Attack Mitigation through Memory Encryption

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Engineering Sciences

By

Michael J. Henson

Thayer School of Engineering
Dartmouth College
Hanover, New Hampshire

September 2014

Examining Committee:

Chairman_____
Stephen Taylor, Ph.D.

Member_____
George Cybenko, Ph.D.

Member_____
Eric Hansen, Ph.D.

Member_____
Jeff Boleng, Ph.D.

_____
F. Jon Kull
Dean of Graduate Studies

Abstract

Historically, full memory encryption (FME) has been propounded as a mechanism to mitigate vulnerabilities associated with code and data stored in the clear (unencrypted) in random access memory. Unfortunately, until recently the CPU-memory bottleneck has represented a roadblock to using this concept to design usable operating systems with acceptable overheads. Recently however, a variety of commodity processors, including the Intel i7, AMD bulldozer, and multiple ARM variants, have emerged that include security hardware -- in particular, encryption engines -- tightly integrated on-chip. This innovation opens the door to a new generation of operating systems that protect data by encrypting code and data in RAM. This thesis explores this idea and introduces a collection of novel operating system technologies that provide automated, transparent confidentiality and integrity protection via memory encryption. These techniques raise the difficulty for attackers, making it significantly more challenging to determine the vulnerabilities present on a system, apply the same attack vector against multiple hosts, steal sensitive information, reverse engineer code, modify data at rest or in flight, and inject code onto a platform.

Acknowledgements or Preface

I would like to thank my advisor Dr. Stephen Taylor, as well as fellow group members Colin, Jason, Kathleen, Morgon, Rob, and Steve. I also want to thank my wife Latasha and children who put up with many late nights and who inspired me to continue striving even when my time was extremely limited. Special thanks to Brianna and Elanna for covering many flights of stairs while providing me with the necessary sustenance to keep me working. Finally, I would be remiss if I did not thank my mother and father, who imparted both a strong work ethic and a love of learning.

This research is supported by the Defense Advanced Research Projects Agency as part of the CRASH program under contract FA8750-11-2-0257. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

“It is not the critic who counts; not the man who points out how the strong man stumbles, or where the doer of deeds could have done them better. The credit belongs to the man who is actually in the arena, whose face is marred by dust and sweat and blood; who strives valiantly; who errs, who comes short again and again, because there is no effort without error and shortcoming; but who does actually strive to do the deeds; who knows

great enthusiasms, the great devotions; who spends himself in a worthy cause; who at the best knows in the end the triumph of high achievement, and who at the worst, if he fails, at least fails while daring greatly, so that his place shall never be with those cold and timid souls who neither know victory nor defeat."

-Theodore Roosevelt

Table of Contents

List of Tables

List of Figures

Chapter 1: Overview

1.1 Problem:

How can attacks against plaintext code and data residing in large random access memories be mitigated?

1.2 Hypothesis:

The vulnerability associated with plaintext in memory can be eliminated, with reasonable performance impact, by *memory encryption* on security-enhanced processors.

1.3 Background Synopsis

Trust as defined in computing systems is concerned with providing guarantees on functionality and security associated with a system design. Typically trust emanates from a careful combination of core hardware and software components that in tandem form a *trusted computing base* (TCB). This base is often used either to protect the critical core associated with a sensitive or proprietary software or it can be *amplified* to protect larger portions of a system.

In operating systems, a trusted base can be used to protect core kernel functions. Several examples of this include system bootstrapping via a trusted computing module (TPM) [Kauer 2007], *data-at-rest* protection through full-disk encryption [Casey et al. 2011], *data-in-transit* protection through link-level encryption [Karlof et al. 2004], and kernel operation using access control and protection schemes [Karger and Schell 1974]. Although all these technologies are valuable, they are not sufficient in-and-of themselves. They only guarantee that the system *began* in a trusted state and that data was confidential *before* it was decrypted. Unfortunately, systems become vulnerable whenever code and data are stored in the clear (unencrypted) within random access memory. This creates numerous *vulnerabilities* at every level of the software stack. These vulnerabilities have consistently been *exploited* to gather confidential information (such as encryption keys) and inject malicious code in order to overcome access controls and other protections.

1.4. Approach

Recently, a new generation of commodity processors have appeared that include security technologies, such as encryption engines, on-chip within the trusted boundary provided by the processor. These processors include the Intel i7, AMD bulldozer, and multiple ARM variants. The creation/use of such processors begs the question: Can these technologies be leveraged with sufficiently low overhead in order to improve operating system security? This thesis explores the idea of enhancing security through *memory encryption.* In particular, it introduces three new technologies:

- ***Static Encrypted Processes:*** This technology employs one-time decryption within

the trusted boundary. Since the one-time cost of encryption is amortized over the life of a programs execution, its overhead is negligible. The technique can be used to protect industrial control systems employing microcontrollers and other real-time processors. These devices typically lack memory management and make little to no use of cache.

- ***Dynamic Encrypted Processes:*** This technology provides a general, full memory encryption mechanism for code and data. It is appropriate to any multi-tasking operating system that employs a memory management unit (MMU) and cache including smart phone and other mobile computing devices. Two micro-benchmark programs targeting the specific areas where overhead is introduced (context switching and cryptopaging of heap and code) show reasonable performance impact of approximately .12% and 1.2% per minute respectively given a page size of 4 KB and typical mobile smartphone workloads.
- ***Mutually Distrusting Processes:*** This technology extends dynamic encrypted processes to protect processes from each other by uniquely keying each process. At its finest granular level, this technique induces a performance penalty of approximately 1920 cycles or 2.4 microseconds per context switch (or about 480 microseconds per minute) for the key search—an extremely small overhead for the additional protections afforded.

Collectively, these technologies *increase attacker workload* by ensuring that both code and data are always encrypted outside the trust boundary afforded by the processor. To overcome this barrier requires physical access and exotic reverse engineering techniques, such as acid etching, that are generally the domain of only a few, highly

skilled, internationally recognized, specialists in reverse engineering. A side-effect of the approach is that it introduces a *synthetic diversity* into code and data: every processor's image is completely different in RAM. This makes it significantly more difficult to determine the vulnerabilities present on a particular system, use the same attack vector against multiple hosts, or steal sensitive code and data, perform reverse engineering of code, modify data, and inject code.

1.4 Contributions

The core contributions of this thesis are:

- The first practical full-memory encryption system implemented on a general-purpose commodity processor.
- A survey and comparative analysis of memory encryption techniques covering three decades of research with proposed solutions; these employ widely varying assumptions and experimental conditions [Henson and Taylor 2014].
- A collection of novel memory encryption techniques providing *synthetic diversity* and *increasing attacker workload.* These techniques protect against software and hardware based confidentiality and integrity attacks; the techniques are portable to currently deployed general-purpose, security-enhanced processors [Henson and Taylor 2013A and B].
- Analytical results that include performance benchmarks and analysis on the overhead of memory encryption down to *process segment granularity* [Henson and Taylor 2013B].
- Empirical evidence and analytical analysis that demonstrate protection through memory encryption against confidentiality and integrity attacks.

- Techniques to employ self-modifying code within the memory hierarchy to achieve memory encryption.

These techniques and technologies have been demonstrated in proof-of-concept implementations and exemplars. Memory encryption has been implemented on the ARM Cortex A8 processor to provide *automatic* and *transparent* protection for applications. This is achieved through extensions to a secure microkernel – *Bear* – under development within our research group at Dartmouth. These extensions involve modifications to linker scripts, initialization, process creation and context switching routines as well as new modules for interfacing with the A8's on-chip encryption decryption unit (EDU). The ideas have been demonstrated by encrypting processes while they reside in external RAM (eRAM) thereby adding synthetic diversity. The concepts cover application deployment regimes that range from unsophisticated microcontrollers, with no memory management unit (MMU) and cache, to full-functioned multi-processing operating systems utilizing a memory management unit (MMU) and L1/L2 cache. Various *granularities* of protection are considered from a complete code base to individual process. Finally, exception-handling routines have been developed and experiments executed to understand the protections afforded against code and data injection.

1.5 Scope and Assumptions

The work described in this thesis extends the base of technologies available for *trusted computing*. While definitions of trusted computing abound, in this thesis it is defined as the process by which a trusted subset (software and hardware) of a system, known as the *trusted computing base* (TCB) is *amplified* to provide security assurances about the operation of the larger application or system [Smith]. Hardware components of

the traditional TCB include encryption coprocessors, random number generators, and small amounts of protected space for operation on sensitive code and data. The main application of trusted computing in operating systems design is for "trusted boot" in which the TCB checks the integrity of each component in the boot process, perhaps halting the boot process when a problem is discovered [trusted boot]. Additionally, the TCB has been used as a means for providing digital rights management. While the underlying security and integrity of hardware are often assumed to be axiomatic by those programming higher layers, this is not typically the case [Arbaugh et al. 1997 Aegis]. The inclusion of security hardware within commodity processors means that these general purpose CPUs may now be treated as part of the TCB. While the processor boundary may not have been designed to meet stringent guidelines, such as the PCI, it does, however, provide natural barriers to penetration and observation [Vandana 2008]. The work in this thesis seeks to expand upon current trusted computing capabilities such as trusted boot by continuing to protect applications dynamically as they execute. While memory encryption provides significant protection against multiple attack vectors, it should be used as part of a defense-in-depth solution including other trusted computing capabilities such as trusted boot as well as encryption of data-at-rest.

Any security can be circumvented given enough resources and motivation and memory encryption is no exception. The goal of the work, to *increase attacker workload*, can be applied under two alternative scenarios: In any time-sensitive operation, as occurs on the battlefield, an increase in attacker workload serves to force the adversary outside of the useful timeframe of any sensitive data collected. For a commercial example, the increased workload would influence the attacker to choose a

weaker attack surface, on a different device (preferably at another business).

The use of any encryption technique begs the question of key generation, delivery and escrow. Additionally, protecting programs from static analysis means that they must only be stored in an encrypted form. While this work explores some core ideas that can be applied to satisfy these questions, they have not been fully implemented. This thesis concentrates on designing and quantifying memory encryption systems on a general-purpose processor; other issues, while important, are ancillary.

Finally, it is useful to point out that there are many similarities between the goals of sophisticated attackers and law enforcement with regard to the acquisition of sensitive information from the memory of a device. Therefore, throughout this thesis, the terms *attacker* and *forensic investigator* may be considered synonymous.

1.6 Outline of the Thesis

The structure of the thesis is divided into seven chapters:

Chapter 2 begins with the background and motivation for memory encryption. Next, a comprehensive survey of the past three decades of memory encryption research including a ground-truth comparative analysis is presented. Closely related works are included at the end of chapter 3 such that the latter part of chapter 2 (beginning with section 2.2) may be skipped without impacting the reader's understanding of the thesis.

Chapter 3 introduces the threat model, core ideas, and fundamental protections associated with memory vulnerabilities and memory encryption. This serves to provide background and motivation for the remainder of the thesis.

Chapter 4 describes *Static Encrypted Processes*, a memory encryption technology created as a part of this research. It describes the design philosophy, the encryption decryption unit (EDU) used, bootstrapping details, and other implementation details.

Chapter 5 describes *Dynamic Encrypted Processes*, its design philosophy, and extensions to include the use of cache and the MMU. This chapter also explores the issues associated with *self-modifying code* in memory encryption systems.

Chapter 6 extends the ideas presented in Chapter 4 to protect *Mutually Distrusting Processes* (MDP) from each other via an increase to *key scope granularity*. Examples of MDP's include applications (apps) downloaded from online stores and the issue of covert channels is discussed and evaluated in this context. Additionally, experimental results evaluating the security properties provided by memory encryption are provided.

Chapter 7 concludes the thesis, including directions for future research and observations.

Chapter 2: Survey and Comparative Analysis of Memory Encryption Techniques[1]

Chapter 2 presents a survey of memory encryption techniques spanning the past three decades, as well as a thorough comparative analysis of those techniques. While all of the information gathered in the survey helped to shape and form the direction of this thesis it is not strictly necessary for the reader's understanding, however, the beginning of this chapter (here through section 2.1) and the conclusion (section 2.9) provide necessary background.

Encryption has been an important part of secure computing for decades, first used in the Department of Defense (DoD) and national agencies and then publicly beginning with public-key encryption in 1977 [Mel et al. 2001]. As public use of computers continued to grow, so did the need to secure sensitive information. In 1991, Phil Zimmerman released the first version of Pretty Good Privacy (PGP) allowing anyone to encrypt e-mail and files. In 1995, Netscape developed the secure sockets layer (SSL) protocol combining public and private-key encryption to protect online financial transactions. Indeed, encryption of *data-in-transit* has become accepted practice especially when interacting with entities where sensitive information is common (e.g. banking, medical, etc.).

Although a more recent innovation, full disk encryption (FDE) in commodity computer systems provides confidentiality of all data stored on disk (i.e. *data-at-rest*).

[1] Significant portions of this chaptered were published in:

HENSON, M. and TAYLOR, S. Memory Encryption: a survey of existing techniques. ACM Computing Surveys 46, 4, Article 53. (March 2014).

Recent advances to the overall speed of processors and hardware-based encryption have resulted in several commercially viable FDE implementations. Software approaches to FDE include TrueCrypt, PGPDisk, FileVault, and Bitlocker. In addition, multiple hard drive manufacturers offer self-encrypting drives (SED) in which encryption is handled entirely by the hard drive microcontroller. Several factors have resulted in increasing adoption of FDE technologies by both individual users and system developers [Brink 2009], [Muller et al. 2011]. Regulations, such as Sarbanes-Oxley and the Health Insurance Portability and Accountability Act (HIPAA), have increased the requirement for privacy. The advent of mobile computing and increased movement of information over the Internet have raised concerns regarding physical access to data. Finally, numerous data breaches have been publicized raising awareness of vulnerabilities.

2.1 Vulnerabilities and Exploits --Motivation

Current operating system designs have sought to utilize a *static* base of trust and extend trust into software through deliberate layering [Arbaugh et al. 1997]. Modern computer systems, even those protected by full disk encryption (FDE) [Brink 2009], exhibit a major weakness in that code and data are stored in the clear, unencrypted, within *memory and its connections*. These sensitive details are not only available to applications. They are known to persist in multiple unexpected locations (kernel and application), for longer than traditionally thought, even after an application exits [Chow et al. 2004], [Dunn et al. 2012], [Tang et al. 2012]. Unfortunately, this invalidates widely held basic security assumptions rendering it possible to gather confidential information, including encryption keys, passwords, PINs etc. that can be used to undermine trust [Halderman et al. 2008], [Boileau 2006], [Steil 2005], [Henson and Taylor, 2012]. To

exacerbate the problem, memory vulnerabilities extend to *every level of the software stack* and the opportunities for exploitation extend well beyond physical attack to include remote attacks over the Internet. Techniques have evolved that allow malicious code to be injected into device drivers, operating system kernels, and user processes.

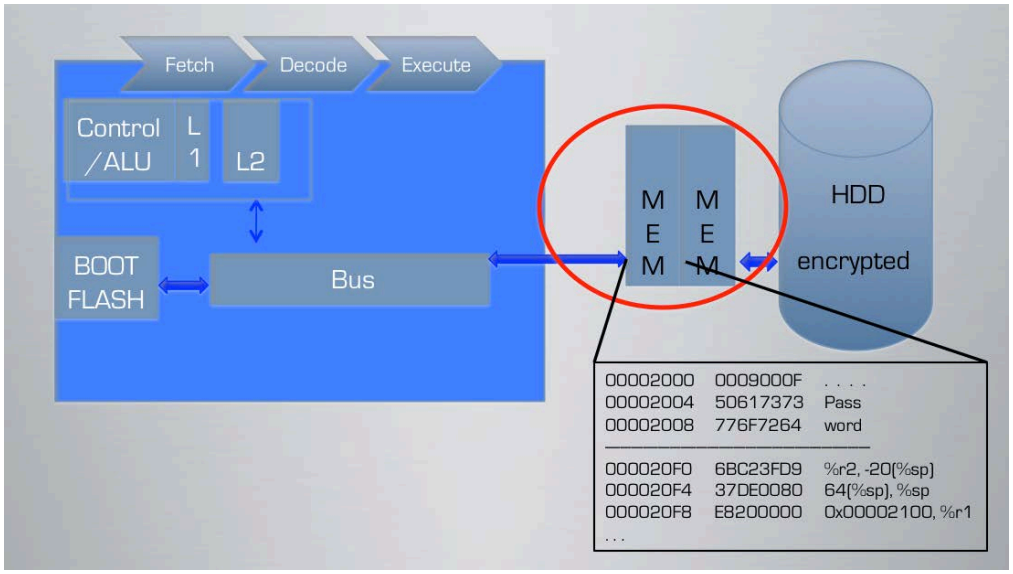

Figure 1: System with Full Disk Encryption but Vulnerable Memory and Connections

To exploit memory vulnerabilities, numerous attack vectors have been developed. In a cold boot attack, for example, memory is frozen using a refrigerant and then removed from the computer. It is then quickly placed into a specially designed system that reads out its content, targeting encryption keys and other sensitive information. This particular attack has recently been shown to be applicable to smart phone devices as well as traditional desktops via the forensic recovery of scrambled telephones (FROST) operating system [Muller et al. 2012]. Besides capturing the encryption key, FROST was used to capture other code and data to include photos, websites visited, e-mails, contact lists, networking credentials and complete ELF binaries. While this approach is novel, the

idea of recovering encryption keys from memory has been described as early as 1998 [Kaplan 2007]. Even without cooling, some information persists in RAM for several minutes [Halderman et al. 2008]. However, cooling slows down the rate of data loss, reducing recovery errors [Chhabra 2011b].

Direct memory access (DMA) attacks make use of Firewire, PCI and/or PC card protocols [Muller et al. 2011]. Direct access of memory allows an attacker to capture a copy of the entire RAM space and/or modify memory contents. The attacks have been demonstrated (with the release of "tools") in black-hat settings for bypassing Windows login screens. The routine for checking the validity of the required password is found and replaced with NOP's allowing any password to be used to gain access to the attacked system [Boileau 2006]. Unfortunately, these techniques are equally accessible to legitimate law enforcement agencies that use it for forensics investigations and criminal organizations and other attackers as well [Freiling and Vomel 2011].

One particularly effective attack, bus-snooping and injecting, allows information to be captured or inserted via the bus lines between system components [Boileau 2006]. This exploitation method has been used to undermine the Xbox gaming system. This system was specifically designed to provide a secure *chain of trust* for enforcing digital rights management (DRM). Bus-snooping was used to capture keys as they transited between read-only-memory and the CPU. These keys were then used to decrypt the secure boot loader, thus undermining the entire chain of trust. Subsequently, low-cost "mod" chips were developed that can be soldered into the gaming system bus, allowing a user to bypass DRM restrictions and play pirated games [Steil 2005]. Alternatively, the same chips can be used to run alternative operating systems on the gaming hardware

allowing it to be used for illicit purposes [Rabaiotti et al. 2010]. Far from being academic or theoretical in nature, these exploits of memory vulnerabilities have been used extensively by criminals for profit.

As encryption of data at rest becomes the status quo, attacks will begin to target vulnerable RAM. In effect, increasing adoption of these techniques has pushed the vulnerabilities associated with persistent data stored on disk down into the next level of the memory hierarchy, which itself has proven equally vulnerable. For example, *memory scraper* viruses, which target selective information from the volatile memory of point of sale (POS) applications running along side them, have been increasing since their appearance in 2008 [Baker et al. 2008]. These viruses are a subset of a larger problem with *mutually distrusting processes* (MDP). Mutually distrusting processes are those, for example, that are downloaded from an application store on a smart phone or tablet. Recent research suggests that between 0.20% and 0.47% of applications downloaded from alternative Android application stores are malicious while 0.02% downloaded from the official Android Market are malicious. While Apple goes to great lengths to review applications for security issues, malicious applications have been approved and downloaded from the official Apple application market as well [Jekyll on iOS 2013].

Fortunately, access to information in conventional dynamic RAM normally (i.e. not in the case of a cold boot attack) presents an adversary with only a *fleeting* opportunity to obtain sensitive information between power cycles. However, dynamic RAM is being augmented or replaced with new non-volatile alternatives such as flash memory, magnetic RAM, and ferro-electric RAM. These provide several benefits including energy efficiency and tolerance of power failure. Flash memory has also been

used to augment traditional RAM in the Vista and Windows 7 “ready boost” feature, whereas the other two technologies are potential RAM replacements. Unfortunately, these non-volatile memories allow information and attacks to *persist* indefinitely [Enck et al. 2008]. Interestingly, Microsoft has anticipated the security issues associated with persistent memory and has designed the ready boost feature to encrypt all contents of flash making it difficult for forensics investigators to recover useful data [Hayes et al. 2009]. If these memories are adopted in future architectures, without adequate attention to encryption or other protections, there is the potential that memory based attacks will become more prevalent.

There are three general attack vectors as described in [ARM security]: software based *hack* attacks, low-budget hardware attacks known as *shack* attacks, and resource intensive *lab* attacks. The attack vectors previously described fall into the first two of these categories. While many efforts concentrate on mitigating software-based attacks, such as buffer overflows, comparatively little effort has gone into preventing shack attacks. This may be partially explained by the difficulty in addressing attacks where an adversary has physical access of a system, for example, insiders. Shack attacks include those highlighted above such as bus-snooping and injecting, cold-boot attack and DMA attacks. Lab attacks involve significant time and equipment and examples include etching away chip walls with acid to reveal internal bus lines for microprobing, or electromagnetic and power analyses among other side channels [Ravi et al. 2004]; [Kocher et al. 1999]. For systems relying on *software based encryption*, key expansion tables (e.g. AES) are subject to cache attacks; a malicious process (MDP) tracks and

times cache accesses [Osvik et al. 2005], [Mowery et al. 2012]. The typical target of all of these attacks is the encryption key hidden within the chip boundary.

In general, encryption is used to provide four basic properties of protection: *confidentiality*, *integrity*, *authentication*, and *non-repudiation*. In trusted computing and operating system security these properties are realized through *authenticated booting*, ensuring that program code is not changed before it is loaded into memory, *memory authentication*, ensuring that program code is not changed during use, and *attestation*, ensuring that hardware and software have not been altered. Trusted software components, which make up part of the trusted computing base (TCB), are booted and verified producing a *chain of trust*, without which the security mechanisms could be compromised before the system is initialized. While few works discuss the implementation of these other mechanisms, most discussions assume that these components are functional and thus focus on the overhead of ME in the steady-state. Other important assumptions often include mechanisms for secure code delivery, key creation and escrow, inter-process communication, and I/O protection among others. Memory authentication is often closely associated with memory encryption solutions; however, a thorough survey of memory authentication mechanisms is available [Elbaz et al. 2009].

Memory encryption is solely concerned with the *confidentiality* of data and code during execution, with the express purpose of increasing attacker workload associated with crafting exploits and stealing sensitive information. It is interesting to note, however, that memory encryption also hampers attempts to inject code, which is generally assumed to require memory authentication. An adversary lacking an

encryption key would be unable to successfully change an encrypted binary, as decryption would result in corrupt code and likely program termination [Barrantes et al. 2003]. Early work associated with full memory encryption (FME) was dominated by the desire to provide digital rights management and in particular to prevent the theft of intellectual property associated with program source code. This is still the primary purpose in some systems (e.g. gaming systems), but more recently these techniques have become recognized as a method for removing vulnerabilities and protecting system users.

There are two general approaches to providing confidentiality with encryption that are commonly used in computer architectures based on *symmetric* or *public* key encryption techniques. Symmetric key encryption is based on a shared secret (key), and is generally held to be more efficient (i.e. on the order of 1,000 times faster) but it does not provide non-repudiation, and it requires a non-trivial trusted key distribution scheme [Kaplan 2007]. Three common algorithms are typically used to realize this approach based on DES, Triple-DES, and AES. Public-key encryption involves the use of two interlocking keys, one that is held privately and the other that is published, from which all four properties of protection, including non-repudiation, can be realized. This scheme has the advantage in that public keys can be distributed across open networks. A broad variety of books are available that describe these core ideas, [Mel et al. 2001] is particularly accessible. In light of the speed and complexity involved in public key encryption, it is unsurprising that the memory encryption literature typically espouses symmetric key cryptography. However, delivery of encrypted code over the network may be facilitated using the public key model [Kgil et al. 2005].

Unfortunately, computer users have consistently demonstrated an aversion to any form of increased response time, even when associated with increased security. Studies suggest that delays of longer than 150 ms are perceptible to users [Muller et al. 2011]. Full disk encryption has only become viable because overheads have been reduced to acceptable levels. Achieving similar levels of acceptable performance for memory encryption offers a far more significant challenge in that there is an existing, growing, and well-documented speed-gap between processors and memory. Improvements in processor speed are outpacing improvements in memory speed by an average of 18% per year [Hennessy et al. 2006]. Adding encryption latency to this already strained interface may require an overhaul of the basic fetch-decode-execute cycle employed by processors.

Added to the complexities of any memory encryption solution is the fact that, unlike the hard disk where data is sequentially stored for access, memory is used in a broad variety of dynamic access patterns. Numerous decisions must be made concerning the granularity of encryption in operating systems. For example, a running program will utilize RAM during execution for both stacks and heap space. The stack is accessed so frequently that adding encryption/decryption overhead to stack operations might prove prohibitive. Unfortunately, during context switches, registers containing sensitive information are normally saved to the stack in external memory. Additionally, the heap size, for any given program, is not normally known a-priori. The complexities of memory mapped input-output peripherals result in an inability to cache mapped regions. This naturally presents a challenge, if the overarching concept involves decrypting memory only after it is brought onto to the processor chip. It is not clear if the entire memory should be encrypted with a single key, or if shared libraries, individual programs, and/or

data should be encrypted independently using separate keys. Alternatively should individual functions or cache blocks be used as the unit of encryption? All of these decisions incur a tradeoff between the number of keys that must be securely stored, versus the degree of protection and overlapping in operations that can be realized.

The literature on memory encryption is largely concerned with three core approaches based on hardware enhancements, operating system enhancements, and specialized industrial applications. These approaches are explored in the sections that follow. Unfortunately, almost all of the hardware and operating system enhancements have only been implemented through simulation or emulation, and as a result, the claims have yet to be validated and quantified on practical systems.

2.2 Monolithic Processor Enhancements

The general scope of hardware enhancements includes a number of approaches that have added specialized encryption units and/or key storage mechanisms to existing processor designs. In addition, several efforts have proposed inserting hardware into the system bus to leverage legacy code and hardware. Although the first patents detailing memory encryption were executed in 1979 [Best 1979; Best 1981; Best 1984], and the first paper detailing their use was published in 1980 [Best 1980], the body of in-depth academic research related to general-purpose memory encryption has occurred primarily in the past decade.

One of the earliest papers, often referenced by others of this genre, highlights an *execute-only memory* (XOM) architecture [Lie et al. 2000]. This architecture was designed to combat software piracy and combines aspects of both public and symmetric key encryption. Public key encryption is used to deliver binary code to the XOM chip,

which maintains a unique private key. This allows vendors to encrypt the code for a particular system and ensures that it cannot be reused on another system. The header associated with the code includes a symmetric key embedded within it, which is used to segment memory into unique compartments at the granularity of a process. In order to map compartments to encryption keys, each compartment is tagged. A single null compartment is created to hold all unencrypted processes and libraries. This compartment enables communication between encrypted processes while allowing all processes to use shared libraries.

The XOM architecture assumes several hardware enhancements to existing processors. Special microcode is required to store the unique private key in a private on-chip memory. A symmetric-key encryption unit is added to the processor, together with a special privileged mode of operation for encryption. A hardware trap on instruction cache misses provides a segue into this encryption mode for encrypted code. When a cache miss occurs, the instruction is decrypted before being loaded into the processors instruction register. Although the authors state encryption could be accomplished in software they acknowledge that this would be very expensive in terms of overhead. Since many of the papers that follow XOM include similar hardware, only the differences or unique contributions of the other systems will be discussed.

XOM encrypts memory in a straightforward manner commonly known by the encryption community as *electronic codebook mode* but referred to in the literature as *direct encryption*. Each code block is decrypted by the encryption unit after it is read from memory, and encrypted before it is written back to memory. Kgil et al. [2005] propose an additional chip enhancement targeted at improving the security of direct

encryption, called ChipLock. This involves storing a small trusted part of an operating system kernel, called TrustCode, in a read-only memory (ROM), termed TrustROM. Additional instructions are added to enable secure communication between the trusted and untrusted parts of the operating system. The TrustCode intercepts all system calls for memory access and performs encryption without the knowledge of the untrusted portion of the operating system. Symmetric keys are assigned at the granularity of the process as in XOM, with additional keys for shared libraries and the concept of a null bit for applications that are not encrypted.

Rogers et al. [2005] attempt to improve on direct encryption using an alternative mechanism, *prefetching*. Prefetching uses stream buffers to capture *spatial locality* in programs by copying additional contiguous blocks of memory into local cache after each miss. These buffers are especially good at speeding up programs that exhibit *spatial locality* and *contiguous access*, such as scientific applications [Hennessy et al. 2006]. An alternative prefetching technique that involves correlation tables to capture and reuse *temporal locality,* i.e. complex and/or non-contiguous sequences of memory access, is also used.

In another direct encryption scheme, Hong et al. [2011] perform a tradeoff analysis on the use of sensitive (encrypted) versus frequently accessed (unencrypted) data in embedded scratch pad memories (SPM). SPM's are software controlled SRAMs, as opposed to caches, which are typically controlled by hardware. There are numerous papers that discuss both static and dynamic policies for SPM utilization to reduce power consumption and memory access latency. DynaPoMP was the first to consider partitioning the SPM into distinct areas with an area dedicated to sensitive code and data.

The authors vary the size of the two partitions in an attempt to find the most efficient ratio. There is a common assumption that an encryption unit and special instructions are available in hardware. Unfortunately, direct encryption schemes involve a one-to-one mapping between blocks of unencrypted and encrypted code. As a result, encrypted code portrays a similar statistical distribution as the unencrypted code, thus allowing a significant amount of information to be gleaned from frequency analysis [Chhabra 2010]. Based on the typical AES encryption block size of 128 bits, programs tend to exhibit multiple redundancies that would lead to information leakage as shown in Figure 2.

After XOM, a number of papers attempt to mitigate this statistical weakness using a one-time pad (OTP) [Suh et al. 2003; Shi et al. 2004; Yang et al. 2005; Yan et al. 2006; Suh et al. 2007; Duc et al. 2006]. A traditional OTP is simply a source of random data that is used exactly once to encrypt a particular communication. This is a form of symmetric-key cryptography since both the sender and receiver require the pad. Although variously referred to as “pseudo one time pads” (POTP) in the literature, this is more commonly known in the encryption community as counter-mode (CTR) encryption. In computing, OTP’s are created by encrypting a unique seed, typically producing a pad of 128 bits in length (i.e. the size of an AES encryption block) as shown in Figure 3. A fixed initialization vector (Nonce) is concatenated with a counter producing a unique seed. The seed is encrypted with a unique key generating the pad, which is then exclusively or’ed (XOR) with the plaintext to produce the cipher text. In memory encryption schemes, the counter is stored either internally, in a cached table that maps to a memory address, or unencrypted within the encrypted memory itself (i.e. RAM) since counter secrecy is not required [Yan et al. 2006]. When a memory reference occurs, the

pad is regenerated, using the counter (and optionally some other component such as the virtual address) and initialization vector, then exclusively or'ed with the encrypted data to produce the original plaintext. Since the encryption operation is no longer dependent upon the data in memory, this regeneration can be overlapped with the memory read, decreasing the performance impact of decryption.

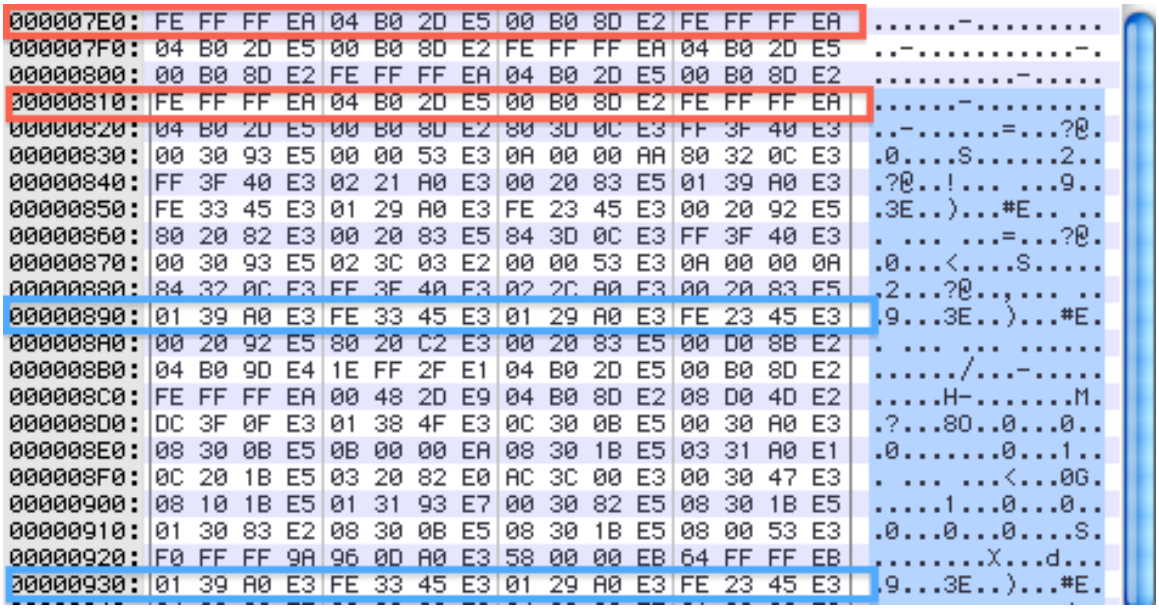

Figure 2: Redundancies in 128-Bit Sections of Binary Code

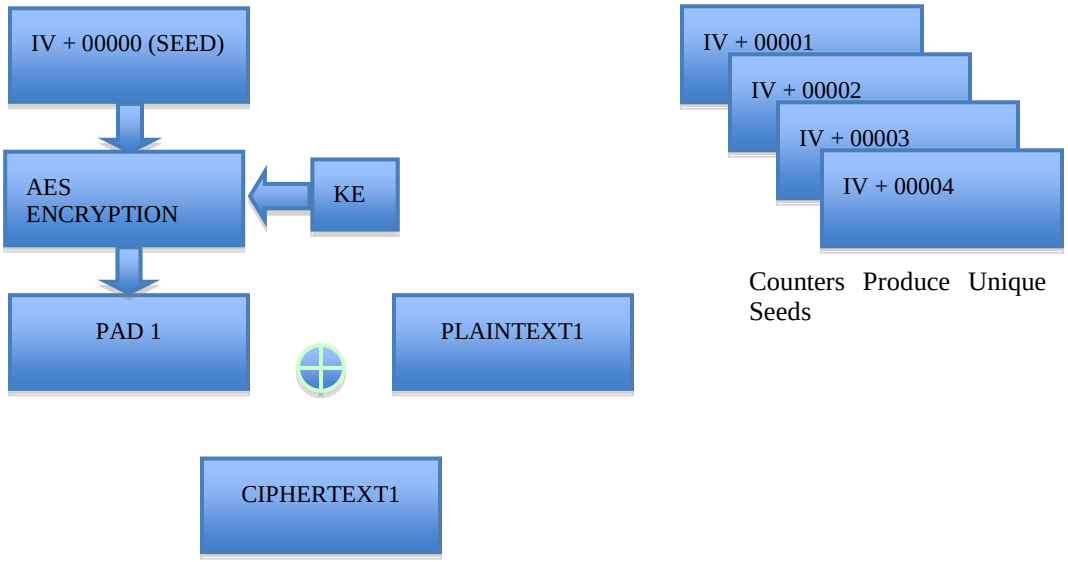

Figure 3: Pseudo One-Time Pad or Counter Mode Encryption

Aegis is a OTP approach that was originally proposed as a direct encryption scheme in 2003. Suh et al. propose the one-time pad approach in [2003b], perhaps illustrating the shift away from direct encryption in the community. One interesting contribution from this paper is the method of creating the unique key. The chip-specific encryption key is created by *physically unclonable functions* (PUF) [Suh et al. 2003a]. These functions make use of unique timing characteristics of "identical" models of the hardware to create the unique keys. Aegis is one of several approaches that include the idea of a small, protected security kernel that is separate from the rest of the untrusted operating system. Unfortunately, this kernel measures 74K lines of code for virtual memory management alone [Chhabra et al. 2011].

In Yang et al. [2005], the authors look to reduce the execution overhead of using one-time pads by adding a *sequence number cache* (SNC) onto the chip below the L2 cache. Sequence numbers, in this paper, correspond to the counters used in Figure 3. However, the initialization vector is unique per cache block and corresponds to the virtual address. Since the addresses are unique across memory, the pads (and thus the ciphertext) will be *spatially* unique. The counters are updated upon each write to memory ensuring *temporal* uniqueness (i.e. pads used for a single location will not be the same over time). The authors suggest that a reasonable addition to a chip would be a SNC of 64 KB. Based on this limitation, two policies for using the SNC are described. In the first policy, only the portion of memory corresponding to the number of available sequence numbers stored in the SNC can be encrypted. The amount of protected memory is therefore limited by the SNC size. In the second policy, additional memory lines are encrypted and sequence numbers that do not fit in the cache are stored in plaintext in

memory. Level two cache is increased in both methods by 4% in order to store the virtual memory addresses used to index into the SNC since only physical addresses are typically available above the level one cache.

In [2006], Yan et al. present *split counter mode* encryption, in which they introduce major and minor page counters. In this scheme, a 4 KB page has one 64 bit major counter and 64 7-bit minor counters (one per 64 Byte cache line). Concatenating the page major counter with the cache line minor counter forms the overall counter. This counter is further concatenated with the memory block's virtual address, and an initialization vector to form the unique seed. The vector can be unique per process, group of processes or system based on security requirements.

In CryptoPage [Duc et al. 2006], the authors again attempt to enhance the OTP encryption scheme. In this case, they modify the translation look-aside buffer (TLB) and page table structures, adding information for pad computation. Since the TLB and/or page table structures are always accessed before a memory read, the authors claim that the pad generation latency can be almost completely removed. This scheme is implemented on top of the *HIDE* memory obfuscation technique whereby access patterns are permuted in memory at designated times [Zhuang et al. 2004].

In *address independent seed encryption* (AISE) [Rogers et al. 2007], the authors propose to use a *logical* identifier, rather than the virtual or physical block address, as the major counter portion of the seed. This scheme closely resembles split mode counters [Yan et al. 2006]. It is claimed that using an address independent seed enables common memory management techniques, such as virtual addressing, paging, and inter-process sharing. In [2011], Chhabra et al. propose to build a secure hypervisor upon the AISE

substrate. The hypervisor implements *memory cloaking*, whereby the operating system only has access to the encrypted pages of applications. The authors suggest that this cloaking will protect processes from vulnerabilities in the insecure underlying operating system, with an order of magnitude fewer lines of code than in Aegis.

In [2007], Nagarajan et al. propose compiler-assisted memory encryption for embedded processors assuming some limited hardware support. They claim that the current counter mode solutions require too much silicon space for small and medium size embedded processors. The compiler supports memory encryption by introducing special instructions to calculate OTP's prior to loads and stores, and assumes the existence of additional process-unique registers used to store the counters. Space for the unique key and global counter is also provided inside the CPU and the availability of a crypto unit is assumed. The compiler attempts to ensure that the counter used for a store is still available for successive loads from the same memory location. A global counter must be available for those loads and stores that do not match one of the process-unique counter registers. The authors claim that since frequently executed loads and stores exhibit highly accurate counter matching, 8 special hardware registers with 32 counters are sufficient for reasonable performance.

2.3 Multi-Processor Enhancements

Chhabra et al. [2010] compare a symmetric multiprocessor (SMP) and a distributed shared memory (DSM) design; they also provide a quick look at monolithic memory encryption. Whereas the efficiency of memory-to-cache confidentiality is the primary concern for monolithic processors, multiprocessor systems must also protect cache-to-cache traffic. In symmetric multiprocessors, the shared bus between caches and

memory can be used as a way to coordinate messages between processors. This sharing is not available in distributed shared memory systems, which must use message passing. Additionally, DSM systems can be observed more easily than monolithic chips via interconnect wires that are exposed at the back of server racks [Rogers et al. 2008].

In [2004], Shi et al. use OTP encryption both for memory-to-cache and cache-to-cache transfers as shown in Figure 4. In this approach sequence numbers (counters) are incremented in lockstep in each separate processor resulting in a claim of "very low" overhead for cache-to-cache encryption. A hardware mechanism in the processors ensures that the sequence numbers begin differently after each reboot. Besides the typical crypto-engines placed within each processor core, a separate crypto-unit is embedded in the north bridge memory controller for memory-to-cache transfers. For these transfers, 64-bit sequence numbers are stored in RAM reducing the available memory by 25 percent.

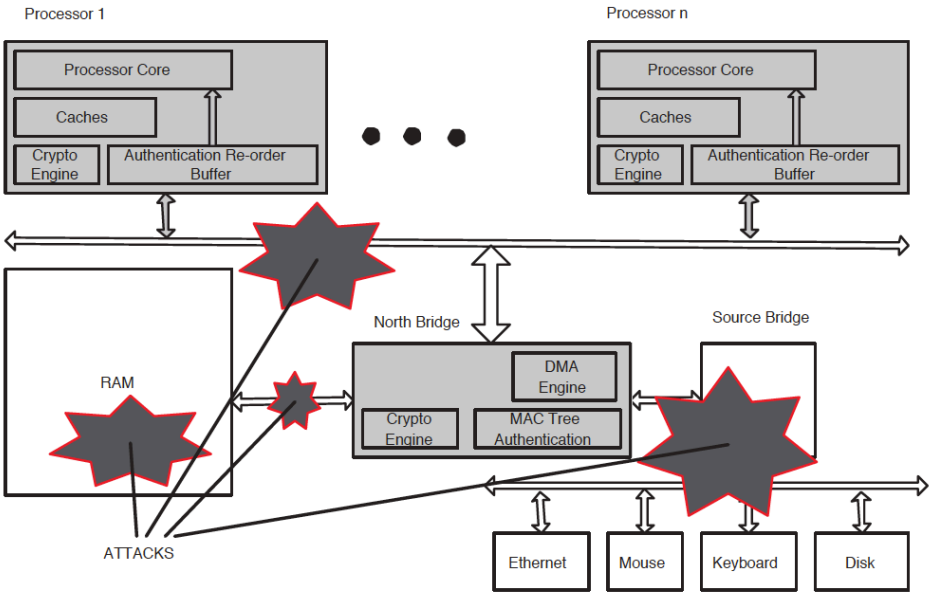

Figure 4: SMP Architecture with Memory Encryption Engine

In SENSS [2005], Zhang et al. utilize OTP's for memory-to-cache transfers and AES cipher block-chaining mode for cache-to-cache transfers. This alternative to direct encryption divides the clear text into blocks and encrypts the first block with an initialization vector; subsequent blocks are chained together such that the output of the previous block is XOR'd with the input of the next before being encrypted. Cipher block chaining implies sequential access since each block depends upon each previous block. RAM is typically accessed in a fairly random pattern, so this mode of operation is impractical except on a very small scale (per cache block for example). Cipher block chaining is acceptable for cache-to-cache transfers as only one previous encrypted block must be stored at each processor (i.e. there is no requirement for access to previously encrypted blocks). The authors propose a secure hardware unit (SHU), located at each processor, comprising an encryption unit with associated storage for keeping track of communication. This storage includes memory for a group processor matrix and group information table. The group processor matrix is used by each SHU to determine if broadcast messages should be read. The matrix is only 640 bytes in size, assuming a maximum of 32 processors. The information table contains the secret information for communicating between groups, such as the symmetric key and pads, and is estimated at 149 KB. An additional 11 bus lines are used for control signals and to pass group id numbers. In Jannepally et al. [2009], the SENSS scheme is improved using Galois Counter-Mode (GCM) AES, which provides both encryption and authentication simultaneously.

In I2SEMS [2007], Lee et al. create a scheme that is claimed to be applicable to both SMP and DSM systems. They propose a global counter cache (GCC) that assigns

different sections of the overall counter space to processors (akin to assigning blocks of IP addresses to groups of computers). The blocks of counters are also broadcast to all processors so that they can begin pre-computation of pads. Each processor has a keystream (pad) queue, keystream cache and keystream pool. The queue and cache both contain pads for encryption. The queue has new pads while the cache contains pads that have been used previously. The authors claim that pads may be reused as long as the plaintext has not been modified and that their scheme scales well to large numbers of processors since over 25% of pads are reused. The keystream pool holds pads for incoming data. The pads are chosen based on prediction with the aid of the broadcast scheme.

The first paper to exclusively address DSM systems [2006] was by Rogers et al., who again make use of counter mode encryption. Since the memory-to-cache scheme is similar to those already discussed, we only focus on the cache-to-cache scheme. The authors propose three methods for managing the pad counters: *private, shared,* and *cached* counter stream. In the first *private* method, tables are kept within each processor with separate counters for send and receive operations to/from every other processor in the system. While this technique allows for nearly perfect pad hit rates, and therefore very low overhead, it suffers from large storage needs (180KB in each processor for a 1024-processor DSM). The second *shared* scheme, aims to reduce the storage requirement by eliminating half of the table: Instead of keeping track of send counters for each processor, only one counter is kept for sending pads. This results in increased execution overhead since messages are less likely to arrive contiguously and therefore must be recomputed. The final *cached* scheme takes advantage of the intuition that

processors in DSM systems often communicate in cliques [Lee et al. 2007]. The overall table size is thus reduced to a quarter of the private scheme's memory with minimal impact on execution overhead. In a subsequent paper [2008], Rogers et al. identify the previous scheme as a two level approach since remote memory requests will first be decrypted by the owning processor and then re-encrypted for cache-to-cache transmission to another processor. In the new scheme, a single mechanism is used for both memory-to-cache and cache-to-cache transfers bypassing the unnecessary decryption and re-encryption. The associated hardware includes a 32-entry buffer (1 KB) for counter prediction and a 32-entry mask buffer that stores a bit vector of recent data block accesses (512 bytes).

2.4 Bus Inserts

Another area of active research involves placing specialized encryption hardware outside of the CPU. The locations include the memory bus (i.e. externally between system memory and the CPU) and within RAM. The primary goal of this approach is to increase the likelihood that this solution will be adopted since re-engineering of commodity processors is not required. One such approach, SecBus [Su et al. 2009] shown in Figure 5, can be located at the frontend of the memory controller. The authors state that this method of modification is required in many user markets when embedding new functionality into systems with legacy CPUs. SecBus is essentially a cryptographic coprocessor with internal storage and bus manager. The page security parameters entry (PSPE) includes information to map pages to corresponding security policy (SP), which includes a confidentiality mode, integrity mode and secret key. SecBus includes the

ability to choose between multiple encryption modes based on the type of memory (i.e. code or data).

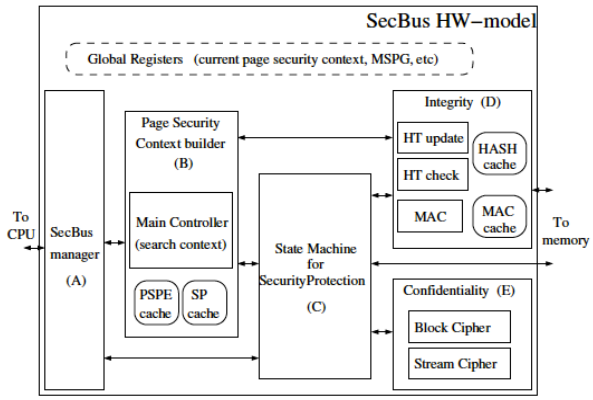

Figure 5: SecBus Hardware Augmentation Model

In [2008], Enck et al. design a memory encryption control unit (MECU) to again be placed on the memory bus between the processor and RAM. The goal of MECU is to provide the same guarantees of security provided by the volatility of traditional RAM when utilizing non-volatile main memory. MECU uses a OTP scheme with internal storage for the array of counter seeds and the encryption engine. A secret key and master counter, which tracks the greatest overall counter, are stored on a removable smart card. In order to reduce the storage requirement, the encryption chunk granularity is increased from one cache line to *n*, where *n* is 256 in the common case but can grow to the entire memory for experimentation.

With the same goal as [Enck et al. 2008], Chhabra et al. [2011] propose placing the cryptographic engine and other required hardware in non-volatile RAM modules. Their scheme keeps most of the RAM encrypted with a smaller group of frequently accessed pages in plaintext in a similar fashion to [Hong et al. 2011]. The authors claim

that by doing this, the remainder of the RAM can be encrypted at power-down within 5 seconds, paralleling traditional RAM volatility.

2.5 Operating System Enhancements

Similar to the bus insert method for enabling memory encryption, software-only approaches seek to provide solutions that can be implemented without major changes to applications or commodity hardware to increase the likelihood of adoption.

In [2008], Chen et al. propose an operating system controlled memory bus encryption technique for systems that offer scratch pad memory (SPM) or cache locking that is software controllable. Both types of memory are available in some embedded processors including the Intel XScale series. A new symmetric key is generated each time the system is booted and random vectors (32 bits generated using /dev/urandom and padded with 0's) are used to initialize AES encryption at the granularity of a page. The vectors are then placed in memory with the pages. This scheme requires a 0.4% space overhead when used with 1 KB pages. When a page fault occurs for a secure process, a specially crafted handler moves the encrypted page into the chip boundary and decrypts it there placing it into the cache, which is then locked to prevent leakage of sensitive data. The locked region holds several pages of data and encryption variables. In order to facilitate this special handling, a Boolean status variable is added to each process descriptor residing in kernel address space. The authors note the scheme is appropriate when embedded systems designers can tolerate a significant performance overhead for protected processes.

In Cryptkeeper [2010], Peterson modifies the virtual memory manager and partitions RAM into two parts; the plaintext *Clear* and the encrypted *Crypt*. Essentially,

this technique aims to reduce the amount of sensitive data available at any time in memory. All pages initially start in the clear and the number of Free Clear Pages (FCP) is reduced with each allocation. The least recently used pages are encrypted and moved to the Crypt when the limit of FCP runs low. This operates under the assumption that the number of high use pages will be small, and therefore most infrequently used pages will be encrypted. This has the unfortunate side effect of maintaining all the important pages in the clear. A prototype Cryptkeeper system was designed based on the Linux 2.6.24 kernel. The kernel page structure was extended to include information indicating whether a page is in the Clear or Crypt portions of memory.

2.6 Specialized Industrial Devices

Industry offers several solutions for memory encryption including low frequency specialized processors for ATM use, expensive tamper resistant coprocessors for financial transactions, proprietary gaming systems and, more recently, adding technologies in commodity processors that enhance trust.

The Dallas Semiconductor 5002FP secure processor is an 8051 compliant processor and runs at a maximum frequency of 16 MHz [Dallas 1997]. The processor encrypts memory addresses to prevent traffic analysis on the memory bus in addition to data. The device uses spare processor cycles to place dummy memory accesses on the bus since analysis of memory access patterns can reveal useful information (e.g. encryption keys or sensitive algorithms) to attackers [Gao et al. 2006]. All external memory is encrypted via a proprietary encryption algorithm with a 64-bit secret key that is stored in a tamper-protected, battery-maintained static RAM. Plaintext code is

uploaded via serial port and a firmware monitor encrypts it and stores it in external RAM. The 5002FP is commonly used in credit card (i.e. point of sale) terminals, automated teller machines, and pay-TV decoders [Yang et al. 2005]. A newer version (DS5250) includes a larger 1 KB instruction cache, which, according to Dallas Semiconductor, reduces the effect of memory encryption on execution speed providing a 2.5X performance improvement. The newer processor runs at a maximum frequency of 25 MHz.

Another active area of secure hardware used in industry is the cryptographic coprocessor such as the IBM PCI-4758. These coprocessors include an impressive array of technology including a secure processing environment, microprocessor, custom encryption and random number generation hardware, and shields and sensors (to help protect against destructive attacks) [Howgrave-Graham et al. 2001]. However, they are generally limited to IBM server platforms under customized contracts and tend to be used for financial and banking systems. A modified version of CP/Q message-passing kernel runs on the system providing a subset of typical features. The secure module is encased in a flexible mesh of overlapping conductive lines meant to prevent any physical intrusion. If such intrusion is detected the system responds by zeroizing the internal RAM which holds the secret key. The stated purpose of the IBM secure coprocessor is to offload computationally intensive cryptographic processes (e.g. specialized financial transactions) from the host server.

While mostly constrained for use in playing games and other entertainment media (unless compromised) gaming systems are some of the most capable (e.g. fast processor speed and relatively large storage) to incorporate memory encryption techniques. As an

example of these systems, the Xbox 360 provides encrypted/signed bootup and executables, partially encrypted RAM, and an encrypted hypervisor [Steil and Domke 2008]. These mechanisms are provided via a Microsoft proprietary processor with 64 KB of internal RAM, random number generation and encryption as opposed to the "off the shelf" processor used in the original Xbox. While it is possible to use the Xbox as a general-purpose platform, this requires compromising the system's security measures first. Alternatively, the Sony Playstation 3 includes many of the same security mechanisms of the Xbox 360, but allows the end user to partition the hard drive for use with a chosen (e.g. Linux) operating system. However, the proprietary security mechanisms of the Playstation 3 are not available to the additional operating system [Conrad et al. 2010].

The trusted computing group (TCG) designed the Trusted Platform Module (TPM) based on the IBM 4758 secure coprocessor [Vandana 2008]. The TPM provides secure key storage and the capability for platform measurements for chain-of-trust booting. The current specification for the TPM calls for it to be attached to a typical motherboard via the low pin count (LPC) bus. The TPM provides non-volatile storage for encryption keys and an encryption engine including support for RSA, SHA-1 hashing, and random number generation. The LPC bus is limited in speed and the cryptographic engine on the TPM is not meant to be a cryptographic accelerator. Over 350 million TPMs were deployed as of 2010 and can be found in many laptops and general-purpose computers (disabled by default) [Dunn et al. 2011]. On its own, the TPM would not be powerful enough to provide general memory encryption with acceptable overhead. However, the TPM may be used to provide secure key storage between power cycles.

Unfortunately, a small weakness still exists in that keys must be sent in the clear over the LPC bus to the CPU, allowing a bus snooping attack to capture them [Simmons 2011]. Other interesting methods to store encryption keys have been described recently in schemes targeted at preventing cold-boot attacks on full disk encryption. For example, Muller et al. describe TRESOR, a technique for utilizing CPU debug registers for encryption key storage [2011]. In order to protect against memory attacks on the key, the decryption routines are carefully written in assembly to avoid using the stack, heap or data segment during decryption. By utilizing AES-NI, TRESOR was shown to perform better than software based full disk encryption (17.04 MB/s vs. 14.67 MB/s) with the additional protection. A similar approach is taken in [Simmons 2011] except that registers used for performance counting are targeted for master key storage with multiple encrypted keys being stored in RAM.

Intel has recently filed several patents for processors incorporating memory encryption, perhaps indicating a move toward support in commodity processors [Gueron 2012], [Gueron 2013]. The patents describe a new processor with hardware including a memory encryption engine (MEE) and on-chip storage for counters. The hardware described in the application modifies the AES-XTS *tweak* mode of operation. XTS stands for XEX based tweaked codebook mode with ciphertext stealing and this mode is typically used for disk encryption [Martin 2010]. A tweak is similar to an initialization vector and is an additional input to a cipher designed to protect against similarities in ciphertext. For disk encryption, the tweak tends to be the sector number. In Intel's patents, the tweak is extended to include a time stamp or counter value along with the memory address. The counter is updated each time a cache line is written, providing

protection against a replay attack where a chunk of memory is copied and inserted back into memory at a later time.

2.7 Commoditized Security Hardware

Most of the approaches in the ME literature assume that several components are necessary for secure, efficient performance: a way to generate and securely store encryption keys (i.e. not in RAM); and hardware to accelerate encryption performance. Although not targeted specifically at memory encryption, nascent technology could be used to form the basis of an encrypted memory solution for general-purpose systems. One of the developers of IBM's 4758 cryptographic coprocessor has suggested, for example, that a general-purpose system with hardware support (such as a trusted platform module) could theoretically be turned into a somewhat less secure but more pervasive and less expensive version of the 4758 [Smith 2004]. Encryption engines have been added to Intel's core i5 and i7, AMD's bulldozer and various embedded processors [Muller et al. 2011]. For X86 systems, Intel's advanced encryption standard - new instructions (AES-NI) include six instructions to speed up key expansion and encryption. Intel states that the new instructions can provide a two to three time performance improvement over software-only approaches for non-parallel modes of operation such as cipher-block-chaining (CBC) encryption [Gueron 2010]. Further, a 10-fold improvement can be realized for parallelizable modes including CBC-decrypt and counter-mode encryption (CTR). As an example of the performance improvement possibilities, the authors ran TrueCrypt's encryption algorithm benchmark test on a MacBook Pro with an Intel i7 dual-core, 2.66 GHz CPU. Using a 5 MB buffer in RAM, the throughput averages 202

MB/s without AES-NI support, and 1 GB/s with it, thus approaching the speed required to overcome encryption overheads on general-purpose systems.

Henson and Taylor [2013] are among the first to take advantage of this commoditization of security hardware for use in implementing memory encryption. The IMX53 development board is used in conjunction with an ARM cortex A8 processor that contains security hardware within its boundary. The hardware consists of encryption and hashing engines and random number generation as well as facilities for trusted booting. A small (~35KB) microkernel called *Bear* is developed and integrated with the A8 and security hardware. As the work is implemented on commodity hardware, it considers many of the details that are not thoroughly addressed in the other surveyed literature (i.e. simulation work). For example, encryption is explored at process component granularity (e.g. stack, heap, code) with analysis of the overhead for encrypting each component. The work takes advantage of on-chip, internal RAM (iRAM) as well as cache to provide the secure processing environment. Outside of this chip boundary, all code and data are encrypted. Most of the ME functionality is tied to the context switching routines in the microkernel. The microkernel fits into the iRAM and is part of the TCB in this work.

2.8 Analysis

Although the primary goal of memory encryption architectures is security, the work tends to focus on the overheads involved, both in chip area and performance degradation. This is unfortunate though unsurprising given that most of the work is simulated and it is within the intricacies of implementation that security vulnerabilities tend to be found. The analysis here focuses on the data available including encryption latencies, performance degradation, simulation environments, operating system

assumptions, overall space requirements, user requirements and general observations regarding security.

Since the performance degradation of memory encryption results in less likelihood of its use, it is an extremely important factor in the comparison of different schemes. One of the major issues with the body of literature is the lack of a common set of measurement standards, with explicit assumptions regarding memory access latency, encryption latency etc. This makes it difficult to directly compare approaches and draw valid conclusions. Encryption latencies are typically given as the number of cycles required to encrypt/decrypt a cache line that varies from 16 to 128 bytes, typically using a value of 64 bytes. The latencies range from 11 to 160 cycles with 80 being the most common value (especially in the multiprocessor work). The authors in [Rogers et al. 2006] state that 80-cycle latency is assumed in order not to penalize the direct encryption scheme (upon which they are trying to improve) since a recent (circa 2006) hardware implementation required over 300 ns. Cycles and nanoseconds are often used interchangeably since many of the systems modeled are based on 1 GHz processors. Low encryption latencies are possible but at the cost of large die area making them appropriate for powerful processors. For example, it is claimed in [Suh et al. 2003] that 40 cycle-latency is achievable with four AES units chained together requiring 300,000 gates. In AEGIS [Suh et al. 2007], a single AES unit is estimated at 86,655 gates, which the authors claim is modest when compared to the size of commercial cores. Unfortunately, the OR1200 soft core used to demonstrate Aegis is only approximately 60,000 gates (meaning one AES unit is 144% of the original core size).

The methods used for determining performance include mathematical models, simulation, kernel prototypes and FPGA prototypes with various benchmarking suites used in the latter three. Simulation is performed with (in order of decreasing usage) SimpleScalar, Simics, SESC, GEMS, SOC designer, RSIM, and M5. Benchmark suites used include SPEC2000, SPLASH2, Mediabench, EEMBC and several user developed varieties such as one entitled “memeater”. A group of the simulations utilize SimpleScalar and [Duc and Keyell 2006] notes that this simulator neglects the impact of the operating system and other running processes. Besides these limitations, some authors admit a lack of model fidelity with significant differences between systems modeled and those targeted. For example, in [Chen et al. 2008] an x86 architecture is modeled since it happens to be better supported by the simulation tool (Simics) even though the scheme is actually targeted for embedded-ARM systems. Unfortunately, even if a system under test were to be modeled perfectly, the simulation tools themselves have been shown to sometimes exhibit behavior unlike real systems. In [Muller et al. 2011], the behavior of CPU registers is interrogated under simulation in QEMU with the contents surviving soft-boot. Such behavior would circumvent the protections afforded in that work, however, real hardware behaves differently and zeroes out the registers.

A summary of the featured techniques is presented in Table I to provide an overview of memory encryption. The table includes basic characteristics of each approach such as complexity information including execution and storage overheads. In order to fairly compare the different schemes, several assumptions were made. For example, the size of internal storage required is sometimes dependent on the size of RAM, and where possible an assumption of 1 GB is made. Similarly, an assumption of

32 processors is made where possible for the multiprocessor approaches. When there is no data available, an element of the table is left blank. Two values are commonly reported in the literature with regard to execution overhead: worst case (max) and the average (based on some suite of benchmark tests) percentage slowdown when compared to non-protected execution. Storage overheads typically break down into internal (cache) and external (RAM) usage (and one example of the increase to overall code size). Operating system approach indicates whether the authors assumed the existence of a secure kernel (A), described hardware to protect the processes from an insecure kernel (H), or ignored the operating system (I) (further discussion of this requirement below). Finally, slightly fewer than two-thirds of the authors included memory integrity (I) along with memory confidentiality (C) mechanisms. Where possible, results (e.g. execution overhead and storage) are provided for memory encryption only. Maturity indicates how the technique was evaluated if not a commercial product. Methods appear in the table as they are presented in the survey and detailed in the approach column: monolithic processor, multiprocessor, bus insert, or software/direct or counter mode encryption.

Security level refers to the overall security of the ME approach with the following factors from the table taken into consideration: category, operating system approach, encryption algorithm, and partial vs. full ME. Specifically, the five sections are scored with maximum points as follows: category (1), operating system approach (2), encryption algorithm (2), and encryption level (1). A score of 6 represents a system capable of addressing a wider range of memory threats than those with lower scores. For category, no points are given for bus inserts and software approaches due to inherent weaknesses of these techniques when compared to hardware approaches. The operating system

approach is scored as follows: hardware (2), assumption of secure OS (1), no discussion (0). The encryption algorithm used receives 2 points for AES and 1 point for DES or unknown algorithms. We will consider partial/full memory encryption and security level in more detail. While partial memory encryption schemes are typically used to decrease both space and execution overheads, they place the onus for identifying secure components, a non-trivial task, on application or system designers. Today, an analog can be observed in the adoption of hard disk encryption technologies, whereby administrators struggling to identify which files (or parts of files) require encryption are opting instead for full disk encryption [Brink 2009]. Since it is difficult for end users to properly determine which processes should be encrypted, partial memory encryption receives 0 points with FME receiving 1.

A comparative analysis on the relative security of these techniques is nontrivial and it is important to note that the analysis in this work favors approaches that aim to mitigate a wide range of threats over those with a narrower scope. For example, full memory encryption will receive a higher score than an approach that adds volatility to magnetic RAM making it behave more like traditional RAM. Additional factors to consider when analyzing these works include consideration of implementation details outside of the "steady state" such as key escrow, delivery of secure code, inter-process communication, etc. although these are not used for the purposes of scoring.

Table 1: Summary of Memory Encryption Techniques

Reference	Category	Execution Overhead Max/Average %	Storage Overhead (I)nternal, (R)AM, (C)ode	Operating System Approach	Maturity	(C)onf (I)ntegrity	Encryption Algorithm	Full/Partial Memory Encryption	*Security Level 1-6*
[Lie et al. 2000]	Mono/Direct	50 /	I – Private Mem & XVMM	H + Virt Machine	Math	C + I	3 DES	PME	*3*
[Kgil et al. 2005]	Mono/Direct	/ 3	I – Key Table	H + Part of Kernel	Sim	C + I	AES	FME	*6*
[Rogers et al. 2005]	Mono/Direct	/ 1	1.5 MB-I	I	Sim	C	UNK	FME	*3*
Suh et al. 2003]	Mono/Counter	32 / 4.5	12 KB-I 6%-R	H + Part of Kernel	P-FPGA	C + I	AES	FME	*6*
[Yang et al. 2005]	Mono/Counter	/ 3.9	64 KB-I 1.5%-R	H	Sim	C	AES	FME	*6*
[Yan et al. 2006]	Mono/Counter	9 / 2	32 KB-I 1.5%-R	A	Sim	C + I	AES-GCM	FME	*5*
[Duc and Keryell 2006]	Mono/Counter	7.4 / 3		H	Sim	C + I	AES	PME	*5*
[Nagarajan et al. 2007]	Mono/Counter	/ 2.3	32 B-I 12.5%-R 3%-C	I	Sim	C	AES	FME	*4*
[Chhabra et al. 2011]	Multi/Counter	13 / 5.2	32 KB-I 1.6%-R	H	Sim	C + I	AES	FME	*6*
[Rogers et al. 2007]	Multi/Counter	13 / 1.6	32 KB-I 1.6%-R	I	Sim	C + I	AES	FME	*4*
[Shi et al. 2004]	Multi/Counter	55 /	32 KB-I 25%-R	A	Sim	C + I	AES	FME	*5*

Table 1: Summary of Memory Encryption Techniques Continued

Reference	Category	Execution Overhead Max/Average %	Storage Overhead (I)nternal, (R)AM, (C)ode	Operating System Approach	Maturity	(C)onf (I)ntegrity	Encryption Algorithm	Full/Partial Memory Encryption	*Security Level 1-6*
[Zhang et al. 2005]	Multi/Counter	/ 12	149 KB-I	H	Sim	C + I	AES-CBC	FME	*6*
[Jannepally et al. 2009]	Multi/Counter	/ 5.2	4.8 KB-I	I	Sim	C + I	AES-GCM	FME	*4*
[Lee et al. 2007]	Multi/Counter	10 / 4	608 KB-I	I	Sim	C + I	AES-GCM	FME	*4*
[Rogers et al. 2006]	Multi/Counter	/ 6	4.5 KB-I	I	Sim	C + I	AES	FME	*4*
[Rogers et al. 2008]	Multi/Counter	7 / 1.6	33.5 KB-I	I	Sim	C + I	AES	FME	*4*
[Su et al. 2009]	Insert/Direct	23,753/ 100		A	Sim	C + I	UNK	FME	*3*
[Enck et al. 2008]	Insert/Counter	4.4 / 2.1	131 KB-I	H	Sim	C	UNK	FME	*4*
[Chhabra and Solihin 2011]	Insert/Direct	20 / 7.2	78 MB-I	H	Sim	C	AES	PME	*4*
[Chen et al. 2008]	Soft/Direct	78 / 37	0.4%-R	A	Sim	C	AES-CBC	PME	*3*
[Peterson 2010]	Soft/Direct	800 / 9		I	P-Soft	C	AES-ECB	PME	*2*
[Henson and Taylor 2013]	Mono/Direct	50 /		Microkernel	P-Hard	C	AES	FME	*6*

Each approach is qualitatively evaluated on the five components listed above receiving a total score ranging from 1-6. As an example, the Aegis approach [Suh et al. 2003] is among the highest security level of the works surveyed (6): the category is monolithic processor with encryption support built in (+1); the operating system approach includes both hardware and a small, trusted kernel (+2); the AES encryption algorithm is used (+2); and full memory encryption is provided (+1). While not part of the score, much of the additional details required for a fully functional, secure implementation are discussed in Aegis. It is not surprising that the approach with the highest security evaluation is also among the most mature (implemented as an FPGA prototype) since implementation allows for exploration of security tradeoffs. In contrast, operating system controlled ME [Chen et al. 2008] is classified among the lowest security levels (3): this approach is software based (+0); *assumes* the kernel is secure (+1); utilizes AES encryption (+2); and targets partial memory-encryption (+0). Additionally, this work lacks sufficient detail for a fully functional system and assumes the attacker is a *clever outsider*.

For direct encryption, the performance overhead ranges from a claimed low of 1% in [Rogers et al. 2005] based on simulation of pre-decryption to a high of 50% for XOM [Lie et al. 2000] using mathematical analysis based on a worst-case scenario. Rogers et al. find an average slowdown for a model of XOM of 21% based on the same 18 SPEC2000 benchmarks used in their own simulation work. In four particular benchmarks (applu, bt, ft, and mcf) the overall execution time for pre-decryption is similar to the direct encryption scheme because prefetching adds mis-predicted memory references to bus traffic increasing contention. Overhead for OTP based encryption, in monolithic

chips, ranges from a claimed 1.6% for AISE (SESC and 21 CPU2000 benchmarks) [Rogers et al. 2007] to up to 50% for the basic model in CryptoPage (SimpleScalar and 10 CPU2000 benchmarks) [Duc and Keryell 2006]. The authors of CryptoPage claim only 1% of this overhead is attributable to the memory encryption.

For multiprocessor systems, the reported overheads range from a low of 4% in I2SEMS (Simics + GEMS and 4 SPLASH2 benchmarks) [Lee et al. 2007] to a high of 55% in [Shi et al. 2004] (RSIM and 6 SPLASH2 benchmarks). I2SEMS is claimed to work equally well on both SMP and DSM systems but the simulation environment is limited to SMP. Cache-to-cache overheads are very low (especially for SMP systems that use the shared bus for synchronization) in these multiprocessor schemes. All of the multiprocessor schemes build upon work in the monolithic memory encryption area and use the counter mode (OTP) model.

There are only two models surveyed for hardware insert and they exhibit very different performance characteristics. MECU [Enck et al. 2008] is based on the OTP scheme and exhibits 2.1% and 4.1% overhead based on block sizes of 256 and 4096 cache lines respectively and SimpleScalar simulation with 5 SPEC2000 benchmarks. SecBus [Su et al. 2009] is based on direct encryption and exhibits worst case slowdowns of 472% based on various EEMBC benchmarks and SoC designer. Besides the method of encryption, the architectures modeled add to the significant differences in overhead. While SecBus is simulated on an embedded system with 16KB L1 cache and no L2 cache, MECU is modeled after an x86 system with 32KB L1 and 256KB unified L2. Clearly, the amount of cache available has a huge impact on performance. If complete

working sets fit into a system's cache, the penalty for memory encryption includes only the initial decryption time, which is amortized across the entire duration of the process.

As might be expected, the software-only approaches suffer from impractical overheads. In [2008], Chen et al. simulate operating system controlled memory encryption and report from 137% to 850% overhead based on Simics and Mediabench benchmarks. In Cryptkeeper [Peterson 2010], the overhead to read a page when compared to an unprotected system is 6015%. In regard to commercial hardware, there is no literature available that reports the performance degradation of either the Dallas Semiconductor chips or the IBM cryptographic coprocessors (e.g. PCIXCC). However, these solutions run at slow overall frequencies (25 MHz and 266 MHz respectively) and are not particularly well suited for general-purpose systems. The IBM PCIXCC coprocessor has a reported AES-128 throughput of 185 MB/s.

In general, the counter mode methods exhibit less computational overhead than the direct encryption techniques and are resistant to direct encryption's statistical weaknesses. However, the choice of size for the counter is critical since a "wraparound", whereby the counter resets to zero, requires a change of key in order that each pad is only used once (a condition necessary to ensure protection from chosen plaintext attacks) [Lipman et al. 2000]. In the case where only one key is used the entire memory then requires re-encryption. This re-encryption can be costly depending on the size of memory and results in a temporary freezing of the system, which is unacceptable for real-time performance [Yan et al. 2006]. Choosing a value too small will result in too many re-encryptions but choosing one too large will require unacceptable amounts of storage space either in cache or memory. For example, in [Suh et al. 2003] the authors suggest

32 bits is an appropriate size for the counter. However, even at this size, and based on their simulations, a re-encryption is required every 5.35 hours on average and every 35 minutes for a particularly memory intensive program. In [Yang et al. 2005], the authors choose to disregard the problem since the provided security is assumed to be no weaker than that of the XOM scheme, whereas the wraparound issue is not considered at all in [Suh et al. 2007]. In [2006], Yan et al. attempt to address the counter size versus re-encryption problem with their split-counter encryption scheme. With larger page counters and multiple smaller per-memory block counters, overruns result in a much finer granularity of re-encryption (per page instead of per process). Since some pages are written back to memory more often than others, the overall necessity for re-encryption is reduced since the fastest incrementing counter would have controlled the entire memory space in previous schemes. Another critical decision is where to store the counters.

Although using cache is obviously faster, it is also problematic as cache resources are typically limited and expensive. If pre-existing cache space is utilized instead, additional memory references occur since part of processes' working sets are forced out of cache (essentially reducing the size of the usable cache causing capacity misses). For example, in [Yang et al. 2005] the authors state that a 1 GB memory space would require over 8 million sequence numbers based on cache line granularity and a cache line size of 128 bytes. Adding a cache that large (~ 28 MB) is unreasonable so the authors suggest adding a much smaller 64 KB one. However, this design decision either limits the security of the system, since a large part of memory would be unencrypted, or some sequence numbers would be stored in memory. There are 32K numbers (2 Bytes each) stored in the SNC covering 32K L2 cache lines and 4 MB of memory. Although RAM is

slower than cache, the seed (which is smaller than a cache line) is the first memory access and would arrive earlier than the rest of the reference. Although this does not hide as much latency as using cache, it is an improvement over the direct encryption scheme. This technique would also render part of RAM unusable, as it would be utilized for additional storage.

In address independent seed encryption (AISE) [Rogers et al. 2007], the authors suggest that all of the previous OTP schemes are flawed in their use of memory address as part of pad computation. Using virtual addresses as a component of the input to the pad seeds may lead to a vulnerability since separate processes will use the same address tweak as part of the seed thus breaking the requirement for pad uniqueness. Additionally, using the virtual address for pad computation can cause problems for shared memory inter-process communication since the pads would be different for the various processes even though both need to access the plaintext. For schemes using the physical address as part of the pad computation there are other issues when swapping to the backing store. Since pages in memory that are swapped out are likely to reside at a new physical address when brought back in, there is a potential for pad reuse or the requirement for a decryption and re-encryption of a page loaded into a different address.

Industrial implementations have been shown to be vulnerable to attack. In [1998], Kuhn demonstrates what is essentially a brute-force attack on the 5002FP. External hardware is used to control input to the processor and force it to power cycle. After each power-on, different encrypted “guesses” (possible instructions) are fed to the system and the output ports are observed. The 5002FP had been described as the most secure processor available for commercial users at the time of this successful attack, which used

a personal computer, and a device built in a student laboratory for about $300. One of the reasons the 5002FP is vulnerable to brute force attack is the small size of the plaintext. Kuhn notes that encryption performed over whole cache lines (of at least 8 bytes) instead of on single bytes would make the brute-force attack impractical. There is no known example of a successful attack against the IBM cryptographic coprocessors. However, these coprocessors tend to be used for highly specialized applications and are difficult to upgrade [Suh et al. 2007], thus making them undesirable for general-purpose computing environments. In fact, one of the designers of the IBM-4758 has noted his frustration with their *expense* and *modest processing environment* [Smith 2003], [Gutmann 2000]. While the TPM chip has been included in various trusted computing schemes, it is potentially vulnerable to the same types of snooping and bus injection attacks used against systems with unencrypted memory [Shi et al. 2004; Suh et al. 2007; Simmons 2011]. When utilizing the TPM with bitlocker drive encryption, the secret key is copied into RAM making it vulnerable to capture via cold-boot and other attacks as demonstrated in [Halderman et al. 2008]. Since the key must be in RAM for bitlocker to function properly, the additional protection of the TPM is potentially nullified.

There are three basic approaches in the literature surveyed with regard to operating systems. There is a problem in that without a secure (trusted) operating system extra protections must be placed in hardware to prevent a compromised system from breaking the confidentiality of other processes. When processes are context switched by the operating system the registers and other internal memory will be in plaintext. The first approach is to explicitly assume the existence of a secure operating system [Chen et al. 2008; Shi et al. 2004; Yan et al. 2006; Suh et al. 2003; Su et al. 2009; Chen and

Morris 2003]. Some of the papers taking this first approach discuss implementation requirements but none have been developed. In the second approach, the complexity of the hardware is increased in order to protect all processes (including the operating system) from each other [Kgil et al. 2005; Yang et al. 2005; Duc and Keryell 2006; Enck et al. 2008; Lie et al. 2000; Platte et al. 2006; Zhang et al. 2005; Chhabra et al. 2011]. One example of such hardware includes special instructions and extra registers which are called before context switches [Lie et al. 2000]. The internal registers are then encrypted strictly by the hardware before the kernel can intervene and complete the context switch as normal. Although several papers note the importance of working on a secure kernel to complement secure architectures we have found no work to date suggesting the completion of any such effort. In the third approach, the requirement for a secure operating system is simply not addressed [Nagarajan 2007; Rogers et al. 2005; Rogers et al. 2007; Hong et al. 2011; Lee et al. 2007; Jannepally et al. 2009; Rogers et al. 2006; Rogers et al. 2008].

2.9 Conclusion

This chapter has considered the research challenges associated with full memory encryption and distinguished three primary groups of techniques that attempt to solve those challenges — hardware enhancements, operating system enhancements, and specialized industrial devices. While the concept of memory encryption has existed for over three decades, there are still no general-purpose, commercial-off-the-shelf (COTS) solutions integrated with secure operating systems. However, there is clearly a growing need for privacy and intellectual property protection on the Internet as evidenced by the increasing use of full disk encryption, recent policy directives such as the Federal Data

Breach Notification Act and components of the Health Insurance Portability and Accountability Act [Brink 2009]. Between 2002 and 2007, a reported 773 breaches of US organizations were reported with a total of 267 million private records lost. Over 42% of these breaches were a result of lost or stolen hardware including laptops, PDAs and portable memory devices [Romanosky et al. 2008]. Additionally, it is apparent that at least one major chip maker (Intel) has recognized this growing need as two recent patent applications for adding memory encrypting hardware to processors attests [Gueron et al. 2012], [Gueron et al. 2013].

The range of overheads reported in the literature is quite large (1% to 6015%). The results on the lower end of the spectrum are possibly overly optimistic given the lack of fidelity in the simulation frameworks and the lack of standards for comparison. If standardization could be injected into the validation methodologies through accepted AES decryption latencies, benchmarks etc. it would enable more meaningful comparative analyses. Even with standardiz ation, the number of assumptions make it difficult to be confident that simulation will provide anything more than high-level information: It ignores the more difficult and interesting implementation issues and associated security impact based on vulnerability and exploit analysis. Where, in the few cases available, the literature addresses these low-level issues, it tends to be with generalization since there is no chance for practical experimentation or empirical evidence [Lie et al. 2000; Shi et al. 2004; Chhabra et al. 2010]. While the security of the encryption algorithm or cipher mode is often pointed out, it is commonly the complexity of the system in which these algorithms run that presents vulnerabilities. The most developed, though not commercially available, general-purpose technologies are FPGA soft-core emulations

[Suh et al. 2007] and the Linux prototype used in Cryptkeeper [Peterson 2010]. While the industrial devices are mature and practical, they are not general purpose, and typically cater to highly specialized operations. Additionally, these devices are either low-frequency or expensive and difficult to upgrade [Dallas 1997; Arnold and Doorn 2004].

Several technologies have been incorporated into general-purpose systems recently and often without the knowledge of those buying them. These technologies include TPM chips for storing keys and encryption engines and instructions. Given a system with these components, it is now possible to experiment with memory encryption providing an opportunity to better understand the difficult implementation details and ultimately provide data on overhead and security enhancement. This data should prove invaluable for determining the feasibility of memory encryption in general-purpose systems and for comparing against (and perhaps validating) the results of previous simulation work.

Chapter 3: Core Ideas and Background

Chapter two presented a general background and motivation for memory encryption including vulnerabilities. Additionally, a thorough survey and comparative analysis of memory encryption techniques was presented beginning with section 2.2. Chapter three presents additional specifics regarding the targeted threat model and core ideas behind the thesis. Additionally, closely related works are explored.

3.1 Threat Model

In contrast to research on intrusion detection, our research group at Dartmouth is focused on exploring methods to *increase attacker workload* while *reducing the attack surface*. The general threat model used in this research is outlined in Figure 6. It may involve several steps including *surveillance* to determine if a vulnerability exists, use of an appropriate exploit or other access method [Kennedy et al. 2011], privilege escalation [Davi et al. 2011], removing exploit artifacts, and hiding behavior [Hoglund and Butler 2005]. Surveillance may involve obtaining a copy of the binary code and using reverse engineering [Eagle 2011;Eilam 2005] or fuzzing [Forrester and Miller 2000] to facilitate a broad range of attack vectors including return oriented programming [Checkoway et al. 2009]. The implant then *persists* for a time sufficient enough to carry out some malicious effect, obtain useful information, or propagate intrusion to other systems. Unlike the time to execute an exploit, the time spent in surveillance and persistence may range from minutes to *months or even years* depending upon the intended effect. Moreover, the presence of an intrusion may *never* be detected by network defenses but instead may be recognized indirectly due to either a deviation from expected behavior, or may be derived from intelligence sources.

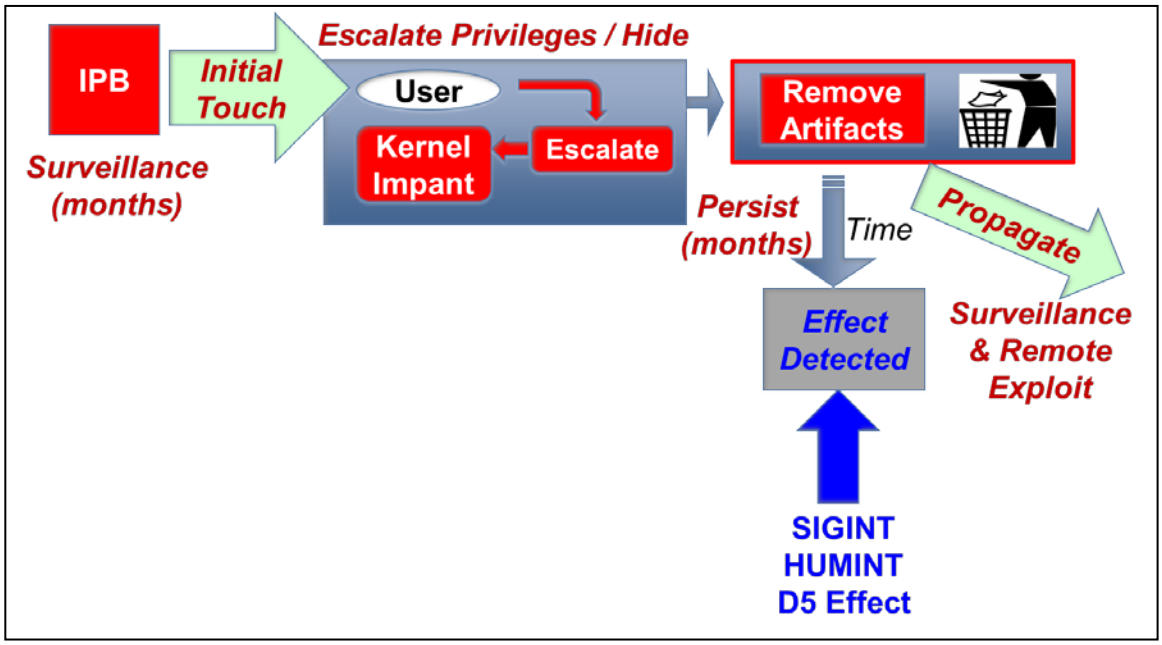

Figure 6: Threat Model for Intrusions with Remote Control

The threat model explicitly includes an adversary gaining remote or physical access (initial touch) with sufficient resources and motivation to pursue vulnerabilities via reverse engineering. There are many situations where this possibility exists: A binary code might be obtained via an insider, by purchase, or via an existing point of presence; The smart phone of a diplomat might be confiscated for a period of time while transiting through airport security; Methods of physical access may be used to capture memory and/or disk contents for offline analysis, data exfiltration, and/or malicious code development and injection; A smart phone or device may be captured on the battlefield and shack attacks used to capture sensitive details of friendly forces locations and mission objectives [United 2012]; A download (accidental or intentional) of a malicious application to a computer system may result in a new point of presence (e.g. memory scraper virus). All of these attack vectors may be used to nullify common protections

including encryption of data at rest and data in transit. Additionally, the protections afforded by recent diversity-based security techniques such as address space layout randomization (ASLR) can be overcome since the randomized layout in memory can be ascertained.

The goals of the protections afforded by the research described here are to ensure that either 1) time-sensitive details are obtained *only after* expiration of their value for military operations or 2) the difficulty of overcoming the additional security mechanisms *outweighs* the value of sensitive personal or financial information associated with the target (i.e. there are other more easily compromised targets). By protecting against hack and shack attacks (described in chapter 2), military mission objectives are safeguarded and the likelihood of compromise and loss of financial data is minimized. As previously mentioned, sophisticated lab attacks are still possible, but the time and resources involved in such efforts suggest that important mission details, critical for only 24-72 hours (i.e., the time window of typical air mission planning and execution) will have expired. Additionally, the return on investment for stealing financial information from one system is not likely viable.

Memory vulnerabilities (as explored in chapter 2) are common in systems ranging from servers and standard desktops to mobile computing devices (e.g. smart phones, tablets, laptops, etc.). However, usage patterns toward the mobile end of the spectrum may exacerbate the problem since many users of smart phones rarely reboot these systems thus maintaining them in an "always on" fashion resulting in the persistence of sensitive information [Karlson et al. 2009]. In a study of the Android operating system, 6 out of 14 applications permanently maintained their passwords in RAM. Additionally,

mobile devices are more likely to be lost or stolen thus providing physical access to possible adversaries. In NYC, for example, 49% of the population has experienced mobile phone theft and/or loss and 60% of those phones are believed to contain sensitive and confidential information [Tang et al. 2012].

Mobile devices, such as Android based smart phones, are beginning to be used in forward deployed military areas. These phones are loaded with information such as local maps, objectives, and blue force tracker (friendly unit) locations. Unfortunately, these phones (and other devices such as remotely piloted airframes with similar embedded processors) could easily fall into enemy hands. In fact, a recent U.S. Air Force document entitled Air Force Cyber Vision 2025 highlights the need for trust-based techniques to protect captured mobile devices in adversarial territory against reverse engineering efforts [United 2012]. While protections should be considered for both standard desktop and mobile devices, the work described here targets the ARM Cortex A8 which is common to many smart phones and tablets, including Apple's iPhone 3GS and 4, iPad first generation, iPod touch 3rd and 4th generations, and Samsung Galaxy Tablet to name a few. Additionally, the architectures of more recent ARM processors including the A9 and A15 are similar enough to that of the A8 to make the approaches in this work applicable with little modification.

3.2 Core Ideas

Encryption was the key concept used to mitigate vulnerabilities on disk: encrypting the disk provided confidentiality preventing access to sensitive information. By migrating the same solution down into RAM, it will be possible to circumvent similar attacks, including those highlighted above, at this lower level of the memory hierarchy.

This constrains the *boundary* available to an attack to lie at the *processor itself*, presenting a barrier that, in most cases, cannot be defeated without mechanical or electrical destruction of the processor chip (exotic techniques) as shown in Figure 7. Attacks on the device are possible, for example, by etching away the chip walls with acid to reveal internal bus lines, or electromagnetic and differential power analyses [Pope 2008], [Kocher et al. 1999]. These approaches clearly increase the attacker workload by at least an order of magnitude, require expert knowledge, and cannot be exploited remotely over a network [Suh et al. 2007]. Moreover, while tamper resistant mechanisms that significantly increase the barrier to entry are already available [Chari et al. 1999], protecting circuits from invasive and side-channel attacks is an open research area that is not addressed in this work. The processor provides a natural boundary within which sensitive information can reside—it is a fundamental component of the TCB in this work. All components outside of the processor are assumed to be vulnerable to include RAM and its interconnections (data and address bus), other I/O devices, etc.

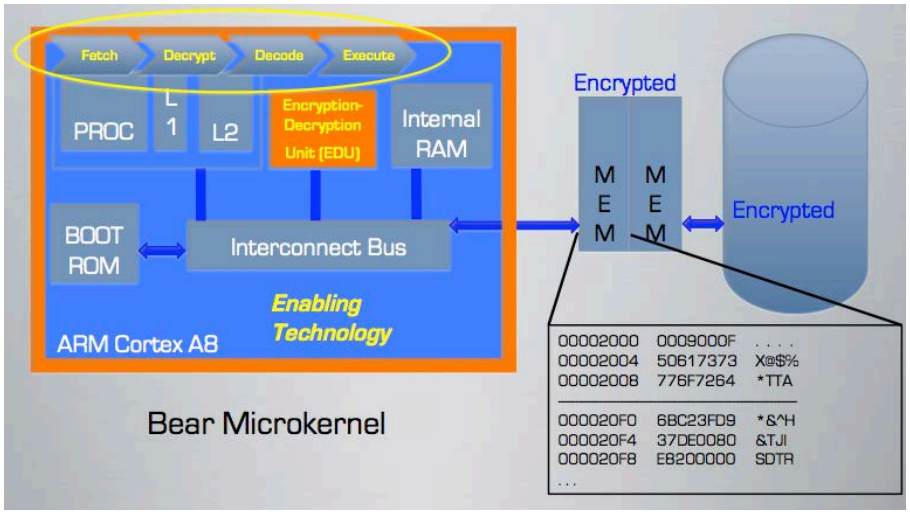

Figure 7: CPU Boundary and Nascent Security Hardware—General ME Approach

Although the concept of *memory encryption* has been an active area of research for over three decades (see chapter 2), it has yet to be used at the core of operating system designs in order to provide confidentiality of code and data [Henson and Taylor 2012]. The literature on memory encryption is largely concerned with three core approaches based on hardware enhancements [Lie et al. 2000], [Rogers et al. 2005], [Su et al. 2009], operating system enhancements [Chhabra et al. 2011], [Chen et al. 2008], [Peterson 2010], and specialized industrial applications [Dallas 1997], [Arnold and Doorn 2004], [Steil and Domke 2008]. Unfortunately, almost all of the hardware and operating system enhancements have only been implemented through *simulation* or emulation, and as a result, the claims have yet to be validated and quantified on practical systems. The few processors that implement memory encryption are characterized by low speeds and small addressable memory (<=16 bits) at use in low throughput (e.g. ATM, set top TV access, etc.) applications or specialized gaming systems.

Memory encryption is solely concerned with the *confidentiality* of data and code during execution (i.e. in use), with the express purpose of increasing attacker workload associated with crafting exploits and stealing sensitive information. It is interesting to note, however, that memory encryption would also hamper attempts to inject code, generally assumed to require memory authentication. An adversary lacking an encryption key would be unable to successfully change an encrypted binary, as decryption would result in corrupt code and likely program termination [Barrantes et al. 2003].

As mentioned in chapter 2, security hardware (see Figure 2), including *encryption engines*, has been integrated within commodity processors such as the Intel i7, AMD bulldozer, and multiple ARM variants--Intel's advanced encryption standard - new instructions (AES-NI) include six instructions to speed up key expansion and encryption. Intel states that the new instructions can provide a two to three time performance improvement over software-only approaches for non-parallel modes of operation such as cipher-block-chaining (CBC) encryption [Gueron 2010]. Further, a 10-fold improvement can be realized for parallelizable modes including CBC-decrypt and counter-mode encryption (CTR). As an example of the performance improvements possible, the authors ran TrueCrypt's encryption algorithm benchmark test on a MacBook Pro with an Intel i7 dual-core, 266 GHz CPU. Using a 5 MB buffer in RAM, the throughput averages 202 MB/s without AES-NI support, and 1 GB/s with it – approaching the speed required to overcome encryption overheads on general-purpose systems. Unfortunately, systems developers have been slow to embrace these specialized, often vendor-specific, features [Vasudevan et al. 2011]. Little practical experimentation has been conducted and the improvements in security and performance have yet to be quantified [Henson and Taylor 2012].

Little work has been performed to explore the trade space of using security enhanced commodity processors to implement *memory encryption* (ME): encrypting all components of a process – stack, heap, code and data. Although more recent processors make memory encryption less costly, it remains unclear if FME is viable for everyday use or is limited to constrained tactical applications. In past ME work, overhead has been measured at the coarse granularity of an entire process without regard to process

components in part due to the limitations of simulation. The relationship between the overhead costs and security gains for encrypting particular process components needs to be understood (e.g. is there a particular component that can be protected with low overhead yet that holds high value code/data). This work is the first to implement ME on a commodity processor, thereby allowing investigation of the low-level implementation details and the cost/security tradeoffs at sub-process component granularity.

3.3 Related Research

A thorough comparative analysis of memory encryption research, over the past three decades, is presented in chapter 2; unfortunately, although the community shares the high-level goals of the work presented here, much of the research stems from different motivations and is largely unrelated. For example, considerable effort has focused on identifying the ideal processor modifications that would enable memory encryption. Additionally, a considerable body of work assumes that the adversary is the end user; it is therefore targeted toward digital rights management (DRM) and protection of proprietary software and algorithms. Only recently has memory encryption been seen as a commodity and an effective technique for protecting end users, and their systems, from attack.

Although security mechanisms in commodity processors have not been used to protect an entire system, there are examples of their use to protect particular applications. Several papers have highlighted approaches used to mitigate confidentiality attacks on memory. For example, Tresor [Muller et al. 2011], aims to protect the FDE key by storing it only inside the CPU and performing encryption/decryption within that boundary. By utilizing Intel's AES-NI hardware, TRESOR was shown to perform better

than software based full disk encryption (17.04 MB/s vs. 14.67 MB/s) with the additional protection against cold boot and other snooping attacks. A similar approach is taken in [Simmons 2011] except that registers used for performance counting are targeted for master key storage with multiple encrypted keys being stored in RAM. While these techniques are an improvement over systems that leave memory unprotected, they are inadequate since it is possible to recover the key via a DMA injection attack on unprotected memory [Blass and Robertson 2012]. Additionally, much of the code/data targeted for protection by the key *resides in the same unprotected memory*. This would be analogous to protecting the key to the front door of a home while having many of the home's valuables sitting on the front lawn. This additional sensitive information includes passwords and PINs, used for access to online resources, which are never stored permanently. The work in this thesis extends the ideas described above to protect all of the sensitive information in memory.

In a closely related approach [2008], Chen et al. propose an operating system controlled memory bus encryption technique for systems that offer scratch pad memory (SPM) or cache locking that is software controllable. Both types of memory are available in some embedded processors including the Intel XScale series. A new symmetric key is generated each time the system is booted and random vectors are used to initialize encryption at the granularity of a page. The vectors are then placed in memory with the pages. When a page fault occurs for a secure process, a specially crafted handler moves the encrypted page into the chip boundary and decrypts it there placing it into the cache, which is then locked to prevent leakage of sensitive data. The locked region holds several pages of data and encryption variables. In order to facilitate this special handling,

a boolean status variable is added to each process descriptor residing in kernel address space. The authors note the scheme is appropriate when embedded systems designers can tolerate a significant performance overhead for protected processes. Unfortunately, the work has only been implemented via simulation and with large reported overheads of from 137% to 850%.

Instruction set randomization (ISR) is based on randomizing the language used by a system. The goal of this field of work is specifically to mitigate *remote* code injection attacks (i.e. hack attacks). Instructions are randomized with various techniques ranging from simple XOR'ing to AES encryption. Since de-randomization must occur just before execution any code injected in normal form would essentially become garbage causing an exception in fairly short order. In one recent effort, the overhead for implementing ISR for a web server and SQL database was 1% and 75% respectively [Portokalidis and Keromytis 2011]. Since instructions are de-randomized into memory before being executed, these techniques cannot prevent any of the shack attacks described earlier when an adversary has physical access to the system. Additionally, ISR mechanisms do not extend to data and so provide no protection against malicious software that scans memory for sensitive information. The research described in this thesis would defend against both remote code injection and physical access shack attacks protecting both the confidentiality and integrity of code and data.

Industry offers several solutions for memory encryption including low frequency specialized processors for ATM use, expensive tamper resistant coprocessors for financial transactions, proprietary gaming systems and, more recently, enabling technologies in commodity processors to enhance trust.

An active area of secure hardware used in industry is the cryptographic coprocessor. Primary examples include the IBM PCI-4758, PCI-XCC, which is shown abstractly in Figure 8, and the latest PCI-e. These coprocessors include an impressive array of technology but are generally limited to IBM server platforms under customized contracts and tend to be used for financial and banking systems. Rather than concentrating on protecting vulnerable RAM, these coprocessors focus on providing a small *safe haven* in which programs can operate on sensitive code and information. Unfortunately, the processor speeds and internal space are limited when compared to typical RAM sizes and the devices tend to cost thousands of dollars thus making them impractical for generalized use. For example, the PCI-XCC is an adapter card including an IBM PowerPC 405GPr microprocessor (266 MHz), 64 MB of DRAM, 16 MB of flash EEPROM, 128 KB of CMOS RAM backed up by battery, tamper-detection circuitry, cryptographic processor and FPGA. It is certified at the FIPS 140-2 tamper resistance standard level 4 [Arnold and Doorn 2004]. The packaging around the unit is designed to detect or prevent all known physical attacks such as acid etching or probing. A modified version of embedded Linux runs on the system providing a subset of typical features. The previous version (4758) used the IBM developed CP/Q message-passing microkernel. The secure module is encased in a flexible mesh of overlapping conductive lines meant to prevent any physical intrusion. If such intrusion is detected the system responds by zeroizing the internal RAM which holds the 168 bit Triple-DES secret key. The cryptographic processor performs at a throughput of 67 MB/s for Triple DES and 185 MB/s for AES-128. The stated purpose of the IBM secure coprocessor is to offload

computationally intensive cryptographic processes (e.g. specialized financial transactions) from the host server. One of the developers of IBM's 4758 cryptographic coprocessor has suggested that a general-purpose system with hardware support (such as a trusted platform module) could theoretically be turned into a somewhat less secure but more pervasive and less expensive version of the 4758 [Smith 2004].

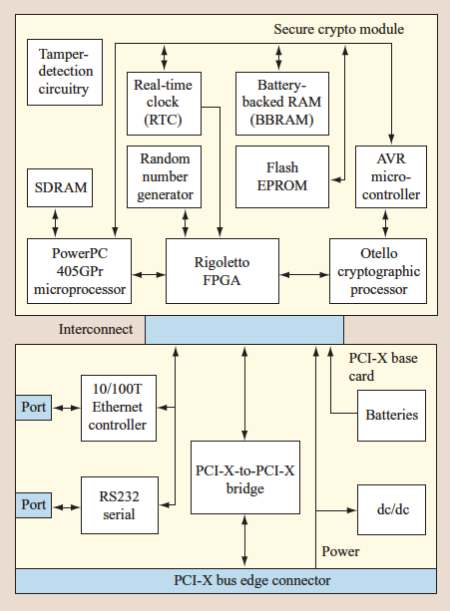

Figure 8: IBM PCI-X Cryptographic Coprocessor

While mostly constrained for use in playing games and other entertainment media (unless compromised) gaming systems are some of the most capable (e.g. fast processor speed and relatively large storage) to incorporate memory encryption techniques. As an example of these systems, the Xbox 360 provides encrypted/signed bootup and executables, partially encrypted RAM, and an encrypted hypervisor [Steil and Domke 2008]. These mechanisms are provided via a Microsoft proprietary processor with 64 KB of internal RAM, random number generation and encryption as opposed to the "off the shelf" processor used in the original Xbox. While it is possible to use the Xbox as a

general-purpose platform, this requires compromising the system's security measures first. Alternatively, the Sony Playstation 3 includes many of the same security mechanisms of the Xbox 360, but allows the end user to partition the hard drive for use with a chosen (e.g. Linux) operating system. However, the proprietary security mechanisms of the Playstation 3 are not available to the additional operating system [Conrad et al. 2010].

3.4 Summary

This chapter described the general concept of memory encryption whereby code and data are never unencrypted outside the trust boundary provided by the processor. The threat model, core ideas and closely related works were also explored. Memory encryption is solely concerned with the *confidentiality* of data and code during program execution (i.e. in use) with the express purpose of increasing attacker workload associated with reverse engineering, crafting exploits, and theft of sensitive information. However, it can also provide various levels of *integrity* protection for code and data. The memory encryption literature has been primarily concerned with designing an idealized processor with encryption hardware integrated into the fetch-decode-execute process and explored through simulation. More recently, software-only encryption has been explored but has suffered from large overheads. The advent of security-enhanced commodity processors appears to offer the opportunity for memory encryption with acceptable overheads.

Chapter 4: Static Encrypted Processes

The information presented in chapters 2 and 3 point to the idea that, in general, for memory encryption to succeed with acceptable overhead, three elements are required: 1) a way to generate and protect key(s), 2) a place to store and operate on sensitive, plaintext information and 3) hardware accelerated encryption and decryption capabilities to overcome the processor-memory speed gap. An additional consideration is the requirement for either a secure operating system kernel or additional specialized hardware to protect all software processes (including the OS) from each other.

Based on these requirements, a secondary survey of existing encryption support in commodity processors was conducted so as to locate an appropriate experimental platform. A Freescale development system (iMX53) shown in Figure 9, including an ARM Cortex A8 processor was chosen for several reasons: it provides direct accessibility to low-level hardware, encryption primitives are available to the programer, and there is increasing market share associated with the ARM architecture (90% of smart phone processors in 2011). Although there are several ARM processors advertising encryption support, in reality only a few are currently available for development work and many suffer from a lack of thorough documentation, which is often the case for nascent technologies [Vasudevan et al. 2011].

Figure 9: IMX53 Development Board

Unlike Intel and AMD, ARM does not manufacture its processors. Instead, ARM designs and licenses the processor architectures to silicon vendors, who supplement the ARM processor with proprietary or 3rd party hardware resources such as memory controllers, input-output and communications peripherals, and advanced power management capabilities. Current generation ARM processors fall into three families: the Cortex-A, Cortex-R and Cortex-M processors. Cortex-A family processors are *application processors* commonly found in smart phones, tablets, and small single-board general-purpose computers. These processors include virtual memory managers, advanced hardware security capabilities and typically include high-performance on-chip multimedia and connectivity peripherals. The Cortex-R family is intended for hard real-time and safety applications. The Cortex-M family contains a broad spectrum of general-purpose microcontroller offerings. Both the R and M series processor families target deeply embedded systems and have many similarities: They provide sophisticated interrupt handling and are typically configured with industrial and/or automation communications interfaces, analog-to-digital and digital-to-analog peripherals, timers,

and (if present) limited display capabilities. Memory protection and ECC logic are common in Cortex-R family processors, but rarely found in Cortex-M family processors. Cortex-R family processors typically have higher maximum clock rates and superscalar architectures, giving them a significant performance advantage over their Cortex-M family processor counterparts.

A simplified block diagram of the IMX53 development platform is shown in Figure 10. This inexpensive ($149) platform is open-source and comes with an ARM Cortex A8 1GHz processor which includes peripheral and hardware accelerated graphics support. Critical components of the IMX53 for this research include the internal RAM (144 KB + 16 KB "secure" RAM), Symmetric/Asymmetric Hashing and Random Accelerator (SAHARA), L2 cache (256 KB) and 2 GB of external RAM.

To generate and protect keys, the IMX53 includes several possibilities: the security controller (SCC) implements "secure" RAM that can be used as general-purpose memory for data/software or as special "confidentiality-preserving" memory that protects cryptographic keys, passwords, PINs etc. The 16KB RAM is divided into four, 4 KB chunks that can be explicitly controlled with respect to access permissions (based on mode of operation etc.). The SCC contains a chip-unique 256-bit AES key initialized at the factory, which is used to encrypt any secure information in the small partition that is sent to external RAM. While this functionality is not powerful enough for general memory encryption, it can be used to provide storage for multiple keys. This will enable encrypted code to persist between boots since the process keys can be encrypted with the system key and stored in non-volatile storage (flash etc.) that resides outside the secure chip boundary. The random number generator is based on two ring oscillators and can be

used to generate keys for new processes or when re-encrypting current processes at given intervals to increase diversity.

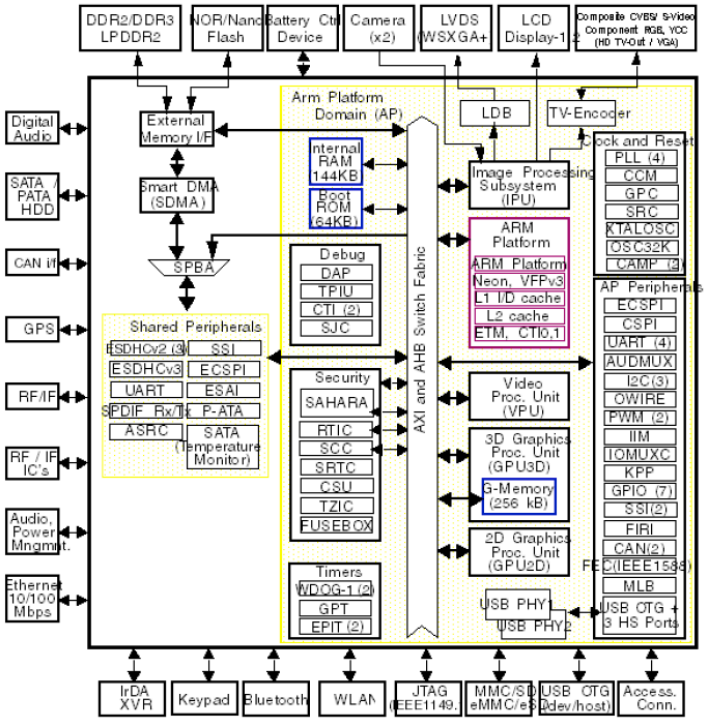

Figure 10: IMX53 Simplified Block Diagram

There are several concepts available in the memory encryption literature with respect to where to store and operate on decrypted information. The most prevalent idea involves the use of L2 cache as the store for plaintext information, and the IMX53 includes 256 KB of on-chip L2 cache. Typical use of L2 cache in encryption work requires hardware modification of the CPU where software control of L2 cache is not available. In other work, especially with embedded systems, there is often another type of internal memory that is *user controllable* either as the primary replacement of cache or in addition to cache. This memory is often referred to as scratch pad memory (SPM) or tightly coupled memory (TCM) [Cho et al. 2007]. The IMX53 includes 128 KB of "on chip RAM" and 16 KB of "secure RAM". For the purpose of this work, this entire RAM

is considered secure since it resides within the chip boundary. The secure RAM includes additional protections against access from a rogue process and can be used for data and/or storage of encryption keys. Initial experimentation takes place utilizing the TCM. Exploration of the IMX53's cache is performed in chapters 4 and 5.

For hardware accelerated encryption/decryption, the IMX53 A8 chip includes the Symmetric/Asymmetric Hashing and Random Accelerator (SAHARA) internal coprocessor. SAHARA implements AES, DES and 3DES encryption, MD5, SHA-1, SHA-224, and SHA-256 hashing and hardware based (ring oscillator) random number generation as shown in Figure 2 [2]. The coprocessor also maintains its own DMA controller with an AHB bus interface so as to reduce the interaction/burden on the primary CPU. For AES encryption, SAHARA includes electronic codebook (ECB), cipher-block chaining (CBC), counter (CTR) and counter with CBC-MAC (CCM) modes of operation. Descriptors are used to notify SAHARA of blocks of memory (internal or external) for encryption/decryption. Internal (secure) registers are cleared after a descriptor chain has completed processing to provide for usage by multiple, mutually distrusting processes. Completion of encryption/decryption is signaled via interrupt. The design of the IMX53 with the SAHARA as a coprocessor provides a significant advantage when faced with the challenge of overcoming memory decryption overhead. For example, it should be possible to decrypt either the next parts of a required process or a separate process while the current process continues execution.

The final component of the TCB required for a memory encryption solution is either specialized hardware support to protect processes (including the kernel) from each other or a *trusted kernel*. When specialized hardware is not proposed for this purpose,

the literature either ignores the requirement or assumes the existence of a secure microkernel. The work in this thesis makes use of a small, from-scratch microkernel--Bear--designed for critical/secure applications and operating through failures, errors, and malicious attacks. Additional information on the Bear microkernel can be found in [Taylor et al. 2011]. Other members of the research group developed Bear for execution on Intel x86/VT-x/VT-d hardware. In order to make use of it for this research, it was ported to the ARM A8 architecture as part of this effort.

While several of the security mechanisms designed into Bear are not available on the ARM architecture due to virtualization limitations, the goal of reducing the microkernel size, and thus reducing the attack surface, is intended to produce a reduction in vulnerabilities. Additionally, the small size of the microkernel allows for a thorough understanding of all system-memory interactions. This reduces the likelihood that unintended copies of sensitive data could persist in multiple locations for longer than expected or desired: A problem that has conspicuously plagued modern operating systems [Chow et al. 2004], [Dunn et al. 2012], [Tang et al. 2012].

4.1 Bootstrapping

To bootstrap the iMX53, an on-chip ROM-based boot-loader is configured by chip pin voltage settings to read the kernel program image from either an SD card or over a serial link from an image server on system reset. The Init.S assembly routine for the iMX53 provides a special program section containing instructions for the ROM-based bootloader to configure the processor's IO multiplexers and electrical properties for the off-chip DDR memory. The ROM-based bootloader then copies the program image from

an SD card or over the serial link into on or off-chip RAM (or a combination) and program execution begins.

In order to facilitate porting and modification of the Bear microkernel on the IMX53, a tool was developed to quickly serve process images to the board. The boot process for the iMX53 on the A8 involves a ROM bootloader first initializing several peripherals (USB, UART) and then loading a second bootloader into RAM and handing off execution. The second bootloader (e.g. Uboot) then loads the full operating system and again hands off execution. While there is a serial downloader protocol, the only available tool is Windows based and writes the image via serial download to a nonvolatile device such as the SD or micro-SD card before booting from one of those devices. This presented several challenges for bare-metal development work on the board. First, the development environment is Linux based thus making the Windows tool impractical. Second, the alternative of loading and testing new code involved removing a micro SD card from the development board, placing the card into an SD adapter, inserting it into the development system and then issuing several low level DD commands to load the image--reversing these steps to get the code back into the development environment and finally powering up the board. Bare metal work (including porting an operating system) requires frequent modification thus rendering this method of initialization and loading impractical.

The tool is an extension of a python script used for interfacing with serial interfaces [PYSERIAL]. After first matching up the serial downloader protocol (chapter 7.8 of the iMX53 manual) [i.MX53] with a captured USB stream from the available Windows tool, it was possible to develop the appropriate routines to initialize the DDR3

memory and place the vector table and executable code at the appropriate locations (bypassing the traditional 2nd bootloader) in RAM and execute the code. Appendix 1 includes the complete code for the developed tool.

On the Cortex-A processor, init.S first disables system interrupts, populates stack pointers for each of the processor's 8 operating modes and overwrites the ROM bootloader's interrupt table. Data and bss sections are then copied out before init.S hands off to main().

4.2 System Initialization

The main() function lives in ~/<platform>/main.c; This file contains platform specific initialization for the peripherals used by Bear. The current version performs the following steps in main():

1. Configures a periodic timer to call the kernel's scheduler
2. Initializes a UART for use as the system console
3. Displays a banner
4. Creates initial tasks
5. Enables interrupts
6. Switches from privileged to user mode.
7. Begins context switching between tasks in the ready queue

In order to port Bear to the A8, a few hardware components were necessary. These included a clock, timer, an input/output interface (e.g. the serial UART) and the ability to service interrupts. The iMX53 has two timers that could be used equally well for Bear: the general-purpose timer (GPT) and the enhanced periodic interrupt timer (EPIT). The GPT is described as a 32 bit up-counter which can generate an interrupt when the timer

reaches a programmed value whereas the EPIT is described as a 32 bit down counter "set-and-forget" timer capable of "providing precise interrupts at regular intervals with minimal processor intervention" [i.MX53]. Although the GPT is used in several Linux distributions the EPIT is more precise and is used in this work. Bringing up the EPIT first required enabling the EPIT peripheral clocks by setting all bits in the second clock gating register (CCM_CCGR2) location CG1 and CG2 (bits 5:2). After that, control is established via five memory-mapped user-accessible registers. Enabling the UART required similar steps to enabling the timer with some additional configuration of input/output muxes. UARTs are vendor-specific peripherals that are common on all but the most rudimentary processors. Custom hardware initialization and user-space driver tasks are required for each port of the Bear microkernel to proprietary hardware. The timer routine was developed to fire an interrupt that calls Bear's scheduler code. Small modifications to Bear's scheduler code were required due to differences in the architectural behavior such as the number of processor modes and amount of automation of register saving when entering those modes.

4.3 Memory Encryption

Recall that achieving acceptable levels of performance for memory encryption offers a significant challenge because there is an existing, growing, and well-documented speed-gap between processors and memory – improvements in processor speed continue to outpace improvements in memory speed [Patterson and Hennessy 1996]. Adding encryption latency to this already strained interface requires an overhaul of the basic fetch-decode-execute cycle employed by processors. Added to the complexities of any memory encryption solution is the fact that, unlike disk encryption where data is simply

stored for access, memory is used in a broad variety of dynamic access patterns. For example, a running program will utilize RAM during execution for both stacks and heap space. Additionally, the heap size, for any given program, is not normally known a-priori. The complexities of memory mapped input-output peripherals result in an inability to cache mapped regions, presenting a challenge since the overarching concept involves decrypting memory only when it is brought on chip. During context switches, registers containing sensitive information normally save it to external memory. Additionally, numerous decisions must be made concerning the granularity of encryption. For example, should the entire memory be encrypted with a single key, or should programs, shared libraries, and data be encrypted independently using separate keys. Alternatively, should individual functions or cache blocks be used as the unit of encryption. All of these decisions incur a tradeoff between the number of keys that must be securely stored, versus the degree of protection and overlapping in operations that can be realized.

While the hardware chosen for experimentation is representative of hardware found in current smart phones and tablets, it is important to note that there are a large number of less powerful processors currently being deployed for industrial use and power grid control and observation. These processors are also being incorporated into vehicles and automated manufacturing equipment among others. The inexpensive nature of this hardware is making it possible to use them to optimize macro processes through the collection and analysis of distributed sensors. However, there is also the potential for attacks on and misuse of the sensitive information collected by these systems.

S*mart electric meters, for example,* are characterized by low-speed processors lacking memory management units (MMU's) and cache with small amounts of RAM similar to low power ARM processors [McLaughlin et al. 2010]. McLaughlin et al. developed an approach to securing these meters, which involves diversity and encryption via encrypting return addresses before they are placed on the stack or heap. If an attacker attempts to overwrite the return address, the overwritten address will be decrypted into garbage or an address that would likely lead to a fault. Developing the Bear microkernel from scratch on the A8 platform afforded the opportunity to emulate this important class of processors by not enabling the MMU or cache. In this way, a baseline for memory encryption could be established with additional measurement of overhead along the spectrum toward a system representative of full-featured processors used in smart phones.

The SAHARA coprocessor includes electronic codebook (ECB), cipher-block chaining (CBC), counter (CTR) and counter with CBC-MAC (CCM) modes of operation. Descriptors are used to notify SAHARA of blocks of memory (internal or external) for encryption/decryption. Internal (secure) registers are cleared after a descriptor chain has completed processing to provide for usage by multiple, mutually distrusting processes. Completion of encryption/decryption is signaled via an interrupt. The encryption-decryption unit (EDU), is controlled via a *descriptor chain*, consisting of six 32-bit words which must begin on a word-aligned base address. These words include a *header*, *length* and *pointers* to blocks of memory to be encrypted, and a pointer to the *next* descriptor (if any). The descriptor chain is built in iRAM starting at address 0xF8000000 and loading this address into the descriptor address register (DAR) begins the encryption process. Within the header, each bit or group of bits (generally 2-3) are selected to enable the

hardware module (e.g., encryption, authentication, random number generation), algorithm (e.g. RSA, DES), mode of operation (e.g. electronic codebook, cipher block chaining) and other details.

A security API was developed to hide proprietary Freescale encryption details and is responsible for building the appropriate descriptor chain. For example, the following function call:

EDU('E', 0x000001A0, 0xF8000000, 0x70000000, 0xF8000040);

causes the encryption unit to encrypt (E=encrypt, D=decrypt) a process block of 416 bytes -- the current size of a process descriptor and stack -- from iRAM at location *0xF8000000*, placing the result in external RAM (eRAM) at location *0x70000000*. For simplicity and repeatability, a 128-bit AES symmetric key is downloaded via JTAG into iRAM at *0xF8000040* (the final parameter above) and used for all process encryption. The IMX53 includes 64 KB of flash memory within the chip boundary that could be used to store keys persistently for use between boots.

In practice an out-of-channel or standard key distribution scheme would be used in a full system implementation [Mel and Baker 2001]. Other techniques for key management are described in the memory encryption literature. For example, several schemes generate new random keys at system reset; these keys are used to encrypt processes, which are initially stored in plaintext [Chen et al. 2008]. While this does present a brief vulnerability during system boot, an argument could be made that the system would either be under the control of the owner at reboot or, if not, that much of the sensitive information that had collected in the memory would be lost. While the contents of eRAM could potentially be captured from memory or the bus, this should only yield the potential for a known plaintext attack (i.e. brute force attack having

corresponding pairs of plaintext and ciphertext) against the key(s) used for encryption—against which AES is known to be currently safe. Recall that the key(s) and microkernel are never in eRAM. Other work describes the method by which binaries are encrypted for protection while stored or transferred.

One such method involves encrypting binaries with a public key. The private key, which is stored inside the processor, is used to decrypt the program in iRAM after it is delivered. The program is then re-encrypted with a randomly generated symmetric key to improve encryption performance. Regardless of the key generation and escrow techniques used, the keys are *never* available in eRAM. In the work described in this thesis, there is space for storage of many keys whereas several of the approaches to protecting FDE schemes rely on internal registers (e.g. SSE, debug, etc.) limiting storage to a small number of keys [Muller et al. 2011], [Muller et al. 2012]. Portokalidis and Keromytis [2010] modify the binutils *objcopy* utility in their instruction set randomization (ISR) work. Binutils is able to parse ELF headers and access a binaries' code—it was modified to encrypt application and shared library code with plans to extend the capability to Windows portable executable (PE) binaries.

Programs that are run on these industrial processors are often load-once run "forever", thus requiring little interaction and often remain unpatched for years. The first memory encryption prototype was developed to address this pervasive class of processors. In this method, only the code is encrypted, using 128-bit AES symmetric-key encryption, and stored on disk as part of the executable binary. As a proof of concept, a hex-editing tool was used to *manually inject* the encrypted code for two simple processes into the binary. Since AES was used in electronic codebook mode (ECB), the code

required padding to align on a 16-byte boundary. This was accomplished via an assembly *skip* instruction padding the code with NOPs (i.e. 0's in ARM) so that each process was 240 Bytes in length. Other process components (data, stack, heap) are never encrypted as they remain within the protected iRAM as shown in Figure 11 (i.e. they are created after process execution begins). A small bootloader stored in internal ROM is responsible for initializing the hardware and loading the microkernel over the JTAG interface directly into iRAM. In a full system implementation, the microkernel could be stored in internal flash or encrypted with the persistent EDU key and stored on external media (e.g. SD card). Next, a shell is bootstrapped using the on-board USART connection so as to allow programs to be executed. The *newproc*() function which builds the process descriptor and stack for new processes was modified in order to load the internal process entry addresses into register 14 (i.e. the link register). The user processes are added to the scheduling queue and the microkernel decrypts the process code, storing it into a buffer in iRAM at 0xF8001000 after which normal process execution begins. This technique, referred to as *static encrypted processes,* only performs decryption once at code loading and is relevant to embedded systems where processes fit entirely within iRAM or flash [Henson and Taylor 2013].

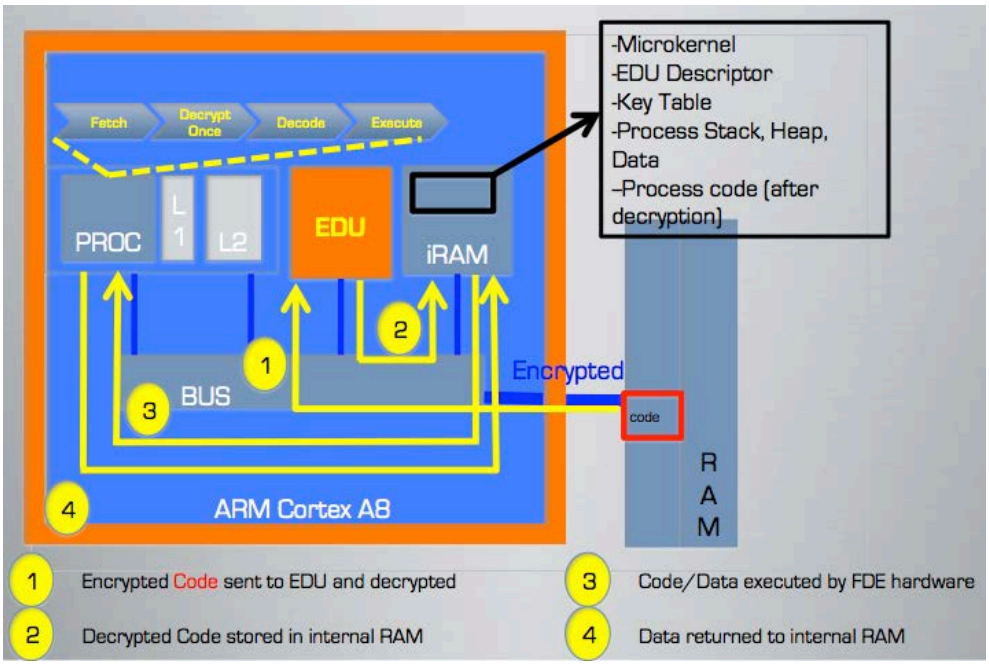

Figure 11: Static Encrypted Processes—One Time Decryption

4.4 Measurement

To quantify decryption speed, generic data was used as the data itself is of no consequence to decryption overhead. The average number of cycles for decrypting chunks of eRAM ranging from 16 Bytes (the smallest size possible) to 128 KB was measured in order to determine performance of the EDU in AES 128 mode. These results are directly applicable to the implemented *static encrypted processes* since the cost for protecting processes in this technique is the one-time cost of decryption of code. The results of the decryption tests are shown in Table 2 below.

Table 2. Overhead for Decryption of Various Sizes (Chunks) of Memory

Data Size in Bytes	Average Cycles	Std Dev	Cycles per bit
Overhead	8096	40	N/A
16	9152	65	71.5
32	9664	60.9	37.7
64 (Cache line)	10496	384.5	20.5
128	11712	55.4	11.4
256	14208	590.7	6.9
512	19776	376.3	4.8
1024	30080	577.2	3.7
2048	50688	578.2	3.1
4096 (Page size)	91776	578.7	2.8
8192	181632	401.8	2.77
16384	355584	716.8	2.71
32768	702720	566.2	2.68
65536	1397184	560.3	2.66
131072	2785792	658.7	2.66

The overhead associated with initializing the EDU (key expansion, etc.) is approximately 8096 cycles (as shown in the first row of the table). This was determined by measuring the cycles between calling and returning from the EDU function without regard to decryption status. In order to measure the average number of cycles for decrypting a chunk of memory, a function was developed to poll the SAHARA status register (bits 2-0) for status 0x04, which signifies a descriptor has been completed successfully. For the other rows, the cycles per bit cost of decryption is calculated by dividing the approximate cycles by the number of bits decrypted. For example, decrypting a chunk at the smallest possible size of 16 Bytes results in a cost of approximately 71.5 cycles per bit (9152 cycles/16*8).

As the decryption chunk increases the overhead remains constant so that the measure of cycles per bit decreases (i.e. the overhead is amortized across a larger execution). Additionally, there are likely architectural optimizations leading to increased throughput at the larger chunk sizes. The trend is shown graphically in Figure 12 below.

After 4KB, the improvement in cycles per bit is reduced dramatically. The ARM Cortex A8 architecture supports page sizes of 4 KB, 64 KB, 1 MB, and 16 MB. These measurements suggest that decryption overhead may be about the same whether 4 KB or larger page sizes are selected in future implementations. They also suggest that any granularity less than 4KB (e.g. a cache line of 64 Bytes) is sub-optimal as the total number of cycles is dominated by those required for initialization and the throughput rapidly deteriorates. For reference, the Bear microkernel binary is approximately 38 KB and would take about 1 millisecond to decrypt (considering the system is running at 800 MHz).

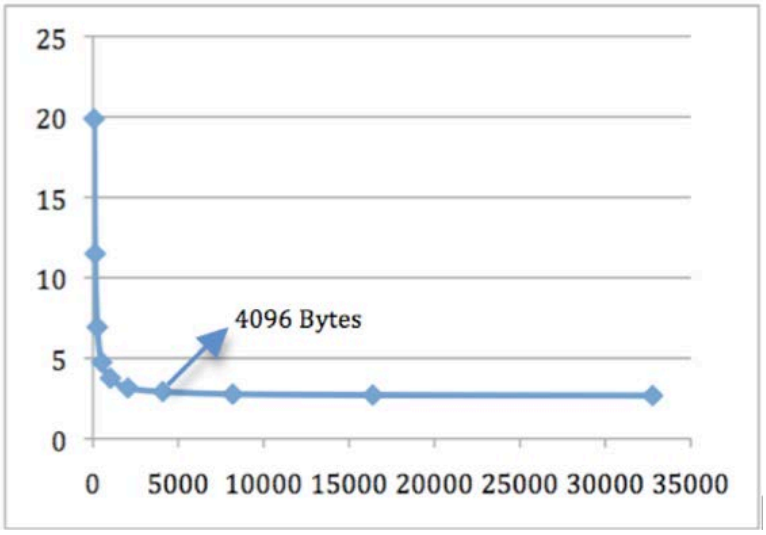

Figure 12: Graph of Cycles/bit Vs. Number of Bytes Decrypted (64 B through 32 KB)

ARM processors are targeted for operations in constrained space and power environments. It is likely because of this that the performance of the EDU on the Cortex A8 is slow relative to figures presented in the memory encryption literature (which tends to target X86 processors). In AEGIS [Suh et al. 2007], a single AES unit is estimated at 86,655 gates. Yet, AEGIS is demonstrated with an OR1200 soft core in FPGA with a total size of approximately 60,000 gates (meaning the AES unit is 144% of the original

core size). Recall that encryption hardware has been added to other processors such as Intel's i5 and i7 and AMD bulldozer chipsets. Intel's advanced encryption standard-new instructions (AES-NI) provide a significant speedup over both software and ARM hardware-enhanced encryption.

Recall that a test was conducted on an implementation of TrueCrypt's encryption algorithm benchmark test on a MacBook Pro with an Intel i7 dual-core, 2.66 GHz CPU. Using a 5 MB buffer in RAM, the throughput averages 202 MB/s without AES-NI support, and 1 GB/s with it – approximately 119 cycles for 64 Bytes. This represents an improvement of 88 times over the 10,496 cycles measured on the i.MX535. While x86 based processors do not tend to include user accessible iRAM, the combination of improved decryption performance and large caches in those systems might enable some form of memory encryption protection. Intel has recently filed a patent for processors incorporating memory encryption, perhaps indicating a move toward support in commodity processors [Gueron et al. 2013].

4.5 Summary

This chapter has described work to implement static encrypted processes – a protection technology targeted to industrial and real-time processors such as those found in smart meters. Other than the one-time initial decryption cost (dependent upon the size of the process code), there is little evidence of overhead using this method. Since embedded processors are continually increasing in on-chip memory, this technique represents an increasingly practical, low-overhead approach to memory encryption.

Chapter 5: Dynamic Encrypted Processes

While the last chapter presented research targeted at lower end industrial processors, recall that memory vulnerabilities are common in systems ranging from servers and standard desktops to mobile computing devices (e.g. smart phones, tablets, laptops, etc.). Additionally, usage patterns toward the mobile end of the spectrum may exacerbate the problem since many users of smart phones rarely reboot these systems maintaining them in an "always on" fashion resulting in the persistence of sensitive information [Karlson et al. 2009]. Further, in a study of the Android operating system, 6 out of 14 applications permanently maintained their passwords in RAM. Additionally, mobile devices are more likely to be lost or stolen providing physical access to possible adversaries. In NYC, for example, 49% of the population has experienced mobile phone theft and/or loss while 60% of those phones are believed to contain sensitive and confidential information [Tang et al. 2012].

Mobile devices, such as Android based smart phones, are beginning to be used in forward deployed military areas. These phones are loaded with information such as local maps, objectives, and blue force tracker (friendly unit) locations. Unfortunately, these phones (and other devices such as remotely piloted airframes with similar embedded processors) could easily fall into enemy hands. In fact, a recent U.S. Air Force document entitled Air Force Cyber Vision 2025 highlights the need for trust-based techniques to protect captured mobile devices in adversarial territory against reverse engineering efforts [United 2012]. This chapter will move away from the static encrypted processes model where code was protected during storage and transit and decrypted once before execution. Instead, it presents a more general approach, *dynamic encrypted processes*

(DEP), where there is sufficient memory pressure (i.e. processes + data are larger than available iRAM) to force processes back to external RAM during execution.

Unfortunately, the memory encryption literature fails to adequately consider the question of performance and *granularity* of encryption. Since the majority of research is conducted via simulation studies, the intricacies of implementation are not considered in detail. The schemes tend to allow granularity only at the process level—either an entire process is considered protected or none of it is. In fact, many of the techniques consider the fact that there may be many processes running which do not require encryption protection and this tends to improve the overall performance for these scenarios. The problem with this approach is that the onus for deciding which processes are sensitive and which are not is placed on the developer or end user. This proposition does not bode well considering a similar situation with regard to encryption of data at rest. One of the reasons for the move toward full disk encryption is the difficulty encountered by system administrators in knowing which parts of a file system to selectively apply encryption protections to [Brink 2009].

Since the work described here involves the first implementation of memory encryption on general-purpose hardware (vice simulation), it is possible to consider additional levels of encryption granularity. To some extent, the previous chapter delves into the overhead associated with the granularity of decryption based on size of the decryption chunk. While this direct measurement based on size is one way to address granularity, another intuitive choice is the process *segment*. Process segments include code (text), data, stack, heap and the process control block (PCB) as shown in Figure 13.

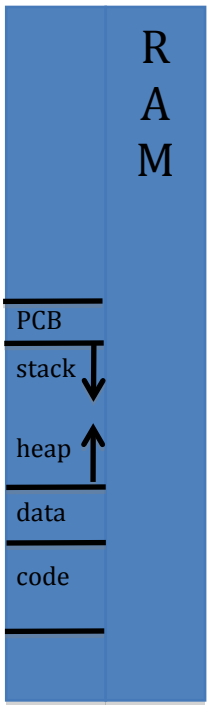

Figure 13: Process Segments in Bear Process

While traditional PCB's include many fields such as the execution state, address space identifiers, and scheduling priority, the simplified PCB in this work maintains only the stack pointer, stack base, process id and message id for communication. The microkernel (i.e. the developer of the microkernel) controls the creation and placement in memory of the process PCB, stack and heap objects while the assembler, linker and loader accomplish this for code and data segments. Statically allocated and global data that are initialized reside in the data segment while uninitialized data is represented in the BSS. Local (i.e. automatic) variables are allocated on the stack, as it grows *downward*. Depending on how deep the stack grows for a particular function call, sensitive information such as passwords may remain in an unused part of the stack indefinitely [Chow et al. 2004]. The location of string literals used in a binary is dependent on the linker used but can be controlled via a linker script and is often added to a *read only* section in the data segment as is the case in this work.

As described by Chow et al. [2004] sensitive information is copied to multiple segments via the complex interaction between the operating system and applications. For example, their work studied the propagation of a password entered into the Emacs editor and found that the password was available in 7 different areas including a global variable and on the stack. What would be more useful was unfortunately reserved for future work—studying how long data persists in particular memory segments. While outside the scope of this work, an additional question that should be explored through empirical study is whether particular segments are more commonly used for sensitive information than others.

5.1 Dynamic Encrypted Processes Implementation

The DEP prototype allows swapping of encrypted processes to eRAM. Process segments (other than the data segment) are stored in eRAM in encrypted form and brought into iRAM, decrypted, and executed on-demand. The PCB/stack and heap objects are updated in iRAM during execution. Once the scheduling quantum expires, the context switch routine sends the PCB/stack to the EDU, which encrypts it and stores it back in eRAM. Segments are re-encrypted before being sent back to eRAM with the exception of code, which does not change and so does not require re-encryption. In the absence of an enabled MMU, this movement of code and data required some virtual memory management (e.g. updating of stack pointers, addresses, program counters, jump addresses, etc.) where all segments of a given type correspond to a single internal buffer. This management was taken care of via modifications to the process creation, context switching and heap allocation routines as described below.

For the bulk of literature surveyed in ME, processes begin execution for a short

period in plaintext in eRAM. In fact, code is first loaded in plaintext and then encrypted in the industrial application processor DS5002FP [Gao et al. 2006]. From one perspective, this short window of vulnerability is not a major concern—processes are generally loaded when the system is under the control of an authorized user. If an attacker has control of the system and can load applications then they would have little need of performing a costly memory attack. Naturally, this intuition is heavily dependent upon the threat model in use. In this work, the threat model considers an attacker who does not have authorized access to a device. In the case where the threat model includes digital rights management (DRM) of proprietary algorithms the attacker may be an authorized user, in which case this short window could be a concern. Additionally, the idea that code is initially loaded in plaintext implies that it is stored in an unprotected form on disk. This means that an attacker could already analyze the binary providing an advantage when attempting to attack the system during operation. In this work, it is assumed that ME would be used *in conjunction* with encryption on disk in a full implementation (e.g. encrypted code would be injected into the binary). For this research, code is loaded to eRAM in plaintext and is encrypted *in place* during initialization. However, although there is only a momentary weakness presented from beginning with processes in plaintext in eRAM, this is an unnecessary risk.

The normal process, *newproc()*, involved allocating space in eRAM and then building and loading the appropriate process stack there. In order to begin with encrypted processes, the sequence in *newproc()* was modified such that after allocation, the process' PCB and stack are built in the internal buffer space (at 0xF801D024) and then encrypted and sent to eRAM. As mentioned previously with SEP, the entry point

for each process is adjusted in the link register to point to the iRAM code buffer at 0xF8001000.

The original heap allocation function in Bear utilized either the 1 GB eRAM or the 128 KB iRAM. In order to be able to use both, allocating iRAM buffers and eRAM segments, a *changeheap*() function was developed. In this way, the bookkeeping mechanisms previously developed in the heap allocation function could be used for iRAM objects rather than carving the internal space in an ad-hoc fashion. Currently, only one iRAM buffer is created for each of the code, PCB/stack and heap segments.

Figure 14 illustrates how the prototype decrypts the process control block (PCB) and stack (as one chunk); dynamically allocated memory and code are decrypted separately. In order to measure overhead at the process segment granularity, encryption can be enabled for each segment independently or in any combination. Changing the segments targeted for encryption requires recompilation of the microkernel and is controlled via preprocessor directive.

Once process execution begins, the *swprocs*() routine handles the decryption/encryption of both code and the PCB/stack segments. Of note is the fact that the cache is “grayed out” because it has been disabled in the initial Bear prototype to include the first dynamic encryption enhancement. The process context switch provides a natural point at which to perform decryption of these segments. Since the prototype does not currently utilize a paging mechanism, there is no similar point at which to intercede in accesses to global/static data, which are solely controlled by the compiler. Therefore, global/static data currently remains in iRAM.

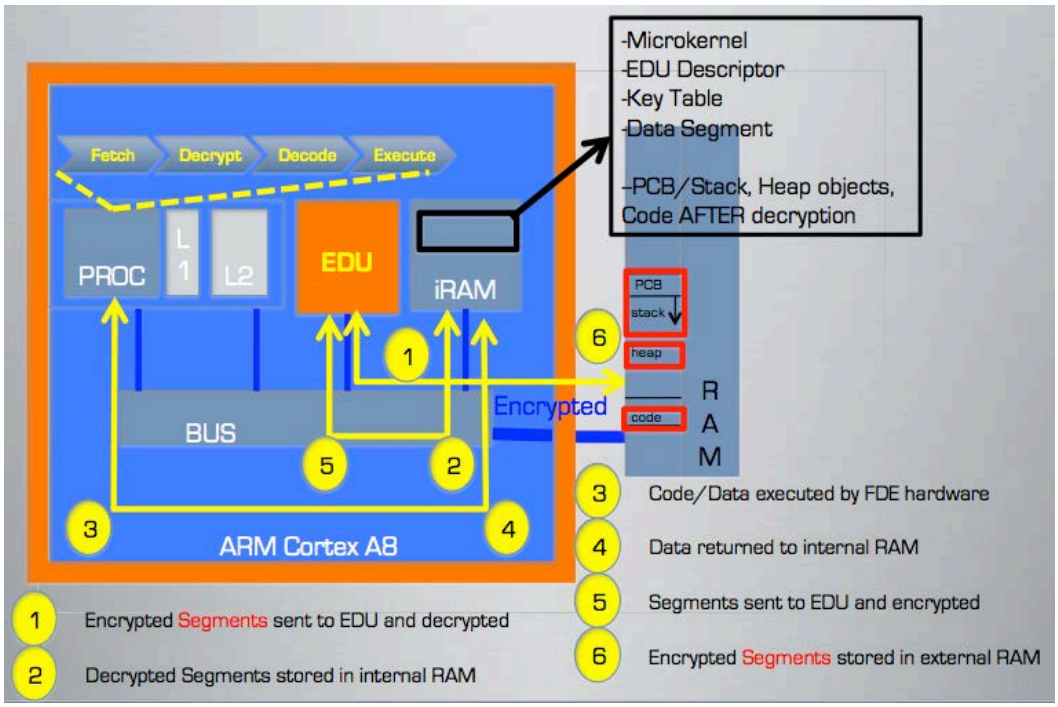

Figure 14: Dynamic Encrypted Processes

A modified linker script (shown in Appendix 2) makes use of the *memory* command describing the available memory regions (iRAM and eRAM). IRAM is defined to begin at 0xF8002000 to account for exception vectors, encryption keys, descriptor chains etc. that begin at 0xF8000000. Initially, protected code segments are placed into external RAM by the compiler and modified linker script while the microkernel is loaded to iRAM. As processes are added to the queue for execution, space is allocated in eRAM for the PCB and stack. The PCB and stack are built in iRAM and then encrypted to the external memory previously allocated. Modifications to the process creation routine include storing the iRAM code buffer location as the starting program counter (PC) for all processes. Once a process is selected for execution by the scheduler, the associated segments (code, PCB/stack) are decrypted by the encryption/decryption unit and placed in the respective iRAM buffers (1,2). Execution is then handed off to the CPU and execution continues in the normal fashion (3). Once the quantum expires, the

context switching routine (*swprocs*) encrypts the PCB/stack and heap (if any) segments back to the appropriate eRAM locations (4,5,6). The cycle continues as the next process is selected for execution and decrypted to the iRAM locations.

The modifications essentially add a simple form of virtual memory management (where all external memory segments map to one internal buffer location) with the addition of the encryption and decryption steps. Recall that the PCB/stack and heap segments require both decryption to iRAM and reencryption to eRAM while code only requires decryption, as it does not change.

Heap objects are more difficult to protect in a transparent manner since they are not necessarily known a-priori whereas the PCB and stack are strictly controlled by the microkernel and the code size is known. Automatic variables, which are created and used within a block of code, make use of the stack *transparently* whereas the creation of heap objects requires user intervention via calls to *malloc* and *free*. Because of this, the first iteration of protection for heap objects involved modification of user programs. However, a different approach allowed for transparent protection of heap objects. The kmalloc function was modified such that heap objects would be allocated to both iRAM and eRAM locations. A large iRAM buffer (~70KB) was set aside for heap object creation. Objects make use of the kmalloc slab allocator within the iRAM buffer. Because objects are allocated at iRAM locations, there is no additional bookkeeping needed when a process references them. At context switch, the entire internal buffer is encrypted to the eRAM location. The kmalloc function was also modified to maintain bookkeeping information about each process' eRAM heap locations in an internal list for use in decryption. Transparency of heap object protection requires strict adherence to

proper memory allocation and access techniques (i.e. using kmalloc instead of directly accessing a memory location via pointers that was not previously allocated). Depending on the size of the allocated segments, two alternative approaches are available for heap protection. If all objects fit within the large iRAM buffer, the set of heap objects is decrypted there. However, if the objects are too large, decryption of data on-demand at the size appropriate to the application (or smallest size possible) is used. For completeness, *both techniques are explored* in this work.

As a consequence of loading protected *code* directly to eRAM, the linker inserts a *veneer* to reach functions located in iRAM. Veneers are sections of code generated by the linker and inserted into a program. They are used in order to be able to mix various types of code (e.g. ARM and Thumb) and when a branch exceeds the architectural limits in ARM. For example, the range of a branch with link (bl) instruction is 32 MB, 16 MB and 4 MB for ARM, Thumb-2 and Thumb instructions respectively. Because we are loading user code to eRAM (0x70000000-AFFFFFFF) but leaving kernel functions in iRAM (0xF8000000-0xF801FFFF) veneers are required. One veneer for the kprintf() function was inserted several words below the last external process as shown in Figure 15. Address 0x720002E8 was the address of the iRAM kprintf() function (0xF8007190) which is immediately loaded into the program counter (PC) after the branch. When the code for a process was moved to iRAM, the veneer address was adjusted to point to an invalid memory location.

```
18                  kprintf("in proc1\n");
72000028:  ldr r3, [pc, #184]    ; 0x720000e8 <Proc1+232>
7200002c:  add r3, pc, r3
72000030:  mov r0, r3
72000034:  bl 0x720002e8 <__kprintf_veneer>
```

```
18                  kprintf("In proc 1\n");
f80035d0:  movw r0, #38632 ; 0x96e8
f80035d4:  movt r0, #63488 ; 0xf800
f80035d8:  bl 0xf8007190 <kprintf>
```

Figure 15: Kprintf() Veneer (left) in eRAM with Normal Kprintf() Call (right) in iRAM

In order to overcome the veneer issue, various techniques were explored. One such technique was *hijacking* the global offset table (GOT). The GOT is used to provide absolute addressing for some variables. This allows for these absolute addresses to be used, for example, with position independent code (PIC). The PIC will search the GOT for absolute addresses when necessary. Unfortunately, examination of the GOT revealed that the veneers are not included there—the address could not be changed to an appropriate iRAM address at run time. Besides GOT hijacking, several compiler directives were explored. While the –fPIC directive produces PC-relative addressing, it does not mitigate the issue with the jump veneers. The compiler directive –mlong-calls tells the compiler to perform function calls by first loading the address of the function into a register and then performing a subroutine call (blx) on that register. Compiling the code with this directive removes the veneer. Internal addresses are hard coded into the instructions for each process. Code was compared before and after moving from eRAM to iRAM. Both the code and data references were correct in the iRAM code as shown in Figure 16 (note the absence of the kprintf veneer as the kprintf() address is now loaded into register 3). This new approach allowed for the loading of code to eRAM and dynamic movement between eRAM and multiple iRAM buffers.

```
f80010a0:   movw r0, #40244 ; 0x9d34
f80010a4:   movt r0, #63488 ; 0xf800
f80010a8:   movw r12, #30464       ; 0x7700
f80010ac:   movt r12, #63488       ; 0xf800
f80010b0:   blx r12
24              kprintf(" PROC %d:automatic variable count == %d\n",
currentp->pid, count);}
f80010b4:   movw r3, #42732 ; 0xa6ec
f80010b8:   movt r3, #63488 ; 0xf800
f80010bc:   ldr r3, [r3]
f80010c0:   ldr r3, [r3, #8]
f80010c4:   movw r0, #40284 ; 0x9d5c
f80010c8:   movt r0, #63488 ; 0xf800
f80010cc:   mov r1, r3
f80010d0:   ldr r2, [r11, #-8]
f80010d4:   movw r3, #30464 ; 0x7700
f80010d8:   movt r3, #63488 ; 0xf800
f80010dc:   blx r3
f80010e0:   b 0xf800101c
```

```
720000a0:   movw r0, #40244 ; 0x9d34
720000a4:   movt r0, #63488 ; 0xf800
720000a8:   movw r12, #30464       ; 0x7700
720000ac:   movt r12, #63488       ; 0xf800
720000b0:   blx r12
24              kprintf(" PROC %d:automatic variable count == %d\n",
currentp->pid, count);}
720000b4:   movw r3, #42732 ; 0xa6ec
720000b8:   movt r3, #63488 ; 0xf800
720000bc:   ldr r3, [r3]
720000c0:   ldr r3, [r3, #8]
720000c4:   movw r0, #40284 ; 0x9d5c
720000c8:   movt r0, #63488 ; 0xf800
720000cc:   mov r1, r3
720000d0:   ldr r2, [r11, #-8]
720000d4:   movw r3, #30464 ; 0x7700
720000d8:   movt r3, #63488 ; 0xf800
720000dc:   blx r3
720000e0:   b 0x7200001c <Proc1+28>
```

Figure 16: Sections of Code after Moving to iRAM (left) from eRAM (right) with –mlong-calls

In order to take advantage of the independent nature of the EDU engine the decryption in the swprocs() routine, including the code and PCB/stack, was placed early in the function. In this way, the decryption work was overlapped with the context switching work (e.g. bookkeeping and queue maintenance) saving an average of ~10,000 (as shown in the measurement section) cycles during context switching. This idea of overlapping work with memory accesses is common in computer architecture [Hennessy and Patterson 2006].

5.2 Cache and MMU Enabled DEP

Caching is one of the techniques developed to overcome the well-known processor-memory speed gap and both the levels and sizes of cache have been increasing. Since this work adds additional cycles to the processor-memory gap, enabling the cache and MMU was extremely important. The iMX535 A8 has a modified Harvard architecture with 32 KB of L1 instruction and data cache, and 256 KB of shared L2 cache. During the initialization routine, the system coprocessor CP15 control register I bit (bit 12) and C bit (bit 2) are used to enable the instruction and data caching respectively. A separate bit in the auxiliary control register, bit 1, must be set to enable

the L2 cache. However, even with caching enabled, *data* will not be cached without a functioning memory management unit (MMU). This behavior is due to the fact that data accesses may include read/write sensitive peripherals and/or components that change the memory in some way, which is not safe without the added protections of the MMU. Setting the M bit (bit 0) in the control register enables the MMU. Unfortunately, the MMU requirement is not listed in the A8 technical reference manual (580 pp), cortex A series Programmer's Guide (455 pp), or the proprietary IMX53 reference manual (4,947 pp). However, it was listed in the V7-A architecture reference manual (2,158 pp) as shown below:

In ARMv7:

• The SCTLR.C bit enables or disables all data and unified caches, across all levels of cache visible to the processor.

• The SCTLR.I bit enables or disables all instruction caches, across all levels of cache visible to the processor.

If the MMU or MPU is disabled, the effects of the SCTLR.C and SCTLR.I bits on the memory attributes are described in:

• *Enabling and disabling the MMU* on page B3-5 for the MMU

The MMU can be enabled and disabled by writing to the SCTLR.M bit, see *c1, System Control Register (SCTLR)* on page B3-96. On reset, this bit is cleared to 0, disabling the MMU

When the MMU is disabled, memory accesses are treated as follows:

• All data accesses are treated as Non-cacheable and Strongly-ordered. Unexpected data cache hit behavior is IMPLEMENTATION DEFINED.

• The treatment of instruction accesses depends on the value of the SCTLR.I bit:

When I == 0

All instruction accesses are Non-cacheable.

When I == 1

All instruction accesses are Cacheable:

• Inner Write-Through no Write-Allocate

• Outer Write-Through no Write-Allocate.

This meant that even though both code and data caching were “enabled” via the cache control register as shown below, no data caching was occurring without the MMU being enabled. In that mode of operation, small improvements to overall performance were noted. After discovering the reliance of data caching on a functioning MMU, the MMU was enabled via the last sequence of ARM instructions below. After doing this, the performance of the system was improved dramatically. The MRC and MCR ARM instructions are used to move data from a coprocessor to a register and vice versa. Coprocessors are represented in the form pn where n represents the number of the coprocessor. In the example below, the coprocessor used is p15. Caching is enabled by or-ing the appropriate bits of the 32 bit control data and then writing that data back into the coprocessor registers (represented by cn where n is the number of the register).

```
mrc p15, 0, r0, c1, c0, 0@read CONTROL REGISTER
orr r0, r0, #(0x1 << 12)@ Instruction Caching
orr r0, r0, #(0x1 << 2)@ Data Caching
mcr p15, 0, r0, c1, c0, 0@ enable instruction and data caching

mrc p15, 0, r0, c1, c0, 1 @read AUXILIARY CONTROL REGISTER
orr r0, r0, #(0x1 << 1)
mcr p15, 0, r0, c1, c0, 1 @enable L2 cache

mrc p15, 0, r1, c1, c0, 0 @read CP15 Register 1
orr r1, r1, #0x1
mcr p15, 0, r1, c1, c0, 0 @enable MMUs
```

Several problems were anticipated as an unintended consequence of enabling the cache since memory encryption is essentially a special case of *self modifying code (SMC)*. Self-modifying code includes any code that purposely changes its instructions including simply copying instructions from one location to another [Cortex-A]. The concern with traditional self-modifying code (not encrypted) is the execution of *stale*

instructions that have been cached before the modifications take place. These coherency issues arise in systems where multiple actors including, for example, the CPU and DMA engines modify multiple levels of code and data. In the IMX53, these levels include the L1 and L2 caches, iRAM and eRAM. When utilizing SMC, the just in time (JIT) compiler or programmer is responsible for manually maintaining the cache (invalidation and cleaning) at the appropriate times as the memory hierarchy is used in a way unanticipated by the cache controller. In the case of a ME system, the stale instructions in eRAM are actually encrypted. If the system were to cache those instructions, an exception would occur upon execution. Additionally, sensitive information might leak from cache to eRAM during cache line eviction. Based on a thorough survey of ME research, *this is believed to be the first work to identify and explore the problem of SMC*.

Enabling both the L1 and L2 cache and code and data caching resulted in a major speedup of the baseline-unprotected system. Unfortunately, enabling the cache and MMU with PCB-stack protection resulted in exceptions, while code protection seemed to work properly. Investigation of these exceptions revealed that initial concerns over SMC issues were well founded. PCB-stack protection worked when caching instructions only, but broke once data caching and the MMU were enabled. Further, this led to the suspicion (which was verified by experimentation) that the instructions being executed were stale—only the instructions from the first process were executing after each context switch. Certain portions of the PCB-Stack that were written during the initial process build were not being encrypted by the EDU as shown in Figure 17 below. This is due to the fact that those words were cached after being written to. These

cached words were not being overwritten by the EDU's DMA engine. Normally, the entire block (0x1A0 or 416 Bytes) would be encrypted. The portions of memory not updated by the EDU fall along 64 Byte cache-line boundaries (e.g. 0xAFFFFCBC is the end of a 64 Byte cache line)—confirming that the caching of these lines was preventing the encryption update.

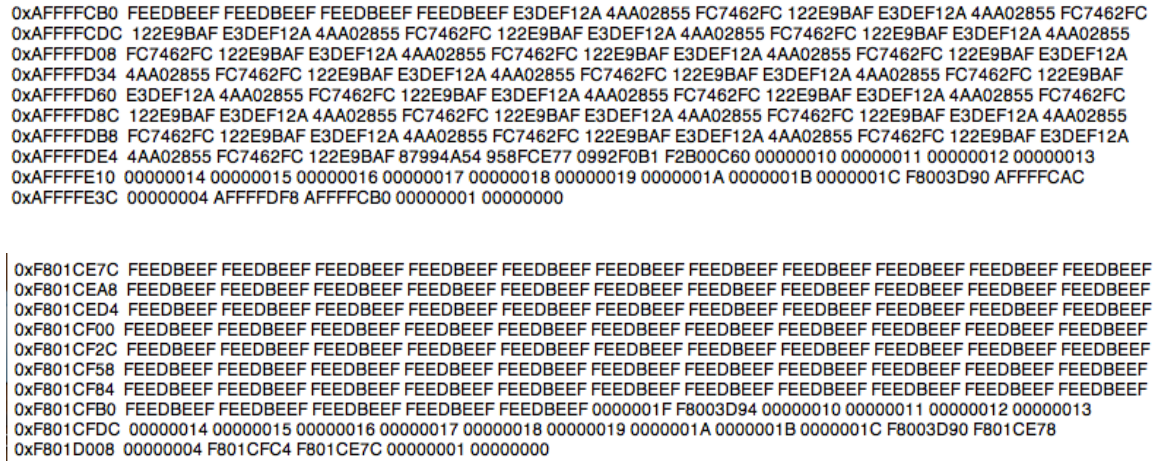

Figure 17: Process Control Block & Stack Before (bottom) and After (top) Encryption

In order to mitigate the SMC issues, routines were developed for cleaning, invalidating and locking the cache. Additionally, the ARM data synchronization barrier (DSB), data memory barrier (DMB) and instruction synchronization barrier (ISB) instructions were explored. *Cleaning* results in any "dirty" memory locations (those updated in the cache but not in iRAM or eRAM) to be updated in RAM while *invalidation* causes the system to read/write the external memory location (updating the code/data in the cache observed by the CPU). In the ARM Cortex A8, the L1 data cache is exclusive (items in L1 are not in L2) while the instruction cache is inclusive. Clean and invalidate functions for both data and instructions were developed for the L1 and L2 caches. These functions could be targeted to invalidate/clean from a single cache line up to the entire cache.

In order to mitigate the issue of incomplete encryption to eRAM, a cache invalidation for the external PCB-stack location was invoked just before encryption. A function was developed to identify the 64 Byte cache line boundary preceding a memory address since invalidation/cleaning must be accomplished along these boundaries. The new sequence of instructions were as follows:

```
chunk=find_chunk((uint32_t)p1->sb); //find the 64 Byte boundary
uni_invbyaddy(chunk);  //invalidate the unified cache (L2) at the proper address
EDU('E',0x000001A0, 0xF801D00C, (uint32_t)p1->sb, 0xf8000040);
```

Invalidating the eRAM PCB-stack location resulted in the entire PCB-stack being encrypted, confirming the SMC cache coherence issue.

Comparing the internal buffer used for each processes stack and PCB before and after an exception occurred with the same buffer while executing single instructions (i.e. in debug mode) verified that the cache is cleaned and invalidated when halted in debug mode and during instruction stepping. In the latter mode, the internal buffer was updated with the next processes data during context switch whereas during full speed operation the data was not updated. Because of this, the system would context switch without issue in debug mode. Considering the issue at hand was one of cache coherence, this made the troubleshooting more difficult since halting the system to examine it would result in an update to iRAM and eRAM from cache. This meant that isolating the cache coherency issues had to be done without the benefit of halting and memory/register introspection. In order to effectively debug in real time, the scheduling quantum was increased to 1 second per process with extensive printing of internal and external memory segments.

Even after the PCB-stack of each process was fully encrypted, the system would only execute the first process, throwing an exception as soon as a context switch to the second process occurred. By stepping through the context switch, and manually updating the program counter and other variables as required, it was possible to force the 2nd process to execute. However, when this was done the process reported the details (process ID and count) from the first process. All three processes used for experimentation are mapped to the same buffer in iRAM. With the original processes data cached, the changes made by decrypting the second process' PCB and stack to the iRAM buffer were *never observed by the processor*. In an experiment to verify this, three internal buffers were created, one for each protected process. In this case, each of the three processes executed once before an exception occurred after the context switch back to the original process. In this case, the data for each process was correct (process ID and count). This meant that the original PCB and stack for each process was loaded into the cache. This allowed for one execution of the process, however the updates to iRAM made during execution were not reflected in the cache causing an exception to occur when the processes attempted to run a second time. A similar approach was required to solve the stale PCB-stack issue as was used for the incomplete encryption to eRAM—the internal buffer had to be invalidated before the next processes data was decrypted there. With the addition of these two properly placed invalidations, PCB-stack protection with the L1/L2 cache and MMU enabled was complete.

After discovering the SMC issues in the PCB-stack, the correctness of the code protection mechanisms with instruction caching enabled was of concern. Although the code protection seemed to be working properly with the instruction cache enabled, this

was not the case. The concern was that since all the code for the 3 processes was using one internal buffer (similar to that used for the PCB-stack protection) the code was indeed working but was not being updated for the 2nd and 3rd processes. Since the data that was being printed out to confirm proper execution was maintained in the PCB-stack (process ID and count) as automatic (stack) variables, that data was being updated and displayed properly. However, only the code from process 1 was being executed, as that was the code that was first cached. In order to demonstrate this, an experiment was conducted whereby the process id was "hard coded" into each processes read-only print string to match up with the data from the PCB-stack. The results of this experiment with instruction caching enabled and disabled are shown in Figures 18 and 19 below.

```
[proc 1 0: sp F801D16C: sb F801D024: count 100000: ]
 PROC 0:automatic variable count == 100000
cycle count = 973 and context switches = 1
[proc 1 1: sp F801D16C: sb F801D024: count 100000: ]
 PROC 1:automatic variablcycle count = 792 and context switches = 2
[proc 1 2: sp F801D16C: sb F801D024: count 100000: ]
 PROC 2:automatic variacycle count = 781 and context switches = 3
[proc 1 0: sp F801D154: sb F801D024: count 200000: ]
```

Figure 18: Process Output with Hard Coded PID--Instruction Caching Enabled

```
[proc 1 0: sp F801D16C: sb F801D024: count 100000: ]
cycle count = 1227 and context switches = 1
[proc 2 1cycle count = 1003 and context switches = 2
cycle count = 1017 and context switches = 3
 PROC 0:automatic variable count == 100000
cycle count = 1002 and context switches = 4
: sp F801D16C: sb F801D024: count 100000: ]
 PROC 1: automatic variable count == 100000
cycle count = 1021 and context switches = 5
[proc 3 2: sp F801D154: sb F801D024: count 100000: ]
cycle count = 1017 and context switches = 6
[proc 1 0: sp F801D154: sb F801D024: count 200000: ]
```

Figure 19: Process Output with Hard Coded PID--Instruction Caching Disabled

The figures show that while the system appeared to be executing properly, with the EDU decrypting code to iRAM during each context switch, only the code of process 1 was truly executing. Note the 1 after the [proc while the PID cycles through 0, 1, 2 and back to 0 with the instruction cache enabled. The hard coded "[proc 1" is from process 1's text segment. In the latter case, the cache is disabled so that the decrypted code from each process is executed from iRAM resulting in both the hard coded PID and the PID from the stack cycling.

Recall that the instruction cache is inclusive on the Cortex A8. Because of this, both the L1 and L2 cache had to be invalidated for instructions (L2 first). With the invalidations in place just before the code decryption, each process was executing its own code and data. The question remained as to the effects on overhead of invalidating various levels of cache during context switches. For example, how do the average cycles for context switch compare with only the L1 enabled vs. both the L1 and L2 (and associated invalidations)? Even with the invalidation of code during each context switch, having both the L1 and L2 caches (and MMU) enabled reduces the number of cycles by about more than half when compared to the case where no cache is enabled.

5.3 Performance Measurements

Since the performance degradation of memory encryption results in less likelihood of its use, it is an extremely important factor in the comparison of different schemes. Recall that the hypothesis in this work is that vulnerabilities associated with memory can be mitigated *with acceptable performance* given security-enhanced commodity processors. In Chapter 4, the static encrypted process (SEP) prototype was

quantified in terms of total number of cycles for decryption of generic data blocks as reported by the Cortex A8 performance monitors. That characterization is appropriate since decryption should only happen once with the code essentially executing in perpetuity after that. However, in order to thoroughly evaluate overhead for dynamic encrypted processes, two micro-benchmarking applications were required—*context switching* and *page allocation*. It is specifically in these two activities that overhead is introduced in DEP. Recall from the implementation details that in order to make ME transparent to the end user the context switch function was targeted. For very small processes where the PCB/stack, code and heap objects are less than 4 KB each, the *only* significant overhead is encountered during the context switch. After some experimentation, it was fairly clear that several factors were key to understanding the performance of ME—the size of protected segments and the amount of spatial and temporal locality exhibited. Segment size, along with the level of caching, are varied in the benchmarking applications. Smartphone workloads tend to be interactive and exhibit a high degree of locality and this assumption is made for the two benchmark programs [Guttierrez et al. 2011]. However, in order to understand the effect of a lack of locality on ME, the fast Fourier transform was implemented. The FFT represents the *worst possible case* performance of the system due to a *pathological lack of locality*. When used for benchmarking in the ME literature, the FFT repeatedly demonstrated significant overhead (normally the worst of any benchmark).

Context Switching Benchmark

To measure the overhead for *segment* protection, the context switching benchmark was instrumented with performance monitoring code. The A8 performance

counter was initialized with counters set to 0 before each measurement. A start and stop count were used to capture the cycles before and after the code of interest. The A8 is a superscalar processor with a dual 13-stage pipeline, which can result in varying performance. Additionally, it is possible under pathological cases (e.g. asynchronous exception at the wrong time) for the performance monitors to be inaccurate. In order to compare the relative performance of ME protection approaches, it was necessary to determine the minimum number of context switches that yielded reliable metrics. The average number of context switches and standard deviation for 10, 100, 500, 1000 and 2000 context switches were compared. Neither the average nor the standard deviation changed significantly for 1000 context switches or more. Therefore, 1000 context switches were sufficient for performance comparisons. For all measurements, the system is running at 800 MHz with a scheduling quantum of 300 ms.

Since the EDU can run in parallel with the primary CPU, it is possible to have it decrypting a component while continuing to process a context switch, for example. While this *overlapping* can reduce overhead, it can also cause errors if not dealt with properly. An interrupt can be enabled to notify the system when the EDU has completed execution. However, servicing multiple nested interrupts would require a significant rewrite of the system as it currently disables all interrupts during the context switch. In order to ensure that memory was appropriately updated before continuing (and thus that the measurements were accurate) a watchword was placed near the end of the section being decrypted. An empty while loop was used to check for the watchword indicating the decryption was complete.

In order to evaluate the relative performance of the system, the total number of cycles required for executing the unprotected system running three simple user processes was measured. Next, the total number of cycles for protecting the various process segments of the user processes was measured independently, allowing for the calculation of accumulated overhead (i.e. slowdown). The combination of system frequency (800 MHz) and cycles was used to determine the approximate execution time for a given experiment. These measurements on commodity hardware are believed to be among the first in the ME literature (the only other example being an FPGA soft core)—allowing for more granular exploration (e.g. process segments) and higher confidence in the results.

Measurements for each component are shown below in Table 3. For comparison, a simple "Hello World" process requires approximately 10 million cycles (~12.5 ms). For measurement, the system schedules the three simple processes in a round-robin fashion with a 300 millisecond-scheduling quantum, resulting in approximately 200 context switches per minute. PCB-stack and code protection is completely *transparent* to processes (and developers) since the protection is implemented within the context switch routine. As previously mentioned, for small processes, costs are incurred only during the context switch, not during process execution. The type of process does not impact the context switching overhead.

Table 3. Context Switch Benchmark Overhead

Size (Bytes)	Protected segment	Average Cycles without Cache	Average Cycles Instruction Caching Enabled	Average Cycles Instruction/Data Caching & MMU Enabled	*Overhead* w/o Cache	*Overhead* Instruction Caching	*Overhead* Instr/Data Caching & MMU
400 Bytes	Unprotected	16384	11712	448	N/A	N/A	N/A
	PCB-Stack	50176	38016	18496	206 %	230 %	4029 %
	Code	24832	23040	17400	52 %	97 %	3784 %
	Heap	50100	39076	19230	206 %	234 %	4192 %
1024 Bytes	Unprotected	16384	11712	448	N/A	N/A	N/A
	PCB-Stack	65472	46400	35264	300 %	296 %	7771 %
	Code	43408	39603	22000	165 %	238 %	4811 %
	Heap	65344	55872	38464	299 %	377 %	8486 %
4096 Bytes	Unprotected	16384	11712	448	N/A	N/A	N/A
	PCB-Stack	105472	102200	96960	544 %	773 %	21543 %
	Code	86208	78204	73200	426 %	568 %	16239 %
	Heap	111744	105200	102750	582 %	798 %	22835 %

The unprotected context switch routine with no caching averages approximately 16,384 cycles (~20.5 microseconds) as shown in the first row of the table. The overhead for protecting the segments is fairly large: approximately 206% each to protect the PCB-stack and heap when compared to the unprotected context switch. This effect is exacerbated with each additional level of caching-the performance of the baseline, unprotected context switch is improved significantly (over 36X) while the boost in performance for protected execution is less impressive. This is because the performance improvements are constrained by the EDU's execution time.

While the overhead is large, it is only incurred on average 200 times per minute. Additionally, the time of each context switch (even with the overhead) is completely dwarfed by the process execution time (scheduling quantum). The context switch time is approximately 1/1000th the 300 ms quantum. Thus the total overhead *per minute* for a system with PCB/stack, code and heap protection is about 100 milliseconds, resulting in a total of ~10 seconds of overhead after 100 minutes of execution. Recall that studies suggest that delays of longer than 150 ms are perceptible to users [Muller et al. 2011]. These delays are measured during an interactive process such as hitting a key on a

keyboard. Since the measured overhead is only 100 milliseconds *per minute*, protection of small processes (i.e. where each segment is only up to ~4 KB) is possible with very reasonable overhead costs.

Note that PCB-stack and heap protection is more costly than code protection in all cases. This is due to several factors—the PCB-stack/heap buffer must both be encrypted and decrypted while it is not necessary to re-encrypt code and there is additional bookkeeping work that must be accomplished with the PCB-stack area. Additionally, code decryption takes advantage of the fact that the code is not needed until after the context switch routine returns. Because of this, much of the decryption work can be *overlapped* with other context switch routine work. For example, the measurement for code protection in a system with no caching is approximately 24,832 cycles. However, by polling the EDU for completion before moving to the next instruction in the context switch routine, the average cycles without overlap can be determined (40,320). Overlapping produces a significant reduction in cycles (~38.4% or 15,488 cycles) and is a technique that is often espoused in computer architecture literature. Unfortunately, there is only enough work within the context switch routine to cover the encryption overhead of one of the segments.

Increased caching highlights the cost of protection as baseline performance is improved. Additionally, the reduction in cycles necessary to carry out non-encryption related work in the context switch routine breaks the overlapping technique discussed above (i.e. there are not enough cycles to cover decryption work). Invalidation adds additional cycles and reduces the effectiveness of caching for code and data that is invalidated. In spite of these issues, caching has an overall

positive effect on performance. Invalidation is narrowly targeted to the process code and data, other code (e.g. kprintf, read only data, etc.) is still cached resulting in additional work being accomplished by each process during the same scheduling quantum. Each simple process executes a count that is printed to the screen. As an example of the overall benefit of caching, the counts were collected after 1000 context switches--with the L2 cache disabled counts of 20800000, 20600000, and 20700000 were recorded whereas with the cache enabled counts were 28700000, 28300000 and 28500000 for processes 1, 2, and 3 respectively.

Page Allocation Benchmark

As mentioned above, small processes only exhibit ME overhead during context switch. However, if a processes code or heap is larger than a single page, additional overhead results from access during the scheduling quantum. Stacks are typically defined at a certain maximum size and so are not considered in this benchmark. For example, in the Android operating system, stacks are commonly defined to be no larger than 4 or 8 KB [Bartel et al. 2012]. The benchmark can be run for any number of pages. Pages are first allocated and then written. The average cycles for this part of the benchmark are recorded. Next, all of the pages are read and again the average cycles are measured. The same number of pages are first allocated, written and read unprotected and then with memory encryption enabled.

Several important differences between the function of the unprotected and protected page allocation benchmark algorithms must be pointed out. For the unprotected benchmark, pages are allocated in eRAM and then written in place. In order to create pages safely when protected they must be written in iRAM and then encrypted

to the allocated location in eRAM. Similarly, when reading an unprotected page it is simply read in place whereas a protected page must be decrypted to an internal buffer before being read.

The results from running the page allocation benchmark are presented in Table 4 below. Measurements were made at 100, 1000, and 10000 page allocations. However, the differences in average cycles for each level were statistically insignificant and so the results are only shown for 1000 pages.

Table 4. Page Allocation Benchmark Overhead

Number of 4 KB Pages	Unencrypted Encrypted	Allocate/Write Read	Average Cycles without Cache	Overhead w/o Cache	Average Cycles with Cache	Overhead with Cache
1000	Unencrypted	Allocate/Write	1,681,600	N/A	39,872	N/A
		Read	1,683,400	N/A	10,112	N/A
	Encrypted	Allocate/Write	1,437,446	- 15 %	38,784	- 3 %
		Read	1,786,300	6 %	98,048	870 %

Without caching, the unprotected allocation/writing takes, on average, 1,681,600 cycles. Allocating and writing the same 4 KB page only takes approximately 1,437,446 cycles when protected (encrypted). This result is, at first, counterintuitive. Recall the differences in how the writes are done under the two modes (i.e. in iRAM when protected or in place in eRAM). Internal RAM (tighly coupled memory) has much faster access times than eRAM. This coupled with the fact that the page encryption can be overlapped with the work of allocating the next page account for the reduction in overall cycles. The average overhead for reading of encrypted pages is 6% wihout caching. Again, while enabling the cache improves the performance of the protected page allocation benchmark significantly, it is dwarfed by the improvement in the unprotected mode resulting in overhead of 870%. It is fairly clear that it is the decryption cycles that dominate the protected mode. Those cycles are not able to be reduced via caching. Note that it costs

approximately 91,000 cycles to decrypt a 4 KB object. This is close to the result we obtain when subtracting the unencrypted read cycles from the encrypted ones (98,048-10,112=~88,000). For these experiments, reading did not begin until the entire page was decrypted. It is possible to begin reading from the top of the page some time before the entire page is decrypted as reading each word takes time which would overlap with the additional decryption. Writing is still faster for the protected mode by about 3%.

These overhead measurements alone are not overly informative. They must be analyzed in consideration of a particular usage model. How many heap and/or code page reads does an average smartphone application make per second? The average size of an application on an Android device was 6 MB in late 2012 [Tom's hardware]. There are hard limits on the amount of heap space that can be allocated to each process similar to the limit on stack space. Further, there is a well-known rule of thumb in computer architecture--programs tend to spend 90% of overall time executing just 10% of instructions [Hennessy and Patterson 2006]. Assuming 50% of the average 6 MB application size are instructions the system will spend 90% of execution time in just 10% of 3 MB or ~76 4 KB pages worth of instructions. Data usage (stack, heap, static, etc.) varies depending on the type of application. When we add that to the other 3 MB for data we have ~844 4 KB pages. Accessing the average program at 100 pages per second would mean running through all of its data and highly used code in just 8.4 seconds. These assumptions are conservative and ignore the benefit of caching on the most highly accessed pages. Additionally, the reduction of cycles for protected writing is not taken into consideration.

Considering that each page read requires an additional ~87,936 cycles we have

8,793,600 cycles per second. The system is running at 800 MHz, so this is roughly equivalent to 11 milliseconds of overhead per second or 0.66 seconds of overhead in a minute. This equates to 1.1% overhead. Increasing the access assumption to 1000 page reads per second (3.9 MB per second or about 66% the size of the average Android application per second) results in 11% overhead. For average mobile application sizes and workloads exhibiting good locality the overhead for protection of all process segments is reasonable. Recall that the context switch costs for protecting all 3 segments are 100 ms per minute. Adding this into the overhead for larger processes that access additional code and data at 100 pages per second yields approximately 0.76 seconds of overhead per minute or ~1.3% overhead.

Worst Case Exploration

The context switch and page allocation microbenchmarks appropriately characterize the specific parts of the operating system where overhead is introduced. The average or typical smartphone workload tends to exhibit good locality and this is taken into consideration when selecting the number of page reads required per second above [Gutierrez et al. 2011]. However, it is important to understand the conditions under which system performance may degrade to unacceptable levels.

In order to approach the upper bound for worst-case ME performance, a radix-2, in-place fast Fourier transform (FFT) based on the Tukey-Cooley algorithm was implemented [Press et al. 1992]. This un-optimized version of FFT displays *extremely poor temporal and spatial locality*—memory accesses are pathological for the traditional memory hierarchy unless the entire data structure fits within the cache [Thomas and Yelick 2001]. The smallest block for decryption in AES is 16 Bytes. Since the data in

each component of the FFT (real and imaginary part) take up one word each (8 Bytes total), additional overhead is introduced in order to align the smaller data with the decryption algorithm requirements. Whereas the unprotected version implements a simple swap of two of the real and imaginary components in eRAM, the protected version must first determine the appropriate 16 Byte-aligned address to decrypt into the internal buffers for each component. Then the proper half of the 16 Bytes must be identified after which the swap is performed in iRAM, data re-encrypted and stored back to eRAM as shown in Figure 20.

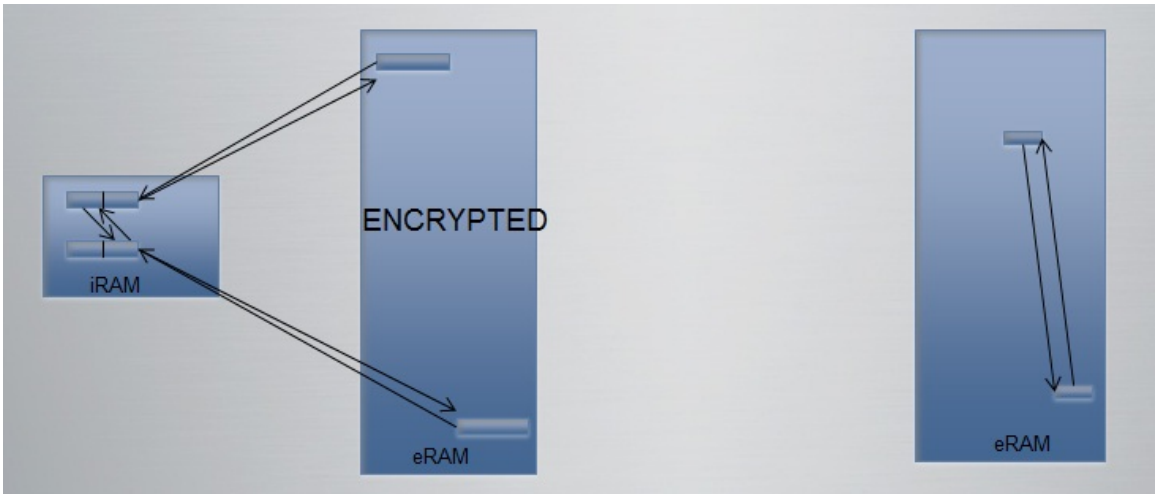

Figure 20: Protected FFT Swap (left) and Normal FFT Swap (right)

Recall that if all objects fit within the large iRAM buffer, the set of heap objects is decrypted there and used normally. However, if the objects are too large, decryption of data on-demand at the size appropriate to the application (or smallest size possible) is used. For completeness, *both techniques are explored* in this work. Because there is no locality to exploit in the FFT exemplar we default to decryption on demand at the 16 Byte size as mentioned above. Additionally, this requires modification of the code to add encryption and decryption calls to the process itself.

Table 5 shows the overhead of decryption of data in the FFT problem with 128 KB arrays holding the real and imaginary data components. Recall that the cycles per bit cost of decryption is large at the 16 Byte size (~71.5). In summary, about 19 billion cycles were required to execute the unprotected FFT. Without caching, encrypting all data for the entire FFT operation requires approximately 18 billion additional cycles (37.6 billion cycles total): resulting in a slowdown of approximately 2 times over the unprotected execution and overhead cycles of 98%. As additional levels of caching are enabled a widening gap between the average cycles for unprotected and protected heap objects emerges. With full caching of data and instructions, the slowdown is about 2.8 times with 185% overhead. For the FFT code, the additional overhead is due to both the EDU decryption cycles and a significant amount of other work required to correctly identify, decrypt and swap FFT components as discussed above.

Table 5. Overhead for FFT of 128 KB Heap Data Object

Security State of FFT Data Structure (128KB)	Average Cycles without Cache	Average Cycles Instruction Caching Enabled	Average Cycles Instruction/Data Caching & MMU Enabled	*Overhead* w/o Cache	*Overhead* Instruction Caching	*Overhead* Instr/Data Caching & MMU
Unprotected	18,958,489,984	7,849,799,296	916,376,832	N/A	N/A	N/A
Protected	37,626,951,104	20,883,119,680	2,607,492,810	98 % (~2X)	166 % (~2.7X)	185 % (~2.8X)

While the overhead for the un-optimized FFT with large data structure is significant, for smaller heap objects that are able to fit within iRAM, the cost of protection is quite reasonable--similar to the SEP case with the cost of one decryption for the heap object. Even in cases where the data structure is too large to fit into iRAM, mapping the FFT problem to the underlying memory hierarchy to improve performance

is an open area of research [Thomas and Yelick 1999; Aboleaze and Elnaggar 2006]. One such technique involves breaking a large FFT data structure into multiple sub-structures that are able to fit within cache and executing the algorithm on those sub-structures.

Table 6 below details the results of multiple runs of the FFT algorithm using a 16 KB data structure. The entire 16 KB data structure is decrypted to an iRAM buffer before executing the FFT. The results at first seem counterintuitive—for the first two cases (no caching and instruction caching) the average number of cycles for the encrypted FFT are actually *less* than in the unprotected case. Recall that decrypting a 16 KB chunk is extremely efficient compared to decrypting smaller sizes (e.g. 16 B). Additionally, unlike the PCB-stack case, there is very little additional bookkeeping necessary with a heap object. In fact, the only changes required were the addition of an EDU call at the beginning of the FFT and a change of address from the eRAM heap object to the iRAM buffer. However, this still does not account for the protected cases having fewer average cycles. This can be explained by the fact that iRAM has much faster access time than eRAM (where the heap object is accessed in the unprotected case).

Table 6. Overhead for FFT of 16 KB Heap Data Object – iRAM Buffer

Security State of FFT Data Structure (16 KB)	Average Cycles without Cache	Average Cycles Instruction Caching Enabled	Average Cycles Instruction/Data Caching & MMU Enabled	*Overhead* w/o Cache	*Overhead* Instruction Caching	*Overhead* Instr/Data Caching & MMU
Unprotected	2,097,284,928	812,444,992	76,247,232	N/A	N/A	N/A
Protected	1,968,859,072	791,610,688	81,605,376	No Overhead	No Overhead	7 %

The combination of an efficient chunk size, few bookkeeping instructions and the faster access results in fewer overall cycles. Another trend demonstrated by the data is the decreasing difference between the average cycles measured with increasing levels of cache utilization—leading to a crossover condition such that the protected case for instruction and data caching requires more cycles than the unprotected case. This can be explained by the fact that the cache exhibits even faster access time than iRAM (e.g. 1 cycle for L1 cache). These results are very promising for heap object protection when objects fit within the available internal space.

While the FFT was considered in this work due to its low level of locality, in reality, most mobile processor packages include single-instruction multiple data (SIMD) cores, such as the NEON processor to optimize algorithms like the FFT. Further, mobile system use tends to be characterized by interactive applications such as chat, e-mail, and those displaying spatial/temporal locality (e.g. photo viewing) [Gutierrez et al. 2011].

5.4 Dynamic Encrypted Processes (DEP) – Conclusion

The methods used in the literature for determining performance include mathematical model, simulation, kernel prototypes and FPGA prototypes with various benchmarking suites used in the latter three. *Simulation* is performed with (in order of decreasing usage) SimpleScalar, Simics, SESC, GEMS, SOC designer, RSIM, and M5. A group of the simulations utilize SimpleScalar and [Duc and Keyell 2006] note that this simulator *neglects* the impact of the operating system and other running processes. Besides these limitations, some authors admit a lack of model fidelity with significant differences between systems modeled and those targeted. For example, in [Chen et al. 2008] an x86 architecture is modeled since it happens to be better supported by the

simulation tool (Simics) even though the scheme is actually targeted at embedded-ARM systems. Unfortunately, even if a system under test were to be modeled perfectly, the simulation tools themselves have been shown to sometimes exhibit behavior unlike real systems. In [Muller et al. 2011], the behavior of CPU registers is interrogated under simulation in QEMU with the contents surviving soft-boot. This behavior would circumvent the protections afforded in that work, however, real hardware behaves differently and zeroes out the registers.

The range of overheads reported in the literature is quite large (1% to 6015%). The results on the lower end of the spectrum are possibly overly optimistic given the lack of fidelity in the simulation frameworks and the lack of standards for comparison. If standardization could be injected into the validation methodologies through common AES decryption latency, benchmarks etc. it would enable more meaningful comparative analyses. Even with standardization, the number of assumptions make it difficult to be confident that simulation will provide anything more than high-level information: It ignores the more difficult and interesting implementation issues and associated security impact based on vulnerability and exploit analysis. Where, in the few cases available, the literature addresses these low-level issues, it tends to be with generalization since there is no chance for practical experimentation or empirical evidence [Lie et al. 2000; Shi et al. 2004; Chhabra et al. 2010]. While the security of the encryption algorithm or cipher mode is often pointed out, it is commonly the complexity of the system in which these algorithms run that presents vulnerabilities. The most developed, though not commercially available, general-purpose technologies are FPGA soft-core emulations [Suh et al. 2007] and the Linux prototype used in Cryptkeeper [Peterson 2010]. While

the industrial devices are mature and practical, they are not general purpose, catering to highly specialized operations.

Various approaches are used in the literature for measuring overhead ranging from worst case through average case to best case with the preponderance of work measuring the latter. It is important to understand the performance characteristics of the average and worst-case (on demand decryption) scenarios where decryption overhead is added directly to memory access time. In this work, it was anticipated that performing memory encryption without the benefit of the MMU and cache (including prefetching etc.) would yield excessively large overheads. In reality, while using caching improved the overall performance of the ME system it was dwarfed by the improvements to the unprotected system as the overhead of the EDU decryption itself could not be mitigated. In other words, while the number of cycles for the protected system with caching was much improved over the system without caching, the overhead when compared to the unprotected system was much higher.

While PCB-stack and code protection added significantly to the average cycles of the context switch, the overall system slowdown would be just 0.6 seconds after an hour of execution. For the majority of smartphone applications (displaying good locality) the page allocation and context switching benchmarks appropriately capture the cumulative overhead which is low at ~1.3%. It is only in *pathological* cases, where there is little to no locality, that the overhead grows significantly. The results of on-demand decryption for heap objects were fairly onerous at 2x-2.8x the unprotected case. However, the slowdown for the FFT is better than that reported in the simulation results of a similar technique that took advantage of caching mechanisms but lacked encryption hardware.

In that work, slowdowns of 2.53x and 8.5x were measured when utilizing a 4 KB page with a 256 KB and 64 KB L2 cache respectively [Chen et al. 2008]. Where the heap object fits into iRAM, the faster access time resulted in fewer cycles in the case of no caching and instruction caching and only 7% overhead with full caching. Further, there are techniques for breaking large data structures down to fit within the underlying cache (and iRAM) architecture meaning that simple optimization techniques can significantly improve the overhead of worst-case applications. These results suggest that ME is a feasible technique today given the commoditization of security hardware.

Chapter 6: Mutually Distrusting Processes and Measuring Protection

The previous chapters have considered protection of a system against hardware-based attacks (e.g. bus snooping, injecting, cold boot, etc.) and against some types of malicious code (e.g. worm or Trojan horse dropped on the local system). One of the assumptions of this type of model is that the user processes are *trusted* to some degree. They do not have access to the encryption key, but they share the same key with all other processes. Unfortunately, the possibility of downloading a malicious application from an application store is quite high [Zhou et al. 2012]. This leads to the idea of *mutually distrusting processes* (MDP). Mutually distrusting processes could be those downloaded from an application store such as those used for Android and even iPhone devices today. While Apple makes an effort to check all such applications, several applications have been included in the application store having malicious characteristics. Although these have been removed from the store (when identified), the damage had likely already been done (e.g. loss of users' private information).

6.1 Mutually Distrusting Processes

The inherent trust placed in processes running at the same level in an operating system is often misplaced. Besides a user accidentally downloading a malicious application, another vector for infection involves a trusted insider purposely installing one. Both Windows and Linux systems provide functions for reading the memory of other processes running on the system allowing for *passive memory scanning attacks* (i.e. stealing data) and *active modification attacks* where code is changed [Gutmann 2000]. One particular version of this attack is increasing in usage and has been dubbed a *memory scraper virus* when used against point of

sale (POS) systems to steal credit card information [Baker et al. 2008]. This issue may be countered to some degree by enforcing a constraint on a group of users to download applications only from a *trusted app store*. One could imagine such a scenario, for example, with smartphones used in military operations. Applications could be encrypted and signed at the trusted app store for delivery to devices in the field. Those applications would then be decrypted with a key shared by a military unit, and re-encrypted with the devices' unique key for use. However, even under this model the potential for an insider attack still exists whereby malicious code could be inserted into a trusted application.

One way to increase the security of each process is to encrypt it with a unique key (i.e. key scope:process). In this way, a malicious process can not attempt to execute a known plaintext-known ciphertext attack on the key itself and, more importantly, can not use techniques to read other processes memory outside of strictly controlled channels. Any attempt to do so through a break in normal control flow would lead to the victim processes code/data decrypting to garbage using the malicious processes key. This technique also increases attacker workload for an attacker attempting to capture large amounts of RAM data for a brute force attack--RAM would be broken up into sections encrypted with disparate keys. Based on this additional protection and the performance measurements below, this technique may be useful even for systems where all processes are trusted.

Unfortunately, if processes are suspect, the problem of covert and side channels (e.g. timing, sharing cache, etc.) produced from sharing internal hardware must be addressed [Wang and Lee. 2004]. Since the IMX53 encryption engine maintains

its own memory for key expansion it is protected against the statistical timing attack on the encryption key. However, as a byproduct of using the same iRAM buffers to hold the decrypted code and data (including heap objects) of each process, a *direct channel* is created. If the code/data of a previous process is not sanitized the next process may be able to access it directly (assuming the next process is small enough to not overwrite the entire buffer). For our iRAM buffers, this can be addressed simply by ensuring that we always decrypt the same number of bytes to overwrite the previous contents (even if the code/data required is smaller). This is especially necessary for the internal heap object buffer as processes are allowed to freely access that space while the iRAM code and PCB-stack buffers are somewhat constrained by normal control flow. One technique to protect against these covert and side channels in cache is to simply disable it. Using multiple internal buffers (one per process constrained by the size of iRAM) could also mitigate the leakage of information. For the cache, the requirement for invalidating and cleaning during context switching should help protect processes from covert and side channels. While these techniques do thwart the obvious side (direct) channels, we do not address subtle attacks such as cache timing, which are outside the scope of this work.

Having an encryption key per process that is strictly under the control of the secure microkernel will help mitigate such attacks on the privacy of data and the control flow of other processes. The modifications required to the dynamic encrypted processes prototype were fairly straightforward. In order to facilitate MDP protection in our system, a key table was added to iRAM in order to store a per-process unique 128-bit AES key. At context switch time, the microkernel selects the appropriate key for encryption/decryption of each process. Since the system

currently uses a round-robin scheduling algorithm the key is selected using the following code:

*nextkey=0xf8000040+(((currentp->pid + 1)%3)*16);*

Note that the key table is placed in iRAM at location 0xf8000040. While a simple if statement would suffice in this situation (and with fewer cycles) the code above generalizes to any number of processes. Depending on the type of scheduling used, and the number of processes (and thus size of the key table) a fast hash lookup could also be used. The measurements for MDP protection without cache or MMU are shown below in Table 7.

Table 7. MDP Measurements

Component within Context Switch	Average Cycles without Cache DEP	Average Cycles without Cache MDP	Overhead for MDP compared to DEP
Unprotected	16384	16384	N/A
PCB-Stack	50176	51096	1.8%
Code	24832	25920	4.4%
Heap	50100	51068	1.9%

Measurement of MDP protection was carried out in the same way as for previous prototypes—averaging the cycles for 1000 context switches of three processes. For comparison, the measurements for DEP protection are also provided. The only additional overhead introduced during context switching is from the key lookup. To verify this the performance monitors were used to measure the average cycles of the key lookup (~1920 cycles). Additionally, since each process uses its key throughout its scheduling quantum there is no additional overhead added to the page allocation benchmark.

6.2 Exploring Memory Encryption Protection (Confidentiality and Integrity)

As espoused in the ME literature, understanding the overhead associated with memory encryption is extremely important in order to gauge its suitability for various users and situations. Because of this, the work in previous chapters was devoted to understanding the intricacies of implementation and measuring the associated overhead for SEP, DEP and MDP. However, an area quite lacking in the literature is a thorough examination of the security properties provided by memory encryption. For the majority of work (as discussed further in Chapter 7) the main threat model includes the end user with the goal of protecting proprietary software and data from theft. Some of the more recent work includes a model similar to that in this work—protection of the end user against malicious code and adversaries with physical access. Regardless of the threat model, the security property highlighted in ME work is *confidentiality* of RAM. This is the first work in ME to explore the additional integrity protections afforded by memory encryption.

In the book Trusted Computing Platforms (2005), Smith identifies a 2X2 taxonomy of goals for the trusted computing platform: confidentiality and integrity for code and/or data. However, as in the work in the ME literature, it is assumed that *confidentiality* is provided by encryption and that *integrity* requires separate authentication mechanisms. The work in this thesis asserts that while integrity mechanisms do not necessarily provide confidentiality--encryption mechanisms (specifically used in memory encryption) do provide integrity protection of varying degrees for both code and data.

An attacker lacking the encryption key will be able to modify (inject) memory in various ways but will inject plaintext code or data. That code or data will be decrypted into random "garbage" in iRAM and scheduled for execution (code). The fetch-decode-execute hardware itself acts as an integrity engine for code segments, producing exceptions to identify modification. While data does not necessarily have a method for quickly identifying modification, changing it in a predictable way is difficult. Further, since memory is encrypted identifying the location of code and data to change is difficult. Additional information can be gleaned from memory access patterns over time and protecting against this subset of pattern analysis is the subject of research as well as various techniques implemented in industrial applications. For example, the Dallas Semiconductor 5002FP secure processor encrypts memory addresses to prevent traffic analysis on the memory bus and uses spare processor cycles to place dummy memory accesses on the bus [Gao et al. 2006].

Analytically, the strength of the protection afforded by an encryption algorithm is described as a function of the block size. For example, if we are using a decryption size of 16 bytes, the strength against a brute force attack is 2^128. Unfortunately, this theoretical strength is often reduced due to the side-channel leakage of information on practical implementations (i.e. the intricacies of implementation). For example, it has been shown that ME processors can be compromised via what has been titled a cipher-instruction search attack [Kuhn 98]. In such an attack, the output busses from the main CPU are used as a form of side-channel to gather information while random "guesses" are placed in encrypted memory, decrypted and executed via system restart. In this way,

the author is able to eventually build a small program to cause the system to write out the internally protected contents. The system attacked used a small block size of 8 bits, making the attack quite feasible with very little cost and time. For the system described in this work, the smallest possible block size is 16 bytes. The size of an ARM instruction is 4 bytes. If we consider a similar attack where the attacker was trying to find an instruction in the first word of a 16 byte sequence the brute force space is actually reduced from 2^128 to 2^128-2^96 because there are 2^96 inputs for the 16 bytes that will produce the correct output for the first word. This means if we have attempted 2^128-2^96 random values without success, the next value must be one that will produce the correct output for the first word. While it is important to consider this type of attack since it has been perpetuated against an actual ME architecture, a brute force attack is still infeasible since the space is essentially still 2^128.

Because of the nature of most of the memory encryption work in the literature (simulations) there is very little information regarding the actual security enhancements. Only one of the papers surveyed mentioned the possibility of protection against *remote code injection* attacks via a reference to another work that highlighted protection against *buffer overflow* attacks in the instruction set randomization (ISR) area. However, after thorough consideration and analysis of several of the ISR techniques it was determined that protection against buffer overflow is not a practical goal of this work. This is because the smallest key-scope granularity considered for this work is per-process. Any information entered into a buffer (legitimate or not) would either be decrypted by the valid key or entered when the buffer is resident in iRAM in unencrypted form. The way that encryption (or randomization) has been explored to protect against such buffer

overflows is to narrow the key-scope all the way down to the return pointer. Since the rest of the stack does not share the same key (or indeed may be unencrypted), any plaintext address written over the pointer will result in an exception as it is randomized (decrypted).

Experimentation was undertaken to provide evidence supporting the claim of confidentiality and especially integrity protection of code and data. In order to test whether our system protected the confidentiality of eRAM, system execution was halted with both code and PCB/stack protection enabled. An examination of eRAM via the JTAG-TAP revealed no evidence of any plaintext information. Besides searching the specific address ranges where code and data was known to reside, large sections of memory were exported for analysis via a *strings* function. While this was a straightforward exercise, it was important to verify that the system was protecting eRAM as advertised.

In order to empirically test protection against code injection, the ARM exception vectors had to be modified. In a fully developed system, the exception vectors would be used to halt execution of the offending process. Additionally, the technique could be used to signal process *resurrection* or selection of a copy already running and encrypted with a new random key to enable resilience [McGill and Taylor 2011]. The offending memory location (and some range of memory surrounding it) could also be captured for forensics analysis to determine if there was an attack or simply an error. The default exception vectors in ARM include Reset, Undefined Instruction, Supervisor Call, Prefetch Abort, and Data Abort. The prefetch and data aborts indicate invalid memory locations for instructions and data respectively. The exception code was modified to

count the number and type of exception and number of instructions executed before exception. Additionally, the code restored the processes stack and program status register value (resetting the test), randomized the encryption key via the SAHARA random number generator (RNG), re-randomized the original plaintext code with the new key and reset the program counter to the original instruction. Valid plaintext code was placed in an internal buffer before being 'decrypted' and executed. Once an exception occurred, the tests continued in an automated fashion (see Figure 21).

Occasionally, the test would become stuck in an infinite loop somewhere in memory. For example, the PC was updated to point to the middle of a counting loop but the counting variable had not been set up properly (which would happen under normal control flow). While this is not the desired behavior, it would be possible to check the progress of a process after some given time after the expiration of which the process could be killed. Additionally, a system with the MMU and paging enabled could prevent such accesses. An additional error that was occasionally observed was the "illegal system call" that is not one of the defined ARM exceptions. It was originally thought that this was an error code from one of the system's coprocessors. However, it was later identified as one of Bear's error codes. Some of the random executions would invoke the 'Supervisor Call', which is the way Bear executes calls for process creation etc. While the random code was able to execute a supervisor call (very infrequently), the additional information required for a valid call was never correctly provided (the probability of that happening is statistically insignificant) [Barrantes et al. 2003]. There was also an issue with imprecise exceptions where an exception is thrown several instructions after it actually occurs (often in the middle of another exception in our attack testing scenario).

Setting a bit in the program status register that disabled all imprecise exceptions rectified this situation.

```
Bear (v1.0) - ARM Cortex-A8 iMX53QSB
PCB NOT ENCRYPTED
ENCRYPT_CODE
[heap: start=F800A0C4, end=F801D400, size=78652 (bytes)]
[checking heap...]
[proc 0: sp F801D16C: sb F801D024: context switches = 1
UNKNOWN-INSTRUCTION-TRAP
DATA-ABORT-TRAP
UNKNOWN-INSTRUCTION-TRAP
DATA-ABORT-TRAP
INSTR-PREFETCH-TRAP
INSTR-PREFETCH-TRAP
UNKNOWN-INSTRUCTION-TRAP
DATA-ABORT-TRAP
UNKNOWN-INSTRUCTION-TRAP
DATA-ABORT-TRAP
DATA-ABORT-TRAP
UNKNOWN-INSTRUCTION-TRAP
DATA-ABORT-TRAP
INSTR-PREFETCH-TRAP
INSTR-PREFETCH-TRAP
DATA-ABORT-TRAP
DATA-ABORT-TRAP
DATA-ABORT-TRAP
DATA-ABORT-TRAP
DATA-ABORT-TRAP
DATA-ABORT-TRAP
```

Figure 21: Small Section of Exception Output from Code Injection Testing

The code injection testing was successful and provided evidence of the integrity protections afforded by memory encryption. On average, about 300 exceptions occurred before an infinite loop. The infinite loops never executed anything useful. The average exception profile was 132 Unknown Instructions, 23 Prefetch Aborts, 123 Data Aborts, and 22 Illegal Supervisor Calls. For additional clarity, the Eclipse disassembly preview of the same section of code before and after scrambling by decryption is shown in Figure 22.

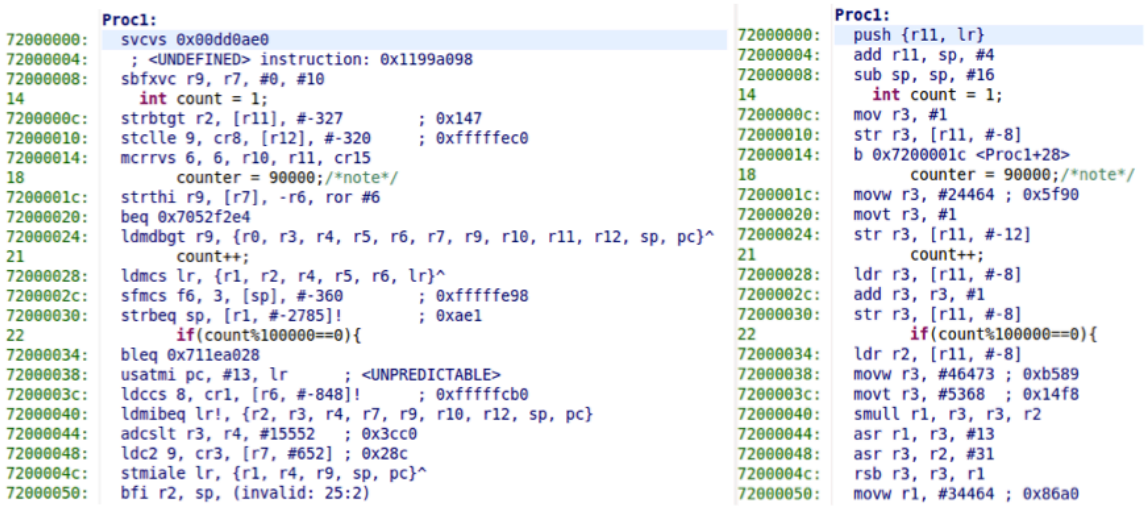

Figure 22: Process 1 After Scrambling via Decryption (left) and Original Plaintext (right)

Instruction set randomization (ISR) work is closely related to this particular aspect of memory encryption. The goal of ISR is to prevent the execution of remotely injected code. This is usually done by rewriting a binary with varying encryption schemes (e.g. XOR to AES) and decrypting the instructions to a buffer that rests in the same external memory. This technique *provides no confidentiality* protections to code since all of the code is available at some point in time and vulnerable to the various attacks described earlier in this work (e.g. bus probing and injecting). Additionally, the ISR work only targets the protection of code, not data. In one of the earliest works on ISR, Barrantes et al. run a similar test of their ISR system to determine the number and types of exceptions encountered on an x86 system. For comparison, the results of that testing is shown below in table 8.

Table 8: Results of ISR Testing

Outcome	Percent	Cumulative
SIGSEGV (illegal mem access)	84%	84%
SIGILL (illegal instruction)	15%	99%
SIGFPE (divison error)	0.6%	99.6%
SIGBUS (illegal memory alignment)	0.3	99.9%
LOOP	0.1%	100%

While the tests on both platforms demonstrate the effectiveness of randomization against injection attack, the ratios of types of exceptions are markedly different. For example, a large percentage (84%) of exceptions in the x86 work are attributed to illegal memory accesses as compared to about half (49%) of the exceptions on the ARM architecture (combining prefetch and data aborts). This is likely because the tests on the x86 system were carried out with paging enabled. This means that memory that does not belong to the current process caused a SIGSEGV exception as well as completely invalid memory addresses (the latter being the only accesses which cause prefetch and data aborts in this work since paging has not been implemented). Additionally, the ratio of illegal instructions in the ARM compared to x86 is about 3:1. ARM instructions are of fairly fixed lengths (ARM 4 bytes, Thumb 2 bytes) whereas x86 instructions are of variable length (1 byte to 16 bytes). The additional constraints on ARM code likely results in more exceptions.

6.3 Summary

This chapter has explored an extension of the basic ME prototype to protect mutually distrusting processes by encrypting each process with its own unique 128-bit

AES key. Additionally, the confidentiality and integrity protections afforded by ME were explored and measured. For very few additional cycles (~1920), mutually distrusting processes can be made more secure. AES is secure against known-plaintext attacks, but even if a process could somehow successfully brute force the key, it would reveal nothing about the other processes' keys. Additionally, if the malicious process were to try and access the memory of victim processes outside of the normal flow control (including access control) of the system it would be unable to decipher any data. Besides protecting the processes from each other, this increase in the number of keys used to encrypt RAM as a whole should further complicate brute force attacks—*increasing attacker workload*.

In the ME literature, the goal of ME is to provide confidentiality of code and data. This is the first work in the genre to identify the additional integrity protections afforded to both code and data. Exploration of integrity protection via randomization of code under a changing key demonstrates the effectiveness of the approach. While there is a small chance of an attack executing pre-existing code in the system, the chance of that code benefitting an attacker is even smaller [Barrantes et al. 2004]. This is especially true considering the experiments above were designed to rapidly test newly randomized code. Under normal conditions the attacked process would be killed and a new process, randomized under a different key (and likely in a different part of memory) instantiated. This process throttles code injection attacks—it takes some amount of cycles/time to accomplish and also forces an attacker to spend additional time to target the new location in memory.

Chapter 7: Future Work and Conclusions

While this work has demonstrated that memory encryption is now viable because of the commoditization of security hardware, there are many more questions to be answered and improvements that could be made. The emphasis in this work has focused on developing and measuring the basic mechanisms for ME with a range of architectural support (i.e. no cache through full cache and MMU support). Measurement in the ME literature varies but the most common measurements are of the average or best case workloads with very few reporting worst case scenarios. By approaching the upper bound on overhead via implementing on demand decryption it is easy to reason about the likelihood of these techniques being used. There are many approaches that can be used to improve the performance of these prototypes.

7.1 Future Work

The first and most straightforward technique has already been explored to some degree in this work—*overlapping*. It was shown how overlapping the decryption work with other work in the context switch routine saved about 10,000 cycles per context switch. Enabling the cache had an overall beneficial effect on the average number of cycles but broke the overlapping technique since most of the switching code was cached (and therefore reduced in cycles). However, a more effective way to use overlapping would be to overlap complete processes. At the beginning of context switching, two processes would be decrypted into two separate sets of internal buffers with a third set remaining empty. As encryption tasks can be queued, at context switch time, the outgoing process would be added to the queue to be encrypted to eRAM, the next process would be added to the queue for decryption into the empty set of buffers. Finally, control

would pass to the decrypted process for execution. In this way, nearly all of the encryption overhead for context switching should be hidden. To increase the size of segments and the number of processes that could be overlapped in this way, the use of other internal storage areas (e.g. 256 KB video buffer in iRAM) in combination with iRAM should also be explored. Basic experimentation with this additional storage demonstrated that it can be used as easily as iRAM.

Another area to be explored is paging. Recall that the measurements of overhead for decryption showed the most efficient decryption sizes were 4 KB and above. Exploration should consider both the performance of crypto-paging and the implications on security. For example, would encryption simply be an on or off decision? Sampling or tracing particular applications to determine the level of locality exhibited could be used for the dynamic selection of even larger page sizes (> 4KB) which should improve overall system performance.

While most of the work of this thesis has concentrated on implementation, there is ample room for developing policy for its use. The granularity of process segment allows for a *graduated response* to protection that could balance currently known threats/goals (e.g. military infocon levels) with associated overhead costs. For example, even if areas prone to having sensitive information are not identified, developers working in a trusted application store can be advised to target specific kinds of storage for sensitive information (e.g. the stack). Further, policies may be developed to link the mode of ME (e.g. PCB-stack, heap, code) to the goal of the protection. For example, digital rights management (DRM) is concerned with the protection of code and so may only require code encryption. An analysis of the overheads involved and types of protection provided

at various levels of granularity may lead to an optimization of the techniques as expressed in usage policy.

Another area that should be examined with regard to policy is when to use ME. For example, if a smart phone has not had its screen locked, there is little need (or likelihood) for an adversary with physical access to attempt a physical memory attack as they likely already have access to anything on the phone. It is worth considering whether a better model for application of ME would be to encrypt eRAM as soon as the phone locks. This idea would also work for x86 systems. For example, recall one of the attacks against a windows system that was at the login screen whereby an iPod was attached that searched through memory and overwrote the password checking routine with NOPs. If that system were to encrypt its RAM as soon as it locked, then this attack would be mitigated. Additionally, this usage model reduces any noticeable overhead for the user. The best solution for protection and usability may come from the proper combination of performance enhancing techniques (e.g. overlapping) and policy. It should be noted that while this policy would protect against many of the hardware based attacks highlighted in this work it would not provide protection against malicious code (e.g. worms dropping executables on a running system) or mutually distrusting processes such as the RAM scrapers described earlier.

Once paging is developed for the ME system, the next major project to consider is virtualization. While the Cortex A8 does not include virtualization hardware, there are other techniques for implementing virtualization (although slower). Further, porting the current system to the A15 architecture, which does include hardware for virtualization support, should not prove too difficult. The Bear microkernel has also been developed as

a micro-hypervisor upon which the Bear microkernel and BSD can boot. The memory encryption techniques explored in this work should be built into a hypervisor that will run on the ARM A15 architecture. Once accomplished, the next step is to enable booting of the Android OS on the hypervisor. In this way, ME protection can be provided *transparently* for Android and all applications running on top of it. Alternatively, the Android dalvik virtual machine (DVM) could be modified to add ME techniques.

Approaching a robust system will require attention to the other parts of a trusted system. For example, Android's file encryption method should be integrated with the ME technique. Automated file encryption is already available in Android and makes use of *on the fly* decryption for every read/write operation on the block device [Skillen et al. 2013]. A fully protected system must make use of both FDE and ME as well as trusted boot mechanisms as discussed in the assumptions and limitations of this work.

Software encryption techniques should be ported and/or developed for the A8 and the performance measured for the same range of experiments used in the hardware tests. For example, the networking and cryptography library (NaCl) suite has been modified to work with ARM NEON hardware at surprisingly fast speeds [Bernstein and Schwabe 2012]. NEON is a single instruction multiple data (SIMD) processor often used for vector operations such as video encoding. NEON is much more ubiquitous in various ARM processors than the proprietary EDU used in this work increasing the likelihood of adoption of techniques developed with it. However, it is unclear whether code and data would be protected everywhere outside of the processor when utilizing the NEON engine.

7.2 Conclusions

This thesis proposed to solve the problem of increasing attacks on *data in use* in memory (e.g. cold boot attacks, memory scraper viruses, bus snooping and injecting, etc.) which has been exacerbated by the growing size of memory, changing usage models which invalidate old assumptions of volatility and increasing adoption of FDE. In order to mitigate this problem, the idea of increasing the artificial diversity of RAM in order to *increase attacker workload* to the point where the costs of attack outweigh the benefits via memory encryption was explored. In this way, commercial systems can be protected since criminals typically target the most vulnerable systems. Further, time sensitive information such as that used in military operations would not be available to attackers until after its useful life (e.g. after a mission is complete).

While there have been three decades of research into memory encryption, that research has focused primarily on the design of the ideal monolithic processor with a hardware engine integrated into the fetch-decode-execute gateway. Other, more recent research has focused on software only approaches but these have proven too costly in overhead. However, recently there has been a commoditization of security hardware into processors such as the Intel AES-NI and various ARM architectures. The hypothesis in this work was that memory encryption could now be implemented with acceptable overheads using this nascent security hardware.

A collection of novel memory encryption techniques was developed--providing *synthetic diversity* and *increasing attacker workload.* The hypothesis was explored through various prototypes from static encrypted processes (SEP) and through dynamic and mutually distrusting processes. SEP sought to introduce synthetic diversity into

memory to protect microcontrollers and other real-time processors commonly used in industrial control systems (e.g. lacking a memory management unit and little to no cache) via a one-time decryption into internally protected space. This technique produced very little overhead. DEP sought to introduce synthetic diversity into memory to protect smart phone and other mobile computing devices characterized by multitasking operating systems including memory management units and cache. For the first time in the ME literature, implementation on commodity hardware enabled exploration of protection at *process segment* granularity. Protection of code and the PCB-stack data was *transparent* to processes (and developers) and the overhead was quite modest since that overhead was only experienced at context switch time (approximately 200 times per minute). Protection of heap objects is transparent for objects that fit into iRAM. However, for large structures that do not display temporal/spatial locality changes were required to application code and the protection was more expensive. Still, the results were better than those in the literature. Further, heap objects that fit into iRAM demonstrated *better* performance than unprotected versions since iRAM has shorter access times than eRAM. Finally, the DEP approach was extended to protect mutually distrusting processes (MDP) from each other via an increase in key granularity (i.e. a unique key per process) resulting in a very modest (~1900 cycles) increase in overhead.

These techniques protect against software and hardware based confidentiality and integrity attacks and are portable to currently deployed general-purpose, security-enhanced processors. An analytical framework was presented to include performance benchmarks and analyses on the overhead of memory encryption at *process segment granularity*. This work is the first in the genre to identify and explore the integrity

protections afforded by ME. The problem of *self-modifying code* associated with memory hierarchy interaction in a memory encryption system was introduced and explored. Finally, memory encryption techniques were explored through a comparative analysis of three decades of research and proposed solutions. Widely varying assumptions and experimental conditions were controlled to provide a basis for comparison of that research.

These systems and techniques have been demonstrated in proof-of-concept implementations and exemplars. Memory encryption has been implemented to provide *automatic* and *transparent* protection for applications. This transparency is achieved through extension of a secure microkernel that was ported to an ARM Cortex A8 processor. The techniques have been implemented as modifications to linker scripts, initialization, process creation and context switching routines as well as new modules for interfacing with the encryption decryption unit (EDU). These techniques have been demonstrated by encrypting processes while they reside in external RAM (eRAM) thereby adding *synthetic diversity*. The implementations cover a range from an unsophisticated processor with no memory management unit (MMU) and cache to one with an MMU and 192 KB of L1/L2 cache. Additionally, various granularities of protection are explored. Finally, exception-handling routines have been developed and experiments executed to understand the protections afforded against code and data injection. The low overhead results for typical workloads (~1.3%) and ability to easily optimize even the worst-case examples to ~7% overhead indicate that memory encryption is viable today on *security-enhanced commodity processors*.

Notice

REFERENCES

ANDERSON, R., and KUHN, M. Tamper resistance – a cautionary note. In *Proceedings of the Second USENIX Workshop on Electronic Commerce*. 2 (November 1996), 1-11.

ARNOLD, T., and DOORN, L. The IBM PCIXCC: a new cryptographic coprocessor for the IBM eserver. *The IBM Journal of Research and Development*. (2004), 120-126.

BAKER, W., HYLENDER, C., and VALENTINE, J. Data breach investigations report. Verizon Business RISK Team, <http://www.verizonbusiness.com/resources/security/databreachreport.pdf> (2008).

BARRANTES, E., ACKLEY, D., FORREST, S., PALMER, T., SEFANOVIC, D., and ZOVI, D. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communications security (CCS '03)*. (October 2003), 281-289.

BARTEL, A., KLEIN, J., ALLIX, K., TRAON, Y., and MONPERRUS, M. Improving privacy on android smartphones through in-vivo bytecode instrumentation. *CoRR*. (2012).

BERNSTEIN, D., AND SCHWABE, P. NEON crypto. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012, 14th*

International Workshop, Proceedings, volume 7428 of Lecture Notes in Computer Science. Springer-Verlag Berlin Hedelberg. (Sep 2012).

BEST, R. Crypto microprocessor for executing enciphered programs. U.S. patent 4,278,837. (14 July 1981).

BEST, R. Crypto microprocessor that executes enciphered programs. U.S. patent 4,465,901. (14 August 1984).

BEST, R. Microprocessor for executing enciphered programs. U.S. patent 4,168,396. (18 September 1979).

BEST, R. Preventing software piracy with crypto-microprocessors. In *Proceedings of the IEEE Spring Compcon*. (February 1980), 466-469.

BOILEAU, A. Hit by a bus: physical access attacks with firewire. Presented at *Ruxcon*. (2006).

BRINK, D. Full-disk encryption on the rise. *Aberdeen Research Group Report*. (September 2009).

CASEY, E., FELLOWS, G., GEIGER, M., and STELLATOS, G. The growing impact of full disk encryption on digital forensics. in *Digital Investigation*. 8 (September 2011), 129-134.

CHAHAL, S., KAMHOUT, D., KOHLENBERG, T., KUMAR, M., MANCINI, S., MORGAN, D., PURCELL, S., ROSS, A., and SMITH, C. Evolution of integrity checking with Intel trusted execution technology: an Intel IT perspective. *Intel white paper*. (August 2010).

CHARI, S., JUTLA, C., RAO, J., and ROHATGI, P. Towards sound approaches to counteract power analysis attacks. In *Proceedings of the CRYPTO'99: 19th Annual International Cryptology Conference*. 1666 (1999), 398-412.

CHECKOWAY, S., HALDERMAN, J., FELDMAN, A., FELTEN, E., KANTOR, B. and SHACHAM, H. Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage. in *Proceedings of the USENIX/ACCURATE/IAVoSS Electronic Voting Technology Workshop*. (August 2009).

CHEN, B., and MORRIS, R. Certifying program execution with secure processors. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems*. (2003), 23-29.

CHEN, X., DICK, R., and CHOUDHARY, A. Operating system controlled processor-memory bus encryption. In *Proceedings of DATE*. (2008).

CHHABRA, S., ROGERS, B., SOLIHIN, Y., and PRVULOVIC, M. SecureMe: a hardware-software approach to full system security. In *Proceedings of the International Conference on Supercomputing (ICS)*. (May 2011).

CHHABRA, S., and SOLIHIN, Y. i-NVMM: a secure non-volatile main memory system with incremental encryption. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. (June 2011).

CHHABRA, S., SOLIHIN, Y., LAL, R., and HOEKSTRA, M. An analysis of secure processor architectures. In *Transactions on Computational Science VII*. Marina L. Gavrilova and C. J. Kenneth Tan (Eds.). Lecture Notes In Computer Science. Springer-Verlag, Berlin. 5890, (2010), 101-121.

CHO, H., EGGER, B., LEE, J., and SHIN, H. Dynamic Data Scratchpad Memory Management for a Memory Subsystem with an MMU. in *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 195-206, Vol 42, Issue 7. (July 2007).

CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., and ROSENBLUM, M. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium*. (August 2004).

CONRAD, S., DORN, G., and CRAIGER, P. Forensic analysis of a sony playstation 3 gaming console. In Advances in Digital Forensics VI. K.P. Chow and S. Shenoi (Eds.). AICT 337, (2010), 65-76.

CORTEX-A Series Programmer's Guide-Version 2.0. Available online at http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0013b/index.html

DALLAS SEMICONDUCTOR. Secure microcontroller data book. Dallas, (1997).

DAVI, L., DMITRIENKO, A., SADEGHI, A. and WINANDY, M. Privilege Escatlation Attacks on Android. Information Security, Springer. (2011).

DUNN, A., HOFMANN, O., WATERS, B., and WITCHEL, E. Cloaking malware with the trusted platform module. In *Proceedings of the 29th USENIX Conference on Security*. (2011), 26.

DUC, G., and KERYELL, R. CryptoPage: an efficient secure architecture with memory encryption, integrity and information leakage protection. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. (2006).

EAGLE, C. The IDA Pro Book. No Starch Press. (2011).

EILAM, E. Reversing. Wiley. (2005).

ELBAZ, R., CHAMPAGNE, D., GEBOTYS, C., LEE, R., POTLAPALLY, N., and TORRES, L. Hardware mechanisms for memory authentication: a survey of existing techniques and engines. In *Transactions on Computational Science*. 4, (2009), 1-22.

ELBAZ, R., TORRES, L., SASSATELLI, G., GUILLEMIN, P., ANGUILLE, C., BARDOUILLET, M., BUATOIS, C., and RIGAUD, J. Hardware engines for bus encryption: a survey of existing techniques. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*. (2005).

ENCK, W., BUTLER, K., RICHARDSON, T., MCDANIEL, P., and SMITH, A. Defending against attacks on main memory persistence. In *Proceedings of the 24th Annual Computer Security Applications Conference*. (December 2011).

FORRESTER, J. and MILLER, B. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In the *4th USENIX Windows Systems Symposium*, Seattle. (August 2000).

FRANTZEN, M., and SHUEY, M. StackGhost: hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*. (August 2001).

GAO, L., YANG, J., CHROBALL, M., ZHANG, Y., NGUYEN, S., and LEE, H. A low cost memory remapping scheme for address bus protection. In *Proceedings of the 15th*

International Conference on Parallel Architecture Compilation Techniques (PACT). (September 2006).

GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., and BONEH, D. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. (2003).

GASSEND, B., SUH, G., CLARKE, D., DIJK, M., and DEVADAS, S. Caches and hash trees for efficient memory integrity verification. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*. (February 2003), 295.

GILMONT, T., LEGAT, J., and QUISQUATER, J. An architecture of security management unit for safe hosting of multiple agents. In *Proceedings of the International Workshop on Intelligent Communications and Multimedia Terminals*. (November 1998), 79-82.

GUERON, S. Intel advanced encryption standard (AES) instructions set. *Intel Technical Report*. (2010).

GUERON, S., GERZON, G., ANATI, I., DOWECK, J., MAOR, M., and CHO, L. A tweakable encryption mode for memory encryption with protection against replay attacks. WO patent number 2012040679. (29 March 2012).

GUERON, S., SAVAGAONKAR, U., MCKEEN, F., ROZAS, C., DURHAM, D., DOWECK, J., MULLA, O., ANATI, I., GREENFIELD, Z., and MAOR, M. Method and apparatus for memory encryption with integrity check and protection against replay attacks. WO patent number 2013002789. (3 January 2013).

GUTIERREZ, A., DRESLINSKI, G., WENISCH, T., AND MUDGE, T. Full-System Analysis and Characterization of Interactive Smartphone Applications. In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC). (2011)

GUTMANN, P. An open-source cryptographic coprocessor. In *Proceedings of the 2000 USENIX Security Symposium*. (2000).

HALDERMAN, J., SCHOEN, S., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J., FELDMAN, A., APPELBAUM, J., and FELTEN, E. Lest we remember: cold boot attacks on encryption keys. In *Proceedings of the USENIX Security Symposium*. (February 2008).

HAYES, D., and QURESHI, S. Implications of Microsoft vista operating system for computer forensics investigations. In *Proceedings of the IEEE Systems, Applications and Technology Conference*. (May 2009), 1-9.

HENNESSY, J., and PATTERSON, D. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, (2006).

HENSON, M., and TAYLOR, S. Beyond full disk encryption: protection on security enhanced commodity processors. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security (ACNS '13)*. (June 2013).

HENSON, M., and TAYLOR, S. Attack mitigation through memory encryption of security enhanced commodity processors. In *Proceedings of the 8th International Conference on Information Warfare and Security (ICIW '13)*, Hart, D. (eds.) pp. 265-268. (March 2013).

HENSON, M. and TAYLOR, S. Memory Encryption: a survey of existing techniques. ACM Computing Surveys 46, 4, Article 53. (March 2014).

HIZVER, J. and CHIUEH, T. An introspection-based memory scraper attack against virtualized point of sale systems. Danezis, G., Dietrich, S., and Sako, K. (Eds.): FC 2011 Workshops, LNCS 7126, pp. 55-69. (2012)

HOGLUND, G. and BUTLER, J. Rootkits. Addison-Wesley Professional Press. (2005).

HOISIE, A. Performance Optimization of Numerically Intensive Codes (Software, Environments and Tools) (19 March 2001).

HONG, D., BATTEN, L., LIM, S., and DUTT, N. DynaPoMP: dynamic policy-driven memory protection for SPM-based embedded systems. In *Proceedings of the Workshop on Embedded Systems Security*. (2011).

HOWGRAVE-GRAHAM, N., DYER, J., and GENNARO, R. Pseudo-random number generation on the IBM 4758 secure crypto coprocessor. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001)*, LNCS 2162, Springer-Verlag, pp. 93-102. (2001).

HUNT, G., LARUS, J., ABADI, M., AIKEN, M., BARHAM, P., FAHNDRICH, M., HAWBLITZEL, C., HODSON, O., LEVI, S., MURPHY, N., STEENSGAARD, B., TARDITI, D., WOBBER, T., and ZILL, B. An overview of the singularity project. *Microsoft Research Technical Report MSR-TR-2005-135*. (2005).

I.MX53 Multimedia Applications Processor Reference Manual. Available online at http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=IMX53QSB&fpsp=1&tab=Documentation_Tab

JANNEPALLY, V., and SOHONI, S. Fast encryption and authentication for cache-to-cache transfers using GCM-AES. In *Proceedings of the International Conference on Sensors, Security, Software and Intelligent Systems*. (January 2009).

KAPLAN, B. RAM is key: extracting disk encryption keys from volatile memory. Master's thesis report. Carnegie Mellon University. (May 2007).

KARGER, P., and SCHELL, R. Multics security evaluation: vulnerability analysis. ESD-TR-74-193, Vol. II, Hanscom AFB, MA. (1974).

KARLOF, C., SASTRY, N., and WAGNER, D. TinySec: a link layer security architecture for wireless sensor networks. In *Proceedings of the Sensys'04*. (2004).

KAUER, B. OSLO: improving the security of trusted computing. In *Proceedings of the USENIX Security Symposium*. (2007).

KC, G., KEROMYTIS, D., and PREVELAKIS, V. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pp 272-280. (2003)

KENNEDY, D., O'GORMAN, J., KEARNS, D. and AHARONI, M. Metasploit: The penetration testers guide. No Starch Press. (2011).

KENT, S. Protecting externally supplied software in small computers. PhD thesis, *MIT Laboratory for Computer Science, MIT-LCS-TR-255*. (March 1981).

KGIL, T., FALK, L., and MUDGE, T. ChipLock: support for secure microarchitectures. *ACM Sigarch*, 33, 1, (March 2005).

KOCHER, P., JAFFE, J., and JUN, B. Differential power analysis. In *Proceedings of the CRYPTO 19th Annual International Cryptology Conference*. 1666, (1999), 388-397.

KUHN, M. Cipher instruction search attack on the bus-encryption security microcontroller DS5002FP. In *IEEE Transactions on Computing*. 47, (October 1998), 1153-2257.

LEE, M., AHN, M., and KIM, E. I2SEMS: interconnects-independent security enhances shared memory multiprocessor systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. (2007).

LIE, D., THEKKATH, C., MITCHELL, M., LINCOLN, P., BONEH, D., MITCHELL, J., and HOROWITZ, M. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS*). (2000), 168-177.

LIPMAN, H., ROGAWAY, P., and WAGNER, D. Comments to NIST concerning AES modes of operations:ctr-mode encryption. (2000).

LOBO, D., WATTERS, P., WU, X., and SUN, L. Windows rootkits: attacks and countermeasures. In *Proceedings of the 2nd Cybercrime and Trustworthy Computing Workshop*. (2010).

MARTIN, L. XTS: A mode of AES for encrypting hard disks. In *Security & Privacy, IEEE* , vol.8, no.3, May-June (2010), 68-69.

MCCLEAN, M., and MOORE, J. Securing FPGAs for red/black systems FPGA-based single chip cryptographic solution. In the *Journal of Military Embedded Systems*. (2007).

MCGILL, K., and TAYLOR, S. Application resilience with process failures. In *Proceedings of the International Conference on Security and Management.* (July 2011).

MEL, H., and BAKER, D. *Cryptography Decrypted*. Addison-Wesley. Upper Saddle River, (2001).

MULLER, T., FREILING, F., and DEWALD, A. TRESOR runs encryption securely outside RAM. In *Proceedings of the 20th USENIX Conference on Security*. (2011).

NAGARAJAN, V., GUPTA, R., and KRISHNASWAMY, A. Compiler-assisted memory encryption for embedded processors. In *HiPPEAC*. (2007), 7-22.

OSVIK, D., SHAMIR, A., and TROMER, E. Cache attacks and countermeasures: the case of AES.

PETERSON, P. Cryptkeeper: improving security with encrypted RAM. In *Proceedings of the IEEE International Conference on Technologies for Homeland Security (HST)*. (November 2010), 120-126.

PLATTE, J., DIAZ, R., and NAROSKA, E. A new encryption and hashing scheme for the security architecture for microprocessors. In *Communications and Multimedia Security*. 4237, (October 2006), 120-129.

POLONSKY, S., KNEBEL, D., SANDA, P., MCMANUS, M., HUOTT, W., PELELLA, A., MANZER, D., STEEN, S., WILSON, S., and CHAN, Y. Non-invasive timing analysis of IBM G6 microprocessor L1 cache using backside time-resolved hot electron luminescence. In *Proceedings of the IEEE International Solid-State Circuits Conference*. (2000), 222-224.

PORTOKALIDIS, G., and KEROMYTIS, D. Fast and practical instruction-set randomization for commodity systems. In *Proceedings of the Annual Computer Security Applications Conference*. (2010).

PROVOS, N. Encrypting virtual memory. In *Proceedings of the 9th USENIX Security Symposium*. (2000).

PYSERIAL. Available online at http://pyserial.sourceforge.net/index.html

RABAIOTTI, J., and HARGREAVES, C. Using a software exploit to image RAM on an embedded system. *Digital Investigation*. (February 2010).

RAMACHANDRAN, Z., and HUANG, D. Computing cryptographic algorithms in portable and embedded devices. *IEEE Portable*. (May 2007), 1-7.

RAVI, A., RAGHUNATHAN, A., and CHAKRADHAR, S. Tamper resistance mechanisms for secure embedded systems. *IEEE Intl. Conf. on VLSI Design*. (January 2004).

ROGERS, B., SOLIHIN, Y., and PRVULOVIC, M. Memory predecryption: hiding the latency overhead of memory encryption. in *ACM SIGARCH Computer Architecture News*, 33, 1, (March 2005), 27-33.

ROGERS, B., CHHABRA, S., SOLIHIN, Y., and PRVULOVIC, M. Using address independent seed encryption and bonsai merkle trees to make secure processors OS and performance friendly. In *Proceedings of the 40th International Symposium on Microarchitecture, IEEE Computer Society*. (2007), 183-196.

ROGERS, B., PRVULOVIC, M., and SOLIHIN, Y. Efficient data protection for distributed shared memory multiprocessors. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT).* (September 2006).

ROGERS, B., CHENYU, Y., CHHABRA, S., PRVULOVIC, M., and SOLIHIN, Y. Single level integrity and confidentiality protection for distributed shared memory multiprocessors. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture*. (2008), 161-172.

ROMANOSKY, S., TELANG, R., and ACQUISTI, A. Do data breach disclosure laws reduce identify theft. Carnegie Mellon Technical Report. (2008)

SHI, W., LEE, H., GHOSH, M., and LU, C. Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT)*. (2004).

SIMMONS, P. Security through amnesia: a software-based solution to the cold boot attack on disk encryption. In *Proceedings of the 27th Annual Computer Security Applications Conference*. (December 2011).

O. W. R. M. J. Marchesini, S.W. Smith. Experimenting with tcpa/tcg hardware, or: How I learned to stop worrying and love the bear. Technical Report TR2003-476, Computer Science Technical Report, Dartmouth College, (December 2003).

SKILLEN, A., BARRERA, D., and OORSCHOT, P. Deadbolt: locking down Android disk encryption. In the 3rd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices. (Nov 2013).

SMITH, S. Magic boxes and boots: security in hardware. In *IEEE Computer Software*. (October 2004), 106-109.

STEIL, M. 17 mistakes Microsoft made in the xbox security system. In *Proceedings of the 22nd Chaos Communication Congress*. (2005).

STEIL, M., and DOMKE, F. The Xbox 360 Security System and its Weaknesses. Google TechTalk available at http://www.youtube.com/watch?v=uxjpmc8ZIxM August (2008)

SU, L., COURCAMBICK, S., GUILLEMIN, P., SCHWARZ, C., and PASCALET, R. SecBus: operating system controlled hierarchical page-based memory bus protection. *EDAA*. (2009).

SU, L., MARTINEZ, A., GUILLEMIN, P., CERDAN, S., PACALET, R. Hardware mechanism and performance evaluation of hierarchical page-based memory bus protection. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. (2009).

SUH, G., CLARKE, D., GASSEND, B., DIJK, M., and DEVADAS, S. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th International Conference on Supercomputing*. (June 2003).

SUH, G., CLARKE, D., GASSEND, B., DIJK, M., and DEVADAS, S. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of the 36th International Symposium on Microarchitecture*. (2005).

SUH, G., O'DONELL, C., and DEVADAS, S. Aegis: a single-chip secure processor. In *IEEE Design and Test of Computers*. (2007).

TAYLOR, S., HENSON, M., KANTER, M., KUHN, S., MCGILL, K., and NICHOLS, C. Bear-a resilient operating system for scalable multi-processors. Dartmouth College, Thayer School of Engineering Technical Report TR 11-005. (2011).

THOMAS, R., and YELICK, K. Efficient FFTs on IRAM. In Proceedings of the First Workshop on Media Processors and DSPs. (November 1999).

VANDANA, G. Exploring trusted platform module capabilities: a theoretical experimental study. Doctor of Philosophy in Computer Science Dissertation. (May 2008).

VASUDEVAN, A., OWUSU, E., ZHOU, Z., NEWSOME, J. and MCCUNE, J. Trustworthy execution on mobile devices: what security properties can my mobile

platform give me? Carnegie Mellon University CyLab Technical Report 11-023. November (2011)

WOLLINGER, T., GUAJARDO, J., and PAAR, C. Cryptography in embedded systems: an overvies. In *Proceedings of the Embedded World 2003 Conference*. (February 2003), 735-744.

YAN, C., ROGERS, B., ENGLENDER, D., SOLIHIN, Y., and PRVULOVIC, M. Improving cost performance and security of memory encryption and authentication. In *Proceedings of the 33rd International Symposium on Computer Architecture*. (June 2006).

YANG, J., GAO, L., and ZHANG, Y. Improving memory encryption performance in secure processors. In *IEEE Transactions on Computing*. (May 2005).

ZHANG, Y., GAO, L., YANG, J., ZHANG, X., and GUPTA, R. SENSS: security enhancement to symmetric shared memory multiprocessors. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. (February 2005).

ZHOU, Y., WANG, Z., ZHOU, W. and JIANG, X. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets," in *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*. (2012).

ZHUANG, X., ZHANG, T., and PANDE, S. Hide: an infrastructure for efficiently protectiong information leakage on the address bus. In *Proceedings of the 11th*

International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). (October 2004). 72-84.

Appendix 1: Python Code for Board Initialization and Bootloader Bypass

```
#/*****************************************************************************
#*
#*
#*          Michael Henson - Python Push
#*          This script is a modification of PYSERIAL serial interface tool.
#*          http://pyserial.sourceforge.net
#*          This script will initialize memory (and other peripherals if desired)
#*          and then load an image file to the location specified in memory jumping
#*          to the execution point specified in the image vector table (IVT).
#*
#*          usage:  python SERV.py write_file 0x70000000 iMXBlinky.img
#*          where 0x70000000 is the starting location in memory to place the image
#*          and iMXBlinky.img is the name of the image to load...could use *.img
#*          Placing the command in the post-processing section of the makefile after
#*          the image has been built will automate push
#*          NOTE: Must swap the IVT_BASE and BOOT_DATA_BASE locations in the LD file
#*          in order for the image to be in the proper format for loading to memory
#*****************************************************************************/

import serial
import time
import os.path
import sys

# resp_var[4:] extract from element 4 up to the end of the list
# resp_var[4:6] extract elements 5 to 6
# resp_var[:-5:-1] extract the last four elements and parse starting from the end.


# function to send the command on port COM
# default response size is 4 bytes if none are specified
def runcmd(cmd,responselength=4):

        cmds = cmd.split()
        if len(cmds) != 16:
                print "Command format is incorrect!"
                return 'error'
        cmdstring = ''
        for cmd in cmds:
                cmdstring += chr(int(cmd,16))

        uart_port.write(cmdstring)
#       time.sleep(1)
        resp_var = uart_port.read(responselength)
        return resp_var

# function to retrieve and print the response on the COM port
def showanswer(resp_var):
        # print as a 32-bit word
        if len(resp_var)==4:
                # resp_var[::-1] parsed the list starting from the end
                for c in resp_var[::-1]:
                        print "%02X" % ord(c),
                print
        else:
        # print in byte view
                print "Byte view:"
                for c in resp_var:
                        print "%02X" % ord(c),
```

```
                print

# function to write the received data to a file
def writetofile(resp_var, o_file):
        # print as a 32-bit word
        if len(resp_var)==4:
                # resp_var[::-1] parsed the list starting from the end
                for c in resp_var[::-1]:
                        print >> o_file,"%02X" % ord(c),
                print
        else:
        # print in byte view
                for c in resp_var:
                        print >> o_file,"%02X" % ord(c),"\n",
                print

# function to read data from serial port
def get_serial_data(length):
        # to do a loop with length larger than 4
        resp_var = uart_port.read(length)
        return resp_var

# function to get the file size formatted to be sent by the serial port
def get_formated_hex(var):
        # create a list of the split hex version of the 32-bit integer
        # if integer is 0x12345678 => ['1','2','3','4','5','6','7','8']
        if type(var) == long:
                hex_list = list("%08X" % var)
        if type(var) == int:
                hex_list = list("%08X" % var)
        if type(var) == str:
                # in case the address is not 4bytes long => add '0's
                var_size = len(var)
                if var_size!=8:
                        for i in range(8-var_size):
                                var = "".join(['0',var])
                hex_list = list(var)
        # create bytes from 2 elements
        byte1 = "".join(hex_list[0:2]) # ['12']
        byte2 = "".join(hex_list[2:4]) # ['34']
        byte3 = "".join(hex_list[4:6]) # ['56']
        byte4 = "".join(hex_list[6:8]) # ['78']
        hex_fmt = " ".join([byte1,byte2,byte3,byte4]) #  # ['12 34 56 78']
        return hex_fmt

#### list of commands ####
#Reference i.MX53 Multimedia Applications Processor Reference Manual section 7.8
GET_STATUS = '05 05'
READ_MEMORY = '01 01'
WRITE_MEMORY = '02 02'
WRITE_FILE = '04 04'
UNK = '06 06'

#### acknowledge ####
ACK_PROD = "".join([chr(18),'4','4',chr(18)]) # <=> '12 34 34 12'
ACK_ENG = 'VxxV' # <=> '56 78 78 56'

#### data size ####
WORD_SIZE = '20'
HWORD_SIZE = '10'
BYTE_SIZE = '08'
```

```
#### file type ####
DCD_TYPE = 'EE'
CSF_TYPE = 'CC'
APPS_TYPE = 'AA'

############ main ###############
def main():
	global uart_port
	global tx_data

	# if nothing is specified, display usage message !
	if len(sys.argv) == 1:
		print "usage error"

	# Start the program
	if os.path.exists("/dev/ttyUSB0"):
		# for Cygwin or Linux Python
		uart_port = serial.Serial('/dev/ttyUSB0', 115200, timeout=2)
	else:	# for Windows Python console
		uart_port = serial.Serial('COM1', 115200, timeout=2)

	no_valid_arg = 0
	#### get status command ####
	cmd_to_send = " ".join([GET_STATUS,'00 00 00 00 00 00 00 00 00 00 00 00 00 00'])
	answer = runcmd(cmd_to_send)
	print 'Status is:'
	showanswer(answer)

	#### initialize memory for QSB DDR3 ####

	access_size = WORD_SIZE

	cmd_to_send = " ".join([WRITE_MEMORY,'53 fd 40 68',access_size,'00 00 00 00','ff ff ff ff','00'])
	answer = runcmd(cmd_to_send)

	cmd_to_send = " ".join([WRITE_MEMORY,'53 fd 40 6c',access_size,'00 00 00 00','ff ff ff ff','00'])
	answer = runcmd(cmd_to_send)

	cmd_to_send = " ".join([WRITE_MEMORY,'53 fd 40 70',access_size,'00 00 00 00','ff ff ff ff','00'])
	answer = runcmd(cmd_to_send)

	cmd_to_send = " ".join([WRITE_MEMORY,'53 fd 40 74',access_size,'00 00 00 00','ff ff ff ff','00'])
	answer = runcmd(cmd_to_send)

	cmd_to_send = " ".join([WRITE_MEMORY,'53 fd 40 78',access_size,'00 00 00 00','ff ff ff ff','00'])
	answer = runcmd(cmd_to_send)

	cmd_to_send = " ".join([WRITE_MEMORY,'53 fd 40 7c',access_size,'00 00 00 00','ff ff ff ff','00'])
	answer = runcmd(cmd_to_send)

	cmd_to_send = " ".join([WRITE_MEMORY,'53 fd 40 80',access_size,'00 00 00 00','ff ff ff ff','00'])
	answer = runcmd(cmd_to_send)

	cmd_to_send = " ".join([WRITE_MEMORY,'53 fd 40 84',access_size,'00 00 00 00','ff ff ff ff','00'])
	answer = runcmd(cmd_to_send)

	cmd_to_send = '02 02 53 fa 86 f4 20 00 00 00 00 00 00 00 00 00'
	answer = runcmd(cmd_to_send)
	cmd_to_send = '02 02 53 fa 87 14 20 00 00 00 00 00 00 00 00 00'
	answer = runcmd(cmd_to_send)
	cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 86 fc',access_size,'00 00 00 00','00 00 00 00','00'])
	answer = runcmd(cmd_to_send)
```

```
cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 87 24',access_size,'00 00 00 00','04 00 00 00','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 87 2c',access_size,'00 00 00 00','00 30 00 00','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 85 54',access_size,'00 00 00 00','00 30 00 00','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 85 58',access_size,'00 00 00 00','00 30 00 40','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 87 28',access_size,'00 00 00 00','00 30 00 00','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 85 60',access_size,'00 00 00 00','00 30 00 00','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 85 68',access_size,'00 00 00 00','00 30 00 40','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 87 1c',access_size,'00 00 00 00','00 30 00 00','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 85 94',access_size,'00 00 00 00','00 30 00 00','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 85 90',access_size,'00 00 00 00','00 30 00 40','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 87 18',access_size,'00 00 00 00','00 30 00 00','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 85 84',access_size,'00 00 00 00','00 30 00 00','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 85 7c',access_size,'00 00 00 00','00 30 00 40','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 85 78',access_size,'00 00 00 00','00 30 00 00','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 85 70',access_size,'00 00 00 00','00 30 00 00','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 85 74',access_size,'00 00 00 00','00 30 00 00','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 85 88',access_size,'00 00 00 00','00 30 00 00','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 86 f0',access_size,'00 00 00 00','00 30 00 00','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 87 20',access_size,'00 00 00 00','00 30 00 00','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 85 64',access_size,'00 00 00 00','00 30 00 40','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'53 fa 85 80',access_size,'00 00 00 00','00 30 00 40','00'])
```

```
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 88',access_size,'00 00 00 00','32 38 35 35','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 90',access_size,'00 00 00 00','40 38 35 38','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 7c',access_size,'00 00 00 00','01 36 01 4d','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 80',access_size,'00 00 00 00','01 51 01 41','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 18',access_size,'00 00 00 00','00 01 17 40','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 00',access_size,'00 00 00 00','c3 19 00 00','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 0c',access_size,'00 00 00 00','55 59 52 e3','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 10',access_size,'00 00 00 00','b6 8e 8b 63','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 14',access_size,'00 00 00 00','01 ff 00 db','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 2c',access_size,'00 00 00 00','00 00 26 d2','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 30',access_size,'00 00 00 00','00 9f 0e 21','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 08',access_size,'00 00 00 00','12 27 30 30','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 04',access_size,'00 00 00 00','00 02 00 2d','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 1c',access_size,'00 00 00 00','00 00 80 32','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 1c',access_size,'00 00 00 00','00 00 80 33','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 1c',access_size,'00 00 00 00','00 02 80 31','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 1c',access_size,'00 00 00 00','09 20 80 b0','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 1c',access_size,'00 00 00 00','04 00 80 40','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 1c',access_size,'00 00 00 00','00 00 80 3a','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 1c',access_size,'00 00 00 00','00 00 80 3b','00'])
answer = runcmd(cmd_to_send)
```

```
cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 1c',access_size,'00 00 00 00','00 02 80 39','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 1c',access_size,'00 00 00 00','09 20 81 38','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 1c',access_size,'00 00 00 00','04 00 80 48','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 20',access_size,'00 00 00 00','00 00 18 00','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 40',access_size,'00 00 00 00','04 b8 00 03','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 58',access_size,'00 00 00 00','00 02 22 27','00'])
answer = runcmd(cmd_to_send)

cmd_to_send = " ".join([WRITE_MEMORY,'63 fd 90 1c',access_size,'00 00 00 00','00 00 00 00','00'])
answer = runcmd(cmd_to_send)

if answer == ACK_ENG or answer == ACK_PROD:
        print "Write was done."

#### get status command--must do after each command ####
cmd_to_send = " ".join([GET_STATUS,'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'])
answer = runcmd(cmd_to_send)
print 'Status is:'
showanswer(answer)

#### write img to i.MX53 external memory ####
infile = open(sys.argv[3],'rb')
        # get file size with 2 methods
#       print "size: %d" % os.stat(sys.argv[3]).st_size
#       print "size: %d" % os.path.getsize(sys.argv[3])
f_size_int = os.path.getsize(sys.argv[3])
f_size_hex = get_formated_hex(f_size_int)

        # provide an address like 0x12784596, and skip '0x' in the string chain
mem_add = get_formated_hex(sys.argv[2][2:10])

cmd_to_send = " ".join([WRITE_FILE,mem_add,'00',f_size_hex,'00 00 00 00',APPS_TYPE])
answer = runcmd(cmd_to_send)
if answer == ACK_ENG or answer == ACK_PROD:
        while True:
                tx_data = infile.read(f_size_int)
                if not tx_data:
                        break
                uart_port.write(tx_data)
else:
        print "No acknowledge => can't transfer file"

        infile.close()

#### get status command--must do after each command ####
cmd_to_send = " ".join([GET_STATUS,'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'])
answer = runcmd(cmd_to_send)
print 'Status is:'
showanswer(answer)
```

```
		#uart_port.close()

		# because sys.argv[1] is not valid, help message is displayed
		if no_valid_arg == 1:
			print "improper usage"

if __name__ == '__main__':
		main()
# End
```

Appendix 2: IMX53 Specific Linker Script (IMX53.ld)

```
/*
 * Linker script for the iMX53
 */

/*
 * Generate little-endian formatted binary image
 */
OUTPUT_FORMAT ("elf32-littlearm")

ENTRY(_init)

MEMORY
{
  RAM                : ORIGIN = 0xf8002000, LENGTH = 119K
  eRAM               : ORIGIN = 0x72000000, LENGTH = 990M
}

EXTERN(__stack_size)
ASSERT(__stack_size, "Must provide a non-zero stack size");
ASSERT(!(__stack_size & 0x7), "Stack not aligned on 128-bit boundary");


irq_stack_top = _text + LENGTH(RAM);            /* IRQ Mode Stack */
irq_stack_bottom = irq_stack_top - __stack_size;
fiq_stack_top = irq_stack_bottom;               /* FIQ Mode Stack */
fiq_stack_bottom = fiq_stack_top - __stack_size;
svc_stack_top = fiq_stack_bottom;               /* SVC Mode Stack */
svc_stack_bottom = svc_stack_top - __stack_size;
abt_stack_top = svc_stack_bottom;               /* ABT Mode Stack */
abt_stack_bottom = abt_stack_top - __stack_size;
und_stack_top = abt_stack_bottom;               /* UND Mode Stack */
und_stack_bottom = abt_stack_top - __stack_size;
sys_stack_top = und_stack_bottom;               /* SYS Mode Stack */
sys_stack_bottom = sys_stack_top - __stack_size;
/* Heap is all memory between code and stacks */
_heap_end = sys_stack_bottom;
_heap_start = _end;
BOOTIMAGE_SIZE = SIZEOF( .text ) + SIZEOF( .data );

SECTIONS
{

  .text : {
                                  Procs.o (.text);

            } >eRAM

  .text1 :
  {
    _text = .;
    _stext = .;
    BOOTIMAGE_BASE = .;
    *(.bootimage)

    IVT_BASE = .;
            *(.IVT)

    BOOTDATA_BASE = .;
    *(.BOOTDATA)
```

```
 DCD_BASE = .;
 *(.DCD)
 *(.text .text.*)        /* Code */
 /**(.rodata .rodata.*)*/     /* Constants, strings, ... */
 *(.gnu.linkonce.t.*)
 *(.glue_7)              /* Glue ARM to thumb code */
 *(.glue_7t)             /* Glue thumb to ARM code */
 *(.gcc_except_table)
 *(.gnu.linkonce.r.*)
 . = ALIGN(4);
 _etext = .;
 _sidata = _etext;       /* Start of data stored in flash */
 _fini = .;
 *(.fini)
} >RAM

.data : AT (_etext)
{
  _data = .;
  _sdata = .;             /* Used for copying data on startup */
  *(.ramfunc .ramfunc.* .fastrun .fastrun.*)
  *(.data .data.*)
  *(.rodata .rodata.*)
  *(.gnu.linkonce.d.*)
  . = ALIGN(4);
  _edata = .;
} >RAM

 .ARM.extab :
{
  *(.ARM.extab*)
} >RAM

__exidx_start = .;
.ARM.exidx :
{
  *(.ARM.exidx*)
} >RAM
__exidx_end = .;

.bss (NOLOAD) :
{
  . = ALIGN(4);
  _sbss = .;              /* Used for zeroing bss on startup */
  *(.bss .bss.*)
  *(.gnu.linkonce.b.*)
  *(COMMON)
  . = ALIGN(4);
  _ebss = .;
} >RAM



_end = .;

.stab 0 (NOLOAD) : { *(.stab) }
.stabstr 0 (NOLOAD) : { *(.stabstr) }
/* DWARF debug sections.
 * Symbols in the DWARF debugging sections are relative to
 * the beginning of the section so we begin them at 0.
```

```
  */
 /* DWARF 1 */
 .debug          0 : { *(.debug) }
 .line           0 : { *(.line) }
 /* GNU DWARF 1 extensions */
 .debug_srcinfo  0 : { *(.debug_srcinfo) }
 .debug_sfnames  0 : { *(.debug_sfnames) }
 /* DWARF 1.1 and DWARF 2 */
 .debug_aranges  0 : { *(.debug_aranges) }
 .debug_pubnames 0 : { *(.debug_pubnames) }
 /* DWARF 2 */
 .debug_info     0 : { *(.debug_info .gnu.linkonce.wi.*) }
 .debug_abbrev   0 : { *(.debug_abbrev) }
 .debug_line     0 : { *(.debug_line) }
 .debug_frame    0 : { *(.debug_frame) }
 .debug_str      0 : { *(.debug_str) }
 .debug_loc      0 : { *(.debug_loc) }
 .debug_macinfo  0 : { *(.debug_macinfo) }
 /* DWARF 2.1 */
 .debug_ranges   0 : { *(.debug_ranges) }
 /* SGI/MIPS DWARF 2 extensions */
 .debug_weaknames 0 : { *(.debug_weaknames) }
 .debug_funcnames 0 : { *(.debug_funcnames) }
 .debug_typenames 0 : { *(.debug_typenames) }
 .debug_varnames  0 : { *(.debug_varnames) }

 .note.gnu.arm.ident 0 : { KEEP (*(.note.gnu.arm.ident)) }
 .ARM.attributes 0 : { KEEP (*(.ARM.attributes)) }
 /DISCARD/ : { *(.note.GNU-stack) }
}
```