Exploiting Memory Hardware for Use in Cryptographic Operations

by

Andrew J. Armstrong

Bachelor of Science, The University of Pittsburgh, 2015

Submitted to the Graduate Faculty of the

Dietrich School of Arts and Sciences in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2016

UNIVERSITY OF PITTSBURGH

Dietrich School of Arts and Sciences

This thesis was presented

by

Andrew J. Armstrong

It was defended on

December 8, 2016

and approved by

Dr. Bruce Childers, Professor, Department of Computer Science

Dr. Daniel Mossé, Professor, Department of Computer Science

Thesis Director: Dr. Adam J. Lee, Associate Professor, Department of Computer Science

Copyright © by Andrew J. Armstrong

2016

EXPLOITING MEMORY HARDWARE FOR USE IN CRYPTOGRAPHIC OPERATIONS

Andrew J. Armstrong, M.S.

University of Pittsburgh, 2016

Recent data breaches have motivated a desire to remove all trust storage platforms (e.g., the cloud). To this end, research has focused on implementing cryptographic access controls on untrusted storage platforms. However, there are issues with the feasibility of implementing such controls, particularly when revocation (i.e., a user losing permission) occurs. This thesis investigates the opportunity to increase the viability of these systems by exploiting new functionality in emerging main memory technology. Technology such as the Hybrid Memory Cube possess the ability to perform certain computations in-memory, without reading data into the CPU. This thesis focuses on implementing a re-encryption scheme, called keystream re-encryption, that computes a stream of key material that can be XOR-ed in-memory to re-encrypt a file, without ever bringing the contents of that file into the CPU. We show that keystream re-encryption can produce 5-10% improvements in Instructions Per Cycle (IPC), while also increasing throughput by 18% and reducing energy consumption by 44-65%.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF EQUATIONS

1.0 INTRODUCTION

Typical access control models run under the assumption of a trusted platform, that is, a platform that will not act in malicious way. Specifically, these models assume a trusted reference monitor to enforce the specific policies. If this assumption is violated, then we can no longer trust our reference monitor, or the access control in general, to enforce our policies. For example, data breaches have become a regular occurrence, with more than 6,500 breaches occurring since 2005, leaking over 850 million records [12]. In response to these risks, research has been conducted into implementing access control models that assume an untrusted platform (e.g., cloud services, shared computers, etc.) since, should the platform begin acting in a hostile way, the system has already been designed to handle that undesired behavior.

In recent years, many access control models have been proposed to address these issues with untrusted platforms. These models have focused on implementing cryptographic access control. Cryptographic access control does not rely on an enforcement mechanism that lives on the untrusted platform, but rather, it relies on keys that are used to encrypt the data, thus rendering the data unusable unless you possess the key. Possession of the key, therefore, is equivalent to having permission to view that file. Unfortunately, current research on cryptographic access control models consists primarily of static data and permissions and new research has shown that these models can inherit excessive overheads when dynamically changing data and permissions

are introduced [5]. Even simplifying assumptions such as lazy re-encryption does not help to mitigate these problems.

When examining these costs, revocation of permission(s) from a user, which occurs when a user has their access to some data removed, provides an exemplary glimpse into the pitfalls of cryptographic access control models. The revocation of permissions from a user in real-world datasets have been shown to incur costs totaling thousands of re-encryptions due to the need to re-encrypt all data the user had access to with new keys [5]. This arises from the fact that most users have thousands (if not millions) of files and those files are typically shared amongst multiple users. When a revocation occurs, this creates a domino effect of re-encryptions. Once all the costs are calculated, it becomes apparent that dynamic systems can lead to poor performance. As access control is desirable in an untrusted environment, the question must be asked of whether there are remedies to these issues.

Revocation provides an opportunity to examine an underlying issue with the access control models proposed. Requiring so many re-encryptions incurs a significant overhead that, if alleviated, could lead toward situations where these cryptographic access controls become more feasible. To that end, this thesis examines the feasibility of expediting that re-encryption process. While there has been extensive work in enhancing cryptographic performance, these results have always required the active participation of the CPU. Thus, these enhancements have still incurred the memory traffic associated with pulling the data to be re-encrypted into the CPU, performing the operations in the CPU, and the traffic generated to return the data to memory. The goal of this thesis is to show that it is not only feasible to shift some of the computation off the CPU, but by doing so, there will be other benefits in terms of decreased memory latency since the data will not need to be brought into the CPU. By leveraging these improvements, this thesis aims to show that

some downsides to cryptographic access control systems can be reduced, thereby increasing the viability of said models.

To this end, we look at emerging memory technologies, which have the capacity to further increase the efficiency of cryptographic operations. Memory technology has advanced in recent years to provide many improved and new features including higher bandwidth, shorter latencies and more recently, in-memory computation [19]. These features provide opportunities to reduce cryptographic overheads, however, while most are speed and performance increases, in-memory computation provides a unique opportunity to reduce the amount of computation performed by the processor as well as the amount of data brought into the CPU during these cryptographic operations. This thesis investigates the ways in which offloading some of the workload can reduce the overheads associated with cryptographic operations while also reducing the impact that cryptography has on co-running processes and thus, increase the viability of cryptographic access control models.

This thesis is organized with the following structure. Chapter 2 discusses related works dealing with hardware solutions aimed at improving cryptographic performance along with a discussion about cryptographic access controls. Chapter 3 addresses the methodology used to investigate in-memory computation. The experimental setup, results and discussion are laid out in Chapter 4. With Chapter 5 concluding the thesis along with a discussion of future works.

2.0 BACKGROUND & RELATED WORKS

We now provide an overview of background information and related works relevant to our investigation. We begin by describing the types of cryptographic functions that will be used in this thesis. We then move on to describe work related to cryptographic access controls and their shortcomings before describing a set of new instructions by Intel to aid in AES cryptography. Finally, we discuss in-memory computation technology and the possibilities that this opens up.

2.1 BLOCK CIPHERS

Block ciphers are a type of cryptographic function that allows for the encryption and decryption of fixed-sized blocks of data. They perform better than stream ciphers, which are used for encrypting/decrypting continuous data and which perform the cryptography on a bit-by-bit level rather than per block [1]. Figure 1 illustrates a simple, high level block cipher. A plaintext input of fixed-size (typically 128 or 256 bits) is provided (P), this is then passed into an encryption function along with a secret key known only to individuals authorized to encrypt and decrypt the data. The key, typically at least 128 bits [13], is used to encrypt the data, producing an output of

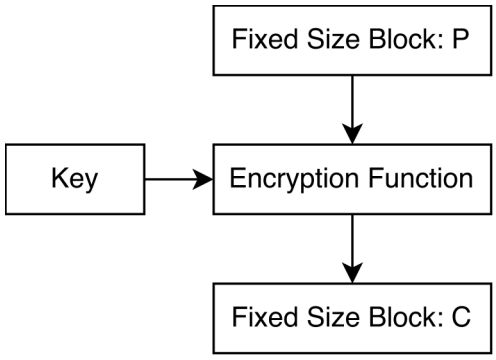

Figure 1: Block Cipher

ciphertext (C), also of a fixed size that is the same as the input size. All encryption and decryption in this thesis uses the Advanced Encryption Standard (AES).

While the typical block size for a block cipher is 128 or 256 bits, most use cases want to encrypt more than 128 or 256 bits of data. Therefore, there are multiple different modes of operations for accomplishing this with block ciphers, including: Electronic Code Book (ECB), Cipher Block Chaining (CBC), Output Feedback Mode (FB), Cipher Feedback Mode (CFB) and Counter Mode (CTR), among others [13]. These different modes of operation are available to facilitate different use cases. For example, CFB can be used to encrypt only m bits of data when the block cipher typically encrypts n bits (where $m < n$). This can be used for applications such as keystrokes at a terminal where we want to send data as soon as it is needed, and not wait until we have a full block of data [14]. For our investigation, we only used Counter Mode (CTR).

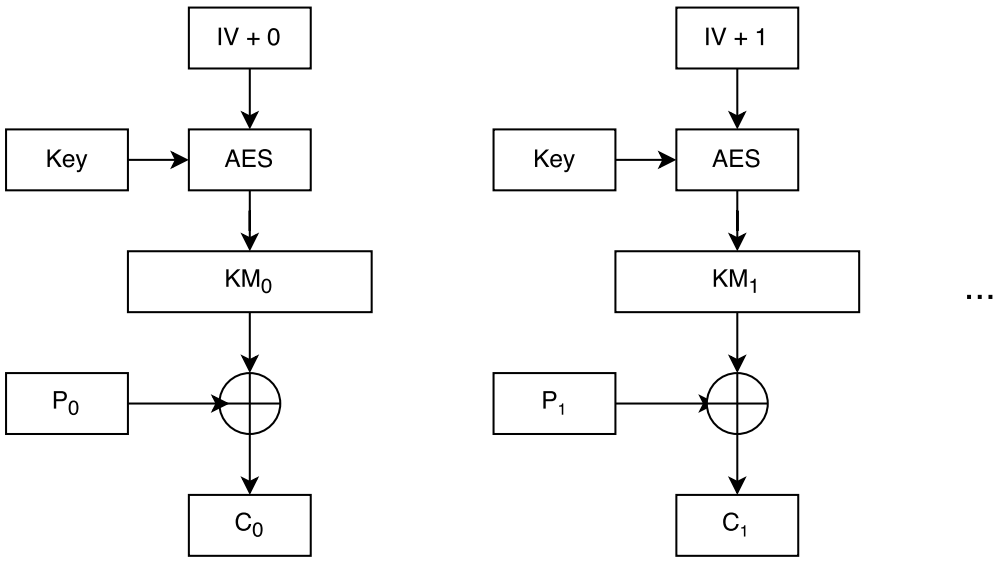

Figure 2: Counter Mode

Counter Mode (Figure 2) allows for encrypting/decrypting blocks of data without relying upon any other blocks. The only values that need to be known are the secret key (Key), initialization vector (IV) and the current block number (0, 1, …). At a high level, Counter Mode uses the secret key (Key) to encrypt the IV. This produces key material (KM) that is the same length as the block of plaintext that is being encrypted. After this key material is computed, it is a simple matter of performing the XOR between KM and the plaintext block being encrypted (P_0, P_1, …), to produce the ciphertext (C_0, C_1, …). If decryption is desired, the same steps are performed except, the key material, KM, is XOR-ed with the ciphertext to derive the original plaintext.

The IV is a random value used to inject non-determinism into the encryption process however, the value does not have to be secret. This is because, without knowing the secret key (Key), it is computationally infeasible to determine the key material (KM) that is produced by encrypting the IV with Key. The IV is incremented to prevent identical plaintext blocks at different positions from encrypting to the same ciphertext, which would leak information even without the secret key being revealed, and could open the door to the possibility of replay attacks. A replay

attack occurs when an adversary injects a previously transmitted and encrypted block of ciphertext into the current ciphertext being sent. The incremented IV helps to protect against this, since the same plaintext at different positions will not produce the same ciphertext, due to the use of a different IV, which will produce different key material.

2.2 RELATED WORK

2.2.1 CRYPTOGRAPHIC ACCESS CONTROLS

Access control is any mechanism that restricts access to a set of resources by analyzing permissions and deciding to allow or deny access based on those set of permissions. For example, a padlock is a very primitive access control mechanism. It restricts access to a resource, a chest for instance, by requiring that the entity that wants to access that resource have some permission – namely a key. If the key is presented to the access control mechanism, then access is granted to the chest and the padlock releases. This is the basic goal behind any access control model. Cryptographic access control is almost identical to the padlock example. Resources, e.g., files, are encrypted with a key that is known to only those users that have access to that data. Should a user wish to interact with the file, they must first provide the key to the access control mechanism, which will then decrypt the file so that it can be viewed and/or modified by the user. Thus, cryptographic access control reduces to a key derivation and/or distribution problem, of which, there are many proposed schemes [4].

A basic cryptographic access control scenario is described in Figure 3. Data (D_1, D_2, …, D_N) are assigned a label, typically denoting a grouping of individuals, for example, Employees.

There is also a key and data associated with that label. That key is used to encrypt the data ($E_K(D_N)$), and, without that key, an individual cannot view the data that was encrypted. This label can then be assigned to users and, through that assignment, those users will gain access to the key used to encrypt the data.

The problem with cryptographic access control is that, should a user need to be removed from a label, then all of the data they had access to must be re-encrypted so that they no longer have access to that data. Referring to Figure 3, if a user has their label revoked, then all the data associated with that label ($D_1 \ldots D_N$) must be re-encrypted with a different key. Thus, if a label has access to N items of data, then there must be N re-encryptions. As pointed out in [5], this incurs severe overhead as most labels have access to large quantities of data. There is also the case where a new key is computed for a label to maintain the freshness of the key. This also requires re-encrypting all the data with the new key.

Despite these issues, cryptographic access controls provide a high level of confidentiality due to the data being encrypted. This means that, unless the private key is leaked, it is computationally infeasible to reconstruct the original data from the encrypted data. So, a user can store encrypted data on an untrusted storage platform with a high degree of confidence that the plaintext data will remain confidential. This degree of confidentiality, provided by cryptographic access controls, has prompted research into how to negate the overheads associated with them.

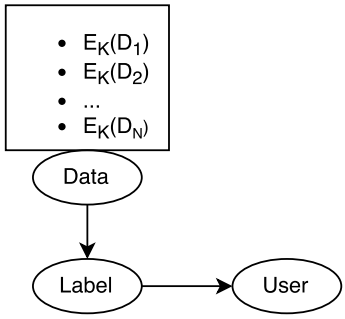

Figure 3: Simple Access Control Diagram

2.2.2 AES HARDWARE ACCELRATION

To address work related to increasing cryptographic performance, it is important to begin with a basic instantiation of the encryption/decryption process. This basic implementation follows a lifecycle of: reading data from disk, decrypting the data, modifying the data, re-encrypting the data and finally, writing the data back to disk. This basic example provides a foundation on which to lay current improvements to cryptography. This implementation exists entirely in software and is not accelerated at all by hardware.

To improve the performance of AES, Intel released a set of instructions to supplement its standard instruction set. The Advanced Encryption Standard New Instructions (AES-NI), achieves performance improvements by removing the need for references such as lookup tables, and instead, implementing AES entirely in hardware, while also mitigating several side channel attacks [6]. Utilizing AES-NI results in an AES encryption/decryption rate of approximately 2 cycles/byte [7]. This is a significant improvement over the software only implementation of AES, which has an encryption/decryption rate of approximately 100 cycles/byte [7]. There is an obvious drawback to this implementation that the underlying system and hardware must support these instructions. While this is a minor drawback, it is still a drawback that some systems will not be able to utilize this performance increase without the proper hardware. While the AES-NI was developed specifically to aid AES, there is the possibility of utilizing hardware not developed specifically for cryptography to further reduce the overheads associated with bulk encryption/decryption operations. An example of this type of hardware is the Hybrid Memory Cube.

2.2.3 HYBRID MEMORY CUBE

Hybrid Memory Cubes (HMC) have been developed to augment standard DRAM main memory by manipulating the layout of the DRAM. HMC consists of three-dimensional stacks of DRAM dies, layered one on top of the other, rather than the traditional DRAM layout. The goal of stacking the dies on top of each other is to increase the parallelism that can be achieved through main memory. All of the dies are then connected with Through-Silicon Vias (TSVs), instead of the standard pin connections used by current DRAM. This allows for extremely fast data transmission within the HMC itself. The HMC is subdivided into *quadrants* which are used for logically partitioning the HMC into four segments. These quadrants are further subdivided into *vaults*, which are analogous to a channel in current DRAM [19]. Vaults extend upward through the HMC from the bottom die to the top. These vaults contain *partitions*, with each partition containing several banks. Figure 4 provides a visual representation of the Hybrid memory cube layout.

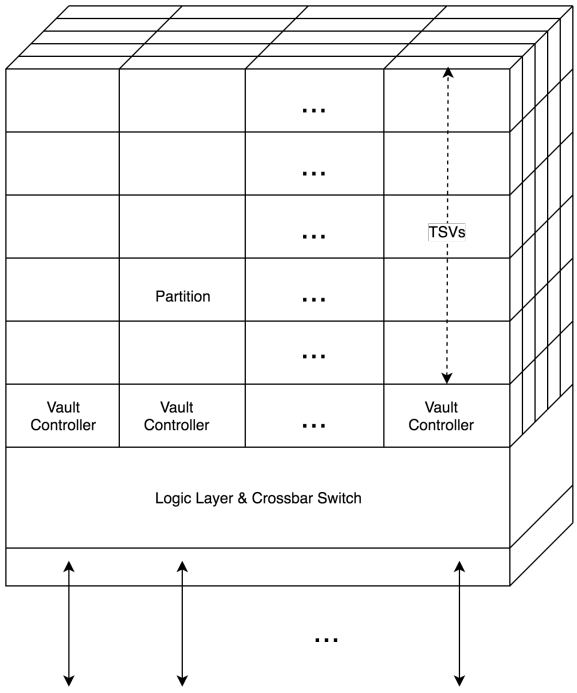

Figure 4: HMC architecture

Each vault within a HMC has a dedicated vault controller. This controller acts as the standard memory controller in current systems: handling the DRAM commands sent to the dies within the relevant vault [19]. Underlying these vault controllers is a logic layer, along with a crossbar switch. The crossbar switch is responsible for handling the routing of requests arriving on the links connecting the HMC to the CPU. Since any request can arrive on any link, some routing is required to send the request to the appropriate vault controller, which will handle the actual memory access. The logic layer controls the actual logic behind the routing while the crossbar switch provides the interface through which to achieve it. While the logic layer is responsible for handling the routing of requests, it also provides some unique functionality, namely in-memory computation.

While in-memory computation is not unique to HMC, the operations have never been implemented in an easy to use way [19]. HMC changes this by supporting a variety of Atomic Request Commands [19]. These atomic commands act on the data at the specified address, performing the appropriate command and then replacing the data, all without the need for bringing the data into the CPU. These commands range from basic arithmetic commands (add immediate, increment), to comparisons, to Boolean commands (AND, NAND, NOR, XOR) [10]. With these commands at the CPU's disposal, it is apparent that there is an opportunity to shift certain types of computation away from the CPU.

Of interest to cryptographic functions is the XOR command. Since XOR plays an integral part in many cryptographic constructions, the possibility of shifting that computation from the CPU into memory poses some interesting implications. First, it relieves the CPU of the burden of performing those instructions. Second, and more importantly, it removes that memory traffic from the bus. When a typical block of data is encrypted, that data needs to be read into the CPU, XOR-

ed with the key and then written back to memory. The ability to perform the XOR in memory allows the process to become streamlined: the CPU needs only to compute the key material before sending an XOR request with that key material to the appropriate memory address. This removes the reading of data into the CPU, essentially halving the memory traffic.

There are a couple of hurdles related to Hybrid Memory Cubes, specifically with the feasibility of testing and research. First and foremost is the fact that the technology is not readily available for testing. Therefore, investigations into the effectiveness of the technology must make do with the HMC specifications and simulators based off of those specifications [9][11][19]. However, another hurdle is that the final timing parameters for HMC simulators are not released due to these parameters being proprietary information. These two issues pose an impasse that severely effects the research that can be conducted into HMC.

Luckily, there has already been some research into the use of XOR in-memory with standard DRAM [8]. By leveraging this existing research, this thesis will examine the benefits available to cryptographic functions by exploiting the in-memory computation.

3.0 EXPERIMENTAL METHODOLOGY

We first describe how in-memory computation can be used with encryption before explaining how normal cryptographic functionality can be adapted for the usages pertinent to our investigation. We then pose three hypotheses that will drive our investigation, before finishing with a small test to confirm the intuition and motivation behind our hypotheses.

3.1 SCENARIOS

Re-encryption requires loading data into main memory and then sequentially accessing that data by bringing it into the CPU to decrypt it and then re-encrypt it with a new key. The data is then written back out to main memory. Our goal is to reduce this back and forth between the CPU and main memory. We can then compare the standard re-encryption, with both reads and writes, with a scenario where there are only writes, and the re-encryption takes place in-memory. While HMCs are not readily available, we can circumvent that issue by simulating everything up until the computation that would be executed within the HMC. We then count this cost, of performing the XOR in-memory, as negligible since all computation would occur within the HMC. Thus, we perform the re-encryption costs that would be experienced by the CPU and the memory traffic it would incur. These costs include computing the key material needed to re-encrypt the data, however, they do not include reading the data into the CPU, only the subsequent write of that key material out to main memory. We then compare those values to the costs seen when performing re-encryption in the typical way. With this setup, we can accurately represent the re-encryption

process that would take place given a main memory that could perform the in-memory computation.

3.1.1 STANDARD RE-ENCRYPTION

The baseline scenario that we start with is the standard re-encryption via Counter Mode. Standard re-encryption reads in an encrypted file (C) and decrypts each block (C_j) with the key material (KM_j) produced by encrypting the $IV + j$ with the secret key (Key). The output of the decryption process is a plaintext block P_j. This plaintext block is then re-encrypted by XOR-ing the plaintext with the new key material (KM'), produced by encrypting the new $IV' + j$ with the new key (Key'). Figure 5 demonstrates this process for a single block, $C_j \rightarrow C_j'$. We examine two versions of this standard re-encryption. The first version, full file re-encryption, decrypts the entire file with the old key material (KM), before re-encrypting it with the new key material (KM'). Cryptographic APIs make this first scenario very easy to implement and thus, it is the default way to re-encrypt data. The second version, block-by-block re-encryption, decrypts each block and immediately re-encrypts that block with the new key material (KM') before moving on to the next block. This method requires more work by the programmer however, it achieves better locality.

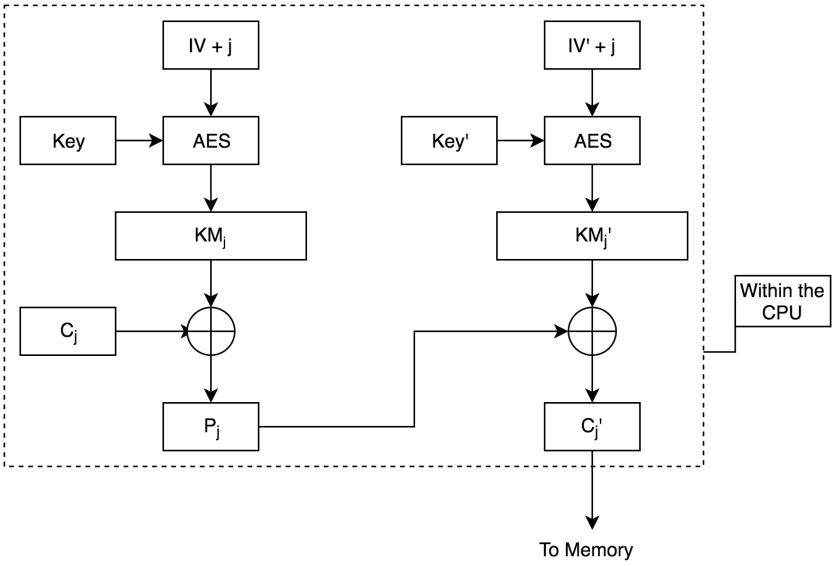

Figure 5: Re-encryption via Counter Mode for a single block j

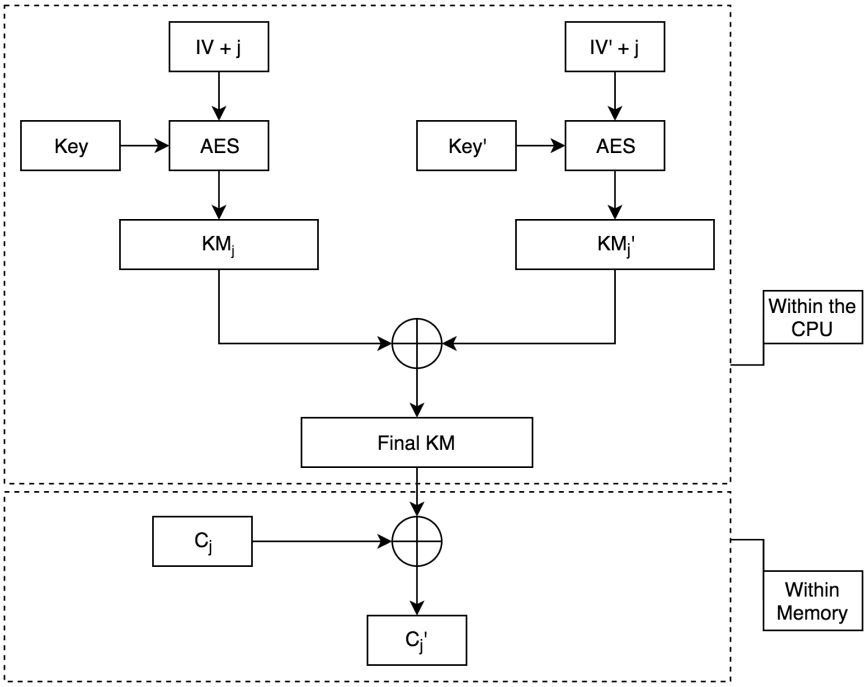

Figure 6: Keystream Re-Encryption

3.1.2 KEYSTREAM RE-ENCRYPTION

Keystream re-encryption is very similar to the block-by-block standard re-encryption. However, it is based on the observation that both decryption and re-encryption require XOR-ing the data with the key material. By slightly modifying the block-by-block standard re-encryption scheme, we can perform the re-encryption in a single XOR with the original ciphertext, which could then be performed in-memory (Figure 6). For this to work, the value that is passed to memory needs to effectively decrypt the ciphertext currently stored there and then re-encrypt that ciphertext with the new key (Key') and IV'. The decryption step within block-by-block standard re-encryption is represented in Equation 1, where KM_j represents the key material produced by encrypting $IV + j$ with the key (Key). Similarly, the encryption, from block-by-block standard re-encryption, is represented by Equation 2 where KM'_j represents the new key material produced by encrypting $IV' + j$ with the new key (Key'). Consequently, if in-memory re-encryption is desired, Equation 1 and Equation 2 must be combined to produce Equation 3. Since the XOR operation is commutative, we can perform the second XOR first. Thus, the desired value is $KM_j \oplus KM'_j$ and the ciphertext C_j will remain in memory. The CPU then only needs to compute the two sets of key material (KM and KM'), and perform an XOR between those. That value (Final KM) can then be sent to memory and XOR-ed with C_j there, to produce C_j'.

Equation 1: $\boldsymbol{C_j \oplus KM_j \rightarrow P_j}$

Equation 2: $\boldsymbol{P_j \oplus KM'_j \rightarrow C'_j}$

Equation 3: $\boldsymbol{C_j \oplus KM_j \oplus KM'_j \rightarrow C'_j}$

3.2 HYPOTHESES

Examining all these scenarios reveals several insights regarding the expected results when comparing the different re-encryption scenarios. First, looking at Figures 5 and 6 it is apparent that both modes of re-encryption will perform almost the same set of operations. The only difference is that keystream re-encryption executes one less XOR in the CPU. Therefore, from a computational standpoint, it is expected that these two modes will be roughly equivalent. The second insight is that, because the keystream re-encryption does not pull data into the CPU to perform the re-encryption, it will generate less memory traffic. This occurs because, with standard re-encryption, data must be fetched from memory and brought into the CPU to be re-encrypted. However, with keystream re-encryption, that data stays in memory and only the key material is computed in the CPU before that data is pushed out to memory.

Standard DRAM has an access time of roughly 50-70 nanoseconds [17]. Therefore, when the standard re-encryption requires data to re-encrypt, it must wait that long until the data has been properly fetched, then it can perform the re-encryption. Keystream re-encryption circumvents this latency by not bringing the data into the CPU. Further, since each block is independent of all other blocks (Counter Mode) then there is no need to wait for that request to even complete. This differs from the block independence during standard re-encryption because while that key material can be generated while the data is being fetched, the key material must still be XOR-ed with the data in the CPU. Therefore, there is still the latency of waiting for that data to arrive in the CPU, although, the CPU can begin computing the key material for the next block. Extrapolating from these insights produces multiple hypotheses regarding the expected outcome of our investigation.

- **Hypothesis 1:** Not moving data from memory to the CPU, which results in less stalling, as well as the CPU performing less computation should lead to an increase in throughput for keystream re-encryption compared to standard re-encryption.
- **Hypothesis 2:** The overall memory traffic will be reduced with keystream re-encryption due to it not bringing data into the CPU. This reduction in data on the bus should result in less interference on other processes.
- **Hypothesis 3:** The DRAM energy consumption for keystream re-encryption will be lower than the DRAM energy consumption for standard re-encryption due to the decreased amount of data reads from DRAM.

3.3 SIMPLE COMPARISON

As an initial investigation into this space, all three scenarios were run on a file already encrypted with an initial key (Key), and the timing statistics were gathered. The machine used for this benchmark had Dual Hyper-Threaded Six-Core 3.33GHz Xeon processors with 96GB of RAM running CentOS 5.5. Using the UNIX *time* command to time the process, all three scenarios were run on a 1GB file of random input produced via `dd if=/dev/urandom`. The results of these runs are summarized in Table 1.

The in-memory re-encryption had almost 20% better throughput (in MB/second) than the standard re-encryption variants. These results are in line with the proposed hypotheses, as the keystream re-encryption does indeed have a higher throughput. While this helps to verify the intuition behind Hypothesis 1, it does not provide enough information to make a judgement on

Hypotheses 2 or 3. A more in-depth approach is required to verify the effect that this mode of re-encryption has on the memory traffic and energy consumption of a system.

Re-Encryption	User	Sys	User + Sys	Filesize	Throughput (MB / sec)
Full File	20.339	3.883	24.222	1 GB	41.285
Block-by-Block	20.576	4.12	24.696	1 GB	40.492
Keystream	17.450	3.332	20.782	1 GB	48.119

Table 1: Simple Time Comparison of Test Cases

4.0 EVALUATION

We begin by detailing our experimental setup. We also discuss the cryptographic library used to implement our test cases, as well as the simulator that the test cases were run on. Since Hypothesis 2 deals with the impact on other processes, we include a discussion of other processes used before diving into the experiment itself and describing the different runs. We conclude with a discussion of our results.

4.1 SETUP

The first step in our setup is to describe the simulator used for our experiment, along with the tool used to instrument the test cases so that they could be run through the simulator. We then describe the cryptographic library used, and provide pseudocode with the appropriate API calls used from the cryptographic library.

4.1.1 HYBRID MAIN MEMORY SIMULATOR

For the actual experiment, HMMSim [2] was used to run the scenarios described in the previous section. HMMSim (Hybrid Main Memory Simulator) simulates the entire memory hierarchy, from the CPU, through the caches, into main memory, including a shared page table among multiple processes, and even down to the level of the banks and buses [2]. Since the hypotheses for this experiment deal with memory traffic, HMMSim was an ideal fit to analyze interactions throughout

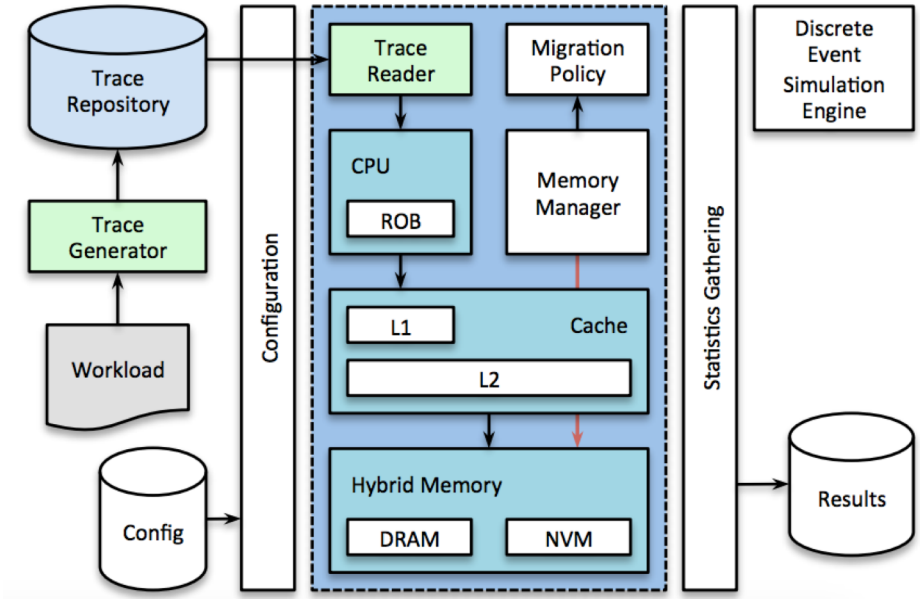

Figure 7: HMMSim Overview [2]

the memory hierarchy. While there are other simulators available, such as DRAMSim2 [20], the fact that HMMSim simulated the complete memory hierarchy, not just DRAM, makes it a desirable choice.

HMMSim's logical construction is laid out in Figure 7. HMMSim relies upon discrete event simulation to allow for scheduling of events anywhere within the system. Then, when that event completes, the component from whom that event originated will receive a callback [2]. HMMSim is a trace driven simulator, therefore, any workloads that need to be run in HMMSim must first be instrumented to generate traces. These traces consist of addresses, sizes and timestamps, subdivided into three categories (instruction, read & write), which are used to drive the simulator. HMMSim provides a tool, built using Intel's Pin [18], to help generate these traces. The tracing tool provided by HMMSim is also equipped with several options for modifying how the workload is instrumented. For instrumenting the scenarios for this experiment the two options changed were `-s` (starting point) and `-i` (instruction count). These values where changed to 750

million for the starting point and one billion for the instruction count. The starting point was set to 750 million instructions to provide confidence that the trace tool instrumented code relevant to the program being executed and not code dealing with starting and setting up the program. One billion was used as the instruction count because it provided a good tradeoff between capturing enough of the program for our investigation to make inferences, while also completing in a reasonable amount of time (the longest runs typically took around 45 minutes).

One drawback of Pin is that it does not instrument code outside the user level. Therefore, traces of operations such as file I/O do not accurately reflect the memory operations happening. A workaround for this problem is to manipulate the data in-memory rather than utilizing file I/O. This workaround, of performing the process in-memory rather than using file I/O, is a standard practice while tracing with Pin [3]. To accomplish this, any data from a file is read into an array, and then the program acts upon the data.

When configuring HMMSim, all default values were used except for those located in Table 2. These changes were made for a couple of reasons. First the value of `dram_ranks` was reduced from 8 to 4 due to the fact that 8 produced a memory configuration with an extremely high level of parallelism, not indicative of the standard memory found in systems nowadays. Secondly, the values of `dram_channels` and `dram_rows_per_bank` were each doubled to take the total system memory size from 4GB to 8GB. Again, this value was more in line with a standard system nowadays. All other configurations were left as the default value. However, it is worth it to point out a couple default values and idiosyncrasies of HMMSim.

When multiple traces are loaded into HMMSim, each trace is given its own CPU and, its own private L1 cache. For example, when 2 traces are loaded, there will be 2 CPUs and 2 private L1 caches. However, there will only be one L2 cache. This is because the default value for

`private_L2` is set to false. Thus, the L2 cache is shared among all traces. Also, while HMMSim is designed to simulate hybrid memory configurations, i.e., PCM and DRAM, the default memory configuration is set to DRAM only. To summarize, all simulations run on HMMSim in this thesis utilized a DRAM-only, 8GB main memory, with multiple private L1 caches (64KB) and a single shared L2 cache (2MB).

Parameter	Default	New
dram_ranks	8	4
dram_channels	1	2
dram_rows_per_bank	16k	32k

Table 2: HMMSim Configuration

4.1.2 MBED TLS

The scenarios described in Chapter 3 were implemented using Mbed TLS. Mbed TLS is an intuitive, simple, and lightweight SSL Library that is extremely easy to setup and use [22]. AES was chosen for two reasons. First, AES is the most popular symmetric key algorithm used today. This is due to its easy implementation and high performance. This performance leads to the second reason AES was chosen, namely, to exploit the AES-NI described in Section 2.2.2. Mbed TLS supports AES-NI by requiring a user to define `MBEDTLS_AESNI_C` within the *config.h* file. This tells Mbed TLS to utilize AES-NI, if the underlying system supports it. The API calls, and pseudocode, used to implement the scenarios are shown in Figures 8, 9, and 10.

All re-encryption methods iterate over the entire data size. The difference between the three methods is that both standard re-encryption variants must decrypt the data and then re-encrypt it with the new key (Key') and IV'. The full file standard re-encryption is described in Figure 8. It

```
// Decrypt the entire input array, consisting of the encrypted file
mbedtls_aes_crypt_ctr( old_key , filesize , … , input_buffer , output_buffer );
// Encrypt the entire decrypted array from previous step with the new <Key, IV>
mbedtls_aes_crypt_ctr( new_key , filesize , … , output_buffer , final_buffer );
```

Figure 8: Standard Full File Re-Encryption Pseudocode

```
for ( i = 0 ; i < filesize ; i += 16 )
{
        // Decrypt the section of input array, consisting of the encrypted file
        mbedtls_aes_crypt_ctr( old_key , 16 , … , input_buffer , output_buffer );
        // Encrypt the decrypted array from previous step with the new <Key, IV>
        mbedtls_aes_crypt_ctr( new_key , 16 , … , output_buffer , final_buffer );
}
```

Figure 9: Standard Block-by-Block Re-Encryption Pseudocode

```
for ( i = 0 ;  i < filesize ; i += 16 )
{
        old_material = mbedtls_aes_crypt_ecb( old_key , … , old_iv , … );
        new_material = mbedtls_aes_crypt_ecb( new_key , … , new_iv , … );
        keystream = new_material ^ old_material;
        increment(old_iv);
        increment(new_iv);
}
```

Figure 10: Keystream Re-encryption Pseudocode

reads the entire file into an array before decrypting the entire array. It then encrypts the entire decrypted array with the new key and IV. Block-by-block standard re-encryption is depicted in Figure 9, and re-encrypts 16-byte wide blocks of data at a time by first decrypting the block and then immediately encrypting the same. Both standard re-encryption scenarios differ from the keystream re-encryption (Figure 10), which only needs to compute the encryption of both the IV with Key (KM) and IV' with Key' (KM'), and then XOR those values together. It then writes that value to an array, simulating a stream of key material values that would be written to main memory in a system that supported in-memory computation. Since the XOR, and therefore the re-

encryption, would be performed in-memory, writing to an array is all that the keystream re-encryption scenario needs to accomplish.

We provided a workload for each of our re-encryption scenarios that consisted of twenty copies of *Moby Dick*[1] (total size: 24 MB), concatenated together in one file. We then instrumented our programs, using the Pin tool, to produce trace files that captured this re-encryption of the file. As mentioned before, these traces were offset by 750 million instructions before capturing one billion instructions.

4.2 SPEC BENCHMARKS

The Standard Performance Evaluation Corporation (SPEC) is a non-profit organization that maintains sets of benchmarks used to evaluate high performance computing systems [23]. The SPEC CPU2006 benchmark suite was used for this experiment and provides a set of roughly 20 different benchmarks. The applications within the CPU2006 suite were selected by SPEC to emphasize the performance of the CPU and memory architecture [21]. The emphasis on the memory architecture makes CPU20006 an ideal candidate for this experiment since this thesis is focused mainly on the interactions of memory traffic. The following benchmarks were selected from the CPU2006 set, representing a range of memory behavior: Bwaves (Fluid Dynamics), Mcf (Combinatorial Optimization), Milc (Physics: Quantum Chromodynamics), ZeusMP (Physics / CFD), CactusAMD (Physics / General Relativity), Leslie3d (Fluid Dynamics), Soplex (Linear Programming, Optimization), GemsFDTD (Computational Electromagnetics) and Libquantum

[1] https://www.gutenberg.org/files/2701/2701-h/2701-h.htm

(Physics: Quantum Computing) [21]. These benchmarks, including our three re-encryption scenarios, constitute the set of processes that will be run through HMMSim.

4.3 RESULTS

We now present the results of our experiment before proceeding to the results recorded for different combinations of the benchmarks and re-encryption scenarios before concluding with an account of the results of our power consumption tests.

4.3.1 SINGLE RUNS

The first step was to take the traces of all the selected benchmarks, listed in Section 4.2.2, along with the traces of our re-encryption scenarios, and run each trace individually through HMMSim. These solo runs provided an opportunity to baseline each benchmark, as well as each re-encryption scenarios. Each run was conducted using the HMMSim configuration described earlier in Section 4.1.1. Table 3 provides a subset of the statistics produced by HMMSim, focusing on computing speed and memory pressure. Instructions Per Cycle (IPC) and Memory Used are both statistics reported directly by HMMSim. While, Misses Per Kilo Instructions (MPKI), using statistics captured in the L2 cache, was calculated using Equation 4.

Equation 4: $(L2\_all\_misses * 1000) / total\_instructions$

4.3.2 IMPACT ON CO-RUNNING PROCESSES

After running each trace individually, we proceeded to co-running processes. HMMSim can run multiple traces concurrently to simulate multiprocessing. Therefore, each trace of a SPEC benchmark was run jointly with the full file standard re-encryption trace, block-by-block standard re-encryption, and finally, with the keystream re-encryption trace (3 simulations per benchmark). This was done to assess the change in memory traffic imposed by each re-encryption process. The IPC calculations resulting from these joint runs are listed in Table 4. The values reported are total IPC values, calculated by adding together the IPC for each of the two concurrent traces (SPEC benchmark and re-encryption scenario) being run during each simulation. This provides an aggregate IPC value that can be used for comparison. For example, the Bwaves benchmark was paired with the full file standard re-encryption trace and run through HMMSim, which then reports statistics on a per trace basis. The IPC values for each trace are then aggregated to get a total IPC for the entire system across both traces (e.g., Bwaves plus full file standard re-encryption). These resulting aggregate IPC values are compared in Table 4. In the final two rows, both tables also report the speedup, in terms of IPC, when using keystream re-encryption instead of standard re-encryption.

4.3.3 THROUGHPUT TEST

While HMMSim provides a very nice overview of the memory architecture it lacks a convenient way to calculate the throughput of the test scenarios. Therefore, the earlier throughput test, using the *time* command, was repeated. However, both block-by-block standard re-encryption and keystream re-encryption were run 100 times on file sizes of 1MB, 10MB, 100MB and 1000MB.

The results of the *time* command were captured and the *user* and *sys* fields were added together to produce a total execution time. These results are reported in Figure 11 and Table 6. This test was done entirely independent of HMMSim. The actual code, not the traces generated by Pin, were executed and timed to produce an empirical value for the throughput achieved by the different re-encryption scenarios.

Benchmark	Bwaves	Mcf	Milc	Zeusmp	CactusADM	Leslie3d	Soplex	GemsFDTD	Libquantum	Standard (Block-by-Block)	Standard (Full File)	Keystream
IPC	1.5	0.57	0.9	1.37	1.89	1.25	0.87	0.79	0.98	3.76	3.75	4.00
MPKI (L2)	19.56	71.74	24.82	11.85	5.37	16.71	23.23	18.61	38.02	0.21	0.43	0.12
Memory Used (MB)	972	1,758	713	526	657	130	162	871	86	51	51	26

Table 3: Individual Benchmark Statistics

Benchmark	Bwaves	Mcf	Milc	Zeusmp	CactusADM	Leslie3d	Soplex	GemsFDTD	Libquantum
Standard Full-File Re-Encryption Total IPC	5.01	3.85	4.38	4.98	5.41	4.79	4.27	4.39	4.42
Standard Block-by-Block Re-Encryption Total IPC	5.2	4.08	4.57	5.14	5.58	4.96	4.44	4.54	4.63
Keystream Re-Encryption Total IPC	5.41	4.40	4.79	5.32	5.81	5.17	4.70	4.74	4.84
KS / Full-File	1.08	1.14	1.09	1.07	1.07	1.08	1.10	1.08	1.10
KS / Block-by-Block	1.04	1.078	1.048	1.035	1.041	1.042	1.059	1.044	1.045

Table 4: Block-by-Block Re-Encryption Dual Benchmark Statistics

Benchmark	Bwaves	Mcf	Milc	ZeusMp	CactusAMD	Leslie3d	Soplex	GemsFDTD	Libquantum	Standard (Full File)	Standard (Block-by-Block)	Keystream
DRAM Reads	19564250	71735883	24819558	11845196	5370566	16709709	23232275	18605602	38022576	430318	207446	122931
DRAM Writes	607062	2231766	8364243	4122291	1551742	3847670	9502741	2453819	655	198084	173994	89845
Total Energy (mJ)	6325	23197	11162	5381	2308	6778	11139	6791	11806	215	136	75

Table 5: Energy Consumption Statistics

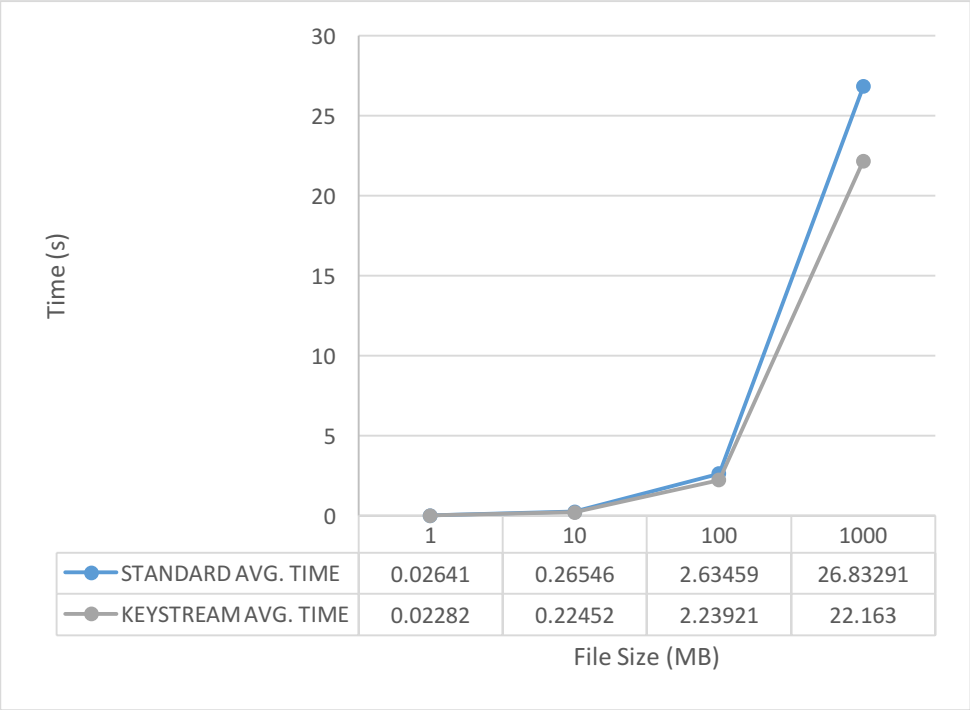

Figure 11: Throughput Comparisons

File Size (MB)	Standard	Keystream	Percent Change
1	37.864	43.821	0.157
10	37.670	44.539	0.182
100	37.957	44.659	0.177
1000	37.268	45.120	0.211
Average	37.690	44.535	0.182

Table 6: Throughput Calculations in MB / sec

4.3.4 ENERGY TEST

Calculating energy consumption does not require running any additional simulations. Rather, the DRAM read and write statistics, gathered during the initial solo runs, were used to calculate this value. The value of 0.000310518 mJ / access was used for DRAM reads and 0.000413123 mJ / access was used for DRAM writes [15]. This lead to a straightforward calculation of total DRAM energy costs; accesses multiplied by the cost of each access. As a note, these values do not include the cost of performing the XOR in-memory. These values are reported in Table 5 and are broken up into separate read and write costs before reporting the final energy consumption.

4.4 DISCUSSION

Hypothesis 1: Not moving data from memory to the CPU, which results in less stalling, as well as the CPU performing less computation should lead to an increase in throughput for keystream re-encryption compared to standard re-encryption.

For evaluating Hypothesis 1, Figure 11 and its associated table provide a useful insight into the throughput advantages of keystream re-encryption. The first result to remark on is the fact that the throughput for both standard and keystream re-encryption is linear. As the size of the file is increased by a factor of 10, the time it takes to re-encrypt that file is also increased by a factor of 10. This is expected, as each block is re-encrypted without any dependencies on other blocks and, therefore, the time to re-encrypt each block is constant.

We can now look at the throughput values calculated during the *time* test. Looking at Table 6 on page 30, we can see that the throughput achieved by keystream re-encryption is, on average, 18% higher than that of block-by-block standard re-encryption. Keystream re-encryption can successfully re-encrypt more data per second than block-by-block standard re-encryption. These results confirm Hypothesis 1; removing the need to pull in data from memory for re-encryption, serves to increase the throughput over standard re-encryption.

Hypothesis 2: The overall memory traffic will be reduced with keystream re-encryption due to it not bringing data into the CPU. This reduction in data on the bus should result in less interference on other processes.

On its surface, Hypothesis 2 is straightforward and trivial to confirm. Standard re-encryption involves both reads and writes, while keystream re-encryption has only writes. Thus, keystream re-encryption has less traffic to and from memory. However, there is something deeper to get at when examining the results. Table 4 contains the results of running full file standard re-encryption with each benchmark, block-by-block standard re-encryption with each benchmark, and keystream re-encryption with each benchmark. Comparing these values for each benchmark reveals an interesting trend: higher IPC values when using keystream re-encryption compared to either standard re-encryption variant. By itself, this seems to be consistent with what is expected. Less memory traffic means less contention between concurrent processes and, therefore, reduced latencies which lead to higher IPC values. This is consistent with Hypothesis 2 in that, due to keystream having less memory traffic, it should produce a higher IPC.

Another interesting portion is seen when comparing the increase in total IPC (the benchmark IPC + test case IPC) between the standard re-encryption and keystream re-encryption. These values are reported in the last two rows of results in Table 4. Keystream re-encryption produces, on average, a 10% increase in IPC over full file standard re-encryption produces, while also producing a 5% increase in IPC, on average, compared to block-by-block standard re-encryption. These changes in IPC are also displayed in Figure 12. While this is not an extreme increase in IPC it still shows a consistent trend of keystream re-encryption having a higher aggregate IPC, even when compared to the block-by-block standard re-encryption. More importantly, it was achieved by reducing a small amount of memory traffic. Table 6 lists the DRAM reads and writes performed by each trace in HMMSim and it shows that the normal and keystream re-encryption perform 1 to 2 orders of magnitude less memory operations than the other benchmarks. Yet that small reduction in memory traffic from standard re-encryption to keystream, produces a significant increase in the IPC.

One more result to draw attention to is the L2 MPKI of the solo runs, located in Table 3. MPKI is a good indicator of memory pressure and we can see that all three re-encryption scenarios have extremely low values. Also, when examining these values between the two standard re-encryption scenarios, we can see the effect of locality. By performing efficient re-encryption (block-by-block) instead of using the naïve full file approach, we can effectively halve the MPKI. This is because full file re-encryption does not preserve locality: a miss is incurred when decrypting a block, and a second miss occurs for each block when it is fetched again for re-encryption. The block-by-block scenario avoids this by performing the decryption and encryption immediately on the same block, thus preserving locality. Keystream re-encryption almost halves the block-by-block MPKI value again because it is removing all ciphertext reads, while the block-

by-block scenario requires both the read into the CPU and the subsequent write out. Therefore, neither re-encryption has much impact on the memory traffic of concurrent processes. Even with re-encryption being computationally bound, there is still a reduction in traffic with keystream and an overall increase in IPC.

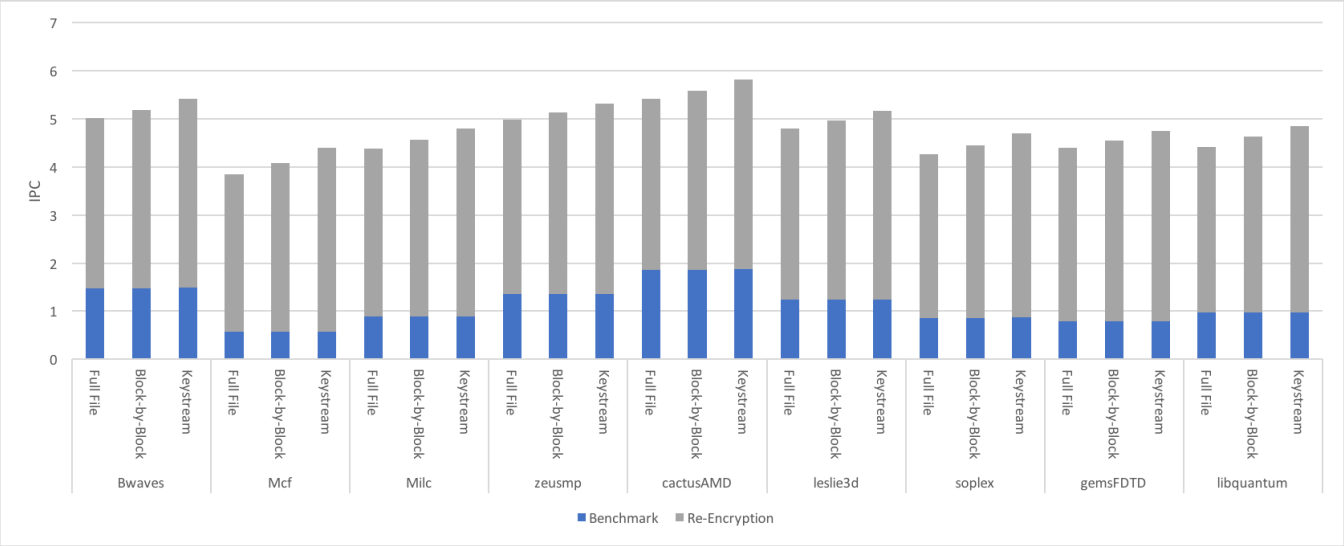

Figure 12: Aggregate IPC Comparisons

Hypothesis 3: The DRAM energy consumption for keystream re-encryption will be lower than the DRAM energy consumption for standard re-encryption due to the decreased amount of data reads from DRAM.

Table 5 calculates the total energy used by DRAM reads and writes for all the SPEC Benchmarks as well as the standard re-encryption and keystream re-encryption. As mentioned before, since re-encryption is computationally bound there is very little energy consumed in DRAM. However, what is notable is the improvement in energy consumption by switching to keystream re-encryption from full file standard re-encryption (65%) and even from block-by-block standard re-encryption (44%). This introduces some interesting scenarios, particularly if the primary process being run is some form of encryption. This is because the amount of energy spent

on DRAM accesses during re-encryption is miniscule compared to the energy consumed by the SPEC benchmarks and, thus, if they are run at the same time, re-encryption produces almost no change in total energy consumption. However, if this is a machine with a high cryptographic workload, then there is a significant decrease in energy consumption due to the use of keystream re-encryption over standard re-encryption. Additionally, these savings are before any consideration of the fact that HMC technology is more energy efficient than standard DRAM due to the short distance (via the TSVs) that data needs to travel [19].

5.0 CONCLUDING REMARKS

We finish this thesis by summarizing the results seen and conclusions drawn from the experiments, before looking at future work. Specifically, we put forth the possibility of shifting even more computation off of the CPU and into memory.

5.1 CONCLUSIONS

Overall, Hybrid Memory Cube technology and other forms of in-memory computation possess some interesting and useful properties to aid in cryptography. The ability to move computation from the CPU and perform it in memory allows for reductions in memory traffic. This reduction in memory traffic not only serves to increase throughput by a significant amount (18%), but also reduces the impact that re-encryption has on the rest of the system (increasing IPC by 5-10%) while also requiring less energy due to the decreased number of DRAM accesses. These results show potential to increase the efficiency of cryptographic access control mechanisms implemented on untrusted storage platforms and thus make those mechanisms more viable for real world applications. By increasing the throughput, there will be a direct correlation to a decrease in the time required to re-encrypt data. Additionally, the lower level of memory traffic will have less impact on other processes while also supporting faster re-encryption due to not having to bring the data into the CPU. Finally, keystream re-encryption will also provide lower energy consumption even before factoring in the lower energy costs of new memory technology like HMC.

5.2 FUTURE WORK

While this thesis only explored the ability to exploit in-memory XOR operations, there are other operations available within an HMC. Just as AES-NI is tightly coupled with the hardware, there exists the possibility of tying cryptographic functionality directly into the HMC. Specifically, if an instruction stream could be computed using only the HMC Atomic Command Requests, then all computation could be moved off the CPU and lead to further reductions in memory traffic. There already exist commands for incrementing a value within the HMC specification. Therefore, if the AES encryption could be carried out in memory, then the IV could be incremented in-memory as well as the XOR, leading to the entire keystream being generation in memory. Even in the likely scenario that AES encryption cannot be implemented entirely using HMC commands, a small dedicated controller could be placed close to the HMC to handle the cryptographic functionality not available within the HMC. With this type of implementation, all the memory traffic and a significant portion of the computation is removed from the CPU. While this may seem like an extreme implementation, not only adding a different memory architecture but also another controller, it brings about some interesting implications. Most notably, data travel is extremely minimal. An encrypted file is loaded into main memory, then each block of the file is XOR-ed with the keystream produced by the micro controller (possibly with help from the HMC to compute the keystream) before returning the data to its memory address. All of that takes place without the data ever leaving the HMC. While this has significant performance benefits, as the latency within the HMC is even less than the latency between the HMC and CPU, there are also added security benefits. Data only ever exists in main memory - it does not travel between main memory and the CPU and it is never in plaintext. There are questions regarding whether the keystream leaks information however, those questions are beyond the scope of this thesis.

BIBLIOGRAPHY

[1] Bishop, M. (2003). Computer security: Art and science. Boston: Addison-Wesley.

[2] Bock, S., Childers, B. R., Melhem, R., & Mosse, D. (2015, August). HMMSim: a simulator for hardware-software co-design of hybrid main memory. In Non-Volatile Memory System and Applications Symposium (NVMSA), 2015 IEEE (pp. 1-6). IEEE.

[3] BZIP 401.bzip2 SPEC CPU2006 Benchmark Description. (n.d.). Retrieved October 14, 2016, from https://www.spec.org/auto/cpu2006/Docs/401.bzip2.html

[4] Crampton, J., Martin, K., & Wild, P. (2006). On key assignment for hierarchical access control. In 19th IEEE Computer Security Foundations Workshop (CSFW'06) (pp. 14-pp). IEEE.

[5] Garrison III, W. C., Shull, A., Myers, S., & Lee, A. J. (2016). On the Practicality of Cryptographically Enforcing Dynamic Access Control Policies in the Cloud.

[6] Gueron, S. (2010). Intel® Advanced Encryption Standard (AES) New Instructions Set. Intel Corporation.

[7] Guo, G. L., Qian, Q., & Zhang, R. (2015, August). Different Implementations of AES Cryptographic Algorithm. In High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conference on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on (pp. 1848-1853). IEEE.

[8] Hamdioui, S., Xie, L., Nguyen, H. A. D., Taouil, M., Bertels, K., Corporaal, H., ... & van Lunteren, J. (2015, March). Memristor based computation-in-memory architecture for data-intensive applications. In Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (pp. 1718-1725). EDA Consortium.

[9] HMCSim. (n.d.). Retrieved June 1, 2016, from http://gc64.org/?page_id=56

[10] Hybrid Memory Cube Consortium. (2015). Hybrid Memory Cube Specification 2.1. Last Revision Oct.

[11] Hybrid Memory Cube. (n.d.). Retrieved December 08, 2016, from https://www.micron.com/products/hybrid-memory-cube

[12] Identity Theft Resource Center. (2016, December 06). Retrieved December 08, 2016, from http://www.idtheftcenter.org/Data-Breaches/data-breaches.html

[13] Kaufman, C., Perlman, R., & Speciner, M. (2002). Network security: Private communication in a public world. Upper Saddle River, NJ: Prentice Hall PTR.

[14] Lee, A. J. (2016, November 11). Symmetric Key Cryptography. Lecture.

[15] Lee, B. C., Ipek, E., Mutlu, O., & Burger, D. (2009, June). Architecting phase change memory as a scalable dram alternative. In ACM SIGARCH Computer Architecture News (Vol. 37, No. 3, pp. 2-13). ACM.

[16] Mell, P., & Grance, T. (2011). The NIST definition of cloud computing.

[17] Patterson, D. A., Hennessy, J. L., & Hennessy, J. L. (2012). Computer organization and design: The hardware/software interface. Waltham, MA: Morgan Kaufmann.

[18] Pin 3.0 User Guide. (n.d.). Retrieved October 07, 2016, from https://software.intel.com/sites/landingpage/pintool/docs/76991/Pin/html/

[19] Rosenfeld, P. (2014). Performance exploration of the hybrid memory cube.

[20] Rosenfeld, P., Cooper-Balis, E., & Jacob, B. (2011). DRAMSim2: A cycle accurate memory system simulator. IEEE Computer Architecture Letters, 10(1), 16-19.

[21] SPEC CPU2006 Documentation. (n.d.). Retrieved December 01, 2016, from https://www.spec.org/cpu2006/docs/

[22] SSL Library mbed TLS / PolarSSL. Retrieved November 01, 2016, from https://tls.mbed.org/

[23] Standard Performance Evaluation Corporation. (n.d.). Retrieved December 01, 2016, from https://www.spec.org/