consumption of our AISE and BMT schemes on a uniprocessor system. Next we show parallel results of our schemes on CMP systems, and finally we present several sensitivity studies.

2.6.1 Uniprocessor Evaluation Results

Performance Results

In our first experiment, we compare 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, we next present figures which 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 [79], 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, our AISE approach is an attractive memory encryption option for secure processors.

![Figure 2.9: Performance overhead comparison of AISE versus the global counter scheme](image)

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 other studies [60, 79]. 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 [79] also obtained low average overheads with their memory encryption and integrity verification approach, however 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.

![Figure 2.10: Performance overhead comparison of AISE with our Bonsai Merkle Tree vs. AISE with the Standard Merkle Tree](image)

To understand why our 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 *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 our AISE+BMT scheme. 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.

Next, we look at the (local) L2 cache miss rate and bus utilization of the base unprotected system, the standard Merkle Tree, and our BMT scheme, 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%), our 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.

Energy Results

Figure 2.13 compares the energy overheads using standard Merkle Trees versus using Bonsai Merkle Trees as the integrity verification mechanism. Once again, both schemes use AISE as the memory encryption mechanism.

It can be seen from the figure that secure processor mechanisms using standard Merkle Tree as their integrity verification mechanism result in an average energy over-
Figure 2.12: L2 cache miss rate and bus utilization of an unprotected system, standard Merkle Tree, and our BMT scheme
Figure 2.13: Energy Overhead of AISE with standard Merkle Tree and our BMT scheme over an unprotected system.

Head of 17.5% across all SPEC2K benchmarks. For memory intensive applications like art, mcf, and swim, the overheads are in excess of 50% with mcf resulting in as much as 90.6% overhead over a system with no protection. These overheads are extremely high considering the fact that the power increase from one processor generation to another, accompanied by a significant performance improvement due to increased clock speeds, is roughly around 10%. For example, Pentium III, running at 500MHz, consumes 7.8% additional power compared to a Pentium II, running at 233MHz. Using BMTs reduces the overall energy overheads from 17.5% on an average to 3.15%, resulting in more than 5× energy savings. In addition, the maximum overheads suffered by memory intensive benchmarks are reduced from 90% to 18%. Hence, the much shallower and smaller BMTs resulting in significantly reduced cache pollution significantly reduce the overall energy overheads of a secure processor.
As seen before, in addition to reducing the energy overheads, BMTs also significantly reduce the performance overheads of the memory integrity verification mechanism. Figure 2.14 compares the Energy-Delay product (EDP) for a secure processor using standard Merkle Tree vs one using BMTs normalized to the EDP of a system with no protection. As can be seen from the figure BMT integrity verification mechanism has a normalized EDP of 1.05, while standard Merkle Tree integrity verification has a normalized EDP of 1.42. In essence, BMTs are very effective in reducing the overall performance as well as energy overheads of a secure processor with an EDP within 5% of a system with no protection.

![Normalized Energy-Delay Product (EDP) for Standard vs Bon-sai Merkle tree](image)

Figure 2.14: Normalized Energy-Delay Product (EDP) for Standard vs Bon-sai Merkle tree
2.6.2 CMP Evaluation Results

Performance Results

In our 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 2.15 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. 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 a 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.

![Figure 2.15: Performance degradation comparison of AISE with our Bonsai Merkle Tree vs. 64-bit global counter scheme with the Standard Merkle Tree](image)

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 2.16 shows the percentage degradation in combined IPC for AISE+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. As can be seen, AISE+BMT maintains a large 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. This is important since CMP systems [8, 53, 66] are likely to find widespread applications in server settings [44]. In such environments, excessive overheads due to security mechanisms are not likely to be tolerated. In addition to the lower average overhead, AISE+BMT achieves a much lower standard deviation of overheads of 5%, compared to that of AISE+MT which has a standard deviation of overheads of 14.5%. The lower average and standard deviation of overheads give server vendors more confidence about the stability of performance of their systems when they use AISE+BMT.

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

Finally, Figure 2.18 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 IPC degradation.

**Energy Results**

Figure 2.19 shows the energy overheads for the benchmark pairs used for the simulated CMP system.

As can be seen from the figure, majority of the pairs suffer from a high energy overhead, with the simulated benchmark pairs suffering an average energy overhead of 23.53% on an average. For memory intensive benchmark pairs, the overheads can be as high as 128% (mcf,art). Hence, the energy overheads of using standard Merkle tree with counter-mode encryption are significantly higher for CMPs that for the uniprocessor case. CMPs are likely to be used in server platforms where Energy (Power) consumption is even more important. A recent article [13] pointed out that power could cost more than the servers themselves. Hence, if hardware mechanisms for security are to be adopted for server platforms, it is even more important that their overheads in terms of power are brought down significantly.

As can be seen from the figure, BMTs are even more effective, compared to the
Figure 2.18: L2 cache miss rate and bus utilization of an unprotected system, standard Merkle Tree, and our BMT scheme.
Figure 2.19: Energy Overhead of AISE with standard Merkle Tree and BMT over an unprotected CMP system

uniprocessor case, in reducing energy overheads of CMP systems with the simulated benchmark pair suffering an energy overhead of 2% on an average representing 11× energy savings. The worst case energy overhead also declines steeply to 5.81% (compared to 128% with Standard Merkle trees). Hence, if hardware mechanisms for security are to be adopted for server systems utilizing CMPs, it is imperative that they use BMTs as their integrity verification mechanism.

2.6.3 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 consortiums 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 2.20 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 MACs cause serious performance degradation in standard Merkle Trees, they do not cause significant performance degradation for our enhanced BMT scheme.

Sensitivity to Cache Size

Figure 2.21 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 over-
Figure 2.20: Performance overhead comparison across MAC size

(a) Average Performance Overhead

(b) Average Cache Pollution
heads for both the schemes decreases, with standard Merkle Tree benefitting 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 which has very little thrashing to begin with. We make two other important observations from this figure. One, 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). Secondly, 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.

Figure 2.21: Sensitivity to cache size
2.6.4 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 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 2.1.

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 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).
Table 2.1: MAC & Counter Memory Overheads

<table>
<thead>
<tr>
<th></th>
<th>MT</th>
<th>Page Root</th>
<th>Counters</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>256b</td>
<td>global64+MT</td>
<td>49.83%</td>
<td>0.35%</td>
<td>5.54%</td>
</tr>
<tr>
<td>MAC</td>
<td>AISE+BMT</td>
<td>33.50%</td>
<td>0.51%</td>
<td>1.02%</td>
</tr>
<tr>
<td>128b</td>
<td>global64+MT</td>
<td>24.94%</td>
<td>0.26%</td>
<td>8.31%</td>
</tr>
<tr>
<td>MAC</td>
<td>AISE+BMT</td>
<td>20.02%</td>
<td>0.31%</td>
<td>1.23%</td>
</tr>
<tr>
<td>64b</td>
<td>global64+MT</td>
<td>12.48%</td>
<td>0.15%</td>
<td>9.71%</td>
</tr>
<tr>
<td>MAC</td>
<td>AISE+BMT</td>
<td>11.11%</td>
<td>0.17%</td>
<td>1.36%</td>
</tr>
<tr>
<td>32b</td>
<td>global64+MT</td>
<td>6.24%</td>
<td>0.08%</td>
<td>10.41%</td>
</tr>
<tr>
<td>MAC</td>
<td>AISE+BMT</td>
<td>5.88%</td>
<td>0.09%</td>
<td>1.45%</td>
</tr>
</tbody>
</table>

These overheads are still significant, however our scheme is compatible with several techniques proposed in [24] 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× less memory compared to global64+MT with 256-bit MACs with the gap widening to 2.3× with 32-bit MACs. Hence our scheme 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 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 overheads from 12.1% to 1.8% for single threaded SPEC 2000 benchmarks and from 15% to 4% for multi-programmed benchmarks, along with a reduction in storage overhead in memory from 33.5% to 21.5%.
Chapter 3

SHIELDSTRAP: Making Secure Processors Truly Secure

This chapter is organized as follows. Section 3.1 discusses background and related work on secure booting, Section 3.2 describes our assumed attack model. Section 3.3 describes our proposed SHIELDSTRAP architecture in detail. To evaluate SHIELDSTRAP, we first present a qualitative security discussion in Section 3.4.1. Then we present quantitative results to summarize the area overheads and boot-time performance overheads of SHIELDSTRAP in Section 3.4.2. Finally, we conclude in Section 3.5.
3.1 Background and Related Work

As discussed previously, due to both the increased opportunity and motivation for attackers to compromise the privacy and integrity of data and computation, researchers have proposed secure processor architectures [24, 25, 31, 40, 41, 59, 63, 64, 65, 69, 70, 79, 80]. These architectures assume that the processor die provides a reasonable security boundary. Thus they protect any data stored off-die (including the main memory) with encryption and integrity verification. Unfortunately, these studies describe such mechanisms for steady state execution after the system is up and running. They do not address the threat of physical attacks during system boot.

Recognizing the vulnerability of the bootstrapping process to the overall security of systems, prior work has proposed various secure bootstrapping mechanisms. Despite the growing importance of the problem, current solutions suffer from critical limitations in terms of security, cost, and flexibility. We categorize the previous works in to three categories.

In Category 1, we have solutions that have a low complexity but do not provide complete security. This is due to the fact, that these solutions rely on an off-chip component like BIOS to form the root of trust for the booting process [5, 33] which makes them vulnerable to hardware attacks. MIT AEGIS [69], on the other hand, proposes a secure boot mechanism where the public key of the security kernel manufacturer is embedded on the processor chip, and is used to authenticate the OS when entering
trusted execution mode. Other components involved in the boot process (e.g. BIOS, expansion ROMs) are not verified and thus are vulnerable to boot-time attacks. In particular, MIT AEGIS cannot provide defense against boot attacks targeting the BIOS (e.g. Modchip attacks), the expansion ROMs and MBR (e.g. rootkits persisted in expansion ROMs and MBR [56]). In essence, the attack model for MIT AEGIS does not consider hardware attacks on boot components to be a possibility. MIT AEGIS also lacks flexibility because the embedded OS-vendor public key prevents installing OSes from different vendors. In comparison, SHIELDSTRAP places the root of trust on-chip and verifies all components during the bootstrap process to protect the secure bootstrap process against even hardware attacks targeting any of the boot components. In addition, SHIELDSTRAP is also flexible in allowing hardware and software upgrades by decoupling the secure bootstrap mechanism from the system configuration.

In Category 2, we have solutions that offer low security and in addition have high complexity in terms of the hardware required. TCG [72] and Microsoft’s Next-Generation Secure Computing Base (NGSCB) fall in this category. TCG still relies on the BIOS to establish the root of trust and is optional for end users and NGSCB despite reducing the reliance on off-chip components continues to be optional for the end users allowing adversaries to bypass the security mechanisms. In addition, these approaches are hardware and software intensive, for example, NGSCB requires changes to the CPU, chipset, USB I/O and GPU components, in addition to requiring
a new OS component called the nexus. SHIELDSTRAP requires much more simple hardware and software changes and is an architectural mechanism which cannot be bypassed by users. IBM Cell BE architecture [32] supports the Runtime Secure Boot feature, which is used to check the integrity of application code as it is loaded or as it runs, but not the bootstrap of the system. The Cell security model, hence, does not protect against boot time hardware attacks. Modchips for Sony Playstation 3 using an IBM Cell processor will soon be available [55].

In Category 3, we have solutions that provide defense against hardware attacks but are hardware intensive and even infeasible to be used for general purpose systems. ARM TrustZone [6] falls in this category. In TrustZone, a new processor mode is introduced, called the Secure Monitor Mode, which is supported by the introduction of a new security bit (S-bit). The caches, TLB and MMU are tagged with the S-bit and partitioned to provide isolation between secure and non-secure applications. A secure application is executed from the on-chip Secure RAM and cannot be evicted from the chip to provide protection against TOCTTOU-style attacks. TrustZone also moves the root of trust on-chip into a ROM which stores the first level boot loader. This solution targets consumer products such as mobile phones and PDAs with small boot code requirements (8-16 KB) and applications with small code footprints. However, general purpose and gaming systems have typical BIOS sizes of 256KB and 1MB (with some of the future systems coming with as much as 8MB of BIOS) respectively, and generally execute much larger applications. The TrustZone approach is impractical
for these types of systems, while SHIELDSTRAP requires much more lightweight on-chip storage overheads of less than 0.1% on-chip area overhead.

We also note that there have been a variety of studies on secure processor architectures [24, 25, 31, 40, 41, 59, 63, 64, 65, 69, 70, 79, 80], which consist of hardware-based solutions for ensuring the confidentiality and integrity of computations even in the face of relatively sophisticated hardware attacks. While the proposed secure processor architectures enforce the confidentiality and integrity of computations at run-time, it is assumed that the system has been booted and initialized securely by including the booting process in the Trusted Computing Base (TCB). This assumption however can render the system vulnerable unless a secure booting mechanism is in place which is resistant to even hardware-based attacks as SHIELDSTRAP is.

The following figure summarizes the solutions to secure booting in terms of security and cost, and we show where SHIELDSTRAP fits.

![Figure 3.1: Comparison of Secure Booting mechanisms with SHIELDSTRAP](image_url)
3.2 Attack Model and Assumptions

As in all prior studies on secure processor architectures which consider hardware attacks, our attack model identifies two regions of a system. The secure region consists of the processor chip itself. We assume that the attackers cannot tamper with code or data stored on-chip (e.g. in registers or caches). The non-secure region consists of all off-chip structures including the buses, memory, disk, BIOS and expansion cards. Any data or code stored in any of these structures can be observed and modified by the attackers.

In this work, we focus on attack scenarios where the attacker has physical access to the system and can perform hardware-based attacks, including the attack scenarios where the owner of the system is the attacker itself. For instance, this is the case in gaming systems where the owners hack the system to bypass the security mechanisms used by the manufacturers to enforce Digital Rights Management. This enables the owners to use the system beyond the intended purposes for which the system was originally designed. Our proposed technique, however, is not restricted in application to gaming systems and is equally applicable to other embedded or general purpose systems. Finally, we also consider software boot attacks to be equally likely and continue to defend against them similar to prior solutions.
3.3 SHIELDSTRAP architecture

3.3.1 SHIELDSTRAP

In light of the problems with current approaches to secure bootstrapping and the growing need for such a mechanism, we propose our solution to secure booting, SHIELDSTRAP. SHIELDSTRAP is specifically designed to provide protection against hardware attacks and continues to provide effective defense against software attacks that target the boot components. SHIELDSTRAP uses an on-chip memory, which we call the SHIELDSTRAP ROM (ST-ROM) as the core root of trust to start our bootstrap verification process. We make no assumptions about the integrity of any of the off-chip components (including the BIOS). The proposed solution is based on the observation that the security attacks in prior solutions were made feasible because off-chip components, namely the BIOS, were included in the TCB. Moving the root of trust on-chip into the ST-ROM addresses this vulnerability. The proposed architecture necessitates several changes to the current boot process, and we discuss these later in this section.

The booting process with the SHIELDSTRAP architecture consists of two phases, the verification phase and the booting phase. On a hard or soft system reset, the verification phase is started where the ST-ROM reads the BIOS from an off-chip ROM and verifies its integrity by checking its signature. If the signature is verified successfully, the system transitions to the booting phase where control is passed to the
BIOS to start the booting process. In the booting phase, we follow a chained integrity verification where each layer verifies the integrity of the next layer in the boot sequence before passing execution control to it. This requires each component involved in booting, namely the BIOS, the expansion ROMs, the primary and secondary boot blocks and the OS image to be signed by their respective manufacturers. An exception to this requirement is the BIOS, which can either be signed by the manufacturer itself or by the processor manufacturer depending on the type of system in question as discussed later in this section. Figure 3.2 outlines the SHIELDSTRAP approach to secure booting.

Secure processors are equipped with hardware memory encryption and authentication mechanisms. Memory encryption provides protection against passive attacks, where the attacker tries to observe data communicated in plaintext between the processor and main memory. This is done by encrypting and decrypting data and code as it moves on and off the processor chip. Memory authentication is achieved by building a tree of MAC (Message Authentication Code) values over the memory (called a Merkle tree). The root of the tree is stored securely in an on-chip register and never goes off-chip. A block loaded from memory is verified for integrity by checking it against the chain of MAC values up to the root. SHIELDSTRAP leverages these mechanisms to significantly reduce the design and storage complexity of the secure booting solution. The boot components do not have to be retained on-chip for verification or execution and can be safely evicted to the main memory where
Figure 3.2: SHIELDSTRAP: Secure bootstrap mechanism
they automatically come under the integrity and confidentiality protection provided by the secure processor. In effect, SHIELDSTRAP effectively eliminates the window between the verification and start of execution of a boot component, *vulnerability window*, where an attacker with physical access to the system can change the boot component, thereby eliminating any possibility of TOCTTOU attacks. However, processors have the ability to read directly from the BIOS ROM chip allowing the code to bypass the memory hierarchy and thereby bypass the secure processor protection. Hence, we propose to make it mandatory to shadow the BIOS ROM and not provide it as an option. With shadowing always enabled, the BIOS will first be copied to the main memory before it can be executed by the processor to boot the system. This will ensure that the BIOS always comes under the integrity protection mechanism of the secure processor substrate used by SHIELDSTRAP. Most systems use this optimization to speed up accessing the ROMs and it is recommended to set this option on anyway. Hence, this requirement does not restrict the design of a system in any way. We describe one TOCTTOU attack in section 3.4.2 and show how SHIELDSTRAP defends against such attacks. We next discuss the requirements for each of the boot components.

**SHIELDSTRAP-ROM (ST-ROM)**

SHIELDSTRAP uses on-chip storage, the ST-ROM, as the root for establishing trust. The ST-ROM lies inside the security boundary of the processor chip. In this
section, we highlight the requirements and operation of the ST-ROM.

The ST-ROM stores the following components:

- Public key of the BIOS or processor manufacturer
- RSA [20] and SHA [18] implementations
- ST-code

**Public key of the BIOS or processor manufacturer**

One may assume that we would want to authenticate the BIOS by having the ST-ROM verify the BIOS signature using the public key \(<e,n>\) of the BIOS manufacturer which is stored on the BIOS itself and is signed with the private key of a Certificate Authority (CA). The public key of the CA would be stored on the ST-ROM. However, we note that simply proving that the BIOS is an authentic one that has been signed by the CA may not be the most desirable solution. The reason is that the CA will likely sign a large variety of software, and which software is signed is not under the control of any particular manufacturer. Thus, a modding attack, where the modchip is equipped with another BIOS that has been signed by the same CA, could be used to supply a BIOS to the processor that will pass the verification check of the ST-ROM, but does not contain the correct code to verify the rest of the components in the boot sequence.

Instead we propose that the ST-ROM should directly store the public key that will be used for establishing the authenticity of the BIOS. This ensures that when
the ST-ROM loads and verifies the BIOS against its digital signature, the BIOS is guaranteed to be one which was intended to run on the system and which contains the necessary code to verify the next components in the booting. An obvious choice is to store the public key of the BIOS manufacturer in the ST-ROM. This public key can then be used to authenticate the BIOS signed by its manufacturer. We believe storing the public key of the BIOS on-chip will be suitable for systems where the system manufacturer also manufactures the BIOS. For example, in gaming systems the system manufacturers produce the BIOS as well, since the BIOS is used for specialized tasks such as DRM enforcement. However, for other secure systems, requiring the public key of the BIOS manufacturer to be stored in the on-chip ST-ROM will reduce the flexibility of the system by tying the processor manufacturer to a particular BIOS manufacturer. In order to not trade flexibility for security, we propose using the processor manufacturer’s key to verify the authenticity of the BIOS. This in-turn would require the BIOS manufacturers to send their code to the processor manufacturer for signing before it could be used by the system. At boot time the processor’s public key stored in the ST-ROM will be used to verify the digital signature off the BIOS. We believe that this requirement is reasonable as processor manufacturers implementing a secure boot mechanism would anyways want to have some form of control over the system configuration. A similar model is currently being employed by the Windows Logo Program [47], where the driver manufacturers are required to get a certification from Microsoft before the driver could be installed on the operating system without
warnings.

*Group Signature scheme*: In order to further decouple the processor manufacturers from the BIOS manufacturers, we explore the use of a Group signature scheme like Direct Anonymous Attestation (DAA) [9] for verifying the authenticity of the BIOS. In DAA, a single group public key can correspond to multiple private keys, that is, a single group public key can be used to verify signatures signed by multiple private keys. For our purposes, the processor manufacturer can embed the group public key in the ST-ROM and each of the BIOS manufacturer, desiring to be compatible with the processor, can obtain a private key from the processor manufacturer to sign the BIOS code. This scheme retains the flexibility advantage of the scheme where the processor manufacturer’s public key is used to verify the BIOS, but at the same time, it also avoids the extra indirection, where the BIOS manufacturers need to get their BIOS signed by the processor manufacturers.

**RSA and SHA implementations**

We propose to verify the signatures of the boot components using software stored on the ST-ROM. This requires the ST-ROM to store an implementation of a digital signature algorithm. We use RSA for signature verification with the SHA algorithm used for generating the hash of the component. Our choice of RSA over Digital Signature Standard [19] was driven primarily by two factors: storage and speed. As we propose storing software implementations of the algorithms on-chip, the algorithm chosen should be space efficient. Also, it is desirable that the algorithm is faster for
the verifier than for the signer. Since each boot component is only signed once by the
manufacturer, but the signature is verified on each boot attempt, this helps to lower
the execution time overheads due to our secure bootstrapping mechanisms. RSA with
SHA compares favorably to DSS with SHA in terms of both storage and verification
speed, hence our choice of RSA plus SHA. Performing the signature verification in
software minimizes additional hardware needed for the proposed SHIELDSTRAP
architecture, but at the same time will be slower than a corresponding hardware
implementation. However, we show in our results that using a software approach adds
an insignificant performance overhead to the boot time compared to a base system
with an unprotected boot process.

**ST-Code**

The pseudo code for the algorithm in the ST-Code is shown below:

```plaintext
// Verification Phase

// Verify the BIOS
biosSignature = read BIOS signature from BIOS
publicKeyBIOS = read public key from ST-ROM
verified = VerifySignature(biosSignature, publicKeyBIOS)
If(verified == false) {
    BOOT FAILURE
}
```
else {
    //Pass control to the BIOS to start the Booting phase
}

VerifySignature(signature, publicKey) {
    verificationBlock = first 512 bits from the BIOS;
    runningHash = hash(verificationBlock);
    While (!endOfComponent) {
        verificationBlock = next 512 bits from the BIOS;
        runningHash = updateHash(verificationBlock)
    }
    expectedHash = hash from the signature using publicKey
    if (expectedHash == runningHash) {
        return true
    }
    else {
        return false;
    }
}

On a system reset, the processor jumps to execute the ST-ROM code. The ST-ROM code which we call the SHIELDSTRAP Code (ST-Code) is a simple algorithm that
carries out the verification phase. The ST-Code starts by reading the signature of the BIOS. A digital signature is generally not computed over the entire range of code/data itself, but instead a much smaller representative form of the code/data is signed. The representative form we use is the running hash of the code/data. This means that a hash should first be computed on the BIOS code and this hash must then be signed with the private key of either the BIOS manufacturer or the processor manufacturer depending on the flexibility requirements of the system as discussed above. The ST-Code will read the BIOS location by location and compute a running hash on it. Once the hash for the BIOS code is obtained, the ST-Code proceeds to verify the computed running hash. The ST-Code uses the public key stored in the ST-ROM to get the expected hash from the signature. If the running hash matches the expected hash, the ST-Code concludes that the BIOS has been verified successfully, and control is passed to the BIOS to start the booting phase. The operations carried out for verifying the BIOS signature can be summarized as follows:

\[
\begin{align*}
\text{public_key} & \leftarrow < e, n > \\
\text{expected_hash} & \leftarrow \text{bios_signature}^e \mod n \\
\text{running_hash} & \leftarrow \text{HASH}(\text{BIOS code}) \\
\text{running_hash} & \equiv \text{expected_hash}
\end{align*}
\]
where, $<e,n>$ is the public key stored in the ST-ROM, $expected\_hash$ is the hash obtained from the signature of the BIOS using the public key and $running\_hash$ is the hash computed by ST-Code over the BIOS that is being verified.

In the event of an integrity failure at any layer, the bootstrap process is halted. We note that preventing the system from booting up in the event of failure and not recovering (as in [5]) may lead to denial of service. However, for the attack model we consider where the attacker has physical access to the system, causing a boot failure on integrity check failures will prevent the attack from being successful. This forces the attacker (the user/owner) to restore the original hardware configuration of the system, thereby preventing the system from being used for purposes other than those intended.

Figure 3.3 shows the minimal architectural modifications required for SHIELD-STRAP. Over a secure processor architecture, we only need to add an on-chip memory, the ST-ROM, which forms the root of trust for the bootstrap process.

**SHIELDSTRAP: Boot component requirements**

**Requirements of BIOS manufacturer**

SHIELDSTRAP requires either the BIOS or the processor manufacturer to sign the BIOS code using their private key. Additionally all further BIOS upgrades should also be signed by the BIOS or processor manufacturer. Signed upgrades are necessary to ensure the system boots up properly on legitimate BIOS upgrades. The ST-ROM
Figure 3.3: SHIELDSTRAP: Architecture changes

will store the corresponding public key of the BIOS or processor manufacturer and verify the digital signature of the BIOS before handing over control to the BIOS to start the normal booting process of the system.

Requirements of other boot components

The other components involved in the booting chain include the expansion ROMs, the primary boot block, the boot loader (in multi-boot systems) and the OS image. SHIELDSTRAP requires the manufacturers of these boot components to sign their components and include their public key signed by a certificate authority with the component. The boot components involved in verifying other components should also include the public key of the certificate authority to verify the signed public key of the next component in the boot chain. We chose this approach as opposed to requiring the boot components in one layer of the boot chain to store the public keys of the components in the next layer to favor flexibility without sacrificing security. If
A boot component stores the public key of the next layer’s components, it limits the flexibility of the system, as a user cannot add new software or hardware components without requiring the lower layer component to be upgraded as well to include the public key of the new component manufacturer.

A Note on Impact of SHIELDSTRAP on usability

The security provided by SHIELDSTRAP does have an impact on the normal usage model for end users. For example, a user trying to upgrade a boot component must ensure that the boot component is signed by the manufacturer and includes its public key signed by a CA. In other words, the boot components must now be SHIELDSTRAP compatible to ensure that the system boots up. In addition, with SHIELDSTRAP in place, a user will not be able to boot a compromised system even if the incompatibility was due to a user error with no intent to bypass the security measures, for example, an erroneous upgrade. The intent behind this design is to ensure that a system with a configuration (intentional or unintentional) that can possibly result in compromising the security of the system should not be allowed to boot up.
3.4 Evaluation

We present a qualitative and quantitative evaluation of SHIELDSTRAP in this section. The primary evaluation of a secure bootstrap mechanism is the ability to defend against boot time attacks, which can either be hardware or software attacks. Hence, we first analyze SHIELDSTRAP qualitatively, by presenting a detailed security analysis of the proposed SHIELDSTRAP secure boot architecture in section 3.4.1.

SHIELDSTRAP adds an on-chip ST-ROM that forms the root of trust for the booting process. Hence, it is critically important for the on-chip area overhead to be within reasonable limits for the proposed scheme to be practical for implementation. Also, despite the fact that booting is not a time critical process, it is desirable that the proposed mechanism does not add an undue amount of time to the boot process. Hence, we qualitatively analyze SHIELDSTRAP by presenting an area and performance analysis in section 3.4.2.

3.4.1 SHIELDSTRAP Security Analysis

In this section, we present a security analysis of the proposed SHIELDSTRAP secure boot architecture and present five case studies which show how SHIELDSTRAP can protect against common boot time attacks.

Since the verification phase checks the BIOS against its signature using the trusted, on-chip ST-ROM, the attacker cannot conduct any attacks before the verification
phase is complete, as it will result in a boot failure due to a signature mismatch. Once the verification phase is complete, control can be passed to the now trusted BIOS which is executing under protection from the secure processor. Thus, any attacks against the BIOS after this point will be detected by the secure processor mechanisms. As such, the BIOS can be trusted to verify the next components in the boot process against their signatures, and subsequently pass control to them as they are now verified and running under the protection of the secure processor. This procedure continues until all boot components have been verified and executed and the OS has been loaded.

We note that not all attacks are detected by the SHIELDSTRAP mechanism itself as a signature failure. The SHIELDSTRAP architecture is used to ensure that all the boot components present in a system, on a reset, have not been tampered, and that they are loaded and execution is started under the control of the secure processor. Once the underlying secure processor takes over, it will detect any attacks that occur against already running programs. Hence, SHIELDSTRAP in conjunction with a secure processor architecture results in end-to-end system security. We now present three case studies to motivate the security of SHIELDSTRAP against hardware attacks. We also present two case studies to show the effectiveness of SHIELDSTRAP against software attacks.
Hardware Attacks

Case study 1: Modding Attack

Attack details: The modding attack has been one of the most widespread boot-time attack used by attackers to bypass various protections enforced in systems. For example, in a gaming system like Xbox, when the system is booted up the BIOS tries to read the disk in the drive. If a disk is found in the drive, the system proceeds to verify the digital signature off of the disk. Only if the verification is successful does the system play the disk. This prevents users from playing pirated or illegally manufactured games for instance. However, some game enthusiasts managed to obtain the Xbox BIOS and modify it. The hacked BIOS performed everything that the original BIOS did, except that it did not have the code for verifying the digital signatures from disks. An alternate form of the modding attack involves storing an alternate BIOS on the modchip, which is capable of booting up the system. Then this alternate BIOS is fed to the processor during boot time. The Cromwell BIOS has been widely used by modders as a replacement BIOS for gaming systems. In either case, the new system has no measures for DRM enforcement. Thus it can be used to play any arbitrary disks including but not limited to pirated games, booting an OS on the system, installing bigger hard drives, and storing games on the system’s hard drive.

Attack penetration: Mod-chips are available for all popular gaming systems in-
cluding the Microsoft Xbox, Sony Playstation, and Nintendo Wii. They are available for all versions of the gaming systems released by their respective manufacturers, including the Xbox 360 and Sony Playstation 3 [3, 49, 48, 50, 51, 75, 76, 77, 78]. Despite the seriousness of the situation, modding continues to be a large problem as no effective solution prior to SHIELDSTRAP has been proposed to provide protection for systems under the control of malicious users from hardware-based attacks.

**SHIELDSTRAP’s attack prevention:** When the system boots up with a modchip bypassing the BIOS, the verification phase will read the BIOS stored on the modchip, as the modchip is imitating the functional BIOS of the system. The modchip can either have a legitimate BIOS, or it can be a cut down version of the original BIOS which does not have the code for verifying the digital signatures from disks played on the system. In either case, when the ST-ROM reads the BIOS to verify its signature, it will fail the verification phase as the modchip manufacturers can in no way forge a signature of the new, modified BIOS based on the private key of the original manufacturer. Hence, SHIELDSTRAP targets the root of the problem and renders such modchips ineffective by causing boot failures and forcing the owner to restore the original configuration of the system.

**Case study 2: TSOP Flashing Attack**

**Attack details:** The flashing attack was the most widespread attack in earlier versions of a few gaming systems. We explain this attack taking the example of an Xbox console. In earlier versions of Xbox (Version 1.0 to 1.5), the BIOS was stored on a
chip called TSOP. Write-access to the chip was disabled before it was distributed to customers. However, it was possible to re-enable the write-access by soldering a few points on the motherboard. With write-access enabled, the BIOS could be replaced with an alternate BIOS, such as the Cromwell BIOS which is capable of loading Linux.

**Attack penetration:** This attack was mostly prevalent in earlier versions of Xbox (Versions 1.0 to 1.5) where the BIOS was stored on the TSOP. Microsoft recognized the problem and made the BIOS non-writable ensuring that the attackers cannot flash the BIOS with an alternate one.

**SHIELDSTRAP’s attack prevention:** When the system boots up with the new BIOS loaded to the BIOS TSOP chip, the verification phase reads in the modified BIOS and performs a signature verification. The alternate BIOS will fail the signature verification even if the BIOS was signed by its manufacturer. This is due to the fact that SHIELDSTRAP’s ST-ROM stores the public key of the BIOS manufacturer on-chip and uses it to verify the signature. Hence, even an alternate BIOS which legitimately signed by its own manufacturer will also not be able to bypass SHIELDSTRAP’s security protections.

Case study 3: A possible Time-Of-Check To Time-Of-USE Attack (TOCT-TOU)

**Attack details:** The phased approach (verification then booting) that SHIELDSTRAP follows may lead one to believe that it may be vulnerable to possible TOCT-
TOU attacks. In such an attack, an attacker could attach a device on the bus connected to the BIOS. The attack could be thought to proceed as follows. Verification phase completion is signalled by the processor when it starts to execute the BIOS in the booting phase. A snooping device attached on the control bus could snoop this action from the processor, and at this time redirect fetch requests from the processor to a modchip containing an alternate BIOS. This attack would trick the processor into executing a malicious BIOS, which it thinks it has already verified as trusted.

**SHIELDSTRAP’s attack prevention:** When the ST-ROM reads the BIOS in the verification phase, the integrity verification mechanism of the underlying secure processor is initiated. As such, the BIOS comes under the protection of the secure processor. As the ROMs are always shadowed, when the BIOS starts executing in the booting phase, it will be copied to the memory and go through the entire memory hierarchy. This ensures that the BIOS is protected by the memory protection mechanisms of the secure processor. Thus, the executed BIOS is guaranteed to be the same as the one that was checked in the verification phase. Hence, such TOCTTOU attacks will be detected by the secure processor as an integrity violation which will cause the system to signal an alert and halt execution.

**Software Attacks**

**Case study 1: PCI rootkits**

*Attack details:* Many PCI cards contain expansion ROMs that hold code required
for the initialization of the card during boot up. During the boot process, the PCI devices are enumerated and the expansion ROM code (if present) is copied to the main memory for execution. Expansion ROMs are attractive to attackers for persisting rootkits primarily for two reasons. First, expansion ROMs are generally stored on an EEPROM device which can be erased in-circuit, electrically and re-flashed by the operating system. Thus, a successful remote attack can re-flash the expansion ROM and persist the rootkit without physical access to the system. Second, rootkits can successfully hide in the expansion ROMs as expansion ROMs are generally not checked for code integrity before execution. Rootkits once installed can turn off all security mechanisms employed by the system, for example, the access control mechanisms implemented by the operating system.

*Attack penetration:* [35] demonstrates an attack which uses a PCI expansion ROM for persisting rootkit code and concludes that all flashable devices are vulnerable to such attacks in the current systems which do not perform any verification prior to execution.

**SHIELDSTRAP’s attack prevention:** The expansion ROMs carry the code to initialize the device. The rootkit code can be simply flashed on the ROM which might prevent the device from starting up. A more sophisticated technique uses a compression algorithm to first compress the expansion ROM code which in turn frees up some space on the ROM for the rootkit code. In either case, the rootkit code will be different from the original ROM code. SHIELDSTRAP requires the expansion
card manufacturers to sign their code. In the booting phase, when SHIEDLSTRAP verifies the signature of the boot components, the expansion card with the rootkit code will fail the signature verification which will result in a boot failure.

**Case study 2: MBR rootkits**

*Attack details:* The MBR is normally stored on the first sector of a storage device used to boot the system. MBR rootkits operate by making a copy of the original MBR on some other sector of the storage device, storing malicious instructions in the original MBR location. During bootup, the BIOS passes control to the MBR (first sector) for execution, which in this case, is the malicious code. The rootkit can safely hide its existence by redirecting accesses to the first sector to the sector where the MBR is actually stored.

*Attack penetration:* [15] presents a proof-of-concept boot sector rootkit. In [56] rootkits in the MBR are identified as one of the major sources of security threats for the year 2008.

*SHIELDSTRAP’s attack prevention:* The malicious code replaces the first sector of the boot device which otherwise stored the original MBR code. This code once executing redirects all accesses MBR to the sector where the original MBR is stored. This prevents any runtime mechanism to detect the presence of the rootkit. However, in SHIEDLSTRAP, the MBR of the boot device is first verified for integrity before the control is passed to it for execution. Hence, the malicious code installed in place of the original MBR will be fail the signature verification in the booting phase.
3.4.2 SHIELDSTRAP Area and Performance Evaluation

Area Overhead

To evaluate the on-chip area required for the ST-ROM, we implemented SHA and RSA in software with space efficiency as the primary goal. The object code sizes for SHA and RSA that will be stored on the ST-ROM measure 8266 bytes and 4894 bytes respectively. Along with the implementations of these cryptographic algorithms, the ST-ROM also needs to store the public key of the BIOS which takes up another 1024 bits (128 bytes), and the ST-Code which uses the above implementations to read in the BIOS and carry out the verification phase. Hence, the total area required for ST-ROM is at most 13288 bytes (13KB), so we propose adding a 16KB on-chip ST-ROM.

The on-chip ROM can be implemented using either the MOS NOR ROM implementation having a cell size of $9.5\lambda \times 7\lambda$ or the MOS NAND ROM implementation having a cell size of $5\lambda \times 6\lambda$, where $\lambda$ is $1/2 \times$ feature size. The dimensions for the two implementations have been taken from [57]. We assume a MOS NOR ROM implementation for ST-ROM to derive an upper-bound on the area overhead. Using Mosis scalable CMOS rules [52], and a feature size of 90 nm, the area taken by the 16KB ST-ROM is $0.017 \ mm^2$. On an Intel Pentium 4 with a die area of $143 \ mm^2$ [38], this amounts to an on-chip area overhead of $0.012\%$. Hence, SHIELDSTRAP requires minimal on-chip area overheads and should be feasible to incorporate onto the
Experimental Setup

We use SESC [34], a cycle accurate, execution driven simulator, to model a secure processor substrate for this research. The secure processor simulated is based on counter mode encryption and Merkle tree authentication mechanisms. 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, while the MAC computation models HMAC [27] based on SHA-1 [18] with 80-cycle latency [36]. The default MAC size is 128 bits.

We simulate the cryptographic operations for SHIELDSTRAP using an open source cryptographic library, Crypto++ version 5.5.1 [16], which provides the implementation for RSA and SHA algorithms used for signature verification. The RSA algorithm is based on Public-Key Cryptographic Standards (PKCS) version 1.5 [20], which defines the RSA encryption standard. The SHA algorithm is implemented
based on the standard published by NIST [18]. We use key lengths of 1024-bit for RSA and SHA-256 which generates a 256-bit authentication code used for signing purposes. As discussed previously, the choice of these cryptographic algorithms was based on two factors other than the proven security strengths of these algorithms: Storage and speed.

**Boot-Time Overhead**

Even though booting is not a time critical process, bootstrapping should be fast enough to give a good end-user experience. We modeled the SHIELDSTRAP booting process using the Crypto++ cryptographic library. We assume a system with a 256KB BIOS, two expansion cards with 32KB of expansion ROM on each, Master Boot Record of 512 bytes, a boot loader with a size of 200KB (most common boot loaders have a a size ranging from 150KB - 200KB), and an OS image of size 2MB (the compressed image size for linux kernel 2.6.9). We implemented code to model the verification phase of SHIELDSTRAP (e.g. reading the boot components, computing their running hash value, and verifying their signature), and we have timed this process on a real system to get an approximation of the boot-time overhead that SHIELDSTRAP will add to a base system with no protection. We find that SHIELDSTRAP will add a negligible overhead of 0.37 seconds on average relative to the normal boot process.

Further, to approximate the impact of the interaction of SHIELDSTRAP and
a secure processor on the boot-time overhead, we ran the code modeling SHIELDSTRAP under SESC. We note that, in a real system, some of the boot components like the OS image, the boot blocks etc. will be loaded from the hard disk. Due to the lack of a hard disk model in our simulator, we model the components as being read from the main memory instead of the disk. However, we further note that as accesses to hard disk take significantly longer than accesses to main memory, the relative overhead for SHIELDSTRAP will further go down if hard disk were modeled. Hence, our simulations provide something of an upperbound on the overhead due to SHIELDSTRAP. Figure 3.4(a) shows the overhead of our scheme normalized to the AEGIS approach. As can be seen from the figure, SHIELDSTRAP adds an overhead of less than 0.8% over AEGIS. This overhead comes primarily from two sources. Firstly, SHIELDSTRAP runs under a secure processor which encrypts and verifies the integrity of off-chip data. Secondly, we propose storing the signed public keys of boot components other than the BIOS on the components themselves to allow for system reconfiguration. Hence, SHIELDSTRAP requires first verifying the component key before proceeding to verify the component itself. In AEGIS, the component keys themselves are stored in the BIOS or the AEGIS ROM. Although this avoids the need to verify the public keys of the boot components, it complicates system reconfiguration since changes in the system configuration will result in a boot failure. Figure 3.4 also shows the overhead of a scheme which does not utilize the secure processor as a substrate. This scheme can be used for a class of systems where hardware attacks
once the system has been powered on are not likely or not beneficial for the attacker.

Finally, Figure 3.4(b) presents the breakdown of overheads over the different boot components. As expected, the OS takes the most time for verification (up to 70% of the total verification time) since it is the largest boot component. We believe that the already insignificant overheads of our scheme will be even lower for specialized systems like gaming and embedded systems as they generally employ a significantly reduced version of the operating system kernel compared to those used in general purpose systems.

(a) Performance overhead of SHIELDSTRAP over AEGIS

(b) Overhead breakdown across the boot components

Figure 3.4: Performance Evaluation of SHIELDSTRAP
3.5 Conclusions

Despite the increasing number of attacks which target systems during the boot phase in order to subvert various security mechanisms employed by the system, there have been few studies on secure booting mechanisms. In addition, prior secure booting approaches are vulnerable in the scenario where the user of the system is considered the attacker and can conduct hardware-based attacks. We propose SHIELDSTRAP, a novel secure booting mechanism that is secure even against relatively sophisticated hardware attacks. Our key insight in SHIELDSTRAP is that we move the root of trust for verifying the integrity of all boot components onto the processor chip which is our natural security boundary. We analyze how SHIELDSTRAP is secure against common boot attacks. We also show that SHIELDSTRAP requires an acceptably small amount of on-chip hardware, and that it adds an insignificant amount of execution time overhead to the normal boot process.
Chapter 4

SecureME: A Hardware-Software Approach to Full System Security

This chapter is organized as follows. Section 4.1 presents a background on the problem and discusses the related work. Section 4.2 discusses our attack model and assumptions. Section 4.3 presents SecureME design. Section 4.4 describes the implementation details for SecureME. Section 4.5 describes our experimental setup, and Section 4.6 discusses our evaluation results and findings. Finally, we conclude in Section 4.7.
4.1 Background and Related Work

In Chapter 2, we presented Address-Independent Seed Encryption and Bonsai Merkle trees to address system-level and performance issues with previously proposed secure processor architectures. We then identified a critical security hole with prior schemes in Chapter 3 in that prior schemes on secure processor architectures work to protect the steady state execution of a system and assume that the system has booted up securely by including the booting process in the trusted computing base. We showed that with the increasing number of attacks targeting the boot phase, this assumption can render the steady state security of the secure processor worthless by making the hardware mechanisms of the secure processor protect a system that has already been compromised in the boot phase. We proposed SHIELDSTRAP to secure the vulnerable initialization of a secure processor.

In this chapter, we identify another security issue with previously proposed secure processor architectures. Software attacks have formed the most widely exploited category of attacks so far. In particular OSes, which are complex programs and often contain millions of lines of code, offer an inherently broad surface to attackers forming a large portion of the system’s code base and once compromised, allow the attacker complete access to system’s critical secrets. However, previously proposed secure processor architectures implicitly trust the OS. For example, in most previously proposed secure processor architectures, an on-chip key is used to decrypt/encrypt data.
moving on/off the processor chip. Now, when the OS runs, it can access all the application pages in plaintext. A compromised OS can leak sensitive information such as user passwords, credit card numbers etc. directly or through a rogue application by mapping its pages to the physical pages of a security sensitive application, thereby compromising the security of the system. Hence, a complete security solution should provide defense against vulnerabilities in both the hardware and the OS. However, much research has assumed a model in which only the hardware or the OS is vulnerable. Figure 4.1 shows a classification of related security works and forms the basis of our discussion in this section.

![Figure 4.1: Classification of related security works.](image)

**Secure Processor Technology (Cluster A).** Research on secure processor architectures [24, 31, 40, 41, 59, 60, 62, 63, 64, 69, 70, 79, 80, 82] assumes that the processor die boundary provides a natural secure boundary, and off-chip components are vulnerable. Secure processors use memory encryption for ensuring data *privacy* and memory integrity verification for ensuring data *integrity*. While secure proces-
sor architectures provide defense against hardware attacks, they implicitly assume a trusted OS.

**Dealing with Untrusted OS but Trusted Hardware (also Cluster A)** Hypervisors or Virtual Machine Monitors (VMMs) running directly on bare hardware (e.g. VMWare ESX Server [73] and Xen [7]) have become popular. Several hypervisor-based security protection schemes have been proposed, assuming software-based attacks but not hardware attacks. SecVisor [61] is a tiny hypervisor designed to ensure kernel code integrity against software attacks. Proxos [71] is a hypervisor-based trust partitioning system. It partitions the system into multiple Virtual Machines (VMs), one running an untrusted full-blown OS and the others running trusted, application-specific OSes. Chen et al. [11] and Yang et al. [81] propose *multi-shadowing*, that presents different cryptographic views of the physical memory (ciphertext or plain-text) depending on whether the data is accessed by the application or by the OS. McCune et al. propose Flicker [46] which relies on a secure co-processor like TPM [26] to provide a secure execution environment. Flicker provides protection from other software and OS, however, it requires the applications to be re-written to identify the code segments requiring security and linked against the Flicker library.

**Dealing with Untrusted OS on Secure Processor (Cluster B).** Lie et al. proposed XOM OS [42], an OS for XOM secure processor. To deal with the untrusted OS assumption, XOM provides compartments to fully isolate one process from others. Suh et al. proposed AEGIS [69], and presented two alternative implementations.
One implementation requires a secure implementation of OS called secure kernels, and another assumes untrusted OS and enforces isolation between processes similar to compartments in XOM OS. A protected application has a compartment ID that is stored as tags for processor registers and all cache blocks. The processor prevents a process to access registers or cache blocks with a different compartment ID. Hence, even if the OS is compromised and allows a malicious application to map its address space to a good application, the malicious application still cannot read the data of the good application due to the compartment ID mismatch. However, compartments are incompatible with how a modern OS manages protection and sharing between processes, through virtual memory management features such as inter-process communication, copy on write, and dynamic linked library. AEGIS prohibits such OS features, while XOM OS uses special hardware and OS mechanisms to provide the features. Significant modifications to the OS make the system incompatible and less applicable for general purpose systems, and may present backdoors to new security vulnerabilities.

**Dealing with Untrusted OS and Untrusted Hardware (Cluster C).** The closest related work to ours is Bastion [10], a hypervisor based solution that assumes untrusted OS and hardware. Bastion security protection revolves around a *security module*, an encapsulation of code section, data it uses, access permission for data, that forms the basic unit of security protection. To enjoy the processor and hypervisor protection from hardware/OS vulnerabilities, an application programmer must
identify statically a module and the permission (read/write/execute) for all pages that the module uses. Module information is given to the compiler which generates a security compartment for that module, which is enforced by the hypervisor and secure processor at run time.

4.2 Attack Model, Security Model, and Assumptions

4.2.1 Attack Model

The attack model that we are interested in is a case in which attackers gain full physical access to the computer systems. The main goal of attackers will be to steal secret data that is processed by the applications. In order to do that, the attackers may mount physical attacks on the hardware (such as by inserting a bus snooper or memory scanner to read the plaintext data stored in off-chip memory, or communicated between the processor and off-chip memory), or mount software attacks on the OS in order to leak secret data that belongs to an application process. Alternatively, the attackers may attempt to change the code or data of the application in order to modify its behavior so that it leaks out its secret data. Therefore, the goal of a secure computing environment on untrusted computing node is to provide privacy (a guarantee that plaintext of code or data will not be leaked to the adversary)
and integrity (a guarantee that program behavior or data cannot be modified by the adversary). We do not consider denial of service attacks or side channel attacks in this chapter.

### 4.2.2 Security Model

Our goal is to provide privacy and integrity protection for an application from hardware and OS vulnerabilities. There are at least two possible security models to achieve this. One possible security model is one proposed by Bastion [10] where users must define a security module, which is a basic unit of security protection that encapsulates code section, data it uses, and access permission for data. With Bastion, the critical question is who should determine modules and what should be the optimum module granularity. If a program is broken into many small modules, figuring out which modules should be allowed to share data or not is tricky. On the other hand, encapsulating the entire program as a single security module presents intractable operational challenges such as (1) Data Sharing: the application must conform to share-everything or share-nothing semantics of a module. (2) System calls: to avoid the OS from having access to the entire application’s data, code implementing system calls must be broken into modules with different security permissions, which complicates OS design. (3) Copy-on-write: The application will need to provide access to its entire address space to the OS to allow this optimization on forking a process. (4) Dynamic memory allocation: Since the location of dynamically allocated memory
is not known at compile time, it is infeasible to define permissions for data residing on the heap using Bastion, hence critical data cannot be placed in the heap or else its security is compromised. Therefore, we view Bastion’s module-based security model to be more suitable for well-defined functions such as subroutines or functions that perform cryptographic algorithms, but is too restrictive for protecting user data or an entire application.

Thus, SecureME uses a security model in which it automatically protects the entire address space of an application from the OS through cloaking. An application must explicitly ask for hardware protection at the start of program by requesting a hardware ID. If two applications want to establish inter-process communication, they have to explicitly request for shared memory to be established.

### 4.2.3 Assumptions

We assume that a proper infrastructure is in place for a hypervisor and applications to be distributed to the end users securely, for use on secure processors. In general, we assume that a feature similar to IBM Cell runtime secure boot [32] is present on the system which verifies an application each time it is launched. Hence, an attempt to modify the application before launch will be caught as an integrity failure and the application will not be launched. We also assume a secure booting infrastructure utilizing establishing a chain of trust and verification to be used [5, 12], to prevent against any attacks that may happen during or before the booting of a
system, and launching of the hypervisor and applications. These features ensure that an application/system has not been compromised before launch/bootup and before SecureME runtime mechanisms can take effect to protect the system.

4.3 SecureME Design

4.3.1 Secure Processor Substrate

In this section, we provide an overview of the secure processor used by SecureME. The secure processor substrate used by SecureME uses Address-Independent Seed Encryption (AISE), presented in Chapter 2 for counter-mode encryption, and Merkle tree for integrity verification [24]. We present it here in order to make the discussion on SecureME mechanisms clearer. In counter mode encryption, data block is not directly encrypted or decrypted. Instead, a seed is encrypted to generate a pseudo random pad that is then XORed with data block in order to encrypt (or decrypt) the data block. Since a data block is not needed for generating the pad (which is the time consuming step), decryption delay can be overlapped with cache miss delay, which is a unique feature of counter-mode encryption.

Figure 4.2(a) illustrates the components of a secure processor, shown in gray boxes. Suppose a data block is evicted from the last level cache, the physical address of the access is sent to the counter cache (Step 1). A counter cache stores a collection of blocks, each block containing a logical per-page identifier (LPID), and
multiple per-block counter values. LPID is unique for a page, assigned at the first allocation of the page, and its value is unaffected by where the page may be located (in physical memory or swap space in disk), and is unrelated to the address of the application. The encryption seed for the block is a concatenation of the page’s LPID, the block’s counter, page offset, and some initial vector (Step 2). The LPID and page offset give the seed spatial uniqueness (each block has a unique LPID and page offset combination), while the per-block counter gives the seed temporal uniqueness (each time a block is written back to the main memory, the block’s counter is incremented to ensure no seed reuse is possible). The encryption seed of a block is input to the encryption engine in order to obtain a pseudo-random pad (Step 3). The pad is XORed with the data block in order to encrypt it. The ciphertext itself is fed into the message authentication code (MAC) generator to produce a hash code (Step 4). Finally, the hash code updates the Merkle Tree, to provide a full integrity protection (Step 5).

Figure 4.2(b) shows a Merkle Tree used for integrity verification. A Merkle Tree is a tree of hash codes where a parent node contains a hash of all its children nodes. A Merkle Tree has been shown to offer the strongest protection for data as it protects against splicing, spoofing, as well as replay attacks [24]. Since a hash code is smaller than the data it protects (e.g. 128 bits of hash code for a 64-byte block), ultimately at some level there is only one root node. This root node covers the entire tree, and is always stored on-chip in order to avoid hardware attacks from tampering it. The
intermediate nodes can be stored on the L2 cache on-chip as needed. By allowing intermediate nodes to be cached, when a data block is modified (read) and is evicted from the cache, the Merkle Tree is updated (verified) until the first parent node that resides on chip, not all the way to the root. Merkle trees with better performance characteristics have been proposed [24, 59]. However, since Merkle Tree is not a focus of this chapter, we use a standard Merkle Tree with 128 bits of hash code covering each 64-byte cache block.

### 4.3.2 Overview of SecureME

Our goal in designing SecureME is to provide security protection against compromised hardware as well as a compromised OS, without requiring modifications to a contemporary OS. The key challenge in providing complete security over untrusted
computing nodes is the fact that in most computing systems, the OS is *implicitly entrusted* to manage the memory resources of the application. OS performs context switching, memory protection, memory allocation, initialization, sharing, etc. on behalf of applications. This automatically exposes an application to security vulnerabilities of the OS. Thus, protecting an application from OS vulnerabilities conflicts with allowing the OS to perform its basic role in managing the application’s memory.

There are basically two approaches to solve the conflicting goals. One approach takes away the role of the OS in managing the application’s memory, and either relegates the role to processor mechanism [42, 69], or ban its use [69]. Such an approach requires significant modifications or restrictions to the OS, making it impractical for general purpose computing systems. Another approach is to *cloak* data of the application before allowing the OS to manage it [11, 81]. In this approach, the OS is allowed to do its memory management functions, but on data that is already encrypted and hashed. Upon the completion of such functions, the hypervisor decrypts the data and verifies its integrity, before returning control to the application. The latter approach requires a thin layer of software between the hardware and the OS, typically inserted as a hypervisor. The root of trust includes the hypervisor.

To provide memory cloaking, SecureME assigns a hardware ID (HID) to an application requesting for security protection, and assigns a different one to the OS. HIDs are directly managed by the hypervisor and the processor, and are immutable from the point of view of the applications or the OS. When memory accesses are associated
with the HID of the OS, rather than that of the application, data that is accessed is cloaked (i.e. encrypted and hashed). The OS is allowed to perform virtual memory management functions according to its own policies, but can only access cloaked (i.e. encrypted) data.

SecureME architecture is illustrated in Figure 4.3. Figure 4.3(a) shows hardware components added to the processor, which mainly consists of re-encryption tracking table to keep track of the progress of page re-encryption, and a control logic that can selectively turn on or off various components of the cryptographic engine. In addition, each cache block is tagged with an “encryption bit” that distinguishes whether a block is stored in the cache in plaintext or ciphertext form.

Figure 4.3: SecureME Architecture: hardware components (a), new instructions (b), and the interaction of applications, the OS, hypervisor, and secure processor (c).

SecureME introduces five new instructions (Figure 4.3(b)). The first two instructions (SECSAVE and SECRESTORE) are to enable secure context saving and restor-
ing in the event of a context switch. They are typically already a component of a secure processor. The next instruction CRYPTO enables the hypervisor to independently turn on or off various parts of the cryptographic engine: the encryption/decryption engine, Merkle Tree update, Merkle Tree integrity checking. The last two instructions allow the hypervisor to encrypt or decrypt some memory address range. The purpose of per-component control and selective encryption/decryption is to provide (1) cloaked access to the OS and (2) additional protection for system calls. Note that protecting against hardware attacks requires the main memory to be encrypted at all time (versus maintaining plaintext and ciphertext versions of a single page in Overshadow [11] or SP3 [81]). Hence, cloaking is achieved by turning off the encryptor/decryptor, and Merkle Tree integrity verification and update, when the OS runs. In addition, we will show that providing spatio-temporal protection for system calls can be achieved simply through controlling various components of the cryptographic engine, deviating the need to explicitly validate memory buffers.

SecureME also introduces four virtual instructions. These instructions do not need to be implemented in the instruction set of the processor. They can use unused opcode, which will cause the application or OS to trap to the hypervisor with an Invalid opcode fault when executed. GET_HID and TERM_HID are instructions executed by the application to trap to the hypervisor, which then assigns or releases the hardware ID of the requesting application. PAGEINIT and PAGECOPY are instructions that are used to annotate page initialization and copying that occur in the virtual mem-
ory management of the OS. Annotating the OS with these instructions is the only change required for the OS, and can be performed with dynamic instrumentation at post-development time.

Figure 4.3(c) shows how the application, the OS, hypervisor, and the secure processor interact, through the execution of various instructions and the occurrence of various processor events. Application’s activities that require the protection of hypervisor always trap to the hypervisor (obtaining and releasing HID, as well as system calls). The hypervisor can control various components of the cryptographic engine, and can instruct the engine to provide secure context switching (to avoid the OS from reading plaintext values in the register file). The OS traps to the hypervisor whenever it attempts to modify an application’s page table. It also traps to the hypervisor when it executes virtual instructions PAGEINIT and PAGECOPY in order to pass information of the occurrence of the events to the hypervisor. Finally, several data structures are maintained by the hypervisor. CurHID keeps track of the hardware ID of the currently running application. Permission domain table keeps track of what pages an application is allowed to map to its address space. This is used in permission paging to support inter-process communication between multiple applications. Finally, system call APIs are the specifications of OS system calls’ interface that include what arguments are used, what memory buffers may be read or written by the OS during the system call execution. The purpose of this is, because an application explicitly trusts the OS to perform a service when it incurs a system call, arguments
cannot be cloaked. Hence, the only protection that is possible, short of validating the semantics of each system call, is to ensure that the system call execution only accesses data that the application intends the OS to access, and only within time duration of the system call. To achieve such protection, SecureME requires all system calls’ interface to be specified to the hypervisor.

4.3.3 Protecting an Application from OS Vulnerabilities

To ensure the privacy and integrity of an application’s data from a compromised OS, SecureME relies on hardware-based memory cloaking. Memory cloaking is a concept introduced in Overshadow [11] and SP$^3$ [81], where the OS is given access only to application data’s ciphertext. Cloaking is enabled based on the observation that for most memory management functions (such as page allocation, copying, and swapping), the OS does not need have an access to the plaintext data of the application. Hence, the OS can carry out most of its functions on cloaked (i.e. encrypted and hashed) data. It may seem straightforward to implement software-based cloaking of Overshadow or SP$^3$ on the secure processor substrate to achieve the same security goal of SecureME. However, the fundamental incompatibility in their attack models makes it hard or even impractical to combine them directly. For example, imagine an access to an application page which is stored in ciphertext in the main memory by the Overshadow mechanisms, because it was previously accessed by the OS. This access will first be decrypted in software by Overshadow mechanisms and the processor
will decrypt it again using its own key before supplying to the processor, resulting in unnecessary power and performance overheads. In addition, software-based cloaking can result in storage overheads of up to 100%, because an application page is kept in both encrypted and non-encrypted forms to minimize performance overheads. Finally, software-based cloaking intrinsically requires direct encryption, which is address independent, to allow OS to operate on pages solely on ciphertext basis. However, the secure processor substrate uses counter-mode encryption due to its performance and security advantages and hence are fundamentally incompatible with current software-based cloaking mechanisms in Overshadow or SP$^3$. XOMOS [42] provides an example where an attempt to add protection against OS attacks to XOM resulted in requiring significant OS redesign.

In contrast, SecureME is designed from bottom-up to make the interaction between the secure processor substrate and hypervisor synergistic, in order to provide the desired protection while meeting the other goals of low performance overheads, compatibility and ease of programming. Unlike Overshadow and SP$^3$, SecureME’s cloaking leverages hardware mechanism already present in the secure processor substrate, by selectively turning on or off various components of the secure processor’s cryptographic engine.

A pre-requisite for providing memory cloaking is that we must distinguish between when an application needing protection is running and when the OS is running. To achieve that, an application requesting protection by SecureME is required to ob-
tain a unique identifier to be associated with it. We refer to this unique identifier as a *hardware identifier (HID)*. A HID must be immutable by the OS, hence it must be assigned and managed by the hypervisor and the processor. The OS itself is given its own unique HID to distinguish it from applications being protected.

To obtain a HID, a secure application executes a special instruction `GET_HID`, which, because it is a virtual instruction, generates an *Invalid Opcode fault*. This fault traps to the hypervisor which then assigns a new HID to the application and initializes its *permission domain table*. HIDs do not need to be unique across system reboots and can be reused by the hypervisor. Hence, the hypervisor can maintain a software counter to assign HIDs to requesting applications. In addition, the hypervisor also keeps track of the HID of the currently running context.

A HID is saved and restored by the hypervisor across interrupts or context switching. An application can give up its HID upon exit by calling another virtual instruction which we refer to as `TERM_HID`. The addition of `GET_HID` and `TERM_HID` to the beginning and the end of the application is the only change required by SecureME for an application which requests for security protection. Unlike modules in Bastion [10] which require programmers to identify static code, data, and security permission to encapsulate as a security module, in SecureME programmers (or compilers) can obtain protection for the program’s entire data, oblivious of various types of code and data structures used in the program.

Note that the notion of cloaked access includes not just privacy protection, but
also integrity protection. In SecureME, the on-chip cryptographic engine alternates between two modes: *cloaked mode* and *uncloaked mode*. In an uncloaked mode, all parts of the cryptographic engine are turned on: encryption/decryption, integrity verification, and Merkle Tree update. Data that is brought on chip is decrypted, and its integrity is verified against the Merkle Tree. If data is modified, the Merkle Tree is updated to reflect the new data. In a cloaked mode, all parts of the cryptographic engine are turned off. Ciphertext data fetched from off-chip memory is not decrypted, hence the OS only views the ciphertext of data. Integrity verification is not performed. And finally, if the OS modifies some of the application’s ciphertext data, the modification does not result in the Merkle Tree being updated. Hence, if a compromised OS modifies cloaked data, the illegitimate modification is detected later when the application accesses the data, and the integrity verification of the data against the Merkle Tree fails. Therefore, cloaked access only grants read permission on the ciphertext form of data.

The hypervisor controls the transition from cloaked to uncloaked mode and vice versa, using the cryptographic control instruction, i.e. CRYPTO<> , which enables the hypervisor to independently turn on or off various parts of the cryptographic engine: the encryption/decryption engine, Merkle Tree update, Merkle Tree integrity checking. When the processor switches mode from user to kernel, for example, the mode switch traps to the hypervisor, which then remembers the new HID as the currently running HID (CurHID).
Note, however, that data that is already cached (i.e. not fetched from off-chip memory), already resides in a certain form, either in plaintext form (if brought in by the application to whom the data belongs), or in ciphertext form (if brought in by the OS). The mix of data forms in the same cache introduces a vulnerability as the OS may directly access an application’s plaintext data that was brought into the cache in the past by the application. To remove this vulnerability, there are several possible solutions. One possible solution is to flush the caches across mode switches. However, flushing the entire on-chip caches is too costly to be practical. Another possible solution that is actually used by Overshadow [11] and SP3 [81] is to unmapping (invalidate) the page of the application that the OS accesses, and remap it to the kernel address space. Unmapping a page removes the corresponding page translation entry (PTE) from the TLB, and causes all data of that page to be flushed from the cache. The flush forces the OS to refetch it into the cache in a ciphertext form. While this solution works, its performance overheads are not trivial since it involves cache and TLB invalidations, and a page table modification. It also incurs subsequent page faults, cache and TLB refills when the application resumes execution.

In order to avoid such costly overheads, in SecureME we exploit the capability of the secure processor. First, in order to distinguish the current form of a particular cache block, we tag each cache block with an “encrypted” bit. The encrypted bit is set to 1 when a block is stored in the cache in its ciphertext form, otherwise it is set to 0. Under an uncloaked mode, data blocks brought from off-chip memory
are decrypted before they are stored in the cache, hence their encrypted bits are set to 0. Under cloaked mode, data blocks brought from off-chip memory are stored in the cache without being decrypted, hence their encrypted bits are set to 1. Since the OS should only be given access to cloaked data, when it accesses a block whose encrypted bit is not set to 1, the block is passed to the cryptographic engine in order to be encrypted and stored back into the cache.

In the previous discussion, we have assumed that in order to perform memory management functions (page allocation, copying, swapping), it is sufficient for the OS to have access to cloaked data. This in turn implicitly assumes that direct encryption scheme is used, in that the encryption function is position-independent, that is, two ciphertext pages corresponding to the same plaintext data are identical, regardless of where the pages are located in the main memory. This allows OS to operate solely on a ciphertext basis, for example it can create a perfect copy of data by copying the ciphertext of a page from one location to another. The same assumption is used by Overshadow [11] and SP³ [81].

However, as pointed out in prior studies [63, 79, 80], counter mode encryption has been shown to be superior to direct encryption in terms of security protection and performance. Direct encryption provides weaker security protection than counter mode encryption because the statistical distribution of the ciphertext and plaintext are identical. In addition, direct encryption has been shown to impose a significant performance overheads of up to 35% [80] even when the decryption is implemented in
hardware, due to the inability of the processor to overlap the decryption delay with cache miss delay. These security and performance drawbacks are eliminated when we use counter mode encryption [70, 80]. However, with counter mode encryption, the same plaintext page will have different ciphertext values depending on where they are located in the memory. Thus, it is incorrect for an OS to copy a ciphertext application page verbatim to a different location.

Therefore, a novel mechanism is needed to achieve seemingly conflicting goals of allowing the OS to work on cloaked data while at the same time allowing the use of counter-mode encryption in the secure processor substrate. As a part of virtual memory management, the OS is frequently required to copy a page that belongs to an application to another location. Page copying is often employed during forking of a child process. When a parent process forks a child process, the two are made to share the memory pages of the parent, in order to return from the fork quickly, and to reduce the memory requirement of the child process. However, the memory pages of the parent are set to read-only. When either of the process tries to write to the shared pages, a write protection fault occurs and at that time, the OS copies the content of the source page to a new destination page, so that now the parent and child have their own page. After the page copying due to “copy-on-write” completes, the write is allowed to resume.

In order to satisfy the contradictory goals of allowing the OS to perform page copying as it pleases, while at the same time allow us to use counter-mode encryption
in the secure processor substrate, the only solution is to allow the OS to perform
the page copying oblivious of the use of counter-mode encryption (i.e. verbatim),
and then fix the destination page afterwards. To support this, there are important
questions that need to be addressed: how to ensure that integrity verification failure
does not occur under correct OS operation, who should fix the destination page and
how, and how to validate that the OS has indeed performed page copying correctly.

Recall that when the OS runs, the cryptographic engine is in the cloaked mode,
where decryption, Merkle tree verification and update are turned off for data that the
OS operates on. This means that a data block from the source page is not decrypted
as it is read by the processor, and is also written verbatim to the destination as it
is written by the processor. Integrity verification cannot fail at this time as it is
turned off. However, once the application resumes execution and accesses data in the
destination page, it will suffer from integrity verification failure as the Merkle Tree
has not reflected the new data. To avoid integrity verification failure, prior to the
copying by the OS, the LPID, counters used for encrypting the page, and hash codes
that cover the page, are copied over to the destination page. When the application
resumes, if the OS indeed has copied the page verbatim, the seed components (LPID
and counters) are available for the page to be decrypted to the correct plaintext. In
addition, integrity verification will succeed using the copied-over hash codes. Failures
in integrity signify that the OS has not performed the copying correctly, which implies
a compromised OS.
Another issue to deal with is regarding fixing the destination page. The destination and source page currently share the same content, LPID, and page root. This is not a security vulnerability because currently the destination page is the direct outcome of OS page copying, which is not a secret operation. However, once the source or destination page is written to, they can no longer share the same LPID because then they may use the same encryption seed and knowledge of the plaintext of data in one of the page can reveal the plaintext of the corresponding data in the other page. Hence, on completion of copying, the hypervisor write protects the page and on the first write protection fault to the newly copied page, the hypervisor triggers re-encryption of the page.

Page re-encryption can be designed to incur very little overheads. In order to re-encrypt a page, all that is needed is to bring each block in the page into the on-chip cache (in plaintext), and as the page is naturally evicted from the cache, each evicted block is encrypted with a new LPID. Hence, re-encryption occurs over a stretched period of time, and it is off the critical path as the processor is never stalled during the process. Furthermore, during OS page copying, blocks of the destination page were already being brought into the on-chip cache, hence few or no extra cache misses occur due to page re-encryption. The only actual overhead comes from the fact that blocks of the destination page that were brought on-chip by the OS are stored in ciphertext form. Thus, they need to be converted into plaintext form once the application resumes. Hence, the cryptographic engine will have additional work
to decrypt the blocks. However, this can be done at the background without hogging the cryptographic engine. To achieve that, for a page that needs to be re-encrypted after page copying, we keep a re-encryption tracking table (RTT), where each entry contains a page address and a bitmap of re-encryption progress. The RTT does not need to have many entries, since only a few pages are being re-encrypted at any given time. Each bit in the bitmap corresponds to a cache block. For example, for a 4KB page, we will have 64 bits to track 64 blocks of the page. As each block is re-encrypted (i.e. decrypted using the old LPID), the corresponding bit is set to 1. Whenever the cryptographic engine is idle, we schedule another block to be decrypted, until all blocks of a page are decrypted. Once that is completed, re-encryption process for the page is completed, and its entry from the RTT is removed.

Page initialization is handled in a similar manner to page copying. We provide a reference zero-page having all zero values. When the OS initializes a page to zeros, SecureME treats it as if it copies the zero-page to a destination page. The rest of the mechanism is then identical to page copying.

One final issue is how the hypervisor knows when the OS performs page initialization or copying. While analyzing the change in the page table may allow us to do that, it is relatively tricky to do so. Hence, we require the OS to be annotated with PAGEINIT and PAGECOPY virtual instructions prior to initializing or copying a page. These instructions can be added to the kernel code base after kernel development, for example, by dynamic instrumentation. Let us examine the security
implications of this new requirement. If the OS adds these instructions, while ma-
liciously doing other operations, it will result in integrity verification failures as the
data in the destination page mismatches the hash codes that are copied over from the
source page. Another possible attack is the removal of the virtual instructions. In
this case, as the kernel runs in the cloaked mode (cryptographic engine turned OFF),
the ciphertext of the source page will be copied to the destination page without its
accompanying LPID, counters, and hash codes. Once again, when the application re-
sumes execution, this will be caught as an integrity verification failure by the secure
processor. Hence, in SecureME, the secure processor mechanisms are used to verify
the outcome of these operations reliably.

4.3.4 Protecting An Application from Other Applications

Different applications must be protected from one another, and at the same time
they must also be allowed to establish data sharing for *inter-process communication*
(IPC). Different applications can be distinguished from their HIDs. One way to en-
force protection between applications is by tagging data with the owning application’s
HID, and only allow access when the currently running application’s HID matches
with the HID tag value of the data. Such an approach is employed in XOM [42] and
AEGIS [69]. Overshadow [11] and SP³ [81] also use a context-based approach, hence
it allows applications to either share nothing, or share everything in their address
space.
Context-based protection is incompatible with the way contemporary OSes allow protection and sharing between different applications. Contemporary OSes allow shared memory-based IPC to be established between two processes by mapping two virtual pages from the two processes to map to a single physical page. One of the goals of SecureME is to be compatible with current OS mechanisms, including allowing different applications to establish a common shared memory for IPC. To achieve that, we extend regular demand paging mechanism by keeping track of an application’s permission domain, which essentially specifies what pages an application should be allowed to have access to. If a page resides in the permission domain of an application, the application is allowed to map the page into its address space. Otherwise, it is not allowed to do so. To establish a shared page, the page identity is added to the sharing applications’ permission domains. Permission domain may be updated or checked when a page fault occurs.

Figure 4.4 illustrates how a page fault is handled in SecureME. When a page fault occurs, the OS constructs the new virtual to physical page translation information, and attempts to record that information in the page table of the application. This attempt will trap to the hypervisor. The hypervisor then checks to see if the page already has an LPID associated with it. If it does not, indicating an allocation of a new page, the hypervisor assigns a new LPID by inquiring the secure processor (e.g. by executing a special instruction called RDAGPC [59]), which in turn gives a globally-unique number for the page’s LPID. The hypervisor then adds the LPID of
the new page to the permission domain of the faulting process. If a page already has an LPID, indicating it is an already-allocated page that needs to be brought back in to the physical memory, the permission domain of the currently executing process is checked for a match. A match indicates that the application should be allowed to map the page to its address space. A mismatch indicates that the application should not be allowed to map the page to its address space, effectively denying the access.

![Page Fault Handling Mechanism for Permission Paging](image)

Figure 4.4: Page fault handling mechanism for permission paging.

If an application with a different HID, or an unprotected application having no HID, attempts to map a page that belongs to a protected application, a mismatch will occur since the page does not belong to them. Therefore, they would not be able to map the page to their address space. For example, consider an attack where the OS changes the virtual-to-physical mapping to allow a rogue application (R) access the data of a secure application (S). It will construct this new translation and write to R’s page table to allow the access. This will trap to the hypervisor. The hypervisor’s permission paging semantics will now verify if the access is allowed or not. Since
the page belonged to S, it already has an LPID associated with it. This will result in the hypervisor to lookup R’s permission domain (Figure 4.4), thereby raising a security exception, because the LPID is not in the permission domain of R. If on the other hand, an application wants to establish a page for IPC with another application having a different HID, it can do so by explicitly giving the hypervisor permission to add the page’s LPID to the other application’s permission domain. This gives applications an ability to share only a part of their address space without revealing the entire address space to one another.

### 4.3.5 Additional Protection for System Calls

Unlike virtual memory management functions where it is sufficient to provide cloaked data access to the OS, system calls imply that an application *explicitly* entrusts the OS to carry out a service on its behalf. Therefore, short of validating each of OS service’s functionality [4], SecureME cannot guarantee that the service will be performed correctly, or data that is passed to the OS will not be leaked to attackers. Hence, it is the responsibility of the application to encrypt system call parameters before making a system call if they need privacy protection.

However, a limited *spatio-temporal* protection is possible in SecureME. The protection ensures that the OS can only access the data that the system call is authorized to access (spatial protection), and only within the window of time of the system call invocation (temporal protection). We will demonstrate that such spatio-temporal
protection can be achieved mostly without explicit validation, relying on the hardware’s cryptographic engine to provide much of the protection.

The requirement for this system call protection is that the hypervisor must know
the input and output of each system call in terms of all register arguments, memory
buffers, and whether they will be read or written by the system call. While some
system calls do not take any arguments, many others require arguments.

To restrict the OS access to just the arguments of the system call and nothing else,
Overshadow [11] uses argument marshalling (system call arguments are copied from
user space to the kernel space) and demarshalling (output of system calls is copied
from the kernel space to user space). We view such copying to be too expensive,
especially for applications that invoke system calls frequently, such as web servers,
network applications, and disk utilities. Hence, in SecureME, we allow the OS to
directly access the part of the application’s address space that stores the argument.
However, the hypervisor controls the cryptographic engine such that spatio-temporal
protection is still provided.

To provide temporary read access to user arguments, Figure 4.5(a) illustrates
the steps taken in SecureME. When the system call is encountered, it traps to the
hypervisor. The hypervisor reads a register that identifies which system call has
been invoked. It checks a table that records the specifications of the system calls:
what arguments are read and their addresses and lengths. It then instructs the
the application’s buffer containing the arguments to be decrypted (using the DECR
instruction). In addition, since only read access is required, the OS should not be allowed to make any modifications to the user buffer. This is achieved by updating the Merkle tree as the buffer is decrypted which brings the decrypted buffer under Merkle tree protection. Then, the cryptographic engine is completely turned off (encryption/decryption, Merkle Tree update, and Merkle Tree integrity verification), before the system call is allowed to proceed. During system call execution, the OS can only view user arguments in plaintext while the rest of the user address space is cloaked. Upon the completion of the system call, the OS tries to return control to the application, which traps to the hypervisor. The hypervisor instructs the user buffer to be re-encrypted (using the ENCR instruction) with the Merkle tree verification mechanism turned on so that any illegal modifications by the OS to user buffer results in integrity verification failure. In parallel, the Merkle tree is updated to reflect the ciphertext of the user buffer. Finally, encryption/decryption is turned on and control is returned to the application.

Figure 4.5(b) illustrates the steps taken to give spatio-temporal write protection to application’s arguments in user buffer. As the system call is invoked, the hypervisor decrypts the user buffer, this time with Merkle Tree not being updated or verified. System call is executed under the cloaked mode. At the completion of the system call, the hypervisor re-encrypts the user buffer, with the Merkle Tree updated (but not verified) to reflect the new values. Modifications by the OS to user buffer will not trigger integrity verification failure in this case because integrity verification is
turned on at the completion of the system call.

Figure 4.5: Temporary access mechanism for System Calls with read-only arguments (a), and read-write arguments (b).

### 4.3.6 Other Design Issues

On an interrupt, the processor saves the register state of the interrupted process in an exception frame on the kernel stack. This leaves a possibility for the OS to read the register state of the interrupted process. Since hypervisors are designed to naturally intercept all interrupts, SecureME lets the hypervisor to use the intercept to securely store the state of the application that is interrupted (by executing SECSAVE). The hypervisor or the processor can encrypt the register state along with the HID of the application, before storing the exception frame on the kernel stack. In addition, a hash code is computed over the encrypted exception frame to prevent the kernel from modifying the register contents. The hash may also include a unique counter value that is incremented across interrupts to provide protection against replay on register state saved across interrupts (similar to XOM [42]). Once the interrupt is
serviced, SecureME can decrypt the exception frame and restore the register state and HID of the application (by executing SECRESTORE), after verifying the hash code associated with it to ensure its integrity and defense against replay attacks.

4.4 SecureME Implementation

While our discussion in the previous section has focused mainly on a hypervisor to manage the secure processor and provide privacy and integrity to the application, the thin software layer in SecureME can also be implemented as an extension to an existing OS kernel through dynamic instrumentation. The main point is that in either implementation, there is no need for the OS source code to be modified, or any functionality of the OS to be changed or rewritten.

In a virtualized environment, the hypervisor virtualizes the hardware resources by causing a trap when the guest OS attempts to modify the privileged state. These include events for virtual memory management such as page initialization and copying, context switches, interrupts, and system calls. Hence, SecureME thin software layer can be captured cleanly as a hypervisor. However, in some cases, assuming a virtualized environment is an overkill if performance or power overheads are a concern, or when the system does not support virtualization. In such a case, SecureME thin software layer can be dynamically instrumented into the OS code. For security reasons, this weaved-in code must be verified for its integrity using a Merkle Tree as
well. In addition, this software must identify itself to the hardware to ensure that the
privileged instructions introduced, cannot be executed by any other part of the OS.
This can be achieved by having the module identify itself to the hardware during its
initialization, for example, through a hash signed by a trusted authority. The module
can then present this hash to the processor along with the instruction execution, to
allow the processor to verify its identity and allow the instruction execution.

In our current implementation, we use the second approach where the OS is dy-
namically instrumented by inserting probes into the Linux OS kernel. The probes
are inserted at the kernel functions to match the traps that will happen in a virtual-
ized environment. Probing the kernel closely models the performance of a virtualized
environment as probes also result in a trap when execution reaches the probe, em-
ulating the behavior of a virtualized environment. In particular, we use \texttt{jprobes}
and \texttt{kretprobes} \cite{58}, which allows us to get a handle to the probed kernel function
arguments and return values respectively. We implement the code that manages per-
mission paging, HIDs, and LPIDs, as trap handlers. In this section, we present the
details of our modifications to the Linux kernel to simulate SecureME.

\textbf{Assigning HIDs:} We use the process ID assigned by the OS as the HID for the
requesting application’s permission domain.

\textbf{Maintaining current HID (CurHID):} The kernel is instrumented at the sched-
uler function \texttt{\_\_switch\_to}. The function is used by the scheduler to do a context swtich
and takes two arguments, the task structure of the outgoing process and the task stru-
ture of the process selected by the scheduler to run next on the processor. We have used jprobes [58] which allows us to analyze the arguments of the function. The kernel module simulating permissions based paging extracts the ID of the next process and updates the CurHID variable.

Maintaining LPIDs: The Linux kernel uses a structure to represent each frame in physical memory, struct page. This structure is modified to have another field, the LPID of the page. When a page is loaded for the first time, it does not have an LPID value associated with it. To get a free page, the kernel uses the allocator API, which consists of multiple functions, so that the correct zone can be chosen for the allocation. However, all the allocator functions ultimately map onto one function, alloc_pages. This function is the heart of the zoned buddy memory allocator used by the Linux kernel. It is responsible for allocating a page, and returns a pointer to the page structure that has been allocated. The kernel is instrumented at alloc_pages using kretprobes [58], which allows us to get a handle to the return value of the probed function. We use this return value stored in the eax register to examine the LPID field of the page structure.

If the page has never been used by the application before, that is, the page is being loaded for the first time, the LPID field of the page will be 0. On seeing an LPID of 0, a global counter maintained inside the kernel module is used to assign an LPID value to the page(simulating the global page counter in AISE). If on the other hand, the page being allocated has an LPID value assigned to it, it implies that the page
is being loaded from the swap space and must already be present in the permission domain of the requesting application.

The LPID of the page structure is reset to zero, when the page is freed by the application. To free a page, the kernel function, \_\_free\_pages is called. It takes the structure of the page to be freed as an argument. Once again, we use \jprobes to get a handle to the page structure, to reset the LPID field to 0.

**Page Initialization:** As described earlier, the kernel makes a call to \_\_alloc\_pages to get a page. The allocation is controlled by a set of flags, \_\_gfp\_flags (Get Free Page flags). To get a zeroed page, the kernel uses flag value of \_\_GFP\_ZERO. The kernel is instrumented using \jprobes at \_\_alloc\_pages to examine the flags used and detect a page initialization. The kernel module simulates the re-encryption required on a page initialization by initiating a read to the entire page. This is done by obtaining the virtual address of the page from the page structure using the \page\_address kernel function, and initiating a read of the next 4096 bytes. This closely models the re-encryption, as a re-encryption, in AISE, requires a read of the page and changing the LPID associated with the page. The blocks will be naturally re-encrypted as and when they are evicted off the processor chip, using the new LPID for the page.

**Copy-On-Write:** On a page fault, the kernel determines the cause of the page fault. If the page fault is due to a write to a page but the controlling area is writable and the count field in the frame is greater than one, the OS determines that a COW is to be performed. The kernel makes a call to \do\_wp\_page, which in turn calls
the \textit{cow\_user\_page} to create the page copy. This function takes the source and the
destination page structures as arguments. We use \textit{jprobes}, to get a handle to the
destination page and simulate the re-encryption required by SecureME on COW as we
did for page initialization.

\textbf{System Calls:} The function corresponding to the system call in the kernel is probed
using \textit{jprobes}, to obtain a handle to the system call arguments. The encryption/de-
cryption of the system call arguments is simulated by reading the buffer passed to
the call, similar to page initialization and copy-on-write.

\section{4.5 Experimental Setup}

\textbf{Simulated machine configuration.} For our benchmark-driven performance evalu-
ation study, we implement our secure processor substrate on a full-system simulator,
Simics [43]. We model a 2GHz, in-order processor with split L1 data and instruction
caches. Both caches are 32KB in size, 2-way set associative and have a 2-cycle round-
trip hit latency. The L2 cache is unified and is 1MB in size, 8-way set-associative,
and has a 10-cycle round-trip hit latency. L1 and L2 cache lines are tagged with
the “encryption bit” to simulate our cache protection mechanism. For counter mode
encryption, the processor includes a 16-way set-associative 32KB 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 350 cycles. The encryption/decryption
engine models a 128-bit, 16-stage pipelined, AES engine with a total latency of 80-cycles. The MAC computation assumes HMAC \[27\], based on SHA-1 \[18\], with an 80-cycle latency \[36\]. The seed used for encryption is the concatenation of a 64-bit per-page LPID and a 7-bit per-block counter. On a counter cache block, a 64-bit LPID is co-located with 64 7-bit counters, occupying exactly a 64-byte block, and corresponding exactly to a 4KB memory page \[79\]. The default authentication code size is 128-bits.

The simulated machine runs Linux kernel that is dynamically instrumented with traps to implement SecureME. We add to the cost of context switch an extra 100 cycles needed to encrypt or decrypt the architecture register file.

**Real machine configuration:** To evaluate just the software components of SecureME, we run SPEC 2006 \[68\] benchmarks on a real machine (not a secure processor). We use the modified Linux kernel version 2.6.24 as our OS, running on a 2GHz Intel Pentium 4 processor. The processor has 32KB split L1 data and instruction caches, and a 1MB L2 cache.

**Benchmarks.** To evaluate SecureME, we use all (but one) C/C++ benchmarks from SPEC 2006 benchmark suite: *astar, bzip2, gcc, gobmk, h264ref, hmmer, lbm, libquantum, mcf, milc, namd, omnetpp, perlbench, povray, sjeng, soplex, sphinx, and xalancbmk*. Only one benchmark, dealII, is excluded, due to compilation error. The benchmarks are run on the simulated machine with reference input set. We intentionally avoid skipping any instructions since the initialization phase of the benchmark is
where most page faults will occur. This stresses the performance of SecureME. Each benchmark is simulated for 2-billion user-mode instructions. The SPEC benchmarks do not involve many system calls or fork a process, hence to test these specific aspects, we wrote three microbenchmarks: ReadBench to test read system call performance, WriteBench to test write system call performance, and COWBench to test copy-on-write performance. We further present results for several real, system call intensive benchmarks. We use Linux find utility, disk usage utility, du, and I/O performance tool iozone3, with tests performed on a 4MB file.

For multiprogrammed evaluation on a CMP, we constructed nine benchmark pairs from SPEC 2006, by pairing two benchmarks. We divide the benchmarks into two groups: those that are cache miss-intensive, measured as having an L2 miss rate of more than 20% when they run alone, and those that are not cache miss-intensive, measured as having an L2 miss rate of 20% or less when they run alone. We pair two cache miss-intensive benchmarks (gcc_libquantum, gcc_mcf, mcf_libquantum), one cache miss intensive benchmark with one non cache miss intensive benchmark (gcc_hmmer, perl_gcc, perl_mcf), and two non cache miss-intensive benchmarks (perl_hmmer, soplex_perl, soplex_hmmer). For each simulation, we use the reference input set and simulate for a combined number of 2 billion user-mode instructions.
4.6 Evaluation

To evaluate our scheme, we present our results for microbenchmarks that stress specific performance aspects in Section 4.6.1, general evaluation results on SPEC 2006 benchmarks in Section 4.6.2, and the sensitivity study results obtained by varying cache sizes and cryptographic latencies in Section 4.6.3. Finally, in Section 4.6.4, we present various evaluation results: execution time overhead results on a real system, and results for memory allocation intensive benchmarks.

4.6.1 Microbenchmark Results

In order to evaluate the overhead of SecureME on system calls and copy-on-write events, we have constructed three microbenchmarks: ReadBench, WriteBench, and CopyBench. ReadBench (WriteBench) reads (writes) 1MB of data from (to) a file, at varying granularity at each call. Hence, a smaller granularity increases the system call frequencies. In CopyBench, we allocate an array of varying size, and fork a child process that writes to the entire array to force the OS’s copy-on-write mechanism to cause page copying. The microbenchmarks only have system calls or page copying and have no computation at all, hence at small granularities, they represent the worst-case execution time overheads on SecureME. Figure 4.6(left) shows the execution time overheads for these microbenchmarks on SecureME (which includes secure processor substrate) over a system with no protection.
The figure shows that in the worst case (at the smallest granularity), ReadBench, WriteBench, and CopyBench incur execution time overheads of 35%, 19%, and 15%, which are significant but relatively low considering that there is no computation performed other than those of the system calls and page copying. The overheads decline steeply as the number of traps reduces with the increase in granularities, reaching 5.3%, 10%, and 11% on 1MB case, respectively. We believe such overheads are manageable, and real-benchmarks suffering less than 1% overhead as shown in Figure 4.6(right) corroborate that.

4.6.2 Real Benchmark Evaluation Results

Figure 4.7(a) shows the execution time overheads of secure processor alone, and SecureME (including secure processor substrate) for SPEC 2006 applications. The overheads are over a system with no security protection at all. On average, SecureME adds an execution time overhead of 5.2%. Roughly 42% of those overheads come from the use of secure processor, with an average of 2.2%. Hence, on top of secure processor,
SecureME adds only a small performance overhead of about 3%.

However, while the average overhead is low, a small number of benchmarks suffer from higher execution time overheads (11% for astar, and 13.5% for bzip). To understand the source of these overheads, we profile the number of user-mode and kernel-mode instructions before and after SecureME is added, in Figure 4.7(b). The figure shows that while the number of user-mode instructions are largely unaffected, the number of kernel-mode instructions (due to the thin software layer of SecureME) may increase significantly. This is especially true for the two benchmarks that suffer from a large execution time overhead with SecureME (astar and bzip), and we observe a strong correlation between the increase in kernel mode instructions and SecureME execution time overheads for other benchmarks as well.

Figure 4.8(a) presents the L2 cache miss rate and the number of misses per thousand instruction or MPKI (right) for SecureME compared to a system with only secure processor substrate. The figure shows that the L2 miss rates are either unchanged or slightly reduced in SecureME. The reason for this reduction in L2 miss rates can be seen in Figure 4.8(b), which shows that the new kernel instructions introduced by SecureME has much lower MPKIs compared to regular kernel instructions. This is because the kernel code and data in SecureME code exhibits better temporal locality than regular kernel code. This better temporal locality and the resulting lower MPKIs help mask the performance overheads that would result from higher number of kernel instructions. Note, however, that the significant increase in the number
Figure 4.7: Execution time overheads of secure processor and SecureME (a), and the breakdown of kernel and user-mode instructions (b).
of kernel instructions can be attributed to the unusually high page fault rate of the benchmarks at the initialization phase of their execution (the first 2 billion instructions are simulated). Steady-state execution page fault rate would be significantly lower than this, and the overheads from SecureME will be significantly lower as well.

To understand the source of SecureME overheads better, Figure 4.9 breaks down the overheads into their components: permission table lookup, page initialization, page copying that results from copy-on-write, and secure interrupt handling. The figure shows that the permission table lookup is the dominant component of overheads (44.7% on average), with page initialization at second place (34.1% on average), followed by secure interrupt handling (20.44% on average), and copy on write that is incurred by the OS and not by the application (0.76% on average). The relatively-high overhead contribution due to permission table lookup is due to our current permission table implementation that must be linearly searched on a page fault. We believe a better implementation, such as a hash table, can improve performance significantly by eliminating linear searches.

Figure 4.10 shows execution time overheads for multiprogrammed workload running on a dual core CMP. The figure shows an average overhead of 1.5% for secure processor alone and 3.4% for SecureME. The total overheads from SecureME for all nine workloads are below 5%.
Figure 4.8: L2 cache miss rate (a), and L2 misses per thousand instructions (b), for secure processor vs. SecureME.
Figure 4.9: Execution time overhead breakdown for SecureME

Figure 4.10: Evaluation results for multiprogrammed workloads on a two-core CMP
4.6.3 Sensitivity Studies

Figure 4.11(a) shows the average execution time overhead for single-benchmark workloads as well as multiprogrammed workloads on CMP when the L2 cache sizes vary from 512KB, 1MB to 2MB, over secure processor only. The figure shows that the overheads for SecureME are relatively unaffected by the cache size, and stay below 3.5% for all cases. Figure 4.11(b) shows the average execution time overheads when the latency of the AES engine is varied from 80 cycles (base case), to 160 cycles (2X) or 320 cycles (4X), which is almost equivalent to the memory latency. The overheads are not much affected.

4.6.4 Other Evaluation Results

Real Machine Evaluation. We also measure SecureME dynamically instrumented Linux kernel on a real machine. The processor (2GHz Intel Pentium 4) has no cryptographic engine, hence the overheads mainly come from the extra traps, code to lookup and update permission domain tables. All benchmarks are run five times using reference input set from the beginning to completion, and the average of five runs is taken. The execution time overheads we observe approximately match the simulation results, averaging 2.3% across the SPEC benchmarks, with individual benchmark overheads showing similar trends to those of simulation runs.

Memory Overheads. The permission table requires a storage of 286KB in the main
Figure 4.11: Sensitivity to Cache size (a), and Sensitivity to AES latency (b).
memory on an average across all SPEC benchmarks, which is less than 0.03% of size of the 1GB physical memory. The theoretical maximum overhead for a 1GB physical memory and 2GB swap space on disk assuming 64bit LPID is 6MB \( \left( \frac{3GB}{1TB} \times 64bits = 6MB \right) \), equivalent to 0.58% storage overhead.

**Memory Allocation-Intensive Benchmarks.** To stress SecureME, we also run eight memory allocation-intensive benchmarks that are often used in memory allocation studies \([17, 37]\) (boxed, cfrac, deltaBlue, espresso, lindsay, LRUsim, richards, and roboop). Even for these benchmarks, the performance overheads of SecureME remains low, at 4% on average.

### 4.7 Conclusions

We have proposed and presented SecureME, a new mechanism consisting of architecture support and a hypervisor, built on top of secure processor substrate, that provides secure computing environment on untrusted computing nodes. SecureME shows how an application can be protected from vulnerabilities of the hardware and the OS, through memory cloaking built on top of secure processor mechanisms. We show how memory cloaking can be integrated with counter-mode encryption and Merkle Tree integrity verification. SecureME also allows traditional inter-process communication to be supported. SecureME requires minimum changes to the OS (dynamically inserted annotation is sufficient) and to the application (only one instruction to get
hardware identifier is needed). We show how spatio-temporal protection can be provided to guard against some attacks utilizing system calls, relying on just selective activation or deactivation of various components of the cryptographic engine of a secure processor. Finally, our evaluation results with single-benchmark and multiprogrammed workloads show that the execution time overheads of SecureME, inclusive of overheads from the secure processor substrate, is low on average (5.2%), and still acceptable even in the worst case (13.5%).
Chapter 5

Conclusion

Traditionally, software attacks have been the most exploited form of attacks. But despite the increased complexity involved in carrying out hardware attacks, it has been shown that given significant payoffs, hardware attacks are realistic threats and can be much more powerful than software attacks. To defend against hardware attacks, researchers have proposed secure processor architectures. Secure processor architectures incorporate hardware-based security mechanisms to maintain the confidentiality and integrity of application code and data even in the face of hardware attacks. This is done by encrypting/decrypting code and data (ensuring confidentiality) and authenticating it (ensuring integrity) as it moves off/on the processor chip. However, currently proposed secure processor architectures suffer from critical performance, system-level and security issues which make them impractical for implementation. In this dissertation, we have presented three studies which aim to resolve
these issues to make secure processor architectures practical for implementation.

5.1 Performance and System-level Issues

In the first part of this dissertation, we looked at the performance and system-level issues with currently proposed designs. In particular, we proposed a new counter-mode encryption scheme, Address Independent Seed Encryption (AISE) which uses logical identifiers instead of using address (virtual or physical) as a component of the seed to encrypt/decrypt data blocks. This enables AISE to support virtual memory and Inter-Process Communication. We also proposed Bonsai Merkle Tree integrity verification scheme (BMT). BMTs are based on the observation 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 enables us to construct BMTs which take less space in main memory, but more importantly much less space in the L2 cache, resulting in a significant reduction in performance overheads from 12.1% to 1.8% for single threaded SPEC 2000 benchmarks and from 15% to 4% for multi-programmed benchmarks, along with a reduction in storage overhead in memory from 33.5% to 21.5%.
5.2 Security Issues

In the second part of this dissertation, we looked at two critical security issues with current architectures. First, currently proposed architectures provide defense against hardware attacks conducted during steady state of the system and assume that the system has booted up securely by including the booting process of the system in the TCB. With the increasing number of attacks which target systems during the boot phase, this assumption makes the security provided by secure processor architectures contingent on the system booting up securely. If a system can be compromised before or during boot up, the secure processor mechanisms just work to protect an already compromised system. Hence, it is necessary to have a secure booting mechanism in place. Unfortunately, there have been very few studies on secure booting mechanisms. In addition, prior secure booting approaches are vulnerable in the scenario where the user of the system is considered the attacker and can conduct hardware-based attacks. We propose SHIELDSTRAP, a novel secure booting mechanism that is secure even against relatively sophisticated hardware attacks. Our key insight in SHIELDSTRAP is that we move the root of trust for verifying the integrity of all boot components onto the processor chip which is our natural security boundary. We analyze how SHIELDSTRAP is secure against common boot attacks. We also show that SHIELDSTRAP requires an acceptably small amount of on-chip hardware, and that it adds an insignificant amount of execution time overhead to the normal boot
process.

Second, current secure processor architectures while assuming untrusted hardware, implicitly trust the OS. In essence, currently proposed secure processor architectures use an on-chip key to encrypt/decrypt application data as it moves off/on the processor chip. Since the OS has access to all pages, when it runs, it can access all application pages in plaintext. A compromised OS can leak this critical application data like passwords, credit card numbers etc. to an attacker either directly or indirectly through a rogue application by allowing it to map its pages to the physical pages of a security sensitive application, thereby rendering the security of the system worthless. Hence, a complete security solution must protect applications against vulnerabilities in both the hardware and the OS. To achieve that, we propose an integrated security protection framework which we refer to as Secure My Execution (SecureME). SecureME consists of a secure processor technology controlled by small trusted software which can be integrated either through OS kernel dynamic instrumentation or as a hypervisor. Firstly, to provide defense against hardware eavesdropping from discovering plaintext data, the entire address space of the application is cloaked (encrypted and hashed) at all time. Only when the application accesses its own data, data is decrypted and its integrity is checked on chip. Secondly, to provide security guarantees even with a compromised OS, SecureME uses a combination of memory cloaking and permission paging. Memory cloaking works on the observation that in order to perform its management functions, the OS does not need plaintext access to the application pages.
Likewise, the OS is always rendered the ciphertext image of a memory page belonging to a secure application. Now, in order to prevent a compromised OS from leaking sensitive data through another application by allowing the rogue application to map to the physical pages of the secure application, we introduce permission paging, a novel extension to the traditional demand paging mechanism, which ensures that an application cannot access a page belonging to another application unless explicitly allowed to do so, for example, through shared memory communication. In addition, SecureME also provides an additional protection mechanism for system calls. By making system calls, an application explicitly trusts the OS to perform service with data it supplies or will receive. Thus, system call parameters must be uncloaked, and there is no guarantee that a compromised OS will perform the service correctly. However, SecureME ensures that the OS only accesses data it is supposed to (spatial protection), and only within the window of time of the system call invocation (temporal protection).

SecureME is compatible with the state-of-the-art secure processor technology, allows a commodity OS to be employed nearly unmodified. Hence the OS is free to implement vital management functions on application’s data using whatever policies it wants, and does not require any effort on behalf of the application programmers. Despite this flexibility, SecureME ensures the privacy and integrity of the application, from both hardware and OS vulnerabilities. Our evaluation results with single-benchmark and multiprogrammed workloads show that the execution time overheads
of SecureME, inclusive of overheads from the secure processor substrate, is low on average (5.2%), and still acceptable even in the worst case (13.5%).
Bibliography


