The Thesis Committee for Donald Edward Owen Jr.
certifies that this is the approved version of the following thesis:

The Feasibility of Memory Encryption and Authentication

APPROVED BY

SUPERVISING COMMITTEE:

Lizy K. John, Supervisor

Andreas Gerstlauer

The Feasibility of Memory Encryption and Authentication

by

Donald Edward Owen Jr., B.S.E.E.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2013

The Feasibility of Memory Encryption and Authentication

Donald Edward Owen Jr., M.S.E.
The University of Texas at Austin, 2013

Supervisor: Lizy K. John

This thesis presents an analysis of the implementation feasibility of RAM authentication and encryption. Past research has used simulations to establish that it is possible to authenticate and encrypt the contents of RAM with reasonable performance penalties by using clever implementations of tree data structures over the contents of RAM. However, previous work has largely bypassed implementation issues such as power consumption and silicon area required to implement the proposed schemes, leaving implementation details unspecified. This thesis studies the implementation cost of AES-GCM hardware and software solutions for memory authentication and encryption and shows that software solutions are infeasible because they are too costly in terms of performance and power, whereas hardware solutions are more feasible.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the advent of nearly ubiquitous, portable, and cheap computing, sensitive data is increasingly put on mobile devices and remote servers. Examples of such data include health records, privileged discussions with lawyers or caretakers, Social Security numbers and other identifying information, and various forms of cryptographic keys. Users concerned about security typically employ several protective methods to ensure the security of their data such as passwords, disk encryption, and hardened software systems that have been tested against attacks. However, for a motivated or ambitious attacker with physical access to the system in question, these protections are frequently not enough. An integral part of most modern computer systems is left largely unprotected: Random Access Memory (RAM).

The "cold boot" attack has been shown to present serious troubles to the security of data that is stored in the Dynamic RAM (DRAM) chips of a modern computer[1]. In short, a cold boot attack exploits DRAM remanence by first obtaining physical control of a computer while it is powered on, then briefly interrupting power, rebooting with a custom operating system, and dumping the still-remnant contents of DRAM out to permanent storage.

Because power is only interrupted for a short period of time, the contents of the DRAM chips are largely intact and can be mined offline for exploitative material such as encryption keys. Variations on this attack include chilling the DRAM modules before performing the attack (thereby slowing decay) and physically transplanting the DRAM modules into an attacker-controlled system, bypassing the need to reboot the attacked system into a new OS.

It should be noted that most cold boot attacks take place offline. Data is read out of memory once and then post-processed for valuable information, such as cipher keys.

Elbaz et al. [2], in a survey covering various hardware mechanisms for memory authentication, also succinctly define active attacks based on spoofing, splicing, and replay. These attacks are illustrated in Figure 1.1, adapted from the same paper. In short, spoofing effectively replaces the contents of memory at a specific address, splicing transposes memory contents from one address to another, and replay changes the contents of an address to contents that were previously observed at the same address. These attacks can be done online and can enable attackers to influence program execution or reveal data, even if the bus traffic is encrypted [3].

Other data leaks are possible, such as passive bus snooping, and can be done while the system under attack is running. The original Microsoft Xbox$^{\text{TM}}$was compromised by the use of a cheap bus snooper that read a cipher key sent in the clear between the processor and a peripheral, with no cold boot attack needed [4]. Once this key was known, it was straightforward to

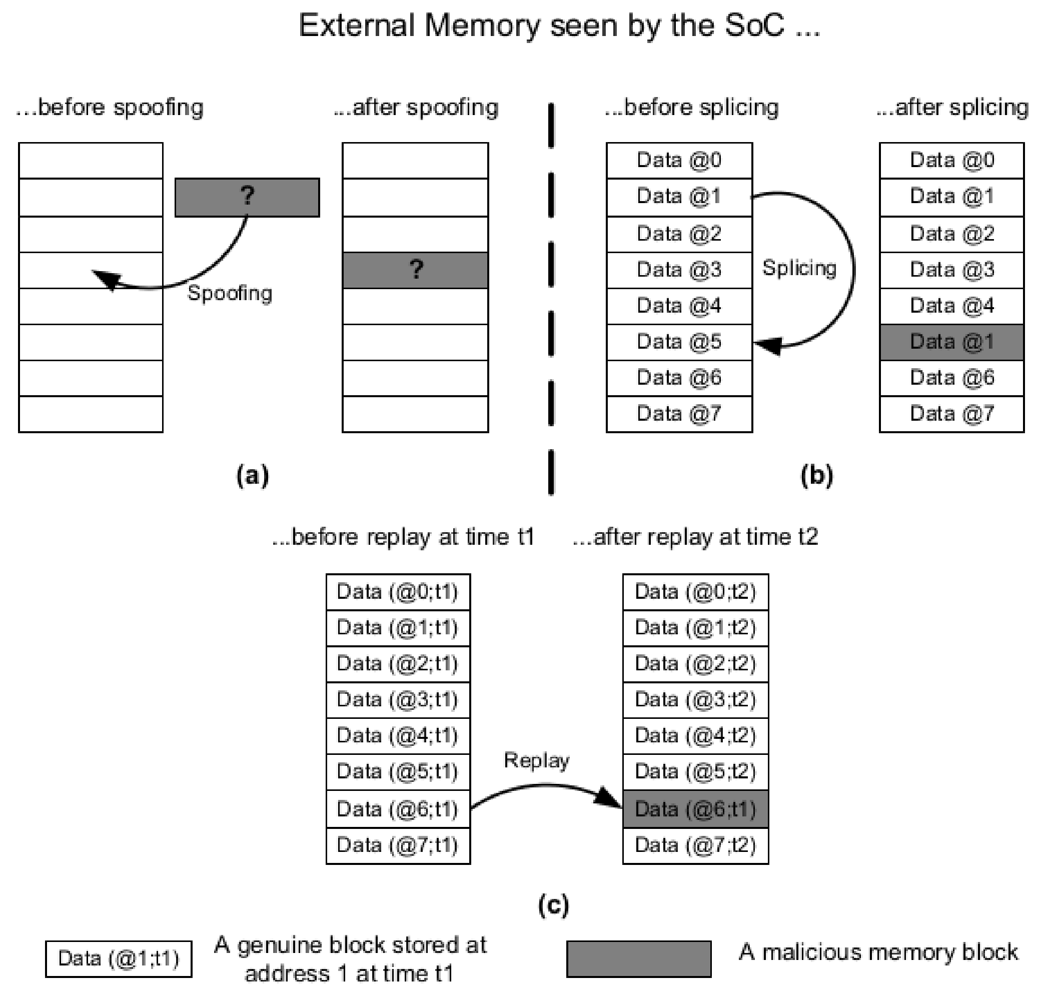


Figure 1.1: Spoofing (a), Splicing (b), Replay (c) Attacks on Memory (Adapted from Elbaz et al. [2])

repurpose the system to run almost anything that could be compiled for it, from new operating systems to cracked, stolen, or pirated games.

A solution to both of these types of attacks is to 1) encrypt and tag all data that is sent to RAM and 2) authenticate and decrypt all data that is read from RAM. Several methods for doing this are described in the survey paper by Elbaz et al. [2]. These methods typically involve using a symmetric key cipher to encrypt data and also building a tree structure over the contents of RAM such that any alterations to the contents of RAM will be detected upon reading the data back into the processor. More details will be discussed in Chapter 2.

The objective of this thesis to provide an analysis of the feasibility of implementation of memory encryption and authentication. This will be accomplished by describing previous work within the field in conjunction with a characterization of the implementation of necessary primitive operations to provide the functionality of encryption and authentication, specifically using the Advanced Encryption Standard (AES) cipher operated in Galois Counter Mode. We will describe multiple software, FPGA, and ASIC implementations of the necessary primitives with a comparison between the different methods in terms of implementation cost, power consumption, area cost, and feasibility of design. This thesis does not provide a new method or process for solving the memory encryption and authentication problem, but instead seeks to provide more perspective on the implementation concerns of previous work.

This thesis is organized as follows. Chapter 2 discusses previous work that has been done to prevent data compromise in the realms of both software and hardware solutions. Chapter 3 covers the basics of the AES cipher and modes of operation, specifically Galois Counter Mode. Chapter 4 covers a software implementation and performance analysis of the necessary pieces for encryption, decryption, and authentication based on a modified MiBench Rijndael codebase. Chapter 5 details the operations of an open-source hardware module capable of performing the necessary encryption, decryption, and authentication procedures. In Chapter 6, a comprehensive analysis of the performance, power consumption, and area consumption of all of the discussed methods is presented. Chapter 7 lists several ways in which this work could be extended and Chapter 8 summarizes the findings of this work.

Chapter 2

Related Work

Various research groups have performed detailed analysis of the performance impact of memory authentication and encryption and have devoted a large amount of effort to reducing the performance penalty that is traded off for security.

2.1 Tree-based Schemes for Memory Protection

One of the early models for protecting the contents of memory was to build a Merkle Tree over the contents of RAM [5]. A Merkle Tree for memory authentication uses a hash function applied to blocks of memory (typically cache lines) to verify that the contents of a piece of RAM have not been modified. Hashes of leaf nodes (cache lines) are then combined into blocks and hashed again, with the process repeated until one final root hash is produced. This root hash is kept on-chip at all times, while other levels of the tree are able to be sent out to RAM and stored, with the assumption that any changes to either data or hash values will produce a root hash that will not match the one stored on the chip. The Merkle Tree can authenticate blocks in parallel because all levels of the tree are available at authentication time but must update the levels of the tree sequentially whenever a leaf node is changed because of the nature of the hashing function.

To address the problems of the Merkle Tree with sequential updates, the Parallelizable Authentication Tree (PAT) was introduced by Hall et al. [6]. This authentication tree functions similarly to the Merkle Tree but is designed in such a way that both authentications and updates may be done on all levels of the tree in parallel. Nonces are used as part of generating a Message Authentication Code (MAC) and then those nonce values form the next level of the tree. This procedure is repeated up to the root. Because these nonce values may be generated at any time, this scheme allows for parallel authentication and updates, offering a significant performance improvement over the Merkle Tree. In the best case, an authentication or update may be performed with a latency of one operation, whereas a Merkle Tree would have n sequential operations where n is the number of levels in the tree.

The Tamper-Evident Counter Tree (TEC-Tree) uses a primitive known as Block-level AREA which combines both encryption and authentication into one operation [7]. A nonce is generated and concatenated onto every data block (cache line) and the resulting block is then encrypted with a symmetric key cipher operating in Electronic Code Book Mode (ECB). The nonces are then combined into new data blocks and the procedure is repeated up to the root, forming a tree structure. The final nonce value is held on chip in secure storage. This tree is also able to process authentication and updates in parallel, similarly to the PAT. An added advantage of the TEC-Tree implementation is that the encryption scheme used provides confidentiality for all traffic to and from memory by encrypting it before sending to external memory, something not provided in the Merkle Tree or PAT.

Other schemes have been devised that meld facets of multiple of the previously described schemes. The Bonsai Merkle Tree uses a MAC function over data blocks and then stores counter values separately in memory. Those counter values are then put together into new blocks and the same MAC is applied, with this process repeated until a tree structure over all of the counter values is formed [8]. Because the counter values are smaller than the data, the tree is smaller and therefore higher-performing. Yan et al. describe using the AES cipher in Galois Counter Mode (GCM) to both encrypt and authenticate memory and use a novel split counter as the seed for GCM [9]. AES operated in GCM has been a NIST standard since 2007 and has become widely regarded as having very high performance relative to the resources needed for implementation [10] [11]. This has the advantage of both encrypting and authenticating, similarly to TEC-Tree, and claims a low performance penalty due to the use of a split counter and a counter cache. The hardware and software implementations described later in this work are very similar to the implementation assumptions detailed in Yan et al [9].

2.2 Software Protection Against Cold Boot Attacks

Previous work has addressed cold boot and physical access attacks in software, with no hardware modifications proposed, primarily by making sure that sensitive data (cipher keys) never leave the boundary of the processor. That is, software routines are handcoded into assembly as part of the operating system such that it is provable that operations using those routines will not let a cipher key be stored in RAM and will, instead, keep the cipher key and all intermediate computed data in registers [12] [13]. An extension of this is to store one "master" cipher key in registers and then to use that key to encrypt a section of RAM or nonvolatile storage that is used in turn to store a number of other keys for various uses, largely bypassing the problem of limited register space. Slowdowns in cipher performance on the order of 2-7x [12] and 2-4x [13] are reported compared against software implementations that do not rely on keeping keys in registers.

The protections provided by these methods largely protect small keys that are then in turn relied upon to be used to protect the contents of non-volatile storage via disk encryption. These methods do not provide confidentiality, integrity, or authentication for arbitrary contents of RAM. While included here for completeness, these methods are not evaluated in this thesis as they do not provide a sufficient level of protection for the problem being considered.

2.3 Hardware Assumptions for Memory Authentication and Encryption

Whereas software methods have the distinct advantage of not requiring hardware or ISA modifications, they also do not provide all of the protections that hardware-based solutions provide.

Kgil et al. [14] propose an architecture under the name of ChipLock. ChipLock is based on AES for encryption and decryption and the Secure Hash Algorithm (SHA) for verification of data. The properties of the assumed AES unit are a latency of 32 cycles on the target system clock and a 5.3 mm^2 area in a 180nm process. The properties of the SHA unit are a latency of 160 cycles and an area of 1.0 mm^2. All cache lines are stated to be 64B in size.

Lee et al. [15] propose a secure architecture processor which selectively encrypts and authenticates specific regions of memory based on whether or not they are tagged to contain sensitive data. The proposed architecture uses AES in the Cipher Block Chaining mode with a Message Authentication Code for authentication (AES-CBC-MAC). The AES modules associated with decryption, encryption, and MAC generation are estimated at 20, 80, and 100 processor cycles respectively. It should be noted that decryption in AES-CBC-MAC mode may process multiple blocks in parallel, whereas encryption must process blocks sequentially; the encryption and decryption operations are in fact at the same speed. Therefore, there must be at least 4 separate units employed in parallel (4 blocks of 16B each for a 64B cache line) in order to account for the difference between encryption and decryption speeds in this work. Overall, the overhead latencies for secure data loads, secure data stores, and secure instruction loads are 100, 120, and 80 cycles.

Yan et al. [9] simulate a 5GHz out-of-order processor coupled with a set of 12 AES engines operated in Galois Counter Mode (AES-GCM) that, together with supporting logic and a small counter cache, implements a tree-based memory protection scheme very similar to the TEC-Tree. Each of these engines has a 128-bit, 16-stage pipeline with a total latency of 80 processor cycles (effectively a 1GHz AES-GCM engine). The authors state that this is an ambitious estimate in order to account for "future technological improvements." If each of these engines were fully occupied and produced a block of data on every 1GHz clock, each engine would then be capable of producing 128Gb of output per second. A counter cache of 32KB is coupled with the AES-GCM engines with a block size of 64B.

Some important common characteristics to note about all of these schemes are that they fundamentally rely upon a symmetric key block cipher and some kind of hashing or MAC function to provide confidentiality, integrity, and authentication. The best performing simulation published used AES operated in GCM [9]. For these reasons, AES operated in GCM is used for the remainder of this study. More specifically, this thesis studies the ciphering and authentication primitives and their implementation. We do not simulate the effects of the tree-traversal algorithm or the effects of a counter or hash cache.

These schemes also all assume that the processor is the trusted security boundary. Anything outside of the processor is assumed fallible or insecure, whereas the processor is assumed infallible and unable to be tampered with.

Chapter 3

The AES Cipher and Galois Counter Mode of Operation

The Advanced Encryption Standard (AES) was standardized in 2001 by the United States National Institute of Standards and Technology (NIST) and is considered to be the modern standard for symmetric-key encryption and decryption [10]. However, the description of AES by itself does not constitute a secure way in which to use the cipher. To this end, the NIST also defines various modes of operation in which to operate AES.

This Chapter will provide a high level description of the AES cipher, including each of the four primitive operations that are part of the cipher. This Chapter will also describe one particular mode of operation, Galois Counter Mode, in a similar amount of detail. GCM is frequently used for its high performance and minimal implementation cost in both hardware and software. This was the reasoning used by several of the previous works described in Chapter 2 for choosing a cipher and mode of operation and is why this thesis uses AES and GCM for study.

3.1 Advanced Encryption Standard

AES is a block cipher, meaning that it operates on a fixed size block of data. In the case of AES, this block size is restricted to 128 bits. During operation, the 128 bit data block is referred to as the “state” of the cipher and is frequently considered to be a 4x4, column-major matrix of bytes. AES operates on the state with four primitive operations: byte substitution, row shifting, column mixing, and an XOR operation with a round key. Decryption defines inverse operations for each of these primitives. These operations are composed into either 10, 12, or 14 rounds, depending on the key size. Keys for AES are either 128, 192, or 256 bits in size (corresponding to the 10, 12, and 14 round variants) and are expanded into the appropriate number of round keys.

AES operates on the state in a defined sequence specified by the standard. First, the key is expanded according to the AES key schedule, which uses operations similar to those defined below. This expands the 128, 192, or 256 bit key into 1280, 1536, or 1792 bits corresponding to the 10, 12, and 14 round variants, respectively. Each round consumes 128 bits of the expanded key in the AddRoundKey step. After key expansion, the rounds are applied to the data. The first 9, 11, or 13 rounds consist of applying AddRoundKey, SubBytes, ShiftRows, and MixColumns in sequence. The last round differs slightly, omitting the MixColumns step. After the last round has been applied, the state of the cipher is output as an encrypted block. A similar process is defined for decryption, except using the inverses defined for each step and in

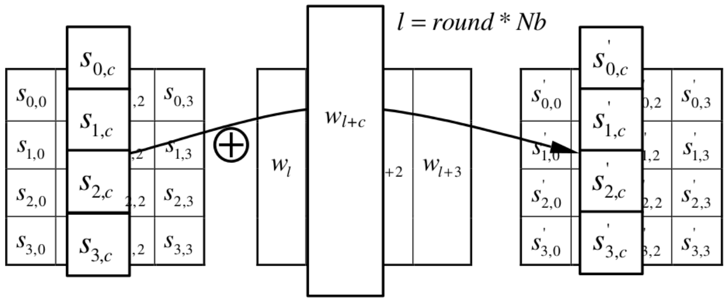


Figure 3.1: AddRoundKey Operation of AES (Image NIST [10])

reverse order.

3.1.1 XOR With Round Key (AddRoundKey)

The AddRoundKey operation is a simple XOR operation, modulo 2, that XORs the state of the cipher with the current round key, as defined by the key schedule. The AddRoundKey operation is illustrated in Figure 3.1.

3.1.2 Byte Substitution (SubBytes)

The SubBytes operation is a non-linear, independent, invertible transformation of each byte of the state. The substitution is constant or hardcoded and specifications for the substitution table may be found in the NIST specification for AES. This primitive is frequently implemented with a look-up-table, either in hardware or software. The SubBytes operation in illustrated in Figure 3.2.

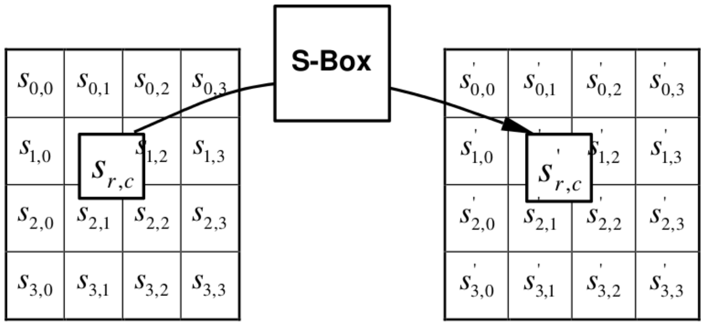

S-Box
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,2}$ $s_{1,3}$
$s_{r,c}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{r,c}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$

Figure 3.2: SubBytes Operation of AES (Image NIST [10])

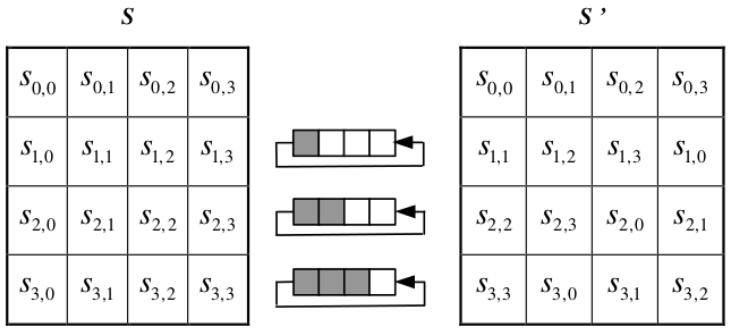

S
S '
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,1}$ $s_{1,2}$ $s_{1,3}$ $s_{1,0}$
$s_{2,2}$ $s_{2,3}$ $s_{2,0}$ $s_{2,1}$
$s_{3,3}$ $s_{3,0}$ $s_{3,1}$ $s_{3,2}$

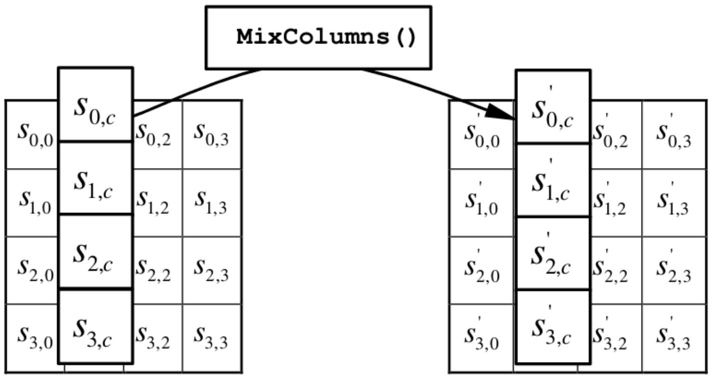
Figure 3.3: ShiftRows Operation of AES (Image NIST [10])

3.1.3 Row Shifting (ShiftRows)

The ShiftRows operation cyclically rotates the rows of the state. The first row is not shifted, but the second, third, and fourth rows are shifted by one, two, and three bytes, respectively. In hardware, this operation can be implemented with simple wiring or shift registers. In software, this operation can be implemented with memory moves, register rotation, or can be integrated with other the operations of other rounds. The ShiftRows operation is illustrated in Figure 3.3.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Figure 3.4: MixColumns Operation of AES (Single Column) (Image NIST [10])



MixColumns()
$s_{0,0}$ $s_{0,c}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,c}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,c}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,c}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,c}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,c}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,c}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,c}$ $s'_{3,2}$ $s'_{3,3}$

Figure 3.5: MixColumns Operation of AES (State Transformation) (Image NIST [10])

3.1.4 Column Mixing (MixColumns)

The MixColumns operation is, conceptually, a matrix multiplication that multiplies each column of the state by a fixed polynomial defined in a Galois Field. This can be implemented directly as a multiplication over a Galois Field in either hardware or software or, as is frequently done, accelerated by pre-calculating look-up-tables for the multiplication results. The MixColumns operation is illustrated in Figures 3.4 and 3.5.

3.2 Galois Counter Mode of Operation

The Galois Counter Mode (GCM) of operation is standardized by the NIST [11] and is an extension of the Counter Mode (CTR) of operation. The Counter Mode of operation is also standardized by the NIST [16] and is shortly described below.

3.2.1 Counter Mode

CTR mode takes a block cipher, such as AES, and generates unique input strings that are used as the input state of the cipher. Inputs are generated by incrementing a counter, leading to the name Counter Mode. After the block cipher has generated an output, corresponding to the encrypted version of the counter value, the output is XORed with the actual data needing to be encrypted. Security for this mode depends on the uniqueness of the counter values, so a sufficiently large counter must be used to avoid reuse of a value.

$$output = block\_cipher(counter) \oplus data\_block \tag{3.1}$$

3.2.2 Galois Counter Mode

GCM is a variant of CTR that adds assurance or authentication in addition to the confidentiality provided by CTR, when used with an appropriate block cipher, such as AES. By using GCM, an authentication tag is generated in addition to the encrypted data. This tag depends upon the encrypted data. Conversely, upon decryption, the tag is supplied in addition to the encrypted data and is used to verify the encrypted data. If the encrypted data is changed or is generated with a key that doesn't match the expected key, it is expected that the tag will not match and the corruption will be detected. GCM also allows for the authentication of data that is not encrypted, referred to as "Additional Authenticated Data." This capability was not used for the work in this thesis.

In the operation of GCM, encrypted data blocks are generated almost identically to CTR mode, with the initial counter value generated from an Initialization Vector (IV) and the incrementing function restricted to 32 bits. A special block of all zeros is also encrypted using the block cipher and the encrypted zero block is used to define the hash subkey for the following steps.

The hashing function of GCM that actually produces the tag takes as input the special block described previously and all of the encrypted data blocks. It iteratively calculates the following algorithm, where $\oplus$ indicates XOR and $\bullet$ indicates multiplication over the binary Galois Field of 2^{128}. The particulars of this multiplication are beyond the scope of this thesis but may be obtained by a number of books and papers on the subject [17]. A similar algorithm is defined for authentication of an already-generated set of blocks with an associated tag.

1. $H = encrypt(zero\_block)$
2. $Y_0 = zero\_block$
3. for i = 1 to m, $Y_i = (Y_{i-1} \oplus encrypted\_block_i) \bullet H$

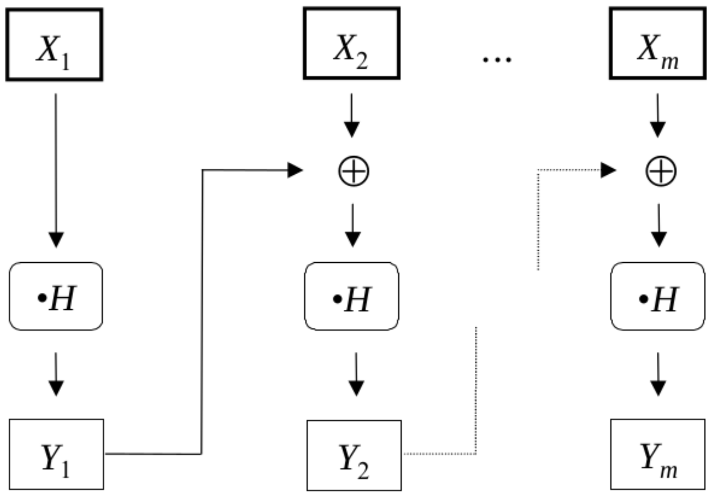


Figure 3.6: GCM Hashing Function (Image NIST [11])

4. $tag = Y_m$

An illustration of this same algorithm is included in Figure 3.6.

Chapter 4

Software AES-GCM Operation

To implement AES-GCM operations in software, the Rijndael benchmark from the Security suite of MiBench [18] was adapted to AES-GCM operation. This benchmark originally implements AES in Cipher Block Chaining (CBC) mode to encrypt or decrypt files stored on disk and is written in portable C with no inline assembly code optimizations. It is a reasonably fast implementation of AES intended for processors that do not necessarily have specific architectural support for cryptography operations. The first major change was to modify the benchmark to operate on blocks strictly in memory and not through file input/output. This was done because it is a more realistic approximation of the way in which an integrated memory authentication and encryption processor may operate. The second major change was to replace the CBC mode of operation with GCM. The original MiBench code does not include libraries that implement GCM, and specifically the Galois Field Multiplication operation of GCM. To address this, open source code was taken from the public domain [19] (also the original author of the code that was adapted to make the Rijndael benchmark in MiBench) and added to the already modified MiBench code. After all modifications, the software implemented AES-GCM in portable C and performed all operations in-memory by repeatedly encrypting or decrypting a variably-sized buffer for a specified number of iterations. Because the software was implemented this way, we believe this is a reasonable approximation of the expected performance if a generic processing core were embedded into the memory system of a processor and applied to the task of memory encryption and authentication. This software assumes no specialized hardware for cryptographic primitives.

Note that the simulations and measurements in this Chapter also intentionally ignore several security concerns in the interest of making a very forgiving test setup that gives every advantage to speed to provide a reasonable comparison point. The processors and setups described in this Chapter require the use of main memory themselves for both instructions and data, for example, and these concerns would have to be dealt with appropriately.

4.1 x86 C Implementation

To profile the speed of the resulting AES-GCM code, the software was compiled with GCC v4.7.2 with *-O3* optimization enabled and evaluated on a system running Fedora 17 GNU/Linux on an Intel Core i7-2620M processor (2.7 GHz). Other optimizations specific to x86, such as *-march=native, -maes, -mpclmul, -funroll-loops, -fomit-frame-pointer* and others were enabled and tested but did not provide any noticeable speedup for the processors evaluated. In order to obtain reliable measurements, power throttling was disabled through the operating system, pinning the processor to its nominal speed of 2.7 GHz. Measurements were collected by inlining the *rdtsc* (read timestamp

Table 4.1: x86 Cycles per Byte Measurements for Pure C Implementation of AES-GCM

Buffer Size	Encrypt	Decrypt
32B	52.2	74.2
64B	37.8	50.4
128B	35.6	39.8
256B	28.3	35.8
512B	25.4	33.0

counter) instruction before and after the AES-GCM operation, accumulating the number of cycles, and then by following Equation (4.1).

$$\frac{cycles}{byte} = \frac{accumulated\_cycles}{num\_iterations * buf\_size} \tag{4.1}$$

The headers controlling various optimizations internal to the code were configured for maximum performance, enabling loop unrolling and four fixed tables (64 KB total) in memory for encryption, decryption, and key scheduling. Being restricted to implementation in pure C code, the speed of the software is restricted as well. Operations on 64B buffers with a 16B tag average on the order of 38 cycles per byte for encryption and 50 cycles per byte for decryption. These speeds allow a throughput of 568 Mbps and 432 Mbps, respectively, per core. Additional speeds for a sampling of buffers sized to be on the order of cache lines are reported in Table 4.1.

4.2 x86 Assembly Implementation

This section will describe implementations of the same AES-GCM algorithms that have been enhanced with assembly-level optimizations for the x86 ISA.Several of these results are reported from other works and not reimplemented for this work.

Gladman's software, the same codebase as used in Section 4.1, also has x86 assembly implementations of key algorithms which reduce processing costs to approximately 22 [20] and 30 [19] cycles per byte, a significant improvement over the results of Section 4.1.

OpenSSL v1.0.0k [21] implements several modes of operation for AES (though not GCM for the platforms tested) and is widely used. Speed tests using the *openssl speed* command report a throughput of 841 Mbps (25 cycles per byte) and 790 Mbps (27 cycles per byte) for CBC [11] and IGE [22] modes, respectively. OpenSSL uses x86 assembly optimizations, hence the comparable but improved performance over a pure C implementation described in Section 4.1.

An Intel whitepaper [20] describes a very high-performance implementation of AES-GCM that reaches approximately 3.5 cycles per byte for performing a full encryption and tag generation, albeit for a large buffer size of 16KB and processing 4 blocks in parallel. This analysis was performed on an Intel "Westmere" server-class processor. The Westmere series of processors have a Thermal Design Power ranging from 35-130 watts. This high-performance implementation takes advantage of the AES and PCLMULQDQ instruction set extensions in modern x86 processors. The AES extensions enable a full round of AES encryption or decryption to be issued with a single instruction and the PCLMULQDQ extension allows for efficient multiplication over a Galois Field, which is critical for GCM operation.

Of notable importance is that good assembly implementations of AES-GCM operations reduce the cycles per byte processing requirement of information by potentially an order magnitude (2-10x) compared to portable C code, but still require the use of a high-performance architecture and potentially dedicated instruction set extensions to extract the most performance.

4.3 Alpha/SimpleScalar/Wattch Implementation

The same codebase detailed in Section 4.1 was also compiled for the Alpha architecture and targeted for SimpleScalar v3.0 and, specifically, Wattch v1.02 [23] [24]. Analysis with Wattch indicates that a software implementation of this type of encryption and authentication process is very expensive in terms of power. Configured similarly to a modern Intel Core i7 processor, Wattch reports an average power consumption of more than 94 watts for a single core running nothing but either encryption + tag generation or decryption + authentication at 900 MHz. SimpleScalar also reports high levels of instruction level parallelism for the AES-GCM software. A 4-wide simulated machine averages 2.7 instructions per cycle for both encryption and decryption.

While consuming this large amount of power, the simulated processor is only able to sustain a throughput of 170 Mb per second with a processing requirement of 41 cycles per byte. This is less than any of the x86 processors measured or described previously and confirms the notion that software solutions both consume large amounts of power and have restricted performance.

4.4 ARM Implementation

The previously described C code without assembly optimizations was also ported to the ARM7 architecture and was run on the gem5 simulator (v. stable_2012_06_28) using the detailed ARM processor model [25]. This model approximates a modern out-of-order ARM core and was configured with 32KB L1 instruction and data caches and a 1MB L2 cache, running at a 1GHz clock speed. Performance statistics for a full run of the encryption and decryption code report an average of 1.5 instructions per cycle committed. Encryption and tag generation performed on 64B blocks comes at a processing cost of 63.1 cycles per byte while decryption and tag verification are measured to be 63.8 cycles per byte. This leads to a peak throughput of 126 Mbps.

Modern ARM cores that are similar in microarchitecture to the gem5 model, such as those offered by Samsung or Qualcomm, are designed with a Thermal Design Power (TDP) of approximately 2 Watts per core [26] [27].

Chapter 5

Hardware AES-GCM Operation

An open-source AES-GCM module from OpenCores was modified to behave in a similar manner to other papers that have assumed the existence of an authentication and encryption module [28]. The module used is intended to be used in the 128b key mode of AES and has an integrated Galois Field Multiplier that implements the required multiplication operations for GCM. The field multiplier is a 16b multiplier capable of processing an AES state block in 8 cycles. This strikes a reasonable tradeoff between high-width operations and speed of processing. The AES-GCM module also has a mutable key, meaning that the key is capable of being changed in between any major operations, but not during an operation. This capability is important as previous works, such as those discussed in Chapter 2, have stipulated that keys used for these operations need the ability to be changed after certain events such as a counter overflow or an operating system directive to flush or change keys.

The referenced open-source module used for these studies was not originally perfectly suited for comparison and a number of modifications were made in order to make it more suitable. An additional top level module was written to route traffic to and from a parameterizable number of the original

AES-GCM modules. This allowed for relatively easy comparisons and measurements with varying module counts without the requirement of changing the top level interface. Several minor modifications were also made to the original code. Some of the original state machines were modified. The GCM specification includes the ability to authenticate additional data (AAD) that is not encrypted. The original code faithfully includes this part of the specification, but is not needed for this study as the intention of almost all of the referenced previous work is to encrypt all data to and from external memory. Therefore, this capability was removed and state machines were made to more closely adhere to only the needed capabilities.

Table 5.1 compares the modified open-source module, implemented on a Kintex 7 FPGA, against the advertised specifications of a AES-GCM module available through Xilinx [29]. The open-source module compares well against the advertised specifications of the commercial module and is of slightly lower performance with slightly lower resource usage to compensate.

The modified AES-GCM module was taken through the toolflow of both Xilinx ISE (v. 14.3) and Synopsys Design Vision (v. E-2010.12) [30] [31]. In both cases, up to 16 AES-GCM modules were synthesized, because it was determined that numbers greater than 16 were increasingly prohibitively long to synthesize and evaluate.

Table 5.1: Open-Source vs. Commercial AES-GCM RTL Module on Kintex 7

Metric	Open Source	Commercial
Startup	19 clocks	0 clocks
16B Enc/Dec	22 clocks	12 clocks
16B Tag(Hash)	17 clocks	12 clocks
64B Cache Line + Tag	123 clocks	60 clocks
Freq. Max	212 MHz	256 MHz
Logic Slices	˜800	˜1000
Block RAMs	8	12

5.1 FPGA Implementation

Power estimates of a single AES-GCM module for the Xilinx Kintex 7 FPGA target are reported in Table 5.2. Xilinx XPower Analyzer was used to estimate power for a single AES-GCM module with the clock speed set to 200 MHz, slightly slower than the maximum reported attainable speed of 212 MHz. Other settings besides clock speed were left at default for these estimates and both the "typical" and "worst-case" power usage estimates were collected. The typical and worst-case usage estimates seem to mainly differ in static power dissipation, with dynamic dissipation minimally affected. Some other settings, such as flip-flop toggle rate, were increased or decreased and showed little variation. Changing the flip-flop toggle rate from 12.5 to 100 increased the estimated power consumption by less than 10% whereas other settings had even lesser effects.

Table 5.2: Power Estimates for Single AES-GCM Module

Target	Clock Speed	Dynamic	Static	Total
FPGA (typical)	200 MHz	228 mW	123 mW	351 mW
FPGA (worst)	200 MHz	230 mW	371 mW	601 mW
ASIC (typical)	225 MHz	11.956 mW	0.428 mW	12.384 mW

5.2 ASIC Implementation

Power estimates for a single AES-GCM module synthesized with Synopsys Design Vision [31] and targeting the FreePDK45 library [32] are reported in Table 5.2. FreePDK is an open-source 45nm cell library commonly used in academic settings. The same code that was used in Section 5.1 was used for this synthesis, with all of the modifications intact. The maximum attainable clock speed reported by the synthesis tools was 250 MHz for a single module and slightly lower for multiple modules; therefore the target clock speed was set at a conservative 225 MHz for all synthesized designs. The power estimates for all Design Vision designs are “typical” use case power estimates. The FreePDK library that was used does not provide “worst case” power estimates.

5.3 Hardware Power Dissipation

Figure 5.1 provides more detailed information on the estimated power consumption of a set of AES-GCM modules synthesized for the Kintex 7. In order to more accurately characterize the incremental power consumption of adding more modules, a linear regression was applied to the data, resulting in the following equations.

$$power_{typical\_FPGA}(modules) = 192.9 * modules + 216.3 (mW) \tag{5.1}$$

$$power_{worst\_FPGA}(modules) = 199.4 * modules + 462.7 (mW) \tag{5.2}$$

Unsurprisingly, a simple linear regression fits the power estimates very well, indicating that adding bandwidth (modules) to a proposed system for memory authentication and encryption should scale linearly.

Figure 5.2 shows the power usage estimates for an ASIC implementation of the AES-GCM module(s) at a 45nm technology node. A linear fit trend was applied to the data resulting from synthesis of 1, 2, 4, 8, and 16 AES-GCM modules and indicates a linear increase in power consumption along with the number of modules, resulting in the following equation.

$$power_{typical\_ASIC}(modules) = 11.05 * modules + 2.12 (mW) \tag{5.3}$$

5.4 Hardware Resource Consumption

Figures 5.3 and 5.4 provide measurements of the resource consumption of the synthesized AES-GCM modules. In the case of Figure 5.3 the measurements are of Look-Up Tables (LUTs) and Flip-Flops (FFs). Block RAMs are not pictured because exactly 8 RAMs are required per module. In the case of Figure 5.4 the measurements are in terms of the native unit of measurement of the FreePDK library, which is square micrometers (μm^2).

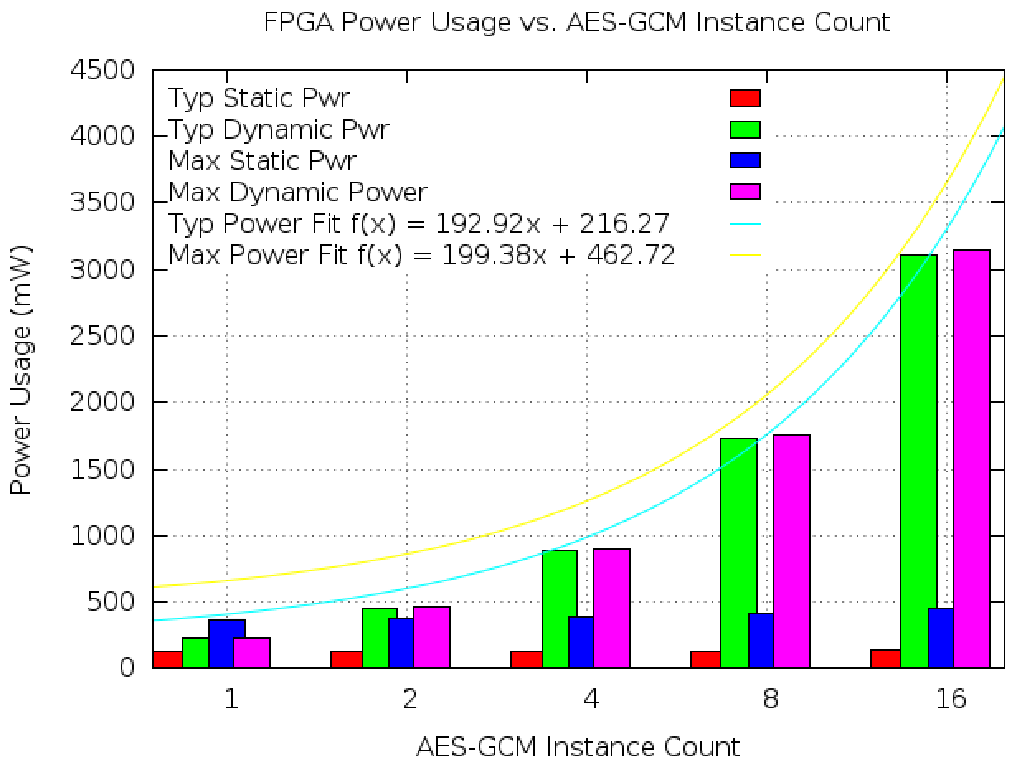


Figure 5.1: FPGA Power Consumption Estimates for AES-GCM Modules

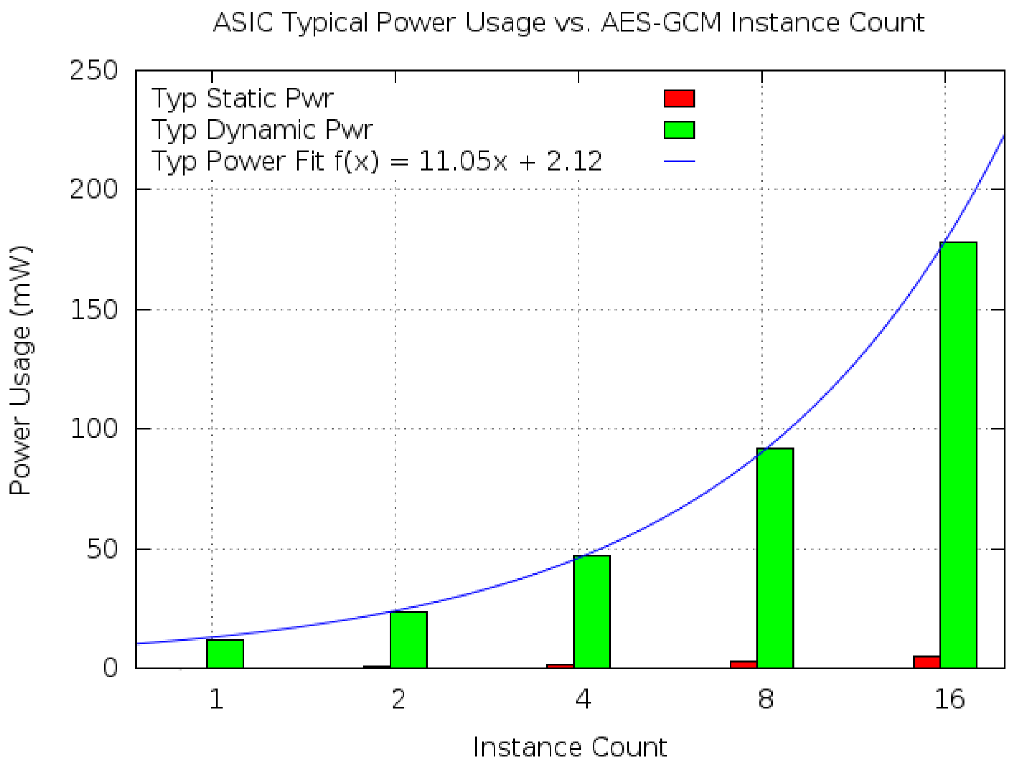


Figure 5.2: ASIC Power Consumption Estimates for AES-GCM Modules

In both the FPGA and ASIC implementations, the resource consumption scales linearly with the number of instances of the module, as shown by the following linear trend fits applied to the data for FPGA resource consumption and synthesized ASIC area.

$$\begin{aligned} resources_{FPGA}(modules) &= 1509.7 * modules - 222.5(FFs) \\ &\quad + 2356.2 * modules - 81.4(LUTs) \\ &\quad + 8 * modules(BRAMs) \end{aligned} \tag{5.4}$$

$$area_{ASIC}(modules) = 74125 * modules + 34762(\mu m^2) \tag{5.5}$$

The FPGA implementation consumes approximately 1500 FFs and 2400 LUTs per module, which fits well with the preliminary estimate of approximately 800 logic slices per module. Each logic slice encompasses multiple FFs and LUTs in the Kintex 7 architecture. The ASIC implementation includes all synthesized logic, FFs, and RAMs.

Combining the results in Section 5.3 and Section 5.4, we are able to calculate an estimate of the $\frac{power}{area}$ dissipation. These results are presented in Table 5.3. The calculation show a relatively constant power density of approximately 0.15 $\frac{watts}{mm^2}$.

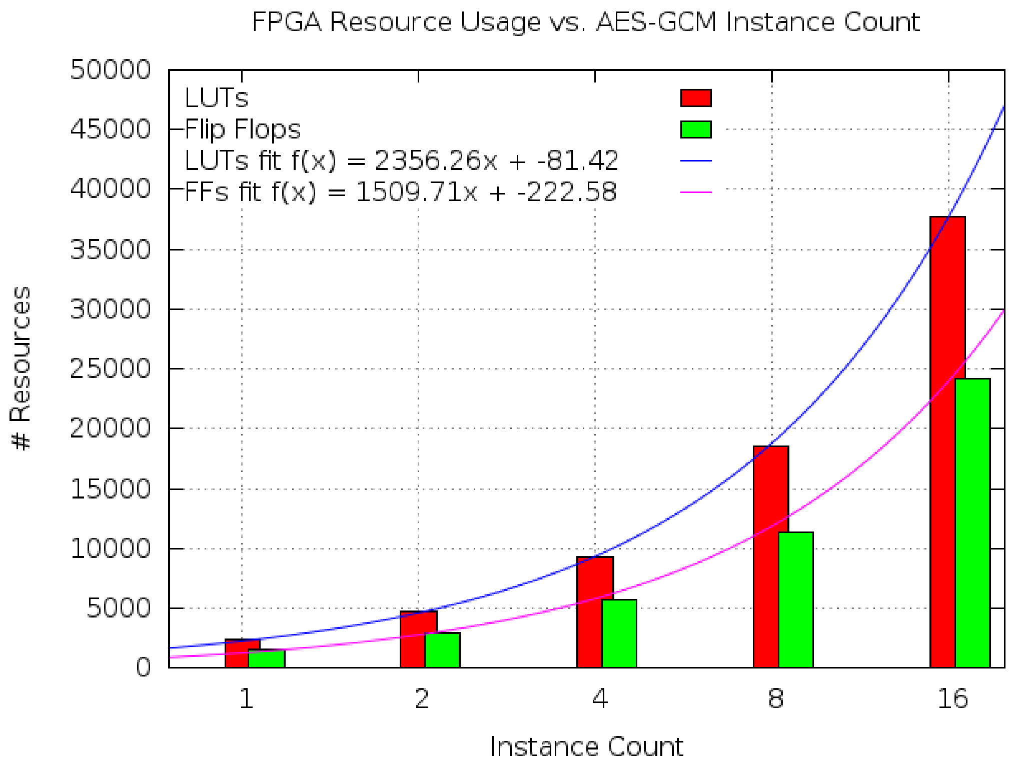


Figure 5.3: FPGA Resource Consumption for AES-GCM Modules

Table 5.3: ASIC Power / Area for AES-GCM Modules

Modules	Area(μm^2)	Power(mW)	Density($\frac{W}{mm^2}$)
1	89k	12.383	0.139
2	176k	24.732	0.140
4	343k	48.724	0.142
8	657k	95.277	0.145
16	1204k	183.162	0.152

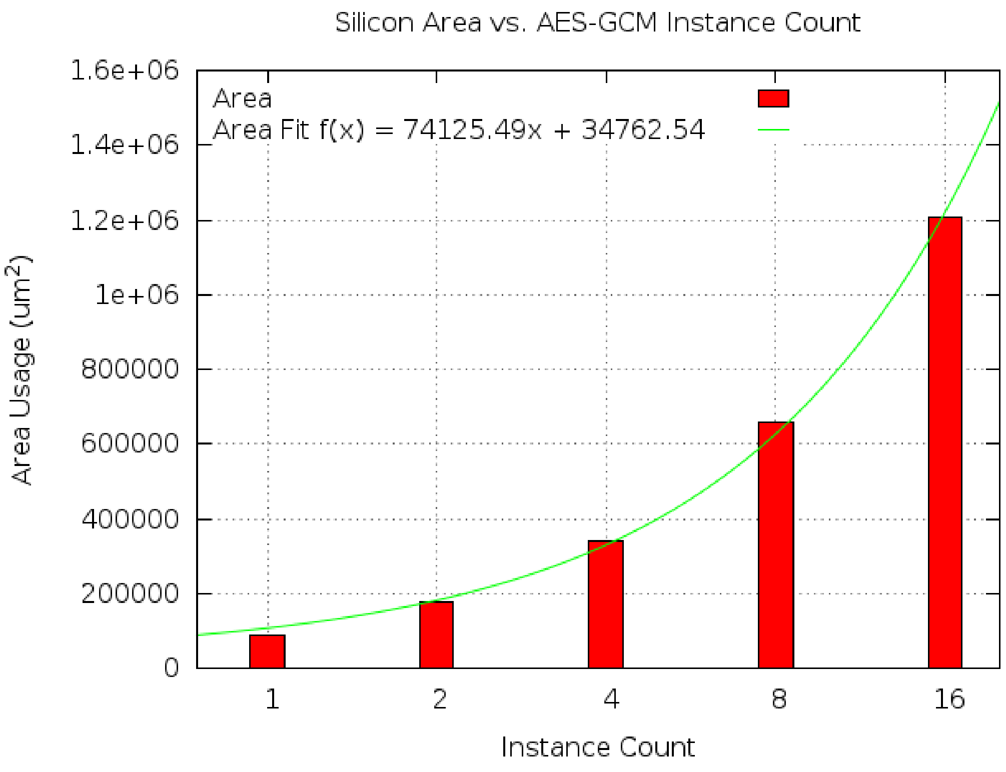


Figure 5.4: ASIC Area for AES-GCM Modules

Chapter 6

Comparative Evaluation

Table 6.1 summarizes all of the relevant metrics for the various implementation methods discussed in this work. The clock rates reported are the speeds at which the modules were tested, which, in the case of the ASIC and FPGA implementations, was slightly lower than the maximum attainable clock speed reported by synthesis tools. The clock rates for x86 were forced to one speed by disabling power management and the Alpha clock was fixed in SimpleScalar. The throughput rates reported here are for 64B of data (cache line sized) with tag generation or authentication. Encryption and decryption speeds were averaged for each implementation for the purposes of calculating throughput.

Table 6.1: Summary of Different Implementation Methods

	ASIC	FPGA	x86	x86	x86	Alpha	ARM
			C	Assembly	ISA Ext.		
Clock (Hz)	225 M	200 M	2.7 G	2.7 G	2.7 G	900 M	1 G
$\frac{Cycles}{Byte}$	1.9	1.9	44	22	3.5	41	63.5
Throughput	$\frac{936.6Mbps}{instance}$	$\frac{882.5Mbps}{instance}$	$\frac{490.9Mbps}{instance}$	$\frac{981.8Mbps}{instance}$	$\frac{6.17Gbps}{instance}$	$\frac{175.6Mbps}{instance}$	$\frac{126.1Mbps}{instance}$
Typ. Power	11.05 mW	192.9 mW	35 W (TDP)	35 W (TDP)	35 W (TDP)	94 W	2 W (TDP)
Typ. Area	$74.1k\mu m^2$	-	-	-	-	-	-
Mbps/mW	84.7	4.57	0.0140	0.0281	0.176	0.00186	0.063

Table 6.2: Memory Bandwidth of Several Modern Mobile Systems

	Nexus 7	Nexus 10	iPhone 5	iPad 3
Peak DRAM BW ($\frac{GB}{s}$)	5.3	12.8	8.5	12.8

Throughput for the software implementations was calculated as follows:

$$throughput_{SW} = \frac{clock\_speed}{num\_clocks/B} \tag{6.1}$$

Throughput for the ASIC and FPGA implementations was calculated as follows:

$$throughput_{HW} = \frac{clock\_speed}{num\_clocks/64B} * 64 \tag{6.2}$$

Power estimates are restated here from Figures 5.1 and 5.2, Intel documentation [33], and Chapter 4.

Once a final metric of $\frac{throughput}{power}$ has been computed, it is easier to compare these different implementation methods for the task of memory encryption and authentication. The various software solutions can be seen to be inferior due to their high power consumption at several magnitudes lower efficiency compared to both FPGA and ASIC implementations. ISA extensions allow modern software solutions to eclipse the studied hardware solutions in terms of pure performance per instance by a significant margin, but at the cost of high power. Although not directly compared here, the area dedicated to obtaining this functionality is also significantly smaller for an ASIC solution.

6.1 Evaluating Feasibility of Implementation

Modern PC systems have peak memory bandwidths up to 25.6 and 21 $\frac{GB}{s}$ (204 to 168 $\frac{Gb}{s}$) for the highest-performing desktop Intel [34] and AMD [35] processors currently available. Modern mobile systems such as tablets and phones are increasingly powerful as well; several representative systems are presented in Table 6.2 [36] [37] [38] [39]. It can be seen that peak bandwidth for mobile systems has almost reached parity with traditional computing systems, in packages that are more power-constrained, making feasibility of implementation for any memory protection system even more important.

Though memory bandwidth is not saturated during typical operation, it is easy to conclude that the capabilities provided for memory encryption and authentication must at least meet the peak bandwidth in order to maintain an acceptable level of performance. For example, in Yan et al. [9], discussed previously, the bandwidth provided by the 12 AES-GCM hardware modules would account for a peak theoretical processing rate of 1.5 $\frac{Tb}{s}$, significantly more than any modern system.

Using only the data for the Intel system mentioned above, we can make an assessment of feasibility for the different methods of implementation.

6.1.1 Embedded Software Implementation

For the methods evaluated in Chapter 4, implementation of enough x86 instances to satisfy peak bandwidth requirements of 25.6 $\frac{GB}{s}$ would take approximately 590, 210, and 34 instances for C, assembly, and assembly with

ISA extension methods, respectively. Implementation of enough ARM cores to satisfy peak bandwidth requirements would take just over 1600 instances. Implementation of Alpha cores was not considered due to extremely low efficiency. The power budget and silicon area that would have to be dedicated to any of these implementations would be tremendous and would make a design based on these methods ultimately untenable.

6.1.2 FPGA Implementation

For the FPGA method evaluated in Section 5.1, implementation of enough instances to satisfy the same peak bandwidth requirements would take approximately 230 AES-GCM modules, as described in Chapter 5. This would come at a power cost of approximately 44.7 W, following the regression curve described previously in Chapter 5. Additionally, this number of modules would not be able to be synthesized into the particular FPGA that was used for this work; a larger, more expensive, and likely more power-hungry chip would have to be used instead.

An interesting line of thought is that a much less aggressive processing core, implemented in an FPGA, may be able to be modified with an appropriate number of memory encryption and authentication modules to provide security and reasonable performance, all on the same chip. A smaller number of modules would easily fit in the FPGA with room to spare and would not consume a large amount of power. This warrants further research into implementation and feasibility of this specific idea, though this is out of the bounds of this work.

6.1.3 ASIC Implementation

For the ASIC method evaluated in Section 5.2, implementation of enough instances to satisfy the same peak bandwidth requirements would take approximately 220 modules as described in Chapter 5. This would consume power at approximately 2.4 W, following the regression curve previously described in Chapter 5. Additionally, an implementation following this method would require approximately 16.3 mm^2 of silicon area at a 45 nm process, again following the regressions described previously.

These costs are much more reasonable than any of the software-based or the FPGA-based solutions described in this work. In short, it seems that implementation of memory encryption and authentication through an ASIC solution is reasonable while software and FPGA-based solutions are simply too power- and space-inefficient to be feasible.

Chapter 7

Future Work

There are a number of ways in which this work can be extended. This will be briefly discussed in this Chapter.

First, information about cache area and power should be integrated with the AES-GCM modules. Almost all of the previous work on hardware memory authentication and encryption designs include a cache dedicated for use with the tags for memory blocks. These caches are an integral part of the system and are considered to be the best way to reduce the performance penalty of memory encryption and authentication. This was not modeled in this work and would serve to make a more complete power and area estimation for such a system.

Second, modeling the effect of tree-traversal for the described tree of counter, hash, or tag values would more accurately characterize performance and power penalties.

Third, additional data points at different process nodes for ASIC implementations of AES-GCM modules would serve to better characterize this work, especially in terms of the embedded market where process technologies typically lag behind high-performance processes.

Fourth, analyzing more RTL implementations of AES-GCM modules or even other modes of AES implemented in hardware would serve to more closely parallel previous work. Not all of the previous work in this area used AES-GCM, and other modes of operation may have differing hardware costs.

Fifth, power annotation into a performance simulator would serve to combine both power and performance information regarding memory encryption and authentication.

Chapter 8

Conclusion

The work described in this thesis demonstrates that implementation costs should be a significant factor in the evaluation of both past and future work on topics in the field of memory encryption and authentication.

Chapter 4 covered implementation and analysis concerning software solutions. Solutions ranging from a pure, portable C implementation to augmentation with hand-coded assembly to full-on ISA extensions with hand-coded assembly were considered. The conclusion was that software implementations of the required primitives to enable AES-GCM operations and to process data are very expensive in terms of power and do not provide a suitable amount of performance to justify their power budget or implementation complexity. Portable software solutions that may be considered as a "drop-in" solution have been demonstrated to have less than acceptable performance considering the amount of hardware and power that would have to be dedicated to run these algorithms. Architectures with ISA extensions, such as modern versions of x86, can perform on par with dedicated hardware, but come at the cost of very high power dissipation. Overall, the approach of assuming a software solution to the memory encryption and authentication problem is infeasible.

Chapter 5 covered implementation and analysis of an open-source hardware module capable of performing all of the required operations to enable a solution to the memory encryption and authentication problem in both FPGA and ASIC formats.

The FPGA implementation of the module shows promise as a potential solution to be added onto an existing or new design that already targets an FPGA. The resources consumed on an FPGA are considerable for each module and a large number of modules would quickly consume a majority of the FPGA resources, having the effect of restricting the usefulness of the FPGA and potentially slowing clock speeds with routing-induced delay. The power dissipation for the FPGA implementation is non-trivial but remains well within the capabilities of the FPGA to dissipate, as confirmed by the Xilinx tools. However, a significant increase in dynamic power consumption can be expected if hardware modules enabling memory encryption and authentication functionality are added to a design previously lacking them.

Implementation of the hardware module as an ASIC at a 45 *nm* process node shows promise as a reasonable, though non-trivial, solution. Power dissipation on the order of ~10 mW per module indicate that solutions of a similar nature may be feasible to implement as part of a larger system. The analysis conducted in Chapter 6 also shows that it is possible to meet the peak memory bandwidth requirements of modern systems using modules of this type.

The analysis of Figures 5.1 and 5.2 show a linear relationship between the number of AES-GCM modules and power dissipation for a design. Although not unexpected, this result indicates that a design needing anything more than minimum memory performance would need to carefully consider the implications of power dedicated to hardware memory encryption and authentication as a first-order design constraint. The works discussed in Chapter 2 assume the existence of at least one, but typically many more, engines capable of performing the low-level work necessary to encrypt and authenticate memory traffic. This work has shown that, as more modules are assumed to be required, area and power can become significant concerns, especially for modern processors that have a large amount of memory bandwidth.

For systems that may not require a large number of hardware modules, the picture is more optimistic. A small number of modules may be integrated into a design without too much of a power or area requirement. For small numbers, power dissipation would only increase a maximum of a few dozens of mW and area would increase by approximately 1 mm^2, assuming a 45 nm process. The integration of these modules, along with some additional hardware such as a dedicated counter cache and appropriate control logic would contribute significantly to enabling secure memory encryption and authentication.

This work contributes an analysis of the feasibility of memory encryption and authentication for the implementation methods included in previous Chapters. We have expanded the work done by previous researchers by taking an in-depth look at the characteristics of several implementations of the necessary primitives to enable memory encryption and authentication, leading to the conclusion that additional power dissipation and area usage will have to be a first-order design constraint and that the only feasible implementation method is with ASICs integrated directly into the design of a system.

Bibliography

[1] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold-boot attacks on encryption keys," *Commun. ACM*, vol. 52, no. 5, pp. 91–98, May 2009. [Online]. Available: http://doi.acm.org/10.1145/1506409.1506429

[2] R. Elbaz, D. Champagne, C. Gebotys, R. B. Lee, N. Potlapally, and L. Torres, "Hardware mechanisms for memory authentication: A survey of existing techniques and engines," in *Transactions on Computational Science IV*, M. L. Gavrilova, C. J. Tan, and E. D. Moreno, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines, pp. 1–22. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-01004-0_1

[3] M. Kuhn, "Cipher instruction search attack on the bus-encryption security microcontroller ds5002fp," *Computers, IEEE Transactions on*, vol. 47, no. 10, pp. 1153 –1157, Oct. 1998.

[4] A. Huang, "Keeping secrets in hardware: The microsoft xbox (tm) case study," in *Cryptographic Hardware and Embedded Systems - CHES 2002*,

ser. Lecture Notes in Computer Science, B. Kaliski, . Ko, and C. Paar, Eds. Springer Berlin Heidelberg, 2003, vol. 2523, pp. 213–227. [Online]. Available: http://dx.doi.org/10.1007/3-540-36400-5_17

[5] R. Merkle, "A digital signature based on a conventional encryption function," in *Advances in CryptologyCRYPTO87*. Springer, 2006, pp. 369–378.

[6] W. Hall and C. Jutla, "Parallelizable authentication trees," in *Selected Areas in Cryptography*. Springer, 2006, pp. 95–109.

[7] R. Elbaz, D. Champagne, R. Lee, L. Torres, G. Sassatelli, and P. Guillemin, "Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks," *Cryptographic Hardware and Embedded Systems-CHES 2007*, pp. 289–302, 2007.

[8] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 183–196.

[9] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ser. ISCA '06. Washington, DC,

USA: IEEE Computer Society, 2006, pp. 179–190. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2006.22

[10] N. F. Pub, "197: Advanced encryption standard (aes)," *Federal Information Processing Standards Publication*, vol. 197, pp. 441–0311, 2001.

[11] M. Dworkin, "Nist special publication 800-38d: Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac," *US National Institute of Standards and Technology*, 2007.

[12] T. Müller, A. Dewald, and F. C. Freiling, "Aesse: A cold-boot resistant implementation of aes," in *Proceedings of the Third European Workshop on System Security*, ser. EUROSEC '10. New York, NY, USA: ACM, 2010, pp. 42–47. [Online]. Available: http://doi.acm.org/10.1145/1752046.1752053

[13] P. Simmons, "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*, ser. ACSAC '11. New York, NY, USA: ACM, 2011, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/2076732.2076743

[14] T. Kgil, L. Falk, and T. Mudge, "Chiplock: Support for secure microarchitectures," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 1, pp. 134–143, 2005.

[15] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, "Architecture for protecting critical secrets in microprocessors," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 2–13. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2005.14

[16] M. Dworkin, "Nist special publication 800-38a: Recommendation for block cipher modes of operation," *US National Institute of Standards and Technology*, 2001.

[17] L. E. Dickson, *Linear groups: With an exposition of the Galois field theory.* Courier Dover Publications, 2003.

[18] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, ser. WWC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. [Online]. Available: http://dx.doi.org/10.1109/WWC.2001.15

[19] B. Gladman, "Aes and combined encryption/authentication modes," Feb. 2013. [Online]. Available: http://gladman.plushost.co.uk/oldsite/AES/

[20] S. Gueron and M. E. Kounavis, "Intel® carry-less multiplication instruction and its usage for computing the gcm mode," *White Paper*, 2010.

[21] "Openssl: The open source toolkit for ssl/tls," The OpenSSL Project, 2013. [Online]. Available: https://www.openssl.org/

[22] C. Campbell, "Design and specification of cryptographic capabilities," *Communications Society Magazine, IEEE*, vol. 16, no. 6, pp. 15–19, 1978.

[23] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, Feb. 2002. [Online]. Available: http://dx.doi.org/10.1109/2.982917

[24] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 83–94. [Online]. Available: http://doi.acm.org/10.1145/339647.339657

[25] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[26] "Samsung exynos," Apr. 2013. [Online]. Available: http://www.samsung.com/global/business/semiconductor/minisite/Exynos/

[27] "Snapdragon s4 product specs," Apr. 2013. [Online]. Available: www.qualcomm.com/snapdragon/processors/s4/specs

[28] T. Ahmad, "Galois counter mode advanced encryption standard gcm-aes," Oct. 2010. [Online]. Available: http://opencores.com/project,gcm-aes

[29] "Aes-gcm - authenticated encryption / decryption," Feb. 2013. [Online]. Available: http://www.xilinx.com/products/intellectual-property/AES-GCM.htm

[30] X. Inc., "Ise design suite," Feb. 2013. [Online]. Available: http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm

[31] "Synopsys design vision," Synopsys, 2010. [Online]. Available: http://www.synopsys.com/home.aspx

[32] J. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. Davis, P. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, "Freepdk: An open-source variation-aware design kit," in *Microelectronic Systems Education, 2007. MSE '07. IEEE International Conference on*, Jun. 2007, pp. 173 –174.

[33] "Intel core i7-2620m processor," 2011. [Online]. Available: http://ark.intel.com/products/52231/

[34] "Intel core i7-3940xm processor extreme edition," 2013. [Online]. Available: http://ark.intel.com/products/71096/

[35] "Amd fx processor model number and feature comparison," 2013. [Online]. Available: http://www.amd.com/us/products/desktop/processors/amdfx/Pages/amdfx-model-number-comparison.aspx

[36] "Apple iphone technical specifications," 2013. [Online]. Available: https://www.apple.com/iphone/specs.html

[37] "Apple ipad technical specifications," 2013. [Online]. Available: https://www.apple.com/ipad/specs/

[38] "Nexus 10 technical specifications," 2013. [Online]. Available: https://www.google.com/nexus/10/specs/

[39] "Nexus 7 technical specifications," 2013. [Online]. Available: https://www.google.com/nexus/7/specs/

Vita

Donald E. Owen, Jr. may be reached by the email address included below.

Permanent address: donald.e.owen@utexas.edu

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.