Parallel Cube Testing on GPUs

by

Sudarshan Rao

Supervised by

Prof. Alan Kaminsky
Associate Professor
Committee Chair

Prof. S. Radziszowski
Professor
Reader

Prof. E. Hemaspaandra
Professor
Observer

Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

June 2010
Project Report Release Permission Form

Rochester Institute of Technology
B. Thomas Golisano College of Computing and Information Sciences

Title: Parallel Cube Testing on GPUs

I, Sudarshan Rao, hereby grant permission to the Wallace Memorial Library reproduce my project in whole or part.

Sudarshan Rao

Date
The project “Parallel Cube Testing on GPUs” by Sudarshan Rao has been examined and approved by the following Examination Committee:

Prof. Alan Kaminsky  
Associate Professor  
Project Committee Chair

Prof. S. Radziszowski  
Professor

Prof. E. Hemaspaandra  
Professor
Acknowledgments

I would like to thank Prof. Alan Kaminsky for his invaluable help during the course of this project. His advise during the design of the project was crucial in creating an extremely flexible framework. We have also used his Parallel Java library for performing some of the statistical analysis. I would also like to extend my gratitude to Prof. Stanislaw Radziszowski for his suggestions on the project document.
Abstract

As computers become more involved in our day to day lives, computer security becomes more of a necessity. Cryptographic primitives are the building blocks of protocols that guarantee safety of our data. Given their frequent use, they must be studied for vulnerabilities extensively. Considering improvements in construction of these primitives, tools that study them must change too.

In this project, we develop a framework for performing property testing on Graphics Processing Units (GPUs). The testing technique used is a relatively new technique known as cube testing. We study two new primitives, Keccak and Threefish, and an older primitive, AES. The framework developed makes use of NVIDIA’s CUDA framework to get a significant performance boost by parallelization. We present the results of a balance test on the three primitives and show that the GPU does indeed provide considerable speedup.
Contents

Acknowledgments .......................................................... iv

Abstract .............................................................................. v

1 Introduction ........................................................................ 1
  1.1 Background ................................................................. 1
  1.2 Our Contributions ....................................................... 1
  1.3 Overview ........................................................................ 1

2 Background .......................................................................... 3
  2.1 Hash functions ............................................................. 3
    2.1.1 Types of hash functions ........................................... 3
  2.2 Block Ciphers ............................................................... 5
  2.3 NIST SHA-3 contest ....................................................... 6
  2.4 Cube Attack ................................................................. 7
    2.4.1 GF(2) operation ...................................................... 7
    2.4.2 The original attack .................................................. 8
    2.4.3 Cube Testing .......................................................... 9
  2.5 CUDA ............................................................................ 10
    2.5.1 Thread Hierarchy ..................................................... 12
    2.5.2 GPU Memory ......................................................... 13
  2.6 \(\chi^2\) Test ................................................................. 13

3 Framework and Experiments ............................................. 15
  3.1 The Primitives ............................................................. 15
    3.1.1 AES ....................................................................... 15
    3.1.2 Threefish .............................................................. 15
    3.1.3 Keccak ................................................................. 17
    3.1.4 Keccak-f[1600] ....................................................... 18
  3.2 The Design of the Framework ......................................... 18
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1 Overall Design</td>
<td>18</td>
</tr>
<tr>
<td>3.2.2 kernel.h</td>
<td>19</td>
</tr>
<tr>
<td>3.2.3 &lt;primitive&gt;.h</td>
<td>22</td>
</tr>
<tr>
<td>3.3 Experiments on the Primitives</td>
<td>22</td>
</tr>
<tr>
<td>3.3.1 Balance Test</td>
<td>22</td>
</tr>
<tr>
<td>3.3.2 Independence Test</td>
<td>24</td>
</tr>
<tr>
<td>3.4 Experiments</td>
<td>25</td>
</tr>
<tr>
<td>3.4.1 Tests on Keccak-f[1600]</td>
<td>25</td>
</tr>
<tr>
<td>3.4.2 Tests on AES</td>
<td>26</td>
</tr>
<tr>
<td>3.4.3 Tests on Threefish</td>
<td>28</td>
</tr>
<tr>
<td>3.5 Timing Measurements</td>
<td>32</td>
</tr>
<tr>
<td>4 Future work</td>
<td>36</td>
</tr>
<tr>
<td>5 Conclusions</td>
<td>37</td>
</tr>
<tr>
<td>Bibliography</td>
<td>38</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

Cryptographic primitives such as block ciphers, hash functions are such an integral part of everyday computing that they are always presumed to be involved in some way. The security of such primitives is paramount as we trust our most private data to these pieces of software. The most conventional and intuitive way to study their security is to try to break them. If such attempts are unsuccessful, then the primitive can claim to be secure.

As cryptographic primitives were being subject to attacks, the designers began inventing newer functions that were resistant to the older attacks. Newer and more sophisticated techniques to study the security of these primitives were needed. The designers of the newer hash functions also increased the size of the hashes, thus requiring the attacker to perform more computations than were practical.

1.2 Our Contributions

In this project, we use a new technique known as cube testing that was introduced in [2] to study three cryptographic primitives. We provide detailed descriptions on cube testing, and how it can be used successfully. We also detail the three primitives that were studied in this project. We describe techniques that can be used to perform statistical tests on the primitives.

The project also presents a framework for performing cube testing on GPUs which can significantly improve performance. The project presents results which show how fast the GPU accelerates the computations.

1.3 Overview

Chapter 2 provides some background into the project. Section 1 and Section 2 discuss hash functions and block ciphers. Section 3 provides some insight into the NIST SHA-3 contest.
Section 4 explains the cube attack, its operation and how it can be used for cube testing. Section 5 explains the CUDA architecture.

Chapter 3 goes into the project itself. In the first section, we will describe how the three primitives that we have chosen work. Section 2 will describe the design of the framework for cube testing. Section 3, provides data on the experiments. In Section 4, we perform some timing measurements.

Chapter 4 concludes the report providing some future work and references.
Chapter 2

Background

2.1 Hash functions

A hash function is a function that takes an arbitrarily sized message and reduces it to a fixed size message digest in a deterministic manner. Hash functions have many uses such as in digital signatures, message authentication codes etc. Cryptographically secure hash functions obey three important properties. $n$ here represents the size of the output of the hash function.

**Preimage resistance** Given the output of a hash function, it must not be feasible to reconstruct the input message. A brute force search can find a preimage in $2^n$ operations.

**Collision resistance** It must be computationally infeasible to find two messages that hash to the same value. A brute force search can find a preimage in $2^{n/2}$ operations.

**Second preimage resistance** Given a message and its hash, it must be computationally infeasible to find another message that is different from the first message that hashes to the same value. A brute force search can find a preimage in $2^n$ operations.

Brute force attacks on hash functions obey the birthday paradox. As mentioned above, finding collisions using brute force takes $2^{n/2}$ operations. As the size of the hash functions increases, brute force attacks are not possible. Newer attacks use more sophisticated techniques to reduce the time taken to attack the primitive.

2.1.1 Types of hash functions

**Merkle-Damgård construction**

Most of the hash functions designed are based on a popular technique known as the Merkle-Damgård construction [14] [6]. This construction uses a compression function that is used repeatedly to get the hash. If the compression function is designed with the above mentioned properties, then the entire hash function obeys the three properties. Let the compression function, say $f$, take a $m$ bit input called the *chaining value* and a message block of size $t$ and reduces it down to $n$ bits. The message, say $x$, which is to be hashed is split into $t$
Figure 2.1: Merkle-Damgård Construction

bit blocks. Let \([x_1, x_2, \ldots, x_k]\) be the message blocks. Initially an \(m\) bit initialization value and the first message block is run through the compression function. The output is now fed to the compression function as the chaining value along with the second message block. The compression function is run as long as there are message blocks to compress. When there are no more message blocks, the output of the compression function is the hash value. Figure 2.1 shows how the construction works.

The Merkle-Damgård construction provided a very good template for hash function designs. All that was needed was a good compression function. But, as it was later discovered this would lead to an attack that would cripple even the most sophisticated compression functions. The attack, known as a length extension attack, simply uses the hash value as a chaining value to hash another message. With this new hash value, the attacker can pretend to be the original message sender.

### Block cipher based hash functions

These hash functions use a block cipher as a component in its construction. The hash function is still based on a Merkle-Damgård contraction, but the compression function is now a block cipher. There are many ways that a block cipher can be used as a compression function. In the NIST SHA-3 contest, there are a few hash functions that are designed in this manner. Threefish, the block cipher, is used as a component in Skein [9], the hash function.

### Sponge construction

Sponge construction[4] based hash functions have the advantage that the output can be of any length. This design of hash function is relatively new. Sponge construction based hash
functions have two phases. In the *absorbing phase* the message is divided into blocks which are gradually added to the state and permuted. Once all the message blocks are added into the state, then the state is squeezed out for as much data as is needed.

Figure 2.2 shows a simple illustration of a sponge function. The message is divided into fixed length blocks of size $r$. Initially, an initialization vector is used as an input to the permutation $f$. The first message block is then mixed into the state, and the new state is run through $f$. This process is repeated until every message block has been mixed into the state. This marks the end of the absorbing phase. The first $r$ units of the state is the first output block. The permutation $f$ is run until the desired number of output blocks are got. It has been proved that if $f$ is indistinguishable from a random oracle, then the entire sponge function is indistinguishable from a random oracle.

### 2.2 Block Ciphers

Block ciphers are cryptographic primitives that encrypt fixed blocks of data. Block ciphers have three components:

**Plaintext** This is the secret message that needs to be encrypted. The length of the plaintext must be equal to the block length of the block cipher.

**Key** This is a secret key shared by the two parties exchanging the data. The secret key encrypts the data in a unique fashion, and also allows the receiver to decrypt the data.

**Ciphertext** This is the message that has been encrypted. The ciphertext usually looks like random bits of data. The length of the ciphertext is equal to the block length of the block cipher.
The security of block ciphers rests on the fact that without the secret key it is computationally infeasible to retrieve the plaintext given only the ciphertext. Most attacks try to get the secret key from a given set of data.

One of the first block ciphers that was standardized was the Data Encryption Standard[19]. The DES algorithm was invented by Horst Feistel at IBM, and was standardized by the National Institute of Standards and Technology(NIST) in 1977. The DES algorithm used a 56-bit secret key, which many prominent cryptographers considered to be very weak. Eventually DES was broken in the first half of 1990’s.

The Advanced Encryption Standard(AES) was the successor to the first standardized block cipher DES. In 1997, the National Institute of Standards and Technology(NIST) announced a contest to determine the next standard block cipher. The NIST encouraged entries to the contest, and submissions were studied closely to uncover any flaws in their design. The idea behind this process was that after an open investigation into all the submissions, the best block cipher would be the one that was the most secure. The candidate block ciphers needed to be secure as well as fast on a variety of hardware. After shortlisting the entries to about 15 Round 1 winners, extensive performance analysis was done. Finally, 5 block ciphers were chosen as the AES finalists in August 1999. They were Rijndael, Serpent, Twofish, RC6 and MARS. In April 2000, Rijndael was chosen as the winner of the contest and standardized as the Advanced Encryption Standard[5].

2.3 NIST SHA-3 contest

Although hash functions are widely used, research into their design and security was absent for the better part of the last decade. MD5 was completely broken[23], and its successor SHA-1 was showing signs of weakness[22]. Despite the warning of MD5 being broken, it was still used by many. In order to drive the point home, there were many papers which showed that MD5 use could be catastrophic. A new form of attack known as the herding attack[13], was used to creatively demonstrate that MD5 was unsafe to use. A more serious attack[20] showed that rogue Certificate Authority could be created if MD5 was used. These new attacks were also showing signs of success on SHA-1. The NIST decided that a new standard for a hash function was necessary, and announced a contest similar to the one that was used to determine the new AES.

The NIST called for hash functions that were free, adhered to standard sizes, hardware etc., must offer security against a set of strong attacks. The deadline for submissions was October 31st 2008. Out of the original 64 submissions, 13 were rejected. The remaining 51 hash functions, advanced to Round 1, and were subjected to attacks by prominent cryptographers. On July 2009, the NIST released 14 candidates that advanced to Round 2. These 14 hash function will now be subject to more cryptanalysis, and extensive performance testing. Both Skein[9] and Keccak[3] are among the Round 2 candidates. Details on the SHA-3 contest can be obtained at the SHA-3 Zoo[8], and the NIST webpage on the SHA-3 contest[15].
Table 2.1: $X + Y$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>$X + Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.2: $X + Y$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>$X \ast Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

2.4 Cube Attack

In 2008, Itai Dinur and Adi Shamir came up with a new cryptographic attack, known as the cube attack[7], which was a new form of an algebraic attack. The cube attack was successful on primitives which are based on a low degree polynomial over GF(2). They successfully mounted such an attack on a reduced version of Trivium, a stream cipher. The attack works by performing a one time preprocessing phase on the black box, to determine the queries which need to made to the black box during the actual key recovery phase. If the degree of the polynomial of the black box is $d$, then the cube attack can recover the key with roughly $2^{d-1}$ evaluations of the black box. If $d$ is sufficiently low, then the attack can be practically mounted.

2.4.1 GF(2) operation

Let us look at addition and multiplication operations in GF(2).

Let $X$, $Y$ be two boolean variables. Table 2.1 shows the truth table for the operation $X + Y$.

The above table tells us that in GF(2) addition of boolean variables is the same as the XOR of boolean variables.

Table 2.2 shows the truth table for the operation $X \ast Y$. The table tells us that multiplication of two boolean variables is the same as AND of the variables.
2.4.2 The original attack

The cube attack works by splitting the input of the primitive into two categories: the public bits and the secret bits. In case of a block cipher, the public bits can be the plaintext to the cipher, and the secret bits can be the secret key.

Let \( p(x_1, x_2, \ldots, x_n) \) be a polynomial and \( I \subseteq \{1, 2, \ldots, n\} \) be the indices of the variables in \( p \). We refer to \( I \) as the index set. Let \( t_I \) be a subterm of \( p \) such that

\[
p(x_1, x_2, \ldots, x_n) = t_I \ast p_{S(I)} + q(x_1, x_2, \ldots, x_n)
\]

where \( p_{S(I)} \) is called the superpoly of \( I \), and \( q \) is a polynomial of terms that miss at least one variable with \( t_I \). \( t_I \) is called a maxterm if the \( p_{S(I)} \) is linear i.e., the degree of \( p_{S(I)} \) is 1.

Consider,

\[
\sum_I t_I \ast p_{S(I)} + q(x_1, x_2, \ldots, x_n)
\]

where \( t_I = x_i x_{i+1} \ldots x_j \). The only instance where the term \( t_I \ast p_{S(I)} \) will be included is when \( x_i = x_{i+1} = \ldots = x_j = 1 \) which happens exactly once in the summation. Thus, \( p_{S(I)} \) is included only once in the summation.

Now consider \( q(x_1, x_2, \ldots, x_n) \). \( q \) has the property that it will miss at least one \( x_i \) such that \( x_i \in t_I \). Thus \( q \) will be added twice, once when \( x_i = 0 \) and once when \( x_i = 1 \). As mentioned before addition in GF(2) is the boolean XOR operation, and \( x \oplus x = 0 \). When all terms have been added, \( q \) will be added even number of times to itself thus nullifying the term.

Hence, we have showed that in GF(2)

\[
\sum_I t_I \ast p_{S(I)} + q(x_1, x_2, \ldots, x_n) = p_{S(I)}
\]

We now briefly describe how this information can be used to recover the key.

Preprocessing phase

The aim of the preprocessing phase is to find many maxterms in the cipher polynomial such that the superpoly of that maxterm contains only secret variables and the maxterm contain only public variables. If we can find many such maxterms, then we will have a set of linear equations in secret variables which we can solve. In order to determine the actual superpoly, we can make use of probabilistic tests which determine if a polynomial is linear. The authors of the original paper suggest a randomized technique to find maxterms.

Online phase

Once we have sufficient maxterms and their corresponding superpolys, we simply evaluate the cipher over those public values present in the maxterm and sum up the results. Since
the superpolys are linear combinations of secret variables we simply solve for their values thus recovering the key.

Let us say that we wish to recover $n$ secret variables. We will need to find $n$ different maxterms and their corresponding superpolys. By evaluating the value of each superpoly, we now have a system of linear equations in the secret variables. We can solve for the values of the superpoly variables by any standard process to solve a system of linear equations such as Gaussian elimination.

Perhaps one of the most important aspects of the cube attack is that during the online phase, each evaluation of the cipher is independent of the rest. Thus, we have an opportunity to parallelize the online phase to speed up the entire process.

### 2.4.3 Cube Testing

Cube testers\[2\] are property testers. Using cube testers we can test if primitives possess given properties. Cube testers use the idea behind the cube attack, treating the primitive as a polynomial in GF(2). Cube testers can be used to find *distinguishers* for primitives. Informally, a *distinguisher* is a function that takes a primitive as its input and determines if it is a random oracle or not. Let $\mathcal{F}_n$ be the set of all function mappings from $\{0, 1\}^n$ to $\{0, 1\}$, $n > 0$. A *random function* is a random element if $\mathcal{F}_n$. Let $\mathcal{F} \subset \mathcal{F}_n$ be the family of functions of the primitive being tested. A distinguisher, given $f$ randomly chosen from $\{\mathcal{F}, \mathcal{F}_n\}$, efficiently determines if $f$ belongs to $\mathcal{F}$, or if $f$ behaves as a random function.

The most important difference between the cube attack and cube testing is that there is no preprocessing phase in cube testing. More importantly, when distinguishing the primitive from a random oracle, the key (secret variables) is preset. Thus it is only with the public variables we can now set values to. Cube testing essentially studies the superpolys of the primitive, noting that superpolys should obey some properties with high probability if they were derived from a random oracle. Cube attack is a special case where we are testing for linear superpolys and using them to recover the key.

As mentioned earlier, cube testers test for properties. Let the primitive be treated as a family of functions with a finite domain and range. The cube tester tests each function and determines if that functions possesses a particular property, otherwise rejects it with a fixed probability. Some of the properties that may be examined are test for balance of the superpoly, test for linearity etc.

Consider the following example. Let

$$f(x_1, x_2, x_3) = x_1x_3 + x_2x_3 + x_3$$

Suppose we choose $x_3$ as the superpoly variable, and $x_1, x_2$ as the cube variables then

$$f(0, 0, x_3) + f(0, 1, x_3) + f(1, 0, x_3) + f(1, 1, x_3) = 0$$

irrespective of the choice of $x_3$. This means that no monomials of the form $x_1x_2x_i\ldots x_j$ can occur in $f$. But if $f$ we random, then the probability of monomials of the above form
existing is non-zero. Thus, if the superpoly for a given set of cube and superpoly variables is constant, then the function is not random.

The original paper suggests a variety of properties, which we will briefly mention here.

**Balance** A random function should contain a “balanced” truth table, where the number of zeroes and ones are equal. We can test the superpoly for balance, and if either the number of zeroes or ones is greater, then we have a distinguisher.

**Constantness** If the value of a superpoly is constant, then this information can be used to determine the maximum degree of $f$.

**Low degree and linearity** By using sophisticated tests with which one can test if the degree of the superpoly is low. If the degree of the superpoly is 1, then this function may be vulnerable to the cube attack. There are also dedicated techniques that determine if a polynomial is linear.

### 2.5 CUDA

GPGPU, or general purpose computing on graphics processing units, is the idea of using the processing cores on the GPU to process, typically large amounts, of data. GPUs are designed with specific features in mind, and not all problems may benefit from them. But massively parallel programs, where each computation of a task is independent from the rest, are best suited to be run on the GPUs. There has been a lot of interesting research in many different fields of computer science using these processing units. CUDA is a parallel computing platform developed by NVIDIA. CUDA programs can be run in parallel on NVIDIA’s GPUs. GPUs are stream processors with hundreds of processing cores on them. Each core can execute instructions in parallel providing ample scope for parallel programming.

Nvidia’s **C for CUDA** [16] is a platform for developing C-like programs to run on the GPU. GPUs are traditionally used for graphics rendering. But some non-traditional uses are to run programs that perform game physics calculations such as the NVIDIA’s PhysX engine. Given that the GPUs are capable of performing fast arithmetic operations, and that their architecture’s are inherently parallel, they become ideal platforms to execute problems which involve many independent computations. GPUs often have many cores, with each core capable of running multiple threads simultaneously. If a problem can be arranged so that it fits the GPU programming pattern, then the performance gain that can be obtained is at most the number of cores on the GPU, which is often much more than one would find on traditional CPUs.

Figure 2.3 shows how a GPU based program runs. Not all aspects of the program are run on the GPU. While the CPU is still used for performing a portion of the tasks, the majority of the computation is offloaded to the GPU. The code that is to be run on the GPU is written in a separate function known as a **kernel**. Once the kernel is done executing, the
Serial code executes on the host while parallel code executes on the device.

**Figure 2.3:** Execution of a CUDA program
CPU takes back control of the program. This kind of flexibility in a program allows for user input/output, pre/post processing of data on the host.

### 2.5.1 Thread Hierarchy

The threads in CUDA have a hierarchy. Threads are grouped into **thread blocks**. Within a thread block, the threads can be arranged as a one dimensional, two dimensional or a three dimensional array. The arrangement of threads is merely for convenience and has nothing to do with performance. Depending on the nature of the problem, certain thread patterns may be more suitable, e.g., if the data to be processed is a matrix, then a 2-d arrangement of threads may be useful. The number of threads per thread block is often limited by the memory consumption per thread. On current GPUs each thread block may contain a maximum of 512 threads.

A **grid** is a container of thread blocks. A grid is only conceptual and has no relevance on the execution of the CUDA program. A grid of thread blocks is designed such that all the data the program needs computed can fit. On GPUs with multiple chips, each thread
block will execute in parallel on each chip. Figure 2.4 shows an representation of a grid.

2.5.2 GPU Memory

The GPU, henceforth referred to as the device, has a hierarchy of memory, each suited for its own purpose. Depending of the nature of the problem, such as memory requirement, if the problem is computation bound or bandwidth bound, arranging the data in the right memory is often necessary.

Global Memory

This memory is the main memory on the device. Data stored here is shared between all the chips on the device. The size of the global memory is often large, when compared to the other kinds of memory on the device. The disadvantage of placing data on the global memory is that access to memory is slow. Thus, its better to store data that is read from or written to only once. Global memory is accessible from the device as well as the host, thus serving as the bridge between them.

Shared Memory

Shared memory resides on each chip of the device. Data stored here is shared among all the threads in a thread block. The size of the shared memory is much smaller than that of global memory. But shared memory offers much faster access to data compared to global memory. The use of shared memory requires deep understanding of the problem, and how the tasks are split among the thread block, and the individual threads. If the problem is such that the thread blocks operate on a large chunk of the problem and each individual thread performs a smaller task, then sharing data that is common to the thread block will offer a great performance boost.

Registers and Local Memory

Each thread is given its own set of registers and local memory (which often resides in global memory). The registers are used to store all the automatic variables in the kernel. Large sized arrays are often not stored in registers, but are off-loaded to the local memory. Access to registers is very very fast. But the number of registers is extremely limited. The key to achieving maximum performance is often in deciding what data goes into the registers and what data is stored elsewhere.

2.6 $\chi^2$ Test

The $\chi^2$ test is the statistical test that we employ in this project to examine all primitives. Each occurrence of every possible outcome is counted and examined against the expected
number of occurrences for that outcome. The $\chi^2$ statistic is calculated as follows

$$\chi^2 = \sum_{i=0}^{i=k} \frac{(\text{observed}_i - \text{expected}_i)^2}{\text{expected}_i}$$

where $\text{observed}_i$ is the value that was observed for a particular outcome. $\text{expected}_i$ is the value that is expected for a particular outcome. The $p$-value of the statistic is calculated as follows

$$p = 1 - P\left(\frac{d}{2}, \frac{\chi^2}{2}\right)$$

where $d$ is the number of degrees of freedom, and $P$ is the incomplete Gamma Function. When the $p$-value falls below a preset threshold, then we consider this event as statistically significant and claim to have disproved the hypothesis. The NIST recommends that we use threshold values in the range of $[0.001, 0.01]$ for cryptographic statistical tests [18]. Based on this recommendation we use 0.001 as the significance level.
Chapter 3

Framework and Experiments

3.1 The Primitives

In this section, we will describe the three primitives that we studied as part of this project. We studied two block ciphers, AES and Threefish and a permutation that is key to the hash function Keccak.

3.1.1 AES

AES’s design is not based on the popular block cipher design, the Feistel network. As with most block cipher designs, AES’s design is based on multiple iterations of a round function. The round function is illustrated in Figure 3.1. The block size of AES can be either 128 bits, 192 bits or 256 bits. The length of the key used by AES is the same as the block size. The round function of AES consists of four sub-routines. The state of the AES cipher is 128 bits long, and is viewed as a 4x4 grid of bytes.

SubBytes The state is run through a S-Box, where each byte is substituted for another.

ShiftRows Each row of the state is cyclically shifted by a fixed amount.

MixColumns Each column of the state is multiplied with a matrix to transform it.

AddRoundKey The round key is mixed into the state with a simple XOR operation.

Depending on the version of AES being used, the number of rounds can be 10, 12 or 14. We have given only a high level description of AES and lead interested readers to the specification[5].

3.1.2 Threefish

Threefish[9] is a block cipher designed as a part of the NIST SHA-3 hash function candidate, Skein. Threefish is a tweakable block cipher, which just means that along with the key that goes into the encryption process, a tweak can also be added to personalize the block cipher. Threefish comes in three sizes of 256 bits, 512 bits and 1024 bits. Threefish is designed with
the philosophy that many rounds of a simple function is more secure than fewer rounds of a more complex function. The simple \textbf{Mix} function that is integral to Threefish is shown Figure 3.2. Part of the overall operation of Threefish-256 is shown in Figure 3.3.

The number of rounds in Threefish is unusually large, given the simplicity of each round. Threefish-256 and Threefish-512 perform 72 rounds, whereas Threefish-1024 performs 80 rounds. All operations in Threefish are kept relatively simple. The key schedule is simply a permutation of the original key, with the key being added to the state every 4\textsuperscript{th} round. The authors have spent considerable time in determining the intricate details such as the rotation constants. The rotation constants were recently revised so as to maximize diffusion in the state. Detailed explanation of the revision can be found in the most recent Skein specification \cite{9}.
3.1.3 Keccak

Algorithm 1 Round\([b](A, RC)\)

{θ step}
for \(x = 0\) to \(4\) do
end for
for \(x = 0\) to \(4\) do
\[D[x] = C[x - 1] \oplus ROT(C[x + 1], 1)\]
end for
for \(x = 0\) to \(4\) do
for \(y = 0\) to \(4\) do
\[A[x, y] = A[x, y] \oplus D[x]\]
end for
end for
{ρ and π steps}
for \(x = 0\) to \(4\) do
for \(y = 0\) to \(4\) do
\[B[y, 2x + 3y] = ROT(A[x, y], r[x, y])\]
end for
end for
{χ step}
for \(x = 0\) to \(4\) do
for \(y = 0\) to \(4\) do
\[A[x, y] = B[x, y] \oplus ((NOTB[x + 1, y])ANDB[x + 2, y])\]
end for
end for
{ι step}
\[A[0, 0] = A[0, 0] \oplus RC\]

Keccak is a hash function that was submitted as a candidate to the NIST SHA-3 contest. Keccak’s design is based on a sponge construction. The hash function uses a permutation as a building block. The permutation is a family of seven functions, which can be used for varying degrees of hash lengths. For purposes of this project we will discuss one of the permutations.

The Keccak-\(f\) permutations are designed with block of length \(25 \times 2^l\) bits where \(l\) can take values from 0 to 6. Each permutation is denoted as Keccak-\(f[b]\) where \(b\) is the block length in bits. The authors of Keccak recommend Keccak-\(f[1600]\) as the permutation.
3.1.4 Keccak-f[1600]

The state of the Keccak permutation is represented by a three dimensional array \( a[x][y][w] \), where \( w = 2^l \). The third dimension can be conveniently represented by a byte, 32-bit word, 64 bit word etc, depending on the function being used. If the state were to be viewed as a bit stream, say \( s \), the mapping between the bits of the state and to \( s \) is given by
\[
s[w(5y + x) + z] = a[x][y][z].
\]

The permutation is based on the idea of a round function being applied iteratively. The round function consists of five operations. All operations are performed in \( GF(2) \).

Algorithm 1 shows a pseudo-code of the Keccak-f[b] permutation. The state of the permutation is represented by \( A[.] \), which is essentially the \( a[\cdot][\cdot][\cdot] \) array, with the third dimension represented by a 64-bit word.

3.2 The Design of the Framework

In this section we will provide some insight into the design of the framework which makes use of CUDA. As mentioned before, the general technique of executing programs on a GPU is through kernel’s which is executed in parallel on the GPU. The following section describes some of the software designs employed in the framework.

3.2.1 Overall Design

The project is designed to be flexible so that testing various primitives is made easy. The software is split into two phases, the data collection phase, and the data analysis phase. The data collection phase, is the actual CUDA program. The idea in this phase is to evaluate the value of the superpoly for a given set of cube and superpoly variables. The CUDA program takes in a random seed to generate the set of the cube and superpoly variables, get a random sample set of assignments to the superpoly variables. It then computes the value of the superpoly’s and outputs them to a file. The file can then be analysed for different sets of significance levels. The general outline of the data collection phase is shown in Algorithm 2

\begin{algorithm}
\caption{Data Collection}
\footnotesize
Choose a random subset of the plaintext bits as the cube variables say \( c_1, c_2, \ldots c_n \)
Choose a random subset of the plaintext bits as the superpoly variables say \( s_1, s_2, \ldots s_m \)
for \( i = 1 \) to \( N \) do
\hspace{1em} Choose a random assignment for \( s_1, s_2, \ldots s_m \)
\hspace{2em} for \( c_1, c_2, \ldots c_n = 000 \ldots 00 \) to \( 111 \ldots 11 \) do
\hspace{3em} \( Q_i = Q_i \oplus F_i(c_1, c_2, \ldots c_n, s_1, s_2, \ldots s_m) \)
\hspace{2em} end for
\hspace{1em} end for
Write the values of \( Q_i \) to a output file
\end{algorithm}
A sample of the file is shown below

786432274
203b3a06433a16480d4077af23830b01
43 102 86 81 10 17
51 72 107 41 45 12 71 31 95 117
16
0 FAC660A226D84441536B6DBE1F4DE419
1 15BD983E24D135969C5F891007805132
2 E6327AEC447FBEA5CFE0D97F0A7A7AD9
3 426A1ABBEB71F6181FA9551967BCAB1CD
4 E907E333D4C476ADB0076DF299FE9C20
5 B4DAEB1D515767B9F5C5DA99CC33DE17
6 FB6AE783E383226EB55B9C41E4FD227
7 0DE3FC6486462065F200CAABACAC6792A5
8 1BCEDAA968130BDE4A8EF056E70751AC
9 457BACD93523ED34AAAF7651A2F38F7B5
10 A0727A3E96CF335CE715CC2AF99028E4
11 C6DA06AE1073934DA3C8F56369312EBB
.
.
.

The first line is the seed that was used to set the random number generator. This allows us to recreate the experiment to get identical results. The second line stores the key that was used. If the primitive is a hash function, then this line is absent. The next two lines show the plaintext bits that were chosen as the cube and superpoly variables respectively. Finally, the next $N$ lines store the value of the superpoly for each bit of output. The data analysis phase is executed by a Java program. The program makes use of the Parallel Java\[11\] library for statistical analysis.

### 3.2.2 kernel.h

This file contains the kernel that will be executed by the program. The task of the kernel is to evaluate the value of the superpoly.

```c
__global__ void computeSum(byte *key, byte *q_g, int nc, int ns, uint16 *c, uint16 *s, uint64 *sval)
{
    uint32 i, cva=0, j;
    byte t;
    byte plaintext[NUM_BYTES];
```
In the above code snippet we can see that each thread gets its own AES variable to perform the encryption. If a different primitive is being used, then this would be the place to get a copy of that primitive.

```c
for(j=0; j<NUM_BYTES; j++) plaintext[j]=0;
for(j=0; j<NUM_BYTES; j++) s_q[threadIdx.x][j]=0;

/** Set the superpoly variables to their respective values */
for(j=0; j<ns; j++)
{
    t = (sval[blockIdx.x] >> (ns-1-j)) & 0x1;
    plaintext[s[j]>>3] |= (t << (7-(s[j]%8)));
}
```

Each thread block evaluates the value of the superpoly for one assignment to the superpoly variables. It is here that we are setting the superpoly variables to their respective values.

```c
for(i=0; i<=upper; i++)
{
    cval = (threadIdx.x << (nc-LOG_NT)) | i;
    for(j=0; j<nc; j++)
    {
        plaintext[c[j]>>3] &= ~(1<<(7-(c[j]%8)));
        t = (cval >> (nc-1-j)) & 0x1;
        plaintext[c[j]/8] |= (t << (7-(c[j]%8)));
    }
    aes_encrypt(&a, plaintext, ciphertext);
    for(j=0; j<NUM_BYTES; j++) s_q[threadIdx.x][j] ^= ciphertext[j];
}
```
The above for loop runs over all possible assignments to the cube variables and evaluates the primitive. The s.q array stores the result of the superpoly for each bit of output.

```c
/** Reduce the shared q values */
__syncthreads();
if(NT>=128 & threadIdx.x <64)
    for(j=0; j<NUM_BYTES; j++) s_q[threadIdx.x][j] ^= s_q[threadIdx.x+64][j];

__syncthreads();
if(NT>=64 & threadIdx.x <32)
    for(j=0; j<NUM_BYTES; j++) s_q[threadIdx.x][j] ^= s_q[threadIdx.x+32][j];

__syncthreads();
if(NT>=32 & threadIdx.x <16)
    for(j=0; j<NUM_BYTES; j++) s_q[threadIdx.x][j] ^= s_q[threadIdx.x+16][j];

__syncthreads();
if(NT>=16 & threadIdx.x <8)
    for(j=0; j<NUM_BYTES; j++) s_q[threadIdx.x][j] ^= s_q[threadIdx.x+8][j];

__syncthreads();
if(NT>=8 & threadIdx.x <4)
    for(j=0; j<NUM_BYTES; j++) s_q[threadIdx.x][j] ^= s_q[threadIdx.x+4][j];

__syncthreads();
if(NT>=4 & threadIdx.x <2)
    for(j=0; j<NUM_BYTES; j++) s_q[threadIdx.x][j] ^= s_q[threadIdx.x+2][j];

__syncthreads();
if(NT>=2 & threadIdx.x <1)
    for(j=0; j<NUM_BYTES; j++) s_q[threadIdx.x][j] ^= s_q[threadIdx.x+1][j];

__syncthreads();
if(threadIdx.x==0)
    for(j=0; j<NUM_BYTES; j++) q_g[blockIdx.x*NUM_BYTES+j] = s_q[threadIdx.x][j];
```

The above section is an efficient way to perform a parallel reduction. Each thread performs only a part of the task of evaluating the value of the primitive for all possible cube variable assignments. Here all the threads aggregate their results into the first thread, and it is this thread that outputs the end result.

Even if the primitive or the test were different, only a minimal number of changes need to be made to the kernel to get it working.

The kernel works in the following manner. The number of cube variables, the number of superpoly variables, the random samples of assignments to the superpoly variables are
passed as parameters to the kernel. Each thread block operates on one random sample of the superpoly assignment. Each thread in the thread block is now tasked with computing the partial sum of the function over a subset of assignments to the cube variables. For simplicity sake, say that the number of cube variables is $C$ and the variables themselves are $c_1, c_2, \ldots c_C$. Let us also say that the number of threads in a thread block is $nt \leq C$. The thread-id of each thread gives the value to the variables $c_1, c_2, \ldots c_{nt}$. The threads now loop over all possible values for $c_{nt+1}, c_{nt+2} \ldots c_C$, and evaluate the value of the function. The result is stored in shared memory (in the $s_{\text{q}}$ array). Once all the threads have computed their tasks, they now need to aggregate the result which is done by a parallel reduction. Once the reduction is complete, the thread with ID 0, writes the result of the block into global memory. Once all the blocks have executed, the host can now run the $\chi^2$ test.

### 3.2.3 `<primitive>.h`

There are a few general rules that are necessary in the design of the primitive.

- The primitive must be defined in a separate header, say `<primitive>.h`. This provides clarity of code more than functionality.

- The primitive header must define a macro by the name `NUMBYTES`. This value of the macro must be the size of the plaintext/ciphertext in bytes. All the primitives we have studied had the same size of plaintext and ciphertext. In the case that both these values are different, the necessary changes must be made to the program.

- Again, more for convenience than functionality, the function that performs the encryption must have the following structure:

  `<primitive>_encrypt(<reference_to_primitive_variable>, plaintext, ciphertext)`

- The plaintext, ciphertext arrays must be of type `byte`. The primitive may transform the plaintext into any other data type, but the input and output of the encryption must be array’s of bytes.

### 3.3 Experiments on the Primitives

#### 3.3.1 Balance Test

Before we begin to detail the experiments that were run, we need to define some of the terms that will be used often in the following sections.

$F_i$ represents the polynomial representing the $i^{th}$ bit of the primitive. $c_1, c_2, \ldots c_C$ represent the set of cube variables chosen. $s_1, s_2, \ldots s_S$ represent the set of superpoly variables chosen.
$F_i$ can be represented as

$$F_i = t_I p_S(s_1, s_2, ..., s_S) + q_i(x_1, x_2, \ldots x_n)$$

where $t_I = x_{c_1}x_{c_2} \ldots x_{c_C}$ and $p_S$ is the superpoly of $F_i$ with respect to a set of chosen cube variables and $q_i$ is the remainder term.

Let $p$ represent the significance level.

The test of balance of one superpoly is shown in Algorithm 3. The value of one bit($F_i$ for some $i$) is observed for $N$ random superpoly assignments. A $\chi^2$ test is performed on the observed number of times the value of the superpoly was 0 and 1.

**Algorithm 3 Test of Balance of one superpoly**

- Run Algorithm 2 for a given seed
- Let $Q$ be the superpoly we are testing
- Run $\chi^2$ test on number of times $q$ evaluated to 0 and 1
- if $p$-value < significance level then
  - Test fail
- end if

Our hypothesis is that we expect the value of the superpoly to be 0 or 1 with probability of 0.5.

The procedure to run a $\chi^2$ test is as follows.

- Let $F_i$ be the output bit being observed
- Choose $N$ random assignments for the superpoly variables
- Let $n_0$ be the number of times $Q_i$ evaluated to 0
- Let $n_1$ be the number of times $Q_i$ evaluated to 1
- Compute $\chi^2$ and the corresponding $p$-value.

Since we are operating the whole primitive, we can simultaneously observe values of $Q_i$ for all $i$. We now extend the test to perform a statistical test on all the superpolys. We hypothesize that for $N$ random samples of $F_i$, we would expect exactly $p \times N$ superpolys to fail their balance test and $(1 - p) \times N$ to pass the balance test. Given the expected data, we can find out what was observed and perform a $\chi^2$ test. In our experiments we set $N$ to be 1000. A general outline of the process is shown in Algorithm 4.

A sample output of our program is shown below

<table>
<thead>
<tr>
<th>ns</th>
<th>nc</th>
<th>pass</th>
<th>fail</th>
<th>$X^2$</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>6</td>
<td>1000</td>
<td>0</td>
<td>1.001001001001001</td>
<td>0.3170684160925148</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>998</td>
<td>2</td>
<td>1.001001001001001</td>
<td>0.3170684160925148</td>
</tr>
</tbody>
</table>
Algorithm 4 Test of Balance

Let the significance level be $P$
Run Algorithm 3 on $N$ random $Q_i$ and count number of passes and failures
Expected number of passes = $N \times (1 - p)$
Expected number of failure = $N \times p$
Evaluate $\chi^2$ and the $p$-value

| 12 | 6  | 999 | 1  | 0.0 | 1.0 |
| 13 | 6  | 999 | 1  | 0.0 | 1.0 |
| 14 | 6  | 999 | 1  | 0.0 | 1.0 |
| 15 | 6  | 999 | 1  | 0.0 | 1.0 |
| 16 | 6  | 1000| 0  | 1.001001001001001001 | 0.3170684160925148 |
| 17 | 6  | 999 | 1  | 0.0 | 1.0 |
| 18 | 6  | 998 | 2  | 1.001001001001001001 | 0.3170684160925148 |
| 19 | 6  | 1000| 0  | 1.001001001001001001 | 0.3170684160925148 |
| 20 | 6  | 999 | 1  | 0.0 | 1.0 |
| 21 | 6  | 1000| 0  | 1.001001001001001001 | 0.3170684160925148 |
| 22 | 6  | 1000| 0  | 1.001001001001001001 | 0.3170684160925148 |
| 23 | 6  | 1000| 0  | 1.001001001001001001 | 0.3170684160925148 |
| 24 | 6  | 1000| 0  | 1.001001001001001001 | 0.3170684160925148 |
| 25 | 6  | 1000| 0  | 1.001001001001001001 | 0.3170684160925148 |
| 10 | 7  | 1000| 0  | 1.001001001001001001 | 0.3170684160925148 |

In the output of the data analysis, the column $nc$ refers to the number of cube variables, $ns$ refers to the number of superpoly variables. The column $pass$ denotes the number of tests that passed the statistical test for a given significance level and $fail$, the number of tests that failed. For the given pass/fail count we evaluate the $\chi^2$ and the $p$-value.

3.3.2 Independence Test

If the primitive were a random function, then the output bits would be independent of each other. Let $Q_i$ and $Q_j$ be a pair of output bits. If the null hypothesis, that $Q_i$ for all $i$ is 0 or 1 with equal probability, then we would expect that for 25% of the trials, $(Q_i, Q_j)$ to be $(0, 0)$, and similarly 25% to be $(0, 1), (1, 1)$ and $(1, 1)$. If there is some dependence between the output bits then one of the above output values will be more likely.

The general outline of our test is shown in Algorithm 5.

In all our experiments we set $N$ to be 4000.
**Algorithm 5 Independence Test**

Run Algorithm 2 for a given seed
For each \((Q_i, Q_j)\) pair, count the number of times the pair evaluates to \((0, 0)\), \((0, 1)\), \((1, 0)\), \((1, 1)\)
Evaluate the \(\chi^2\) and p-value for each \((Q_i, Q_j)\) pair
Pick \(N\) random samples and count the number of pass/fail results

---

**3.4 Experiments**

We ran the balance test and the output/output independence test on each of the three primitives. The machine that we used clairaut.cs.rit.edu is a Sun Microsystems Ultra 40 Workstation, with a 1GHz AMD Opteron 2218 CPU. The GPU coprocessor installed on the machine is a NVIDIA Tesla C870 card. The card has 16 multiprocessors each with 8 thread processors. The total number of thread processors is 128. The card has a clock speed of 500 MHz and has 1.5GB of main memory.

**3.4.1 Tests on Keccak-\(f[1600]\)**

We ran the balance test described above on Keccak-\(f[1600]\). The significance level for all tests was set at 0.001. For the balance test on a particular superpoly, we sampled the primitive for 100 random superpoly assignments to observe the value of the superpoly.

We ran the above test for the number of cube variables ranging from 6 to 22 and the number of superpoly variables ranging from 10 to 25.
Table 3.1: Data Analysis with different seeds

<table>
<thead>
<tr>
<th>Test</th>
<th>seed=3474359</th>
<th>seed=91934823</th>
<th>seed=2843756</th>
<th>seed=7209345</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balance</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Independence</td>
<td>9</td>
<td>9</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

The plot shown in Figure 3.4 shows the result of running the balance test on Keccak-f[1600] with a significance level of 0.001. The test was run as follows. For a given seed, we run Algorithm 4. We then sample 1000 superpolys to see their pass/fail outcomes. We then repeat the above process for one other seed. Thus at the end of this, we have 2000 pass/fail results. A $\chi^2$ test is then run on the data, and plotted as shown above. Each square in the plot represents the p-value of the experiment for the corresponding number of cube and superpoly variables. The color of a square indicates the p-value; white indicating a p-value of 1, and black indicating a p-value of 0. If the p-value is lesser than the significance level set for the experiment, then a white dot is placed in the center of the square. The plot shows that none of the tests failed. This shows no indication of non-random behaviour. The plots were generated by a Java program part of [12].

The plot shown in Figure 3.5 is the result of running the independence test on Keccak-f[1600]. In the independence test, we use the value of the superpolys to analyse output/output dependence in Keccak. The number of output bit pairs are 1,279,200. We sample 2000 of these, and note their pass/fail outcomes. We then repeat the above process on data obtained using a different seed. We now have 4000 pass/fail outcomes which we use to run the $\chi^2$ test on. We see a few more failures than in the balance test. For the 272 different tests that were performed, a failure of 12 tests can be considered significant at a significance level of 0.001.

In the data analysis phase, if we were to change the seed of the random number generator, then we will be picking a different random sample of superpolys in the balance test and different $(Q_i, Q_j)$ pairs in the independence test. We tested the data collected for four seeds and the results are tabulated in Table 3.1.

3.4.2 Tests on AES

With AES, we could do something similar to the Keccak-f[1600]. For a particular key we perform the above test. But by changing the key, we are essentially getting a completely different instance of the primitive. In order for us to find a distinguisher, we must be able to show that independent of the choice of the key, our distinguisher can indeed distinguish AES. Given that the version of AES we have chosen has $2^{128}$ possible key, we cannot practically do this. We can however hope to find weak key, where if the primitive was set with that particular weak key, then our test could detect non-randomness. Finding weak keys however
requires deep understanding of the primitive being studied.

We ran the balance and independence test for 4 different randomly chosen keys. We ran tests for cube sizes of 6 to 22, and varied the number of superpoly variables from 10 to 25.

The following data shows the run of our framework for 4 AES instances each with a different key. We ran the above test for the number of cube variables ranging from 6 to 22 and the number of superpoly variables ranging from 10 to 25. For our experiments, we set a significance level of 0.001.

Figure 3.6, Figure 3.7, Figure 3.8 and Figure 3.9 show the result of the balance test on the AES-128 primitive with the 4 keys. For each key, we run the data collection routine with 10 different seeds, which gives us values for 1280 superpolys. We sample 1000 of them at random to run the \(\chi^2\) test. The balance test does not show any non-random behaviour in AES-128. All four keys showed 0 or 1 failures (white dots), which is well within the acceptable range.

Figure 3.10, Figure 3.11, Figure 3.12 and Figure 3.13 show the result of the independence test on the AES-128 primitive with the 4 keys. With AES-128, we have 8,128 output pairs. We sample 1000 of these at random. We repeat this with 3 data obtained using three other seeds, giving us 4000 pass/fail outcomes, which we use to run the \(\chi^2\) test. Now we are seeing many failures in the experiment. For the given number of test (272), the number of failures detected can be considered statistically significant at a significance level of 0.001. We have run the experiment of four different keys, each of which show similar behaviour. This tells us that the non-randomness is not an isolated incident with respect to the output/output
In the data analysis phase, if we were to change the seed of the random number generator, then we will be picking a different random sample of superpolys in the balance test and different \((Q_i, Q_j)\) pairs in the independence test. We tested the data we collected for four seeds and the results are tabulated in Table 3.2.

### Tests on Threefish

The balance test on Threefish is similar to the one performed on AES. For a given key and tweak we run Algorithm 2. Now with the data collected we perform a balance test and an output/output independence test.

Figure 3.14, Figure 3.15, Figure 3.16 and Figure 3.17 show the output of the balance test on Threefish-256 for four different keys. For the given number of tests that we performed, we would expect to see no more than 1 failure, if the null hypothesis were true. Our findings fail to disprove the hypothesis that \(Q_i\) is 0/1 with equal probability.
Figure 3.10: AES-128 Independence Test 1

Figure 3.11: AES-128 Independence Test 2

Figure 3.12: AES-128 Independence Test 3

Figure 3.13: AES-128 Independence Test 4
But when we get to the test of independence, the results become far more interesting. In case of Threefish-256, the number of \((Q_i, Q_j)\) pairs are \(256 \times 255/2 = 32,640\). Out of the 32,640 pairs, we pick 1000 at random, and look at the pass/fail outcome. For a given key, we had run the data collection program with four different seeds. This now gives us 4000 pass/fail outcomes. The figures 3.18, 3.19, 3.20 and 3.21 show the result of the \(\chi^2\) test on Threefish-256 with 4 different keys. The plots show many white dots, indicating that that particular instance of the test failed. This is now clearly indication of some non-random behaviour in Threefish. The fact that the output/output independence test fails for 4 different random keys, gives us some confidence that the result was not just an isolated incident.

In the data analysis phase, if we were to change the seed of the random number generator, then we will be picking a different random sample of superpolys in the balance test and different \((Q_i, Q_j)\) pairs in the independence test. We tested the data we collected for four seeds and the results are tabulated in Table 3.3.

**Table 3.3:** Data Analysis with different seeds

<table>
<thead>
<tr>
<th>Test</th>
<th>seed=3474359</th>
<th>seed=91934823</th>
<th>seed=2843756</th>
<th>seed=7209345</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balance</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Independence</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>
Figure 3.16: Threefish-256 Balance Test 3

Figure 3.17: Threefish-256 Balance Test 4

Figure 3.18: Threefish-256 Independence Test 1

Figure 3.19: Threefish-256 Independence Test 2
3.5 Timing Measurements

In order to measure the performance of our framework, we ran experiments to measure the time the program took to run. We ran the data collection program on AES-128, changing the number of threads per block and the number of thread blocks in each run of the program. The number of cube variables in each experiment was set to 16.

For the first set of experiments we set the number of thread per block fixed at 1, 32 and 64 and varied the number of thread blocks. The speedup plots are shown in Figures 3.22, 3.23 and 3.24 and the efficiency plots are shown in Figures 3.25, 3.26 and 3.27.

The running times of for the above experiment is shown in Table 3.4. In the table, NTPB refers to the number of threads per block and NTB refers to the number of thread blocks.
Figure 3.24: Speedup (64 thread per block)  Figure 3.25: Efficiency (1 thread per block)

Figure 3.26: Efficiency (32 thread per block)  Figure 3.27: Efficiency (64 thread per block)
The results tell us that we do get linear speedup when we increase the number of thread blocks. It is interesting to observe that when the number of threads per block is 64 we are getting sub-linear performance. Possible reasons for this are mentioned below.

- When a thread block is executing on a core of a multiprocessor, not all 64 threads get executed in parallel. The threads get divided into warps, and all threads in one warp execute in parallel. The number of threads in a warp is 32 for the Tesla C870 GPU. Given that there may be some sequential execution of the warps, this maybe one reason contributing to the sub-optimal performance.

- When the number of threads is lower than 64, the amount of time spent in computing the value of the superpoly by a thread block is significant when compared to the remaining sequential portion of the program. But when the number of threads is 64, the kernel’s execution time is low, and the sequential fraction weighs in on the overall running time. If we increase is size of the problem, making the kernel perform more computation, then we expect to see this sequential fraction disappear.

In another experiment, we kept the number of thread blocks constant at 20 and varied the number of threads in each block. We set the number of threads in each block to 1, 8, 16, 32 and 64 and measured the running time. The speedup and efficiency plots are shown in Figure 3.28 and Figure 3.29. The speedup we observe is about half of what we expect to see in an ideal case. The kernel we have designed does not have many synchronization points, except for the parallel reduction at the end of the kernel. However, this synchronization cannot account for the sub-optimal efficiency of the kernel. Given that the thread scheduling of the CUDA architecture is not known, we are unable to satisfactorily account for the performance. Further investigation may provide insight into this issue.
Figure 3.28: Speedup (20 thread blocks)  

Figure 3.29: Efficiency (20 thread blocks)
Chapter 4

Future work

This project shows only a very preliminary study of the three primitives, treating each as a black box. By delving into the primitives, and uncovering potential weaknesses, one can strengthen the test to a great extent. We also only test the primitives for balance, leaving ample scope to investigate other properties in the primitives. The test for balance and the output/output independence test look at the primitive superficially. Successful tests for linear variables will on a primitive show that it may be susceptible to the cube attack. The NIST SHA-3 contest winner will only be announced in 2012, leaving plenty of time to investigate other primitives. In [12] the authors perform similar statistical tests on CubeHash and Skein, two SHA-3 candidates.

Cube testers have been shown to be effective on some stream ciphers. In the original paper[2] the authors show the presence of neutral variables in Trivium, a stream cipher. There has also been some research into hardware implementations of cube testers to study specific primitives. The authors of [11] perform such a study on Grain-128, a stream cipher, by using FPGAs to accelerate cube testing. There are many other stream ciphers part of the eStream portfolio[17] that can be studied.

The framework we have designed is rigid to a certain extent given the restrictions imposed by the CUDA architecture. As the CUDA framework expands to provide support for more flexible features, the framework too can be extended for greater flexibility. We have also not performed extensive performance analysis on the kernel to maximize performance. The framework was tested on a Tesla C870 GPU which is outdated compared to newer models. The newer GPUs are more sophisticated and have larger amounts of memory. This allows more computation to happen simultaneously on the GPU. The most recent SDK also supports C++ as a programming platform. With an object-oriented design the flexibility of the framework can be greatly improved.
Chapter 5

Conclusions

This project has provided clear evidence that GPUs are excellent platforms for running cube tests. Programming the GPUs is now easy, and this framework can be used to study many other primitives. OpenCL[21], an open source alternative to CUDA, is gaining popularity as it can be used to program a wide variety of stream processors. The results in this project show that GPU coprocessors are extremely effective at solving massively parallel problems.

We have also seen that while we failed to disprove the hypothesis that the value of the superpoly was balanced, there is non-randomness in the output/output independence. We have used a low significance value of 0.001 as per the recommendations in the NIST special publication. Non-randomness at such a significance level means that the probability of our detection of non-randomness being valid is high.
Bibliography


