COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

Analysis and Implementation of eSTREAM and SHA-3 Cryptographic Algorithms

by

Deian Stefan

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Engineering

May 10, 2011

Advisor

Dr. Fred L. Fontaine

COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

_____

Dr. Simon Ben Avi
Dean, School of Engineering

_____

Dr. Fred L. Fontaine
Candidate's Thesis Advisor

Abstract

Invaluable benchmarking efforts have been made to measure the performance of eSTREAM portfolio stream ciphers and SHA-3 hash function candidates on multiple architectures. In this thesis we contribute to these efforts; we evaluate the performance of *all* eSTREAM ciphers and *all* second-round SHA-3 candidates on NVIDIA Graphics Processing Units (GPUs).

Complementarity, we present the first implementation of the cube attack in a multi-GPU setting. Our framework proves useful in the practical analysis of algorithms by providing a generic black box interface and speedup factors over $100\times$. Demonstrating its use we analyze two eSTREAM stream ciphers, MICKEY v2 and Trivium. We find that MICKEY is not susceptible to low-degree cube attacks, while our Trivium analysis confirms previous results, in addition to several new equations applicable to a partial key recovery.

We also extend the linear differential cryptanalysis framework introduced by Brier, Khazaei, Meier and Peyrin at ASIACRYPT 2009 using two new trail search algorithms, and several optimizations. We find several collision and second preimage attacks on simplified and round-reduced variants of BLAKE and CubeHash, two SHA-3 second round candidates. Using the extended framework we also present improved collision attacks on CubeHash, when compared to previous results. In combination with the condition function concept, our new trail search algorithms lead to much faster collision attacks. We demonstrate this by providing a real collision for CubeHash-5/96. Additionally our randomized trail search finds highly probable linear differential trails and leads to significantly better attacks for up to eight rounds of CubeHash.

Acknowledgments

I would like to thank my advisor Fred L. Fontaine for his encouragement and support on this work and my Electrical Engineering studies at The Cooper Union. I am especially grateful for his efforts in providing an excellent environment, the S*ProCom^2 lab, in which I have flourished as both a student and researcher; our collaboration is one of the most rewarding and influential experiences to date.

I am grateful to Peter Cooper and The Cooper Union for providing me with the opportunity to gain an exceptional education. I am thankful for having had the opportunity to work with and study under Jeff Hakner, Danfeng Yao, Kausik Chatterjee, William Donahue, and Alan Berenbaum. They have in many ways influenced my choice to pursue research studies.

I would like to thank all my coauthors for the fruitful discussions and great moments we exchanged. In particular, I would like to thank David B. Nummey, Christopher Mitchell, Jared Harwayne-Gidansky, Ishaan L. Dalal, and Matthew Epstein. Furthermore, I am especially grateful to Arjen K. Lenstra, Joppe Boss and Shahram Khazaei for my summer study at LACAL, the work of which led to this thesis.

Finally, I would like to thank my friends and family for supporting me during my studies. Most of all, I am grateful to my parents for their ongoing support and encouragement.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Security has become a crucial aspect in the design and use of computer systems and networks. Whether one is designing a wireless communication system, web application, or network protocol, addressing security is an essential engineering criterion. Though a well-designed system is built from a multitude of components, the use of *cryptography* as a building block is almost unanimous.

Cryptography is used to address many security issues, the most pertinent of which are *confidentiality*, *integrity*, and *authentication*. Cryptography encompasses the design of (cryptographic) *primitives*, basic building blocks, and *protocols/schemes* that use these building blocks to construct complex security systems. Dually, *cryptanalysis* entails the analysis and evaluation of cryptographic algorithms, including primitives and protocols. In this thesis we focus on the implementation and analysis of two kinds of cryptographic primitives: stream ciphers and hash functions.

1.1 Stream Ciphers

Stream ciphers are cryptographic algorithms that transform a stream of plaintext messages of varying bit-length into ciphertext of the same length, usually by generating a *keystream* that is then XORed with the plaintext. Using a shared secret key, stream ciphers can be used to provide confidentiality, i.e., restrict access to secret data to the parties in possession of the key by encrypting the plaintext secret data. In general, stream ciphers have very high throughput, strong security properties, and use few resources, thus making them ideal for mobile applications; well-known examples of stream ciphers include the RC4 cipher used in 802.11 Wireless Encryption Protocol [50], E0 cipher used in Bluetooth protocol [50], and the SNOW 3G cipher used by the 3GPP group in the new mobile cellular standard [110].

Stream ciphers are widely-used primitives, core to many security systems, and as such the security and efficiency of these primitives is crucial to many applications. In this thesis, we focus on implementing the stream ciphers selected for the eSTREAM portfolio, and analyzing a subset of them: the MICKEY v2 and Trivium stream ciphers. The eSTREAM portfolio is a result of the European Union sponsored four-year project, whose gole was to identify new stream ciphers as alternatives to widely-used, though cryptographically insecure, ciphers [27,74]. We believe that our implementations and analysis is a contribution to the efficient use and better understanding of these ciphers.

Stream ciphers are designed to satisfy various security properties. Specifically, we expect it to be infeasible to recover the secret key or internal state given the keystream output. In this thesis we analyze the MICKEY v2 and Trivium steam ciphers using the *cube attack* [45, 46]. The cube attack is a very recent general

cryptanalytic technique, that can be used to carry out algebraic attacks on cryptosystems with low degree polynomials, a design weakness many stream ciphers are susceptible to. Although the cube attack can be used to analyze a wide range of primitives, including block ciphers and keyed hash functions, we limit our analysis to Trivium and MICKEY v2 as our main contribution, in this aspect, is in providing a multi-GPU cube attack framework.

1.2 Hash Functions

Like stream ciphers, hash functions are important cryptographic primitives. However, hash functions transform arbitrary-length input messages into fixed-length message digests. They are used in many applications, notably in commitment schemes, digital signatures and message authentication codes. To this end they are required to satisfy different security properties. These security properties include i) preimage resistance, i.e., given $f(x)$ it is infeasible to find x, ii) second preimage resistance, i.e., given x it is infeasible to find $x' \neq x : f(x) = f(x')$, and iii) collision resistance, i.e., it is infeasible to find $x, x' : x' \neq x$ and $f(x) = f(x')$. Informally, a hash function is collision resistant if it is practically infeasible to find two distinct messages m_1 and m_2 that produce the same message digest.

It is clear that the security properties of hash functions are desirable when building security systems where integrity is a concern. For example, in a system where two parties exchange secrets by employing a stream cipher a third party modifying the exchanged ciphertexts could easily meddle with the stream. Hence, a receiver could end up decrypting a message that was not sent by the corresponding sender, but instead the *man-in-the-middle*. An approach addressing such issues consists of using message authentication codes (MACs) which

produce a message digest that can only correspond to the MAC'd message. The result is that the intermediary modifying intercepted messages will be detected as it will not be able to produce a proper digest without the secret key. Among their key role in signature schemes and other authentication methods, hash functions are commonly used to construct MACs. The use of such authentication schemese widely used, e.g., in web applications, highlights the significance of efficiently implementing and analyzing the security properties of such primitives.

Though many existing applications already use standard hash functions, like Message Digest 5 (MD5) and the Secure Hash Algorithm-2 (SHA-2), the design and analysis of cryptographic hash functions have come under renewed interest with the public competition[1] commenced by the US National Institute of Standards and Technology (NIST) to develop a new cryptographic hash algorithm SHA-3. In this thesis we focus on the second-round SHA-3 candidates, presenting performance estimates supported by actual implementations. As in the stream cipher case we also analyze a subset of these algorithms: BLAKE and CubeHash.

Similar to the cube attack framework, we use a generic linear differential cryptanalysis framework to analyze BLAKE and CubeHash. Though linear differential cryptanalysis is a more mature technique, widely applied to the analysis of many cryptosystems, including block ciphers, and stream ciphers, we use the more recent results targeting hash functions [34, 66].

1.3 High-Performance Cryptologic Computing

Racing with Moore's law [80, 81], processor design techniques have shifted towards incorporating multiple processors on a single die from the more tradi-

[1] See http://csrc.nist.gov/groups/ST/hash/sha-3/index.html

tional designs that simply increased the complexity of instructions and processor frequency. This paradigm shift is increasingly being regarded as the *many-core* revolution.

At the forefront of the many-core revolution are graphics processing units (GPUs). The once game-specific processors have become very powerful and general purpose processors. Modern GPUs are equipped with hundreds to thousands of streaming processor cores, clocked at rates comparable with those of a CPU. Thus, leveraging the vast computational power of these devices, we can build secure systems that do not suffer in performance simply because of the added security features. Moreover, it is expected that new algorithms be designed for and implemented on such platforms; for example, one of the motivations behind this work is NIST's predisposition for algorithms with greater flexibility [88] — they state that is it preferable if "*the algorithm can be implemented securely and efficiently on a wide variety of platforms.*"

In this thesis we evaluate the performance of the eSTREAM portfolio algorithms and SHA-3 second round candidates. Given the general trend in architecture design, the low prices, and wide availability, it is of valuable interest to evaluate the performance of GPUs as cryptologic accelerators. For example, computing the message digest or encryption of a batch of fixed-length input messages, e.g., in high-end servers, can be efficiently accomplished with the implementations proposed in this work.

We further stress that these high performance devices can also be used to carry out cryptanalytic attacks and evaluations. To this end, we implement the cube attack to target a multi-GPU system and evaluate its performance when compared to a CPU; we find the multi-GPU framework to provide speedup factors of up to $100\times$. A direct consequence of this is the ability to carry out more sophisticated

attacks in less time, which further allows for the confirmation of existing and new results by third party researches. Supporting this scientific endeavor we make most of our code publicly available.

1.4 Related Work

The PlayStation 3 (PS3) video game console, which contains the Cell architecture, has been previously used to find chosen-prefix collisions for the cryptographic hash function MD5 [108]. Graphics cards have similarly been used for MD5 collision searches [22], password cracking [76], and accelerating cryptographic applications [75,109]. In [32] we presented our SHA-3 results; to the best of our knowledge, there is no previous work implementing the eSTREAM stream ciphers or second-round SHA-3 candidates on the NVIDIA GT200 GPUs.

In a closely related work, Kaminsky presented a parallel implementation of cube testers with application to the CubeHash hash function in [65], and Lathrop presented several cube attack results on SHA-3 cadidates in [71]. The best cube attack results on Trivium are due to Mroczkowski and Szmidt [82], building on the works of Dinur and Shamir [46], and Vielhaber [111]. In this work we present a complimentary multi-GPU cube attack framework and as an example analyze MICKEY v2 and Trivium. Our results confirm previously found attacks.

Chabaud and Joux [38] presented the first differential collision attack on SHA-0. Using a linearized model of the hash function, they found message differences that lead to a collision of the original hash function with a higher probability than the birthday bound. Similar strategies were later used by Rijmen and Oswald [101] on SHA-1 and by Indesteege and Preneel [64] on EnRUPT.

Pramstaller *et al.* [99] related the problem of finding highly probable linear

differences to the problem of finding low weight codewords of a linear code. A recent work of Brier *et al.* [34, 66] more precisely analyzed this relation for hash functions whose non-linear operations only consist in modular additions. They reformulate the problem of finding message pairs that conform to a linear differential trail to that of finding preimages of zero of a condition function. The search for such preimages is accelerated by the implicit use of message modification techniques. Given a linear differential trail, the concept further allows to estimate the corresponding complexity of the collision attack.

Section 2. B. 5 of [19] presents a complete survey of cryptanalytic results on CubeHash. The currently best collision attacks on CubeHash-r/b for message block sizes $b = 32, 64$ were presented in [34]. For $b = 32$ they present attacks of complexity $2^{54.1}$ and $2^{182.1}$ for $r = 4$ and $r = 6$ rounds, respectively. For $b = 64$ an attack of complexity 2^{203} for 7 rounds is given. No collision attack for more than 7 rounds was presented so far. Generic attacks are discussed by Bernstein in the appendix of [18]. In [67] we presented the two different methods discussed in this thesis for finding appropriate trails for variants of CubeHash, also discussed in [66].

Similarly, for BLAKE, Aumasson *et al.* present several cryptanalysis results in [8]. Various differential and invertibility properties of BLAKE's compression function are presented in [5]. Compared to their work, we present an analysis of BLAKE's susceptibility to general linear differential attacks, by extending the framework of [34], but do not, however, find any attacks better than those mentioned in [5, 8].

1.5 Contributions

The contributions of this thesis can be divided into two categories: implementations and analysis. Below we state these contribution, relating them to existing work in literature.

1.5.1 Implementation Contributions

We present a new software performance analysis of all eSTREAM stream ciphers and second-round SHA-3 candidate hash functions on the GPU. Our results are four-fold:

1. We presents an in-depth performance analysis of all algorithms by investigating their internal operations. It is worth noting that the aim of this work is not to claim that our techniques are optimal (hence, the provided estimates are indeed subject to change). Rather, our intended goal is to make a fair, reliable, and accurate comparison between all eSTREAM stream ciphers and, respectively, all second-round SHA-3 candidates. To facilitate the analysis of the SHA-3 candidates, we separate the AES-inspired candidates from the others. For the former case, we make extensive use of the work by Osvik et al. [95], which introduced the fastest results of AES on our target architecture. For the latter case, however, a more careful analysis, starting from scratch, was required.

2. We propose specific optimization techniques for our target platform; in combination with our estimation framework, more precise GPU estimates are given for all algorithms.

3. We complement this framework by providing real implementations of all

but one eSTREAM stream cipher, and, respectively, all non-AES based candidates on the target platform. We show that our techniques are indeed applicable, and that the base estimates are usually realistic.

4. In implementing the eSTREAM stream ciphers we developed an open source GPU stream cipher framework, called gSTREAM. The framework provides boiler-plate code for developers to easily port stream cipher implementations to the GPU; though we only show the implementation of the eSTREAM ciphers, other ciphers can easily be used with gSTREAM. A stream cipher implementation using gSTREAM transparently takes advantage of GPUs to accelerate system components.

We also present and benchmark two cryptanalysis frameworks:

1. To our knowledge we present the first multi-GPU cube attack framework. Our framework implements the preprocessing stage of the cube attack, i.e., it consists of a maxterm search algorithm, superpoly reconstruction algorithm and maxterm linearity test algorithm. As an example we apply the framework to the MICKEY v2 and Trivium stream ciphers, measuring speedup factors of up to $100\times$.

2. We extend the implementation of the linear differential framework presented in [34, 66] by parallelizing the code using OpenMP, providing subbyte message modification techniques, a reverse differential trail search algorithm, an adaptive backtracking algorithm, and a generic interface. Our framework provides for a linear speedup in the number of CPUs, the ability to find trails that are practical and leading to real collision, and finally the ability to analyze algorithms other than CubeHash, previously limited in [34].

1.5.2 Analysis Contributions

Using the cube attack and linear differential cryptanalysis methods we analyze four cryptographic primitives:

1. We use the multi-GPU cube attack framework to analyze simplified variants of the MICKE v2 and Trivium stream ciphers. For the former, we do not find any attacks using cubes of dimensions up to 20. However, for Trivium we confirm previously found results and present several new equations that can be used in an online attack to carry out a partial key recovery.

2. We use the linear differential framework to analyze variants of BLAKE and CubeHash. We apply two different methods for finding appropriate trails, including a reverse trail search and randomized trail search algorithm. For toy versions of BLAKE we find two collision attacks on both the 32-bit and 64-bit variants. We also find a 2^{508} second preimage attack on the FLAKE32 variant with 5 rounds, only slightly better than the theoretical 2^{512}.

   For several round parameters r and message block sizes b we present better collision attacks on CubeHash-r/b than those presented so far. Specifically, we analyze new variants of CubeHash-r/b for $r \in \{1, \ldots, 10\}$ and $b \in \{10, 20, 24, 36, 96\}$, find collisions of CubeHash-5/96 and give a theoretical attack of CubeHash-8/96 with estimated complexity of 2^{80} compression function calls. This improves over the generic attack with complexity of about 2^{128} and is the first collision attack on more than 7 rounds.

1.6 Thesis Outline

This thesis is organized as follows. In Chapter 2 we introduce the eSTREAM and SHA-3 algorithms, detailing the designs of the four algorithms we analyze. Chapter 3 introduces the OpenMP and CUDA programming models we use to implement our high-performance frameworks. In Chapter 4 and Chapter 5 we, respectively, describe the cube attack and linear differential cryptanalysis frameworks. Then, in Chapter 6 and Chapter 7 we present our implementation evaluation and cryptanalysis results, respectively. We conclude in Chapter 8.

Chapter 2

Target primitives

As previously mentioned, stream ciphers and cryptographic hash functions are widely-used primitives, core to many security systems. The security and efficiency of these primitives is crucial to many applications, and therefore, much research effort has been put into addressing these issues. In this thesis, we focus on analyzing and implementing various stream ciphers and cryptographic hash functions. Specifically, we evaluate the performance of the eSTREAM portfolio stream ciphers and Secure Hash Algorithm-3 (SHA-3) second-round candidates. Additionally, we cryptanalyze the stream ciphers Trivium and Mutual Irregular Clocking KEYstream generator (MICKEY), and hash functions BLAKE and Cube-Hash. Below, we briefly introduce the eSTREAM project and SHA-3 competition, along with a more detailed description of the four algorithms we analyze. We leave the analysis and implementation details to later chapters.

Profile 1 (Software)	Profile 2 (Hardware)
HC-128 [117]	Grain v1 [59]
Rabbit [31]	MICKEY v2 [11]
Salsa20/12 [16]	Trivium [42]
SOSEMANUK [15]	—

Table 2.1 eSTREAM portfolio algorithms.

2.1 eSTREAM Stream Ciphers

The European Union sponsored the four-year eSTREAM project in the hopes of identifying new stream ciphers as alternatives to the widely-used, though cryptographically insecure, ciphers [27, 74]. Moreoever, the project was initiated as a response to the failed attempt of the earlier New European Schemes for Signatures, Integrity, and Encryption (NESSIE) project to identify new promising stream ciphers [100]. The goal of the eSTREAM project was to create a portfolio of novel stream cipher designs that address security, performance, and resource-utilization. Submissions were considered for either high-performance software-oriented ciphers (Profile 1), or low-power and low-resource hardware-oriented ciphers (Profile 2). However, some submissions were initially considered for both profiles. Since the September 2008 revision [9, 10], the eSTREAM portfolio contains seven algorithms, listed in Table 2.1. In this thesis, we implement all but the SOSEMANUK stream cipher and analyze Trivium and MICKEY v2, two hardware-oriented stream ciphers. Below, we focus on the details of Trivium and MICKEY v2; we refer to the references given in Table 2.1 for details on the other ciphers.

2.1.1 Trivium

Trivium is a synchronous stream cipher designed by De Canni'ere and Preneel [41–43]. The cipher was selected to be part of the eSTREAM portfolio as one of the promising hardware-oriented stream ciphers to be widely adopted. Trivium supports 80-bit keys and up to 80-bit initial values, with which it can generate up to 2^{64} keystream bits. The very simple design structure makes Trivium a desirable target for cryptanalysts. Much effort has been put into breaking round-reduced versions of the cipher [3,46,79,103,111], yet, the full Trivium has withstood these efforts and remains secure.

Algorithm Specification

Trivium operates on a 288-bit internal state $s = s_1, \ldots, s_{288}$, which is organized as three shift registers of length 66, 69, and 66 respectively [79]. When clocking a register, the non-linear combination of several bits from one of the other registers is mixed with bits from its own state. During the state update (register clocking) the output of the three registers is combined to generate the keystream output. Although the three-register description is constructive in making several cryptanalytic observations on Trivium [79], we focus on a 'vectorized' approach. We assume that 288-bit operations such as shifts are available, when describing the algorithm.

Given a 80-bit key $k = k_1, \ldots, k_{80}$ and 80-bit initial value $IV = IV_1, \ldots, IV_{80}$, the Trivium algorithm can be broken down into two steps:

1. Key and IV setup:
    - Set $s_1, \ldots, s_{80}$ to k.
    - Set $s_{94}, \ldots, s_{174}$ to IV.

Algorithm 2.1: Trivium state update function

Data: Internal state $s = s_1, \ldots, s_{288}$.
Result: Clocked state $s = s_1, \ldots, s_{288}$.
Output: Keystrem output z.

1 **begin**
2 $t_1 \leftarrow s_{66} \oplus s_{93}$;
3 $t_2 \leftarrow s_{162} \oplus s_{177}$;
4 $t_3 \leftarrow s_{243} \oplus s_{288}$;
5 $z \leftarrow t_1 \oplus t_2 \oplus t_3$; `// Keystream output bit`
6 $t_1 \leftarrow t_1 \oplus (s_{91} \wedge s_{92}) \oplus s_{171}$;
7 $t_2 \leftarrow t_2 \oplus (s_{175} \wedge s_{176}) \oplus s_{264}$;
8 $t_3 \leftarrow t_3 \oplus (s_{286} \wedge s_{287}) \oplus s_{69}$;
9 $s \leftarrow s \gg 1$;
10 $s_{93} \leftarrow t_3$;
11 $s_{94} \leftarrow t_1$;
12 $s_{178} \leftarrow t_2$;

- Set $s_{286}, s_{287}, s_{288}$ to 1's, and remaining bits to 0's.
- Clock the state $N_{\text{pre}} = 4 \cdot 288$ times, disregarding the output.

2. Keystream generation: clock the state and return the output.

Denoting a i-bit right shit of α by $\alpha \gg i$, XOR by $\oplus$, and bitwise-AND by $\wedge$, the state update (clocking function) is given in Algorithm 2.1. Note that with the exception of line 9, all operations are on single bits.

Brief Design Rationale

As can be inferred from the above description, Trivium's design is "an exercise in exploring how far a stream cipher can be simplified without sacrificing its security, speed or flexibility" [42]. Despite its simplity, the design does not trade security for speed or area. Consider, for example, the non-linear state update. This protects against correalation attacks that attempt to recover the state from either the keystream bits alone, or a combination of keystream and state bits [41,42].

The non-linear state update used in Trivium does, however, makes the analysis of its period more difficult to determine. However, the designers estimated the probability that a cycle (after a large number of iterations) is smaller than 2^{80} to be negligible (2^{-208}) [42].

2.1.2 MICKEY v2

MICKEY[1] is a synchronous stream cipher selected for Profile 2 of the eSTREAM portfolio, along with Trivium and Grain. MICKEY was designed by Babbage and Dodd [11], targeting applications requiring high security ciphers in resource-constrained environments. The design is based on the mutual (and irregular) clocking of two shift registers. MICKEY provides for the secure generation of 2^{40} keystream bits using a 80-bit key and (at most) 80-bit IV.

Algorithm Specification

The MICKEY algorithm has an internal state of 200 bits, evenly divided between registers $r = r_0, \ldots, r_{99}$ and $s = s_0, \ldots, s_{99}$. Similar to Trivium, MICKEY uses a state update function througout the algorithm. However, its update function, called `ClockKG`, is slightly more complex than that of Trivium. `ClockKG` takes a mixing bit m and input bit i as parameters, thus we denote it by $\texttt{ClockKG}(m, i)$. Before delving into the details of the update funciton, we note that, like most stream ciphers, the MICKEY algorithm may be split into two steps:

1. Key and IV setup:
    - For each IV bit $IV_i : 0 \leqslant i \leqslant 79$, update the state with $\texttt{ClockKG}(1, IV_i)$.
    - For each key bit $k_i : 0 \leqslant i \leqslant 79$, update the state with $\texttt{ClockKG}(1, k_i)$.

[1] Unless explicitly noted, from this point, when referring to MICKEY we imply MICKEY v2.

- Update the state $N_{\text{pre}} = 100$ times with $\texttt{ClockKG}(1, 0)$.

2. Keystream generation: update the state with $\texttt{ClockKG}(0, 0)$ and return $r_0 \oplus s_0$ as the keystream output.

Following [11], ClockKG is described in Algorithm 2.2. The core functions $\texttt{Clock}_r$ and $\texttt{Clock}_s$, used on lines 6 and 7 to carry out the actual state update, are presented in Algorithm 2.3 and Algorithm 2.4, respectively. The constant tap vector T, and bit vectors C_0, C_1, F_0, and F_1 are given in Appendix B.

Algorithm 2.2: MICKEY state update function

Input : Mixing bit m.
Input bit i.
Data: Internal state $r = r_0, \ldots, r_{99}$ and $s = s_0, \ldots, s_{99}$.
Result: Clocked state $r = r_0, \ldots, r_{99}$ and $s = s_0, \ldots, s_{99}$.

1 **begin**
2 $\quad c_r \leftarrow s_{34} \oplus r_{67}$;
3 $\quad c_s \leftarrow s_{67} \oplus r_{33}$;
4 $\quad i_r \leftarrow i \oplus (s_{50} \wedge m)$;
5 $\quad i_r \leftarrow i$;
6 $\quad r \leftarrow \texttt{Clock}_r(r, i_r, c_r)$;
7 $\quad s \leftarrow \texttt{Clock}_s(s, i_s, c_s)$;

We note that the two (vectorized) algorithms $\texttt{Clock}_r$ and $\texttt{Clock}_s$, individually clocking the r and s registers, respectively, are equivalently expressing the original individual-bit descrition of [11], though in a more compact form. For example, line 3 is equivalent to the following line from [11]: $r'_i = r_{i-1}$ for $1 \leqslant 1 \leqslant 99$; $r'_0 = 0$.

Brief Design Rationale

As explained in [11], MICKEY uses the r register to guarantee the period and local statistical properties of the keystream generator. We observe that when updating r (using $\texttt{Clock}_r$) with the control bit $c_r = 0$, the update of r is linear, i.e., r simply

Algorithm 2.3: $\texttt{Clock}_r$ - clocking MICKEY's r register

Input : Internal state regiester $r = r_0, \ldots, r_{99}$.
Input bit i_r.
Control bit c_r.
Output: Clocked state register $r' = r'_0, \ldots, r'_{99}$.

begin
  $f \leftarrow r_{99} \oplus i_r$;
  $r' \leftarrow r \gg 1$;
  if $f = 1$ **then**
    $r' \leftarrow r' \oplus T$; `// if` i `is a tap position, add` f
  if $c_r = 1$ **then**
    $r' \leftarrow r' \oplus r$;
  return r';

Algorithm 2.4: $\texttt{Clock}_s$ - clocking MICKEY's s register

Input : Internal register state $s = s_0, \ldots, s_{99}$.
Input bit i_s.
Control bit c_s.
Output: Clocked register state $s' = s'_0, \ldots, s'_{99}$.

begin
  $f \leftarrow s_{99} \oplus i_s$;
  $s' \leftarrow (s \gg 1) \oplus ((s \oplus C_0) \wedge ((s \ll 1) \oplus C_1))$;
  $s'_0 \leftarrow 0$;
  $s'_{99} \leftarrow s_{98}$;
  if $f = 1$ **then**
    if $c_r = 0$ **then** $s' \leftarrow s' \oplus F_0$;
    else $s' \leftarrow s' \oplus F_1$
  return s';

behaves like a linear feedback shift register—a common approach to guaranteeing the period of a cipher. Conversely, when the control bit $c_r = 1$, the update function is effectively clocking r a total of $2^{50} - 157$ times [11]. The careful and irregular variable clocking protects against several statistical attacks attempting to guess the number of times the state has been clocked.

Complementary to the r register, MICKEY's s register provides for high non-linearlities in the keystream and state bits. More specifically, the goal of the s register (and its update function) is to address correlation and distinguishing attacks, as explained in [11]. Finally, because bits from both registers are used to (mutually) control the state update functions, as shown in `ClockKG`, divide-and-conquer attacks become infeasible, i.e., it is not possible to predict future r (or s) values with full knowledge of only one register's state.

2.2 SHA-3 Candidates

Similar to the growing interest in stream cipher design and analysis research as a result of eSTREAM, the design and analysis of cryptographic hash functions have come under renewed interest with the public hash function competition [89] commenced by NIST. The goal of the competition is to develop a new cryptographic hash algorithm to replace the current standard, SHA-2 [87]. The new hash algorithm will be called 'SHA-3' and will be subject to a Federal Information Processing Standard (FIPS), similar to the Advanced Encryption Standard (AES) [90]. The competition is NIST's response to recent advances in the cryptanalysis of hash functions [44, 69, 72, 106, 115], especially to serious attacks against the widely-deployed algorithms MD5 [102] and SHA-1 [86]. Although these cryptanalytic breakthroughs have no direct effect on SHA-2, a successful

BLAKE [8]	BMW [55]	CubeHash [20]	ECHO [13]	Fugue [56]
Grøstl [54]	Hamsi [70]	**JH** [118]	**Keccak** [21]	Luffa [37]
Shabal [33]	SIMD [73]	SHAvite-3 [25]	**Skein** [49]	–

Table 2.2 SHA-3 second-round candidates.

attack on SHA-2 would have catastrophic effects on the security of applications that rely on it. The structural similarities between SHA-2 and its broken ancestors have lead many cryptographers and cryptanalysts to believe that successful attacks are reachable in the near future.

Since the commencement of the SHA-3 competition in October 2008, the number of candidate algorithms have been narrowed down (in July 2009) from 51 to 14 (second-round) candidates, shown Table 2.2, to the 5 finalists (highlighted in Table 2.2). The new hash function standard(s) will be announced in 2012. Similar to the eSTREAM criteria, these candidates are reviewed based on security, cost, and algorithmic and implementation characteristics [88]. In this thesis, we present performance estimates for all 14 algorithms in addition to analyzing the BLAKE and CubeHash hash functions. Below we focus on the latter, the details for the remaining algorithms can be found in their corresponding NIST submission documents[2].

2.2.1 BLAKE

BLAKE is a family of hash functions designed by Aumasson *et al.* for the SHA-3 competition [8]. BLAKE is one of the 14 second-round candidates, having both strong security and efficient implementation properties. The family consists of

[2]See http://csrc.nist.gov/groups/ST/hash/sha-3/

BLAKE-$\{28, 32, 48, 64\}$, each respecively corresponding to NIST's requirement of hash functions with output lengths of $h = 224, 256, 384$ and 512 bits.

The first two hash functions operate on 32-bit words, while the latter two operate on 64-bit words. Additionally, we note that BLAKE-28 and BLAKE-48 are simply truncated versions of BLAKE-32 and BLAKE-64, respectively, with different initial vectors. As BLAKE-64 is only a slightly modified 64-bit version of BLAKE-32, we focus on the description of the latter.

Algorithm Specification

BLAKE-32 operates on a 512-bit internal state, composed of 16 words $v = v_0, \ldots, v_{15}$. Each v_i is a 32-bit word, and all of the operations are truncated to the word length, i.e., addition of two 32-bit words is truncated modulo 2^{32}. BLAKE is an iterated hash function following the HAsh Iterative FrAmework (HAIFA) [24], and like most iterated hash function it can be broken down into three steps:

1. Pad the input message M to a sequence of 512-bit blocks $M' = M'_0 \| \cdots \| M'_N$:

   - Append a 1 bit to M.
   - Append the least number of 0 bits to reach a length congruent to 447 mod 512.
   - Append a 1 bit.
   - Append the 64-bit representation of the message length ℓ: $[\ell]_{64}$.

   Thus, $M' = M \| 10 \cdots 01 \| [\ell]_{64}$.

2. Using the compression function, process every 512-bit block M'_i.

3. Output the h bits of the last compressions's output.

Algorithm 2.5: BLAKE's compression function

Input : Chain value $h^i = h_0^i, \ldots, h_7^i$.
Message block $m = m_0, \ldots, m_{15}$.
Salt $s = s_0, \ldots, s_3$.
Counter $t^i = t_0^i, t_1^i$.
Output: New chain value $h^{i+1} = h_0^{i+1}, \ldots, h_7^{i+1}$.

begin
  `// Initialization:`
  $$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} \leftarrow \begin{pmatrix} h_0^i & h_1^i & h_2^i & h_3^i \\ h_4^i & h_5^i & h_6^i & h_7^i \\ s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\ t_0^i \oplus c_4 & t_0^i \oplus c_5 & t_1^i \oplus c_6 & t_1^i \oplus c_7 \end{pmatrix};$$
  for $r \leftarrow 0$ **to** $N_r - 1$ **do**
    `// Column step:`
    $\mathtt{G}_0(v_0, v_4, v_8, v_{12})$;
    $\mathtt{G}_1(v_1, v_5, v_9, v_{13})$;
    $\mathtt{G}_2(v_2, v_6, v_{10}, v_{14})$;
    $\mathtt{G}_3(v_3, v_7, v_{11}, v_{15})$;
    `// Diagonal step:`
    $\mathtt{G}_4(v_0, v_5, v_{10}, v_{15})$;
    $\mathtt{G}_5(v_1, v_6, v_{11}, v_{12})$;
    $\mathtt{G}_6(v_2, v_7, v_8, v_{13})$;
    $\mathtt{G}_7(v_3, v_4, v_9, v_{14})$;
  `// Finalization:`
  for $j \leftarrow 0$ **to** 7 **do**
    $h_j^{i+1} \leftarrow h_j^i \oplus s_{(j \mod 4)} \oplus v_i \oplus v_{i+8}$;

We note that the BLAKE-64 hash function is identical except for the working sizes: the v_i's are 64-bit words, the message blocks are 1024-bit, and the length is 128-bit.

The compression function, which we call Compress, takes as input a 8-word chain value $h^i = h_0^i, \ldots, h_7^i$, a 16-word message block $m = m_0, \ldots, m_{15}$, a 4-word salt $s = s_0, \ldots, s_3$, and a 2-word counter $t^i = t_0^i, t_1^i$, producing a new chain value $h^{i+1} = h_0^{i+1}, \ldots, h_7^{i+1}$. The three-step $\mathsf{Compress}(h^i, m, s, t^i)$ function is given in Algorithm 2.5. The number of rounds N_r for BLAKE-32 is 10, while for BLAKE-64 it's 14.

Algorithm 2.6: BLAKE-32 $\mathtt{G}_i$ function

Data: Message block $m = m_0, \ldots, m_{15}$.
Four state words a, b, c, d.
Result: Transformed state words a, b, c, d.

begin
  $a \leftarrow a + b + (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)})$;
  $d \leftarrow (d \oplus a) \ggg \omega_0$;
  $c \leftarrow c + d$;
  $b \leftarrow (b \oplus c) \ggg \omega_1$;
  $a \leftarrow a + b + (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)})$;
  $d \leftarrow (d \oplus a) \ggg \omega_2$;
  $c \leftarrow c + d$;
  $b \leftarrow (b \oplus c) \ggg \omega_3$;

The core of the compression function is the $\mathtt{G}_i$ function, a modification of Bernstein's ChaCha cipher [17]. Denoting a right-rotate of α by i bits as $\alpha \ggg i$, we present the $\mathtt{G}_i$ function in Algorithm 2.6. The rotation constants $\omega_0, \ldots, \omega_3$ are $\{16, 12, 8, 7\}$ for BLAKE-32, and $\{32, 25, 16, 11\}$ for BLAKE-64. The c_i constants and permutation functions $\sigma_r(i)$ are given in Appendinx A.

Using the initial values given in Table A.1 of Appendix A, Step 2 of the BLAKE algorithm can be expressed according to the relation: $h^{i+1} \leftarrow \mathsf{Compress}(h^i, M'_i, s, t^i)$. The counter t^i is the number of message bits in $M'_0 \| \cdots \| M'_i$, excluding the bits added by padding; if the last block contains only padding bits, then $t^N = 0$. Finally, Step 3 simply consists of returning the (truncated) h^N.

Brief Design Rationale

BLAKE's design is based on pre-existing, and well-studied components [8]. As mentioned, BLAKE's iteration mode is based on HAIFA, which allows for hashing with salt and randomized hashing. Hashing with a salt is used in various applications, the most common of which are password-based authentica-

tion applications[3]; a salt, usually a random nonce or counter, is used to effectively provide a different hash function using a common algorithm [8]. More importantly, the HAIFA iteration mode addresses some of the issues found in the widely-used Merkle-Damgård construction (including resistance to preimage and second-preimage attacks). The local wide-pipe structure, i.e., having an internal state larger than the chain-value, and message-dependent rounds provide for security against collision-attacks [6,8]. Finally, the compression function being based on the stream cipher ChaCha [17], which has better diffusion properties over the well-regarded Salsa20 cipher [16], greatly increases the confidence in its resilince to generic attacks.

Toy Versions

Encouraging external cryptanalysis, the designers provide four simplified, toy, versions of BLAKE. Since part of our analysis is on the toy versions, the details of the four variations, as described in [7], are given below:

BLOKE: The σ_r's are simply the identity.

FLAKE: The compression finction has no feedforward, i.e., line 13 of Algorithm 2.5 is changed to: $h_j^{i+1} \leftarrow v_i \oplus v_{i+8}$

BLAZE: The c_j constants of the $\mathtt{G}_i$ functions are all 0.

BRAKE: Combination of BLOKE, FLAKE, and BLAZE.

[3] For example, password authentication applications commonly store a salt along the hash of the salt$\|$password to prevent attackers from pre-computing tables that can be used in speeding up a brute force password search.

2.2.2 CubeHash

CubeHash, designed by Bernstein [18], is also a second-round candidate in the SHA-3 competition along with BLAKE. The cryptographic hash function was designed with tweakable parameters r, b, and h, which specify the number of rounds, the number of bytes per message block, and the hash output bit length, respectively. We denote the parametrized function as CubeHash-r/b. Currently, the official proposal for all NIST-required digest lengths $h = \{224, 256, 384, 512\}$ is CubeHash-16/32, truncated to the desired output bitlength. The initial proposal of CubeHash-8/1 was tweaked to CubeHash-16/32 for the second round [19]; the new parameter choices effectively speed up the function by a factor of 16, while still keeping the security margin very high. Furthermore, the author explicitly encourages external cryptanalysis with larger values of b and fewer number of rounds r.

Algorithm Specification

CubeHash operates on a 1024-bit internal state X, composed of 32 words $X = X_0, \ldots, X_{31}$, where each X_i of the internal state is a 32-bit word. All of CubeHash's operations (add, XOR, and rotate) are 32-bit operations, i.e., all additions are modulo 2^{32}. The algorithm consists of five steps:

1. Pad the input message M to a sequence of b-byte blocks $M' = M'_0 \| \cdots \| M'_n$:
   - Append a 1 bit to the input message M.
   - Append the least number of 0 bits required to reach a multiple of b-bytes.

   Thus, $M' = M \| 10 \cdots 0$.

2. Initialize the internal state X:

   - Set X_0 to $h/8$, X_1 to b, X_2 to r, and the remaining X_i's to 0.
   - Using a round function, transform the state through $10r$ identical rounds.

3. For every b-byte block M'_i:

   - XOR M'_i into the first b-bytes of X.
   - Transform the state through r identical rounds.

4. Finalize the state:

   - XOR 1 into X_{31}.
   - Transform the state through $10r$ identical rounds.

5. Output the first h bits of X.

Denoting an i-bit left rotate of α by $\alpha \lll i$, the aforementioned round function is given in Algorithm 2.7.

Brief Design Rationale

Compared to most other SHA-3 second-round candidates, CubeHash has a very simple design. The designer specifically avoids block counters, message padding methods that append the message length to the input, or other techniques commonly used to prevent collision attacks. Shown in [19], the very large state of CubeHash is itself a countermeasure against such attacks. Furthermore, the high degree of symmetry in the cipher allows for very efficient hardware and software implementations, while the constant-time operations and lack of complex message-dependent lookups (common in many designs) prevents possible timing-related attacks, such as cache attacks [94].

Algorithm 2.7: Round funtion for CubeHash

Data: Internal state $X_0, \ldots, X_{31}$.
Result: Round transformed internal state $X_0, \ldots, X_{31}$.

begin
  for $i \leftarrow 0$ **to** 15 **do**
    $X_{i+16} \leftarrow X_i + X_{i+16}$;
    $X_i \leftarrow X_i \lll 7$;
  for $i \leftarrow 0$ **to** 7 **do** `Swap`(X_i, X_{i+8});
  for $i \leftarrow 0$ **to** 15 **do** $X_i \leftarrow X_i \oplus X_{i+16}$;
  foreach $i \in \{16, 17, 20, 21, 24, 25, 28, 29\}$ **do** `Swap`(X_i, X_{i+2});
  for $i \leftarrow 0$ **to** 15 **do**
    $X_{i+16} \leftarrow X_i + X_{i+16}$;
    $X_i \leftarrow X_i \lll 11$;
  foreach $i \in \{0, 1, 2, 3, 8, 9, 10, 11\}$ **do** `Swap`(X_i, X_{i+4});
  for $i \leftarrow 0$ **to** 15 **do** $X_i \leftarrow X_i \oplus X_{i+16}$;
  foreach $i \in \{16, 18, 20, 22, 24, 26, 28, 30\}$ **do** `Swap`(X_i, X_{i+1});

Chapter 3

SMP and GPU Parallel Programming

Although the transistor count per chip is still doubling every two years according to Moore's law [80, 81],'traditional' processor (CPU) design techniques that increase clock rates and add complex features are no longer advancing at a matching pace. Among other limitations, power dissipation has become an increasingly difficult problem for high clock rate processor designs. Alternative approaches, the most successful of which is the *multi-core*, have become the norm [97]. Rather than adding more complex functional units that run at very high speeds, multi-core CPUs take advantage of the increasing data- and task-parallelism to keep up with Moore's law. Incorporating multiple processors on a single die, the multi-core CPU is a direct improvement on *multiprocessor* designs, which have been extensively used in server environments for many years [61]. Additionally, many CPU manufactures are persistently researching methods to increase the number of cores on a die—Intel's recent release (for research) 48-core Single-chip Cloud Computer [63] is an example of this progress.

Similar to the CPU design trend to achieve tera-scale computing, addressing the increasing computational requirements of graphics-related applications (e.g.,

games and high-definition video) graphics processing units (GPUs) have become very powerful and highly-parallel processors. Many GPUs are equipped with hundreds to thousands of streaming processor cores[1]. For example, the NVIDIA GTX 285, GTX 480, and ATI Radeon 5870 have 240, 480, and 1600 streaming processors, respectively. Additionally, these streaming processors are clocked at reasonably high clock rates, when compared to CPU clock rates, usually between 850 MHz and 1.5 GHz. The vast raw computational power of modern GPUs has incited research interest in computing outside the graphics-community. Recently, GPUs have become a common target for numerically-intensive applications given their ease of programming (relative to previous generation GPUs), and ability to outperform CPUs in data-parallel applications by orders of magnitude.

In this chapter we review the *shared-memory parallel computer* (SMP) programming paradigm with Open Multi-Processing (OpenMP) and GPU parallel programming with the compute unified device architecture (CUDA). We limit our discussion as it pertains to programming SMPs and GPUs for cryptologic applications; for more complete introductions to these topics see [39] and [68].

3.1 OpenMP and SMPs

There are a number of application programming interfaces (APIs) that facilitate parallel programming for multiprocessor and multi-core architectures. Of these, the most commonly used APIs are Message-Passing Interface (MPI) [53], POSIX threads (Pthreads) [107], and OpenMP [30]. In parallelizing various parts of out cryptanalytic attacks, discussed in Chapters 5 and 4, we use OpenMP for its sim-

[1]These streaming processors or 'cores', as named by ATI and NVIDIA, are essentially complex arithmetic logic units (ALUs).

plicity and support for incremental parallelization.

Parallelizing already-implemented algorithms using MPI and Pthreads usually requires major code rewriting and explicit handling of thread launch, join, and communication. Conversely, with OpenMP little effort is required to parallelize data- or thread-parallel programs. One may incrementally parallelize different sections of the code without having to focus on low-level thread execution and communication details. Moreover, the programmer focuses on algorithm parallelization, rather than the threading details.

3.1.1 SMP Architectures

As previously mentioned, we limit our OpenMP programming details to architectures where the address space is shared, i.e., SMPs, and the caches are coherent. Extending OpenMP to large-scale clusters is beyond the scope of this chapter and we refer to [62] for details (or taking a MPI+OpenMP hybrid approach). Following [39], our definition of SMP encompasses all shared-memory architectures, including *uniform memory access* (UMA[2]) architectures and *cache-coherent non-uniform memory access* (cc-NUMA) architectures. UMA architecture, shown in Figure 3.1, consists of multiple processors, each with its own private cache, sharing a single main memory. In this architecture, memory access latency is the same for all the processors (hence the name uniform memory access) and a cache coherence protocol is usually implemented to address inconstancies that might arise if multiple CPUs operate on the same data.

[2]Note that, in literature, processors implementing the UMA architectures are often called symmetric (shared-memory) multiprocessors, which is also abbreviated by SMP. In this thesis, when using the term SMP we do not explicitly imply symmetric shared-memory multiprocessors, rather we refer to the more-general shared-memory parallel computer architecture.

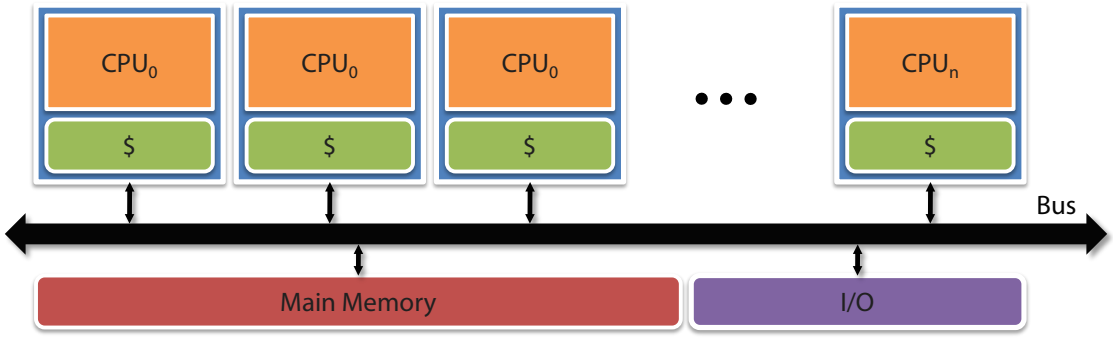

Figure 3.1 Uniform memory access architecture with n processors, caches ($), main memory and input/output (I/O) system.

Although most of the current multi-core processor designs are UMA multiprocessors, NUMA scales beyond the 10–16 core UMA systems and is a popular architecture for multiprocessor designs. Unlike UMA, in NUMA, the memory (in addition the cache) is distributed among the CPU nodes. As shown in Figure 3.2, the latency of CPU_0 accessing data from CPU_n's memory is considerably greater than that of a nearby processor, hence the name non-uniform memory access. We point out that Figure 3.2 has been simplified for clarity; it is very common for the processor nodes to be multi-core processors. Additionally, although it is possible to handle the cache coherency problem in software, popular multiprocessor NUMA architectures commonly implement this in hardware and are referred to as cache-coherent NUMA architectures.

3.1.2 OpenMP Programming

The OpenMP C/C++ and Fortran APIs are composed of compiler directives (to create threads and distribute work), a runtime library (to provide for thread and environment information at runtime), and environment variables (to control

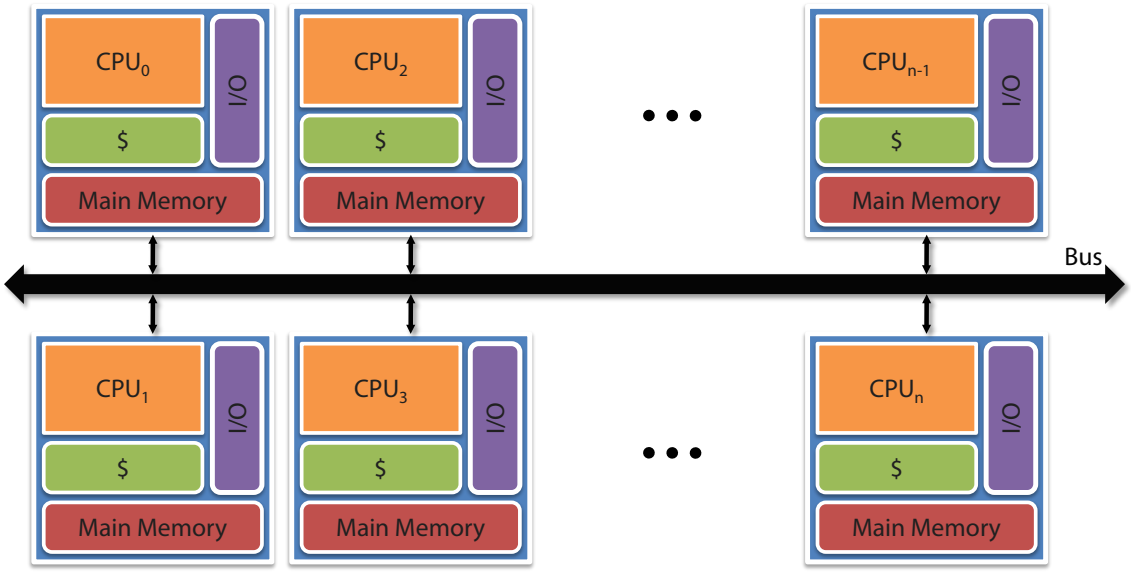

Figure 3.2 Non-uniform memory access architecture with n interconnected nodes. Each node consists of a CPU, local caches, memory and an I/O system.

the parallelism). The programming paradigm use by OpenMP is the fork-join model [39], shown in Figure 3.3. In this programming model, a single thread (called the *master thread*) *forks* into multiple threads, working in parallel and usually on different cores. Once the parallel work is completed, the threads *join* (to possibly combine results) and the master thread continues execution. In Figure 3.3, we denote a thread by ⇝, a point where the master thread forks by ○, and a point where the team of threads join by ○. As shown in the figure, there can be multiple parallel and multiple intermediate serial sections of an OpenMP-parallelized program. Moreover, the number of threads per parallel section (or block) need not always equal the number of cores—the number of threads may be specified at compile-time or dynamically set at runtime.

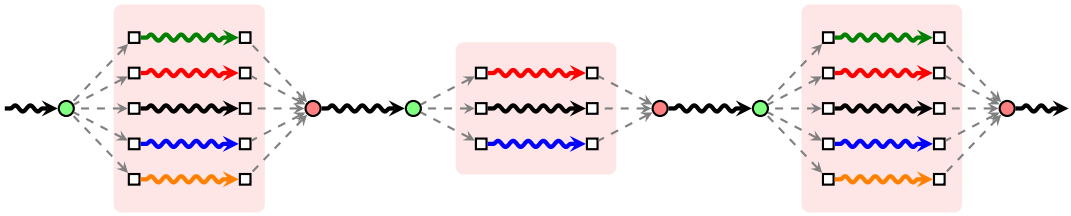

Figure 3.3 Fork-join programming model. The master thread forks (at the green (circle) synchronization point) into a number of threads which work in parallel (highlighted in pink) and join at a later point (indicated by the red (circle) synchronization point).

Creating Threads

To create a group of threads, i.e., fork, in an OpenMP C program, the programmer pre-pends a *parallel* directive (also known as *parallel construct*) to a (compound) statement[3]. Consider the following trivial example:

```
1 #pragma omp parallel
2 {
3     printf("Thread %d in parallel block!\n",omp_get_thread_num());
4 }
```

The first line is the OpenMP parallel directive that instruct the C compiler to parallelize the code in the compound statement. The compound statement, in this case, is a simple statement that uses the OpenMP runtime library function omp_get_thread_num() to get the thread id of the current thread and prints it using printf(...). On a quad-core machine the example produced the following output:

```
Thread 3 in parallel block!
Thread 2 in parallel block!
Thread 1 in parallel block!
Thread 0 in parallel block!
```

As the example shows, the threads in a parallel region are not necessarily scheduled to execute in sequential order, i.e., thread n can finish executing ahead of thread $m : n > m$, and thus care must be take avoid programming dependencies

[3]We assume the reader is familiar with the C programming language. For a reference, see [57].

Clause	Description
if(*expression*)	If the compile-time *expression* evaluates to false, the block is not executed in parallel, i.e., it is inactive.
num_threads(*expression*)	The *expression* (must evaluate to positive integer) specifies the number of threads per team.
private($iden_0, \ldots, iden_n$)	Make the variables $iden_0, \ldots, iden_n$ private to the thread.
firstprivate($iden_0, \ldots, iden_n$)	Extend **private**() to initialize variables to the respective values set before reaching parallel region.
shared($iden_0, \ldots, iden_n$)	Share the variables $iden_0, \ldots, iden_n$ amongst all threads.
default({none, shared})	Specify the data-sharing attribute of all variable used in the parallel region and declared before the directive.
reduction(*op*:$iden_0, \ldots, iden_n$)	Perform a parallel reduction on shared variables $iden_0, \ldots, iden_n$ using operator *op*.

Table 3.1 Commonly used parallel construct clauses.

on the scheduler's algorithm.

More formally, the **parallel** construct has the form:

#pragma omp parallel [$clause_0$],[$clause_1$], . . . , [$clause_n$]

where the optional clauses are used to specify various attributes of the parallel region following the pragma directive. We present some of the most commonly used clauses in Table 3.1; the interested reader is referred to [39] for additional details and a more complete list of clauses. We note that since the **if** clause specifies whether the parallel region is active or inactive, and the **num_threads** specifies the number of threads executing the parallel region, each may only appear once in the directive.

To clarify the use of the **if**, **num_threads**, **private**, and **firstprivate** clauses consider

the following C OpenMP function.

```
1  void simple_example(void) {
2      int i=0x2A, j=0x539, s=0xA5;
3      int iter=0;
4
5      for(iter=0;iter<2;iter++) {
6          printf("Iteration %d...\n",iter);
7  #pragma omp parallel num_threads(5) \
8                                  private(i) firstprivate(j) if(s!=0xBADCAFE)
9          {
10             printf("Thread %d: i=0x%08X, j=0x%08X, s=0x%08X\n",
11                                                     omp_get_thread_num(),i,j,s);
12
13             i=0xDEADBEEF; j=0xBADF00D;
14             if(omp_get_thread_num()==4) { s=0xBADCAFE; }
15
16             printf(" %d: i=0x%08X, j=0x%08X, s=0x%08X\n",
17                                                     omp_get_thread_num(),i,j,s);
18         } /* implicit barrier */
19         printf("Serial : i=0x%08X, j=0x%08X, s=0x%08X\n\n",i,j,s);
20     }
21 }
```

We first point out that in the variable s, which does not appear in any clause, is a shared variable. Unless the **default**(none) clause is explicitly used, and thus every variable must explicitly appear in a **shared**, **private**, or **firstprivate** clause, the compiler implicitly uses the **default**(shared) clause. Furthermore, a *barrier* is implicitly inserted at the end of the parallel compound statement where threads join (each waiting until the last thread completes execution of the block). On a quad-core machine the example produced the following output:

```
Iteration 0...
Thread 0: i=0x54E9B300, j=0x00000539, s=0x000000A5
Thread 1: i=0xF33875D5, j=0x00000539, s=0x000000A5
       1: i=0xDEADBEEF, j=0x0BADF00D, s=0x000000A5
Thread 3: i=0x00000000, j=0x00000539, s=0x000000A5
       3: i=0xDEADBEEF, j=0x0BADF00D, s=0x000000A5
       0: i=0xDEADBEEF, j=0x0BADF00D, s=0x000000A5
Thread 4: i=0x00000000, j=0x00000539, s=0x000000A5
       4: i=0xDEADBEEF, j=0x0BADF00D, s=0x0BADCAFE
```

```
Thread 2: i=0x00000000, j=0x00000539, s=0x000000A5
       2: i=0xDEADBEEF, j=0x0BADF00D, s=0x0BADCAFE
Serial  : i=0x0000002A, j=0x00000539, s=0x0BADCAFE

Iteration 1...
Thread 0: i=0x54E9B300, j=0x00000539, s=0x0BADCAFE
       0: i=0xDEADBEEF, j=0x0BADF00D, s=0x0BADCAFE
Serial  : i=0x0000002A, j=0x00000539, s=0x0BADCAFE
```

Notice that the number of threads in the first iteration is equal to the expression in the **num_threads** clause, 5. Additionally s is set to 0xBADCAFE in the first iteration, and thus, the **if** clause evaluates to false in second iteration—the parallel block is inactive and only executed by the master thread. For both iterations, we highlight (in red) the output of the private variable i before setting it to a new value on line 12; unlike j, which appears in the **firstprivate** clause, the value of i upon enterance to the parallel block is unknown. Additionally, as one would expect, modifying a private variable in the parallel block has no effect on the outside scope; as shown by the "Serial..." output lines, the changes on line 12 to i and j do not appear outside the parallel block. Finally, note that the modification of the shared variable s by thread 4 is read by thread 2 (highlighted in blue)[4].

Assigning and Sharing Work

In the previous section we introduced the **parallel** construct which is used to create a team of threads to execute a block of code. However, in practical applications, we usually want to distribute the work amongst the different threads. As explained in [39], to do this, OpenMP provides three different constructs: **for**, **sections**, and **single**. The **for** (or loop) construct breaks the iterations of a for-loop

[4]This latest-modification read is, however, not deterministic; to guarantee coherence one must use barriers or atomically update the value. In this example, thread 2 could likely have read the old value.

Clause	Description
lastprivate($iden_0, \ldots, iden_n$)	Extend **private**() to set the variables outside the construct to last respective values in the parallel block.
copyprivate($iden_0, \ldots, iden_n$)	Broadcast the value of the private variables $iden_0, \ldots, iden_n$ to other threads upon exiting **single** construct block.
nowait	Ignore any implicit barriers.
ordered	Execute parallel block of code in-order.
schedule(*kind*[, *size*])	Specify the loop iteration distribution amongst threads.

Table 3.2 Clauses supported by the work-share constructs.

amongst the various threads; the **sections** construct explicitly divides the work into sections to be executed in parallel by different threads; and the **single** construct specifies a block that is to only be carried out by one thread.

In addition to the **private**, **firstprivate**, **shared**, and **reduction** clauses, the loop construct also supports the **lastprivate**, **ordered**, **schedule**, and **nowait** clauses. Similarly, the **sections** construct has addtional suport for the **lastprivate** and **nowait** clauses, while the **single** construct only supports the **private**, **firstprivate**, **copyprivate**, and **nowait** clauses. Table 3.2 gives a brief explanation of these clauses.

We refer the interested reader to [39] for a more formal description of the work-sharing constructs and their clauses and, instead, present four examples that concretely use these concepts. The first example demonstrates the use of the **single** construct. In this example we use one of the synchronization constructs, specifically the **barrier** construct, that forces all the threads to wait at the synchronization point before continuing execution. Below is the aforementioned example:

```
1 #pragma omp parallel
2 {
3    int p_i=omp_get_thread_num();
```

```
4     printf("-Thread %d: p_i=%d\n",omp_get_thread_num(),p_i);
5  #pragma omp barrier
6
7  #pragma omp single copyprivate(p_i)
8     {
9        printf("\tThread %d is special!\n",omp_get_thread_num());
10       p_i=omp_get_thread_num();
11    } /* implicit barrier*/
12    printf("+Thread %d: p_i=%d\n",omp_get_thread_num(),p_i);
13 }
```

Only one thread executes the compound statement following the **single** construct, setting the private variable p_i to its thread id and broadcasting the new value to the remaining threads in the team. The example produced the following output:

```
-Thread 2: p_i=2
-Thread 0: p_i=0
-Thread 3: p_i=3
-Thread 1: p_i=1
        Thread 3 is special!
+Thread 2: p_i=3
+Thread 3: p_i=3
+Thread 0: p_i=3
+Thread 1: p_i=3
```

Similarly, consider:

```
1  #pragma omp parallel num_threads(3)
2  {
3  #pragma omp sections
4     {
5  #pragma omp section
6           printf("Thread %d can add!\n",omp_get_thread_num());
7  #pragma omp section
8           printf("Thread %d can subtract!\n",omp_get_thread_num());
9  #pragma omp section
10          printf("Thread %d can multiply!\n",omp_get_thread_num());
11    }
12 }
```

This example uses the **sections** construct to break the work in the parallel block into three sections, to be executed by different threads. In this case the work is

trivial and only consists of printing out the thread id, as the output shows:

```
Thread 2 can subtract!
Thread 1 can add!
Thread 0 can multiply!
```

Most programs spend a major part of their execution in loops. Therefore, optimizing and parallelizing loops is very important. Many compilers already perform various loop optimizations, including loop- unrolling, -fusion, -distribution and -tiling [1, 83]. As mentioned, OpenMP provides a special loop construct to support loop iteration distribution. We refer the reader to [39] for a description of the static, runtime, and guided scheduling methods; below, we consider a simple example that uses the **schedule** clause with *kind* set to dynamic:

```
1  void example_for(void) {
2      int N=1000,i,sum=0;
3      int array[N];
4
5      for(i=0;i<N;i++) { array[i]=i; }
6
7  #pragma omp parallel shared(sum,array)
8      {
9          int tid=omp_get_thread_num();
10         int i_sum=0; /* intermediate sum */
11 #pragma omp for schedule(dynamic,20)
12         for(i=0;i<N;i++) {
13             i_sum+=array[i];
14         }
15         printf("Thread %d i_sum=%d\n",tid,i_sum);
16
17 #pragma omp atomic
18         sum+=i_sum;
19     }
20
21     printf("\nFinal sum=%d\n",sum);
22 }
```

In this example, the array entry array[i]=i, and we wish to compute $\sum_{i=0}^{N-1} i$ in parallel. The loop construct dynamically dives the loop iterations into chunks of at most 20, to next available thread. Each thread computes an intermediate sum

i_sum which is added atomically to the shared sum variable sum. The execution of the above code on a quad-core machine produced the following output:

```
Thread 0 i_sum=169400
Thread 3 i_sum=76950
Thread 2 i_sum=81350
Thread 1 i_sum=171800

Final sum=499500
```

We can easily confirm that the output is correct using $\sum_{i=0}^{N-1} i = \frac{(N-1)N}{2}$. For $N = 1000$ this evaluates to 499500, confirming the OpenMP result.

As previously mentioned, OpenMP provides the **reduction** clause (see Table 3.1) to facilitate recurrence calculations. The above example computes such a recurrence, and can be implemented more efficiently (and naturally) using the **reduction** clause, as shown below:

```
1  void example_reduction(void) {
2      int N=1000, sum=0, i;
3      int array[N];
4
5      for(i=0;i<N;i++) { array[i]=i; }
6
7  #pragma omp parallel for schedule(dynamic,20) shared(array) reduction(+:sum)
8      for(i=0;i<N;i++) {
9          sum+=array[i];
10     }
11     printf("Reduced sum=%d\n",sum);
12 }
```

The produced output, as expected, is:

```
Reduced sum=499500
```

In this example we used a more compact OpenMP form: we combined the **parallel** and the **for** constructs into a single directive. We previously omitted the compact form as to distinguish between the thread-creation construct and the work-sharing constructs, but as this example shows the thread creation and work sharing can be combined into a single compact directive.

Construct	Description
barrier	Synchronize execution of the team of threads. Threads wait until all the threads of the team reach the barrier.
atomic	Assert that the expression following the construct updates a shared variable atomically. A thread may update a shared variable with no interference.
critical	Assert that only one thread at a time may execute the block. Similar to **atomic**, although protects a whole section, i.e., not just a memory update.
ordered	Assert that the code in a (parallel) block is executed sequentially.

Table 3.3 Commonly used synchronization constructs.

Synchronization

To synchronize shared memory access or organize thread execution, OpenMP provides a number of constructs that can be used alongside the implied work-sharing construct barriers. We have already encountered two such constructs in our examples; specifically the **barrier** construct and the **atomic** construct. In Table 3.3 we describes the effects of these barriers, in addition to the **critical** and **ordered** constructs.

Runtime Library

In addition to the compiler directives used to create threads and distribute the work among them, the OpenMP API also provides several runtime library functions to modify and query the environment. We have already encountered one such function in our previous examples, specifically, the function omp_get_thread_num() function. Table 3.4 presents the prototypes for this and several other functions; we refer the reader to [53] for a complete description of the OpenMP runtime library.

Function	Description
int omp_get_thread_num(**void**);	Get the thread number.
int omp_get_num_threads(**void**);	Get the number of threads/team.
void omp_set_num_threads(**int** nr_threads);	Set the number of threads/team.
int omp_get_num_procs(**void**);	Get the number of processors available.
int omp_in_parallel(**void**);	Check if function is in a parallel region.

Table 3.4 Commonly-used OpenMP runtime functions.

3.2 CUDA and GPUs

Until recently, programming GPUs has been limited to graphics libraries such as OpenGL [104] and Direct3D [29]. For many general, non-graphics, applications, especially ones dependent on integer-arithmetic, implementations using such libraries only marginally outperformed CPU results. In certain cases, using GPUs even led to performance degradation. Moreover, the difficulty in writing non-graphics code using OpenGL-like libraries deterred the GPU's advancement into the general computing market. However, the introduction of the NVIDIA G80 series and ATI HD2000 series GPUs, both of which implemented the unified shader architecture, has drastically increased the use of graphics cards as accelerators for non-graphics numerically-intensive applications. This trend in computing is, however, also due in part to the release of high-level general purpose programming language support; NVIDA's CUDA [92], ATI's Close to Metal (CTM) [2, 96] and the more-recent Open Computing Language (OpenCL) [84] facilitate the development of massively-parallel GPU applications. In this thesis, we focus on

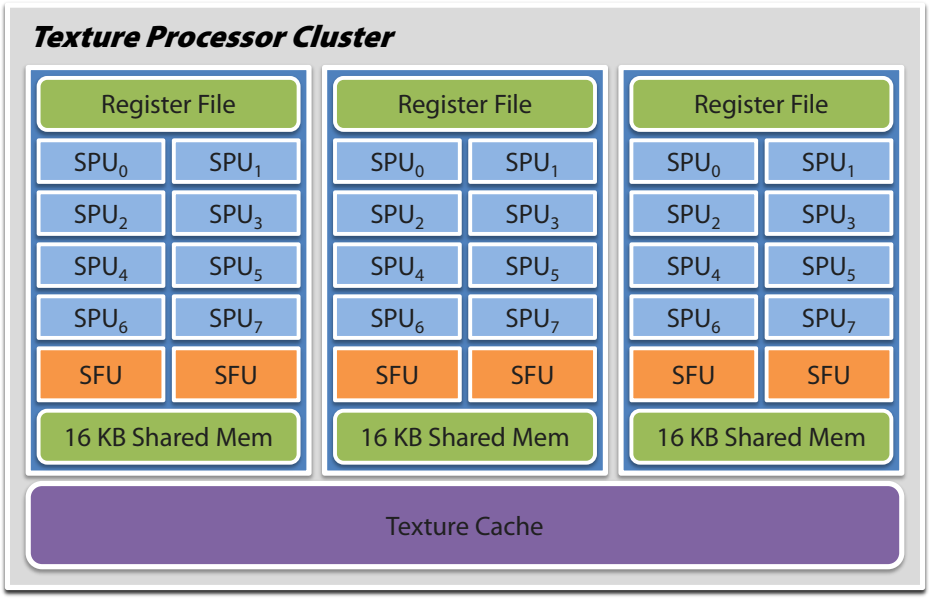

Figure 3.4 GT200 Texture Processor Cluster, containing 3 Streaming Multiprocessors. We note that the constant memory caches are not shown for simplicity.

programming NVIDIA GT200 GPUs with CUDA. Although the older G80 GPUs have also been used for cryptologic applications [58,75,76,93,95,109,119], and our code is generally well-suited for these GPUs as well, with the release of the Fermi architecture (400 series GPUs) interest in and availability of these older devices has been rapidly decreasing.

3.2.1 GPU Architectures

The GT200 GPUs consist of multiple Texture Processor Clusters (TPCs), device memory (logically divided into constant, global, local, and texture memories) and input/output blocks (PCI Express communication with CPU). Each TPC consists of 3 Streaming Multiprocessors (SMs) [91,92], shown in Figure 3.4, each of which is composed of:

- 8 Scalar Processors (SPs),
- 2 Special Function Units (SFUs),
- 1 double-precision floating-point unit (DFPU),
- 16384 32-bit registers,
- 16-way banked 16KB on-chip fast shared memory,
- 8 KB constant cache,
- 1 texture cache interface,
- 1 multithreaded instruction scheduling unit.

The SPs are 32-bit in-order processors capable of computing various (single-precision) floating-point, integer, and bitwise operations. Among these, the SPs can compute 32-bit additions, subtractions, and multiplications (see [91] for the Parallel Thread Execution ISA). Similarly, the SFUs can compute single-precision floating-point multiplications and transcendental functions ($\sin, \cos, \log$), while the DPFU supports double-precision floating-point operations.

The multithreaded scheduler issues a single instruction for a group of 32 threads, known as a *warp*, to be simultaneously executed on the SM. Hence, with each SP and functional unit executing an instruction per cycle, it takes 4 cycles for single-precision floating-point or integer warp instructions to be executed (on the SPs); 16 cycles for single-precision floating-point transcendental instructions (on the SFUs); 32 cycles for double-precision floating-point instructions (on the DPFU).

As with the arithmetic instructions, 32 load/store shared memory instructions can be issued and executed in 4 cycles. However, because the shared memory has

16 banks, a restriction on addressing must be satisfied. Specifically, threads of a *half-warp*, the actual scheduling unit dispatched every 2 cycles, must not (write) access the same bank. Hence, if there are no bank-conflicts in either the first or second half-warp, 32 different loads/stores are completed in 4 cycles. The constant and texture memory throughput is, however, lower than that of shared memory; for the texture memory, the read port is shared with the other SMs in the TPC, while for the constant memory the read is expected to be 'broadcasted' to all the threads of the warp. Further, accessing global memory comes with a large cost in latency (several hundred cycles), which is also present for texture and constant memories upon cache misses.

3.2.2 CUDA Programming

CUDA is an extension of the C language that employs the new massively parallel programming model, the single instruction multiple threads (SIMT) paradigm. Although explicit SIMD access of the SM compute units (the SPs) is desirable when one wishes to highly optimize an application, with CUDA, the programmer is restricted to writing parallel code at the thread level [92]. Specifically, the programmer writes code for a *kernel* which is executed by multiple *threads*. All the threads execute (on the SPs) the same instructions of the kernel, operating on different data–hence the name of the paradigm: single instruction multiple threads [92]. In the SIMT programming model, threads are grouped into a *thread block*, which is executed on a single SM, and consequently, these threads may synchronize execution and use the on-chip shared memory to facilitate communication. Because the multithreaded scheduler issues an instruction for a warp at a time, it is important that the block size be a multiple of 32. When launching a kernel, it is common (and highly recommended) to execute multiple thread

blocks, grouped in a *grid*, which the hardware then assigns to the available SMs; a maximum of 8 blocks may simultaneously execute on a single SM, and in order to hide various latencies, it is recommended that at least 2 be available per SM [92].

Before delving into kernel code, it is important to understand the execution sequence of a typical GPU-accelerated application. This simplified scenario consists of the following steps:

1. A host thread copies input data to the device memory.
2. A grid of threads is launched by the host to be executed on the device.
3. Device executes the kernel.
4. Host thread copies output data back from device.

Though the memory copies and execution may be asynchronous, and the host thread may execute in parallel to the GPU code, the programming model is very similar to the OpenMP fork-join model. Additionally, it is common to execute multiple kernels in sequence (repeat steps 2 and 3), the output of one kernel being used as input to the subsequent kernel (without the need for costly host-device copies). In the multi-kernel scenario it is especially important to use the asynchronous API as to interleave memory copies and kernel execution, riding of unnecessary latencies.

CUDA Kernels

With the exception of certain restrictions, such as no recursion or variable arguments, CUDA kernels are C **void** functions declared with the **__global__** specifier and callable from host code. Below, we focus on introducing CUDA programming through several examples. Additional details are discussed in [92].

Consider the definition of a kernel performing a bitwise XOR of two vectors a and b of N 32-bit words. Additionally, suppose the specified grid executing the kernel consists of N threads, each thread performing a single XOR. As mentioned, each thread executes the same instructions, and therefore a method for diving the work amongst the threads, allowing each thread to read different elements from a and b, is required. Addressing this, CUDA defines the special variables **threadIdx**, **blockIdx**, **blockDim**, and **gridDim** storing the thread index (within a block), thread block index (within a grid), thread block dimension, and grid dimension, respectively. The thread index (and thus the thread block dimension) is a 3-component vector (x, y, z), while the block index (and thus the grid dimension) is a 2-component vector (x, y). It is clear that using the special variables each thread can execute the same instructions, operating on different data. Below is the definition of the kernel performing the vector XOR.

```
1 __global__ void vector_XOR(int *result,int *a, int *b) {
2     int tID = blockIdx.x*blockDim.x+threadIdx.x;
3     result[tID] = a[tID] ^ b[tID];
4 }
```

We reiterate that each thread executes the same same instructions. In the kernel above, the tID is the thread index with respect to all the threads in the grid[5] that the thread uses to read an array element from vectors a and b and write their XOR into the result array.

Let us now consider the position of the kernel within a C application, i.e., the interaction of the host and device code. The code below provides a host wrapper function that initializes a CUDA context on the first device, copies the host array inputs, and launches a grid on the device. Finally, the wrapper copies the results

[5]In this example we assume the grid and blocks are 1-dimensional. For 2- and 3-dimensional the global thread index is computed in a similar fashion.

from the device back to the host.

```
1  void vector_XOR_wrapper(int *result, int *a, int *b, int N) {
2      int device_no = 0; // device number
3
4      int *a_dev, *b_dev, *result_dev; // buffers on device
5      size_t vector_size = N*sizeof(int);
6
7      int n_T=128; // number of threads / block
8      int n_B=N/n_T; // number of blocks / grid
9
10     assert(N>=n_T);
11
12     // Set the device to use
13     cudaSetDevice(device_no);
14
15     // 1a. Allocate buffers on device
16     cudaMalloc((void**)&a_dev,vector_size);
17     cudaMalloc((void**)&b_dev,vector_size);
18     cudaMalloc((void**)&result_dev,vector_size);
19
20     // 1b. Copy input buffers from host to device
21     cudaMemcpy(a_dev,a,vector_size,cudaMemcpyHostToDevice);
22     cudaMemcpy(b_dev,b,vector_size,cudaMemcpyHostToDevice);
23
24     // 2. Launch grid and 3. execute on GPU
25     vector_XOR<<<n_B,n_T>>>(result_dev,a_dev,b_dev);
26
27     // 4. Copy output from device to host
28     cudaMemcpy(result,result_dev,vector_size,cudaMemcpyDeviceToHost);
29
30     // Free buffers on device
31     cudaFree(a_dev);
32     cudaFree(b_dev);
33     cudaFree(result_dev);
34 }
```

In the above, the context is created on line 13; global-memory input and output buffers are allocated on the device on lines 16–18 using a malloc-like function; the host input is copied to the device, using a memcpy-like function, on lines 21–22; kernel is launched on the device on line 25; device result is copied from the device to host on line 28; the initially allocated temporary buffers are deallocated using

a fee-like function, on lines 31–33. We note that with the exception of page-locked memory (discussed in [92]), memory space must be allocated on the device and input/output must be explicitly copied from/to the host. Except for remembering the different memory restrictions, the CUDA API is terse and easily recognizable to the C programmer. A new and unfamiliar syntax to C is, however, introduced by CUDA: the kernel launch, line 25. Executing a kernel is essentially a function call from host code, the instructions of which are executed on the device. Hence, the syntax is similar to calling a C function, as shown below:

```
1  my_kernel<<<gridDim, blockDim, dyn_shmem_size>>>(parameter-list);
```

Following the kernel function name, in the <<<>>>, the programmer specifies the grid dimensions, block dimensions, and the amount of dynamic shared memory per block. The dimensions can be of type **dim3**, a structure with x,y,z integral components, or simply an integer type for 1-dimensional grid/block. The optional dyn_shmme_size is an integral type (specifically, **size_t**) specifying the shared memory size per block, in bytes, to be allocated upon launch.

Shared Memory, Functions, and Atomic Instructions

In the previous trivial example we introduced the most basic of CUDA, device/host interaction and most basic execution of GPU code. We made no use of device functions, atomic instructions, page-locked memory, or the GPU texture, constant and shared memory. For completeness, we consider another simple example showing the use of dynamic shared memory, device functions, and atomic instructions. We refer the interested reader to [92] for additional details on the aforementioned and other more-advanced topics.

Consider computing the sum $\sum_0^{N-1} a_i \cdot x_i \mod 2$, for $N = 2^{7k}$, and any (bounded) integers k, a_i and x_i. First, we define a device function to compute the multiplica-

tion modulo 2:

```
__device__ int mul_mod2(int a, int b) {
    return a & b;
}
```

Unlike kernels, device functions are declared with the **__device__** specifier and are only callable from within device code, i.e., unless the function also has the **__host__** specifier it is only compiled for the GPU architecture. Restrictions, such as no recursion or variable arguments, hold for device functions as they do for kernels. This is especially imposed because device functions are, by default, inlined.

Secondly, consider the kernel computing the actual sum and use of (dynamic) shared memory to cache partial sums.

```
extern __shared__ __align__ (__alignof(void*)) int smem_cache[];

__global__ void weighted_reduce(int *result, int *a, int *x) {
    int tID = blockIdx.x*blockDim.x+threadIdx.x;
    int *reduce_csh = (int*) smem_cache;

    /* write partial result to shared memory */
    reduce_csh[threadIdx.x]=mul_mod2(a[tID],x[tID]);
    __syncthreads();

    /* reduce results of threads in same block */
    for(int i=blockDim.x/2;i>0;i/=2) {
        if(threadIdx.x<i) {
    /* addition mod 2 is XOR */
            reduce_csh[threadIdx.x]^=reduce_csh[threadIdx.x+i];
        }
        __syncthreads();
    }

    /* write result from first thread */
    if(threadIdx.x==0) {
        atomicXor(result,reduce_csh[0]);
    }
}
```

Because the number of threads per block may vary, depending on the kernel

launch parameters, we declare smem_cache on line 1 as **extern**, residing in shared memory and 4-byte aligned It is important to note that, as variables are private to threads, shared memory arrays are private to thread blocks. In other words, one thread block's modification of smem_cache[k] is not visible to another; only threads within the same thread block have shared access to the buffer. On line 8 $a_i \cdot x_i$ mod 2 is computed and cached in shared memory, where the (global) thread index corresponds to i. Note that each thread computes one multiplication and caches the result to the localized shared memory, to be summed. However, before loading the cached multiplications and computing the partial sum on each thread block, it is imperative that all threads within the block have finished computing the multiplication and written the result to shared memory. Hence a barrier or synchronization point is necessary, and this is introduced on the subsequent line. When a thread reaches the **__syncthreads**() barrier it waits until all the other threads within the thread block reach the same point before executing the next instruction. Therefore, a thread waits on line 9 until all the remaining threads have written their respective multiplication result to shared memory, after which the results may be summed, with no error.

The summing can be computed using an arbitrary number of threads. However, a large number of active threads directly corresponds to a more optimal device utilization ratio. Hence for each partial sum on lines 12-18 the number of threads decreases only by a factor of 2, while the number of iterations is logarithmic. Figure 3.5 illustrates this reduction step for thread blocks with 8 threads. Note that each thread writes the partial sum to the cache indexed by its thread index, and thus at the completion of the loop the cache indexed by thread 0 contains the full thread block sum. To combine the results from each thread block global memory must be used – the result output buffer. However, simply reading

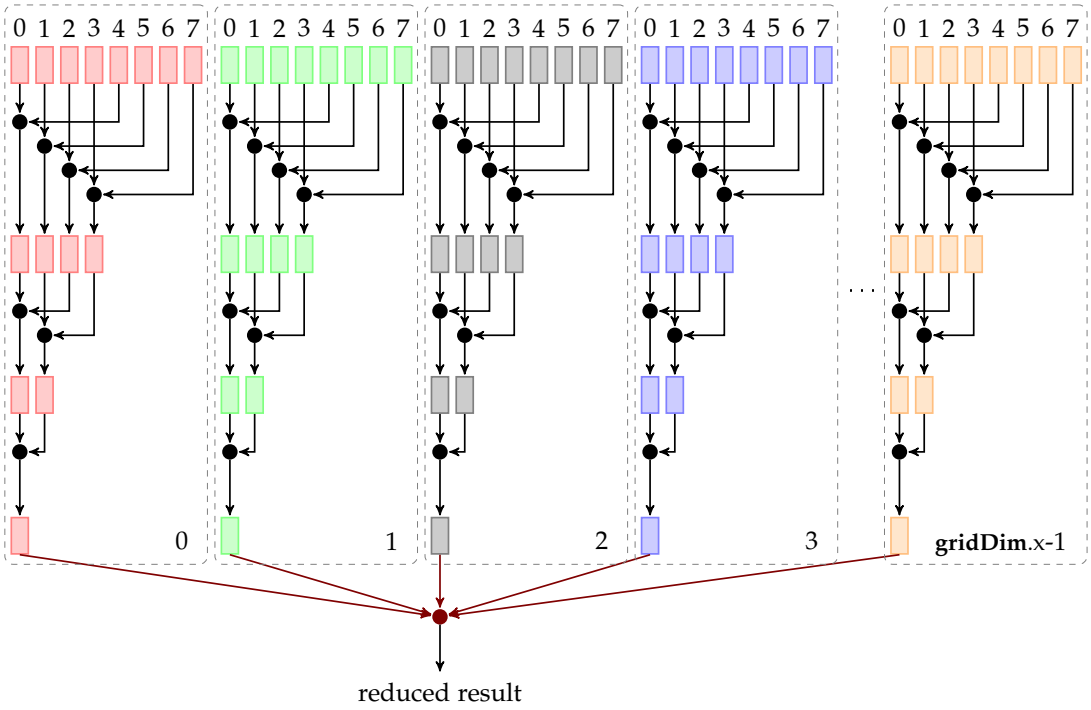

Figure 3.5 Multi-block parallel reduction using shared memory. Each thread block is composed of 8 threads. The rectangles represent shared memory array elements; the circles correspond to the reduction operator (e.g., addition in a field); the red arrow and circle imply that the operation is atomic.

the current result, adding the thread block sum to it and writing it back is not sufficient; the actions must be atomic, as shown on line 22, since different threads (of different blocks) might attempt to modify the value simultaneously and thus violate coherency.

Finally, for completeness, we present the host wrapper function. The function below assumes $N \geqslant 2^7$; for small N's it very likely for CPU code to be an order of magnitudes faster since the memory copies and kernel launch are quite costly.

```
1  int weighted_reduce_wrapper(int *a, int *x, int N) {
2      int device_no = 0; // device number
3
4      int *a_dev, *x_dev; // buffers on device
5      int result, *result_dev;
```

```
6     size_t vector_size = N*sizeof(int);
7
8     int n_T=128; // number of threads / block
9     int n_B=N/n_T; // number of blocks / grid
10
11    assert((N>=n_T) && (N%n_T==0));
12
13    // Set the device to use
14    cudaSetDevice(device_no);
15
16    // 1a. Allocate buffers on device
17    cudaMalloc((void**)&result_dev,sizeof(result));
18    cudaMalloc((void**)&a_dev,vector_size);
19    cudaMalloc((void**)&x_dev,vector_size);
20
21    // 1b. Copy input buffers from host to device
22    cudaMemcpy(a_dev,a,vector_size,cudaMemcpyHostToDevice);
23    cudaMemcpy(x_dev,x,vector_size,cudaMemcpyHostToDevice);
24    cudaMemset(result_dev,0,sizeof(result));
25
26    // 2. Launch grid and 3. execute on GPU
27    weighted_reduce<<<n_B,n_T,n_T*sizeof(int)>>>(result_dev,a_dev,x_dev);
28
29    // 4. Copy output from device to host
30    cudaMemcpy(&result,result_dev,sizeof(result),cudaMemcpyDeviceToHost);
31
32    cudaFree(a_dev);
33    cudaFree(x_dev);
34    cudaFree(result_dev);
35
36    return result;
37 }
```

We further note that the number of threads per block is essential to the performance of the GPU application. A very large block size would result in a large number of inactive threads during the log-number of reductions. Similarly, an odd number or non-multiple of 32 would cause further slow downs due to divergence that would arise during the reduction. Finally, we note that for this specific example having a thread perform multiple multiplications while accumulating their sum would also lead to additional speedups for large N.

Chapter 4

Cube Attack

Most cryptosystems can be described by a polynomial in n private variables and m public variables, corresponding to the secret key and initial value (or plaintext), respectively. Therefore, several cryptanalytic techniques focus on attacking a cryptosystem by targeting weaknesses of the underlying polynomial. Notable techniques include the *chosen initial vector (IV) attacks* [3,48,51,52,103], previously used to cryptanalyze Grain-128, Trivium, Decim, and Lex, among others. Similar to these powerful statistical techniques, though algebraic in its approach, is Dinur and Shamir's *cube attack* [45,46]. The cube attack is a very recent general cryptanalytic technique, generalizing Vielhaber's Algebraic IV Differential Attack (AIDA) [111–113]. Depite AIDA being the first of such algebraic attacks, in this thesis we follow the work and notation of Dinur and Shamir.

The cube attack is suitable for cryptanalyzing a cryptosystems whose underlying polynomials, in $n + m$ variables, have a low degree d. Using the cube attack, an adversary can efficiently carry out a key-recovery attack with complexity $O(n^2 + n \cdot 2^{d-1})$, requiring only black box access to the cryptosystem, i.e., it is not necessary for an adversary to have knowledge of the internal structure of

the target cryptosystem. Moreover, the technique is applicable to a wide range of primitives, including block ciphers, stream ciphers, keyed hash functions, and MACs; any cryptosystem with $m \geqslant d + \log_d n$ public variables and a low degree d is susceptible to a key-recovery attack using this technique [46]. The latter requirement is because the cube attack, like chosen IV attacks, sums over all possible values of subsets of the public variables (i.e., over *cubes*) when attacking the cryptosystem.

In the following sections, we describe the preliminaries necessary to understand the cube attack, followed by a detailed presentation of the two-phase technique. A multi-GPU implementation of the cube attack is presented in Chapter 6, and applications to Trivium and MICKEY are detailed in Chapter 7.

4.1 Preliminaries

Any Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ can be uniquely expressed in the algebraic normal form (ANF) as a polynomial $p(x)$, such that evaluating $p(x)$ is equivalent to computing $f(x)$. Since most cryptosystems are complex Boolean functions in public and private variables, we focus on a cryptosystem's underlying *master polynomial* $p(x)$. For simplicity, however, we (usually) do not distinguish between public and private variables. Instead, let us consider a n-variable polynomial, $p(x_1, \ldots, x_n)$, over $GF(2)$ in ANF. Formally, the ANF of a polynomial has the form:

$$p(x_1, \ldots, x_n) = \sum_{i=0}^{2^n-1} a_i x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}, \tag{4.1}$$

where the monomial coefficients $a_i \in \{0,1\}$, and $i = \sum_{j=1}^{n} i_j 2^{j-1}$, i.e., $(i_1, \ldots, i_n)$ is the n-bit binary representation of i.

The first observation of the cube attack is that, given a set of indexes $I \subseteq$

$\{1, \ldots, n\}$ and a monomial $t_I = \prod_{i \in I} x_i$, any polynomial p can be represented as:

$$p(x_1, \ldots, x_n) = t_I \cdot p_{S(I)} + q. \tag{4.2}$$

Here, $p_{S(I)}$, called the *superpoly* of I in p, and monomial t_I do not share any common terms, i.e., $\nexists x_k : x_k \mid t_I$ and the quotient $p_{S(I)}/x_k \neq 0$. Similarly, q does not have any monomial containing t_I, i.e., the quotient $q/t_I = 0$. In other words, in (4.2) we decompose p by collecting all the terms containing t_I into the superpoly $p_{S(I)}$, leaving the remainining terms in q. Furthermore, if we let $k = |I| = \deg(t_I)$, we observe that degree of the superpoly $\deg(p_{S(I)}) \leqslant n - k$, since p's degree is bounded by n.

Definition 4.1. The monomial t_I is a *maxterm* of p iff the superpoly $p_{S(I)}$ is a non-constant linear (or affine) polynomial such that $\deg(p_{S(I)}) = 1$.

Example 4.1. Consider the following degree-4 polynomial in 6 variables:

$$p(x_1, \ldots, x_6) = 1 + x_1 + x_6 + x_2x_3 + x_2x_3x_4 + x_2x_3x_4x_5 + x_3x_4x_5x_6.$$

(a) Let $I_0 = \{2, 3\}$. We represent p as:

$$p(x_1, \ldots, x_6) = x_2x_3 \cdot (1 + x_4 + x_4x_5) + 1 + x_1 + x_6 + x_3x_4x_5x_6,$$

where $t_{I_0} = x_2x_3$, $p_{S(I_0)} = 1 + x_4 + x_4x_5$ and $q = 1 + x_1 + x_6 + x_3x_4x_5x_6$.

(b) Similarly, let $I_1 = \{3, 4, 5\}$. We represent p as:

$$p(x_1, \ldots, x_6) = x_3x_4x_5 \cdot (x_2 + x_6) + 1 + x_1 + x_6 + x_2x_3 + x_2x_3x_4,$$

where $t_{I_1} = x_3x_4x_5$, $p_{S(I_1)} = x_2 + x_6$ and $q = x_6 + x_2x_3 + x_2x_3x_4$.

(c) Finally, let $I_2 = \{1, 2, 3, 4, 5\}$. We represent p as:

$$\begin{aligned} p(x_1, \ldots, x_6) = x_1x_2x_3x_4x_5 \cdot (0) + 1 + x_1 + x_6 + x_2x_3 \\ + x_2x_3x_4 + x_2x_3x_4x_5 + x_3x_4x_5x_6, \end{aligned}$$

where $t_{I_2} = x_1x_2x_3x_4x_5$, $p_{S(I_1)} = 0$ and $q = p$.

It is important to note that t_{I_1} is the only maxterm in this example since $\deg(p_{S(I_1)}) = 1$. The terms t_{I_0} and t_{I_2} have quadratic and null superpolys, respectively.

Definition 4.2. For $k = |I| < n$, the index-set I defines a k-dimensional Boolean *cube* C_I, over n, by setting the k variables indexed by I to all $0/1$ permutations, leaving the others undetermined. For $I = \{i_1, i_2, \ldots, i_k\} : 0 \leqslant i_t \leqslant n$,

$$C_I = \{(x_1, x_2, \ldots, x_n) : (x_{i_1}, x_{i_2}, \ldots, x_{i_k}) = (j_1, j_2, \ldots, j_k), 0 \leqslant j \leqslant 2^k - 1\},$$

where $(j_1, j_2, \ldots, j_k)$ is the k-bit binary representation of j.

Directly from the defintion, the size of the cube C_I is $|C_I| = 2^k$. Additionally, is clear that for any $v = (v_1, \ldots, v_n) \in C_I$, k of the v_i variables ($i \in I$) are set to $0/1$'s, and the remaining $n - k$ are undetermined. Hence, for each k-bit value, the corresponding (defined part of) vector v can be thought of as a corner of the k-dimensional unit hypercube.

Example 4.2. Continuing with Example 4.1, the cube defined by $I_0 = \{2, 3\}$ is

$$C_{I_0} = \{(x_1, 0, 0, x_4, x_5, x_6), (x_1, 0, 1, x_4, x_5, x_6), (x_1, 1, 0, x_4, x_5, x_6), (x_1, 1, 1, x_4, x_5, x_6)\}.$$

For any vector $v = (v_1, \ldots, v_n) \in C_I$, we define a new polynomial $p_{|v} = p(v)$ in $n - k$ variables such that $\deg(p_{|v}) \leqslant \deg(p)$. Summing over all the vectors in the cube C_I, we obtain a new polynomial

$$p_I = \sum_{v \in C_I} p_{|v}, \tag{4.3}$$

whose degree $\deg(p_I) \leqslant \deg(p) - k$.

Theorem 4.1 (Theorem 1 [46])**.** *For any $GF(2)$ polynomial p in n-variables, and subset of indices $I \subset \{1, \ldots, n\}$ we have $p_I = p_{S(I)}$.*

Proof. Using the representation defined by (4.2),

$$p_{|v} = p(v) = t_I \cdot p_{S(I)} + q = \prod_{i \in I} v_i \cdot p_{S(I)} + q,$$

where $t_I = \prod_{i \in I} v_i \in \{0, 1\}$, as the variables indexed by elements of I are, by Definition 4.2, assigned to 0/1. Taking (4.3) and splitting p_I into two summations,

$$p_I = \sum_{v \in C_I} p_{|v} = \sum_{v \in C_I} t_I \cdot p_{S(I)} + \sum_{v \in C_I} q,$$

we have

$$\sum_{v \in C_I} q = 0,$$

since every term in q is added an even number of times (resulting in a cancellation, since p is over $GF(2)$). Finally,

$$\sum_{v \in C_I} t_I \cdot p_{S(I)} = p_{S(I)}$$

since $t_I = \prod_{i \in I} v_i = 1$ iff all the variables indexed by I are 1 and

$$|\{(x_1, x_2, \ldots, x_n) \in C_I : (x_{i_1}, x_{i_2}, \ldots, x_{i_k}) = (1, 1, \ldots, 1)\}| = 1.$$

□

Example 4.3. Taking the polynomial of Example 4.1, and the cube defined in Example 4.2, we have $\forall v \in C_{I_0}$ the new polynomials $p_{|v}$:

$$p(x_1, 0, 0, x_4, x_5, x_6) = 1 + x_1 + x_6$$

$$p(x_1, 0, 1, x_4, x_5, x_6) = 1 + x_1 + x_6 + x_4 x_5 x_6$$

$$p(x_1, 1, 0, x_4, x_5, x_6) = 1 + x_1 + x_6$$

$$p(x_1, 1, 1, x_4, x_5, x_6) = 1 + x_1 + x_6 + 1 + x_4 + x_4 x_5 + x_4 x_5 x_6$$

which are then used to obtain p_{I_0}:

$$\begin{aligned} p_{I_0} &= \sum_{v \in C_{I_0}} p_{|v} \\ &= p(x_1, 0, 0, x_4, x_5, x_6) + p(x_1, 0, 1, x_4, x_5, x_6) \\ &+ p(x_1, 1, 0, x_4, x_5, x_6) + p(x_1, 1, 1, x_4, x_5, x_6) \\ &= 1 + x_4 + x_4 x_5. \end{aligned}$$

In Example 4.1 we showed that $p_{S(I_0)} = 1 + x_4 + x_4 x_5$, which, as expected by Theorem 4.1, is the same as p_{I_0}, i.e., $p_{I_0} = p_{S(I_0)}$ holds.

Having established these preliminaries, consider a cryptosystem with a n-bit key and m-bit initial value (or plaintext) that is described by a $n + m$ variable polynomial $p(x_1, \ldots x_n, v_1, \ldots, v_m)$. We denote the private and public variables by $x_1, \ldots, x_n$ and $v_1, \ldots, v_m$, respectively. The corresponding ciphertext bit is given by the value of p. The basis for the cube attack is that an attacker with knowledge of ℓ maxterms in a subset of the public variables (setting the remaining public variables to a constant) and ℓ corresponding (linear and independent) superpolys in the private variables can compute p_I for each maxterm and solve the linear equations to recover ℓ-bits of the secret key.

The cube attack is divided into two phases: a *preprocessing phase* and an *online phase*. During the preprocessing phase, the attacker searches for maxterms and their respective superpolys. Once a sufficient number (ℓ for ℓ-bit key recovery) of independent superpolys are found, the attacker can carry out the online phase on any implementation of the algorithm, to (partially) recover the secret key. We describe the two phases below.

4.2 Preprocessing

During the preprocessing phase, the attacker (adversary $\mathcal{A}$) may adaptively modify both public and private variables and subsequently perform multiple queries to the black box cryptosystem using these input vectors. We can assume that the black box is simply an evaluation of the (unknown) master polynomial, given the input vectors. Figure 4.1 illustrates the preprocessing phase, where the adversary (during each query i) provides input vectors $\boldsymbol{v}_i = (v_{i,1}, \ldots, v_{i,m})$ and $\boldsymbol{x}_i = (x_{i,1}, \ldots, x_{i,n})$, and the black box returns the corresponding ciphertext bit $p(\boldsymbol{x}_i, \boldsymbol{v}_i)$. After a sufficient number of queries, the adversary returns a set of maxterm and superpoly pairs: $\{(t_{I_j}, p_{S(I_j)}) : j \leqslant n\}$. If the adversary wishes to recover l-bits ($l \leqslant n$) in the online phase, the preprocessing phase is halted and the set is returned once l independent superpolys are found.

Continuing this attack scenario (see Figure 4.1), let us consider the internal details of $\mathcal{A}$'s preprocessing phase method. As in [46], we divide the preprocessing phase into two parts: finding maxterms and superpoly reconstruction.

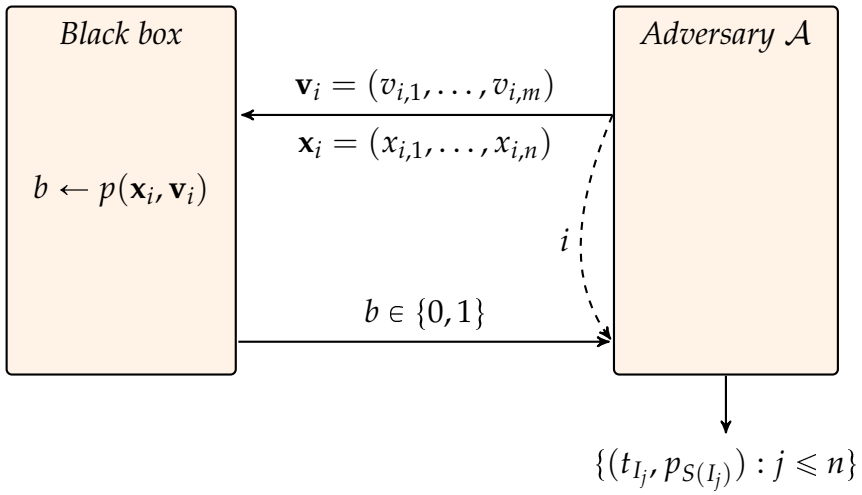

Figure 4.1 Preprocessing phase of the cube attack

4.2.1 Finding Maxterms

Using the notation of (4.1), the ANF of the polynomial p in n private variables and m public variables is

$$p(x_1, \ldots, x_n, v_1, \ldots, v_m) = \sum_{i=0}^{2^{n+m}-1} a_i x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n} v_1^{i_{n+1}} v_2^{i_{n+2}} \cdots v_m^{i_{n+m}}. \tag{4.4}$$

We denote the $b-a+1$ tuple $(i_a, \ldots, i_b)$, consisting of bits a through b of i, by $i_{[a:b]}$. Let $\mathrm{wt}(\alpha)$ dennote the Hamming weight of the w-bit variable α: $\mathrm{wt}(\alpha) = \sum_{j=1}^{w} \alpha_j$. We recall the definition of random, d-random, and $d_{|p}$-random polynomials. Note that our defintions are slightly different from those of [46]; our d-random and d_p-random correspond to their definition of random and d-random, respectively.

Definition 4.3. A polynomial p is *random* if each monomial appears in the ANF of p with probability $\frac{1}{2}$, i.e., for $0 \leqslant i \leqslant n$, $\Pr[a_i = 1] = \Pr[a_i = 0] = \frac{1}{2}$.

Definition 4.4. A polynomial p is *d-random* if each monomial of degree d appears in the ANF of p with probability $\frac{1}{2}$, i.e., for $1 \leqslant i \leqslant n : \mathrm{wt}(i) = d$, $\Pr[a_i = 1] = \Pr[a_i = 0] = \frac{1}{2}$.

Definition 4.5. A polynomial p is $d_{|p}$*-random* if each monomial of degree d, consisting of 1 private variable and $d-1$ public variables, appears in the ANF of p with probability $\frac{1}{2}$. As in the previous definitions, for $1 \leqslant i \leqslant n : \mathrm{wt}(i) = d$ and $\mathrm{wt}(i_{[1:n]}) = 1$, $\Pr[a_i = 1] = \Pr[a_i = 0] = \frac{1}{2}$.

From the definitions, it is clear that a random polynomial is d-random, and a $d_{|p}$-random polynomial is a 'special' d-random polynomial. Moreover, following [48], we let M be the total number monomials in polynomial p, M_k the number of monomials of degree k, and $\mathrm{Bin}(N, q)$ denote the binomial distribution in N trials (each with success probability or q). The random case imposes the restriction

that $M \sim \text{Bin}\left(2^{n+m}, \frac{1}{2}\right)$, the d-random requires that $M_d \sim \text{Bin}\left(\binom{n+m}{d}, \frac{1}{2}\right)$, and the $d_{|p}$-random only requires a subset of the M_d variables to be randomly distributed. For the cube attack, we only require the target cryptosystem to behave under the $d_{|p}$-random, or weakest, assumption. Distinguishers, such as cube testers, however, can be used to attack cryptosystems that are $d_{|p}$-random but not d-random or random, as shown in [3,48,51,103].

Assuming the target master polynomial is $d_{|p}$-random and has maximum degree d, finding a maxterm of degree $d-1$ can be accomplished by testing random terms of degree $d-1$ in the public variables. The number of such terms is $M_{d-1|p} \sim \text{Bin}\left(\binom{m}{d-1}, \frac{1}{2}\right)$, and the probability that the term's superpoly is linear is $1-2^{-n}$. To clarify, consider terms consisting of $d-1$ public variables and another variable u: $v_{i_1}, \ldots, v_{i_{d-1}}u$. As explained in [46], the probability of not selecting a private variable for u is 2^{-n}, since there are n private variables each of which is selected with probability of $\frac{1}{2}$. Consequently, the probability of a $d-1$ dimensional term also being a maxterm is $1-2^{-n}$.

Under these assumptions, we summarize Dinur and Shamir's proposed random-walk approach to finding maxterms [46]:

1. Choose a random size $k \leqslant m$, and random index-set $I : |I| = k$.

2. For several random private variables compute $p_I = \sum_{v \in C_I} p_{|v}$:

   - If the cube sums are constant, I is too large: remove a random variable from I and repeat Step 2.
   - If the cube sums are non-linear, I is too small: add another variable to I and repeat Step 2.
   - If the cube sums are linear, t_I is a maxterm and $d \approx |I| + 1$.

If the algorithm fails to find a maxterm after a large number of tries, it is suggested to simply restarting from Step 1.

In this thesis, we use the Blum Luby Rubinfeld (BLR) test [12, 28, 46] to check for linearity. Given random variables in the domain of f, $k_{i,1}$ and $k_{i,2}$, $1 \leqslant i \leqslant 3N$, the BLR test checks $f(0) + f(k_{i,1}) + f(k_{i,2}) = f(k_{i,1} + k_{i,2})$. If the equality fails for any i, then f is non-linear. Otherwise f is linear with a probability of approximately $1 - 2^{-N}$.

Example 4.4. Let us consider finding maxterms for the following degree-3 polynomial in 6 variables:

$$p(x_1, x_2, x_3, v_1, v_2, v_3) = v_1x_1 + v_3x_3 + v_3x_1x_2 + v_1v_2x_1 + v_1v_2x_2 + v_2v_3x_3$$

We denote the cube sum with a given key $\boldsymbol{x} = (x_1, x_2, x_3)$ over C_I by $p_{C_I}(\boldsymbol{x})$. In this example we compute the cube sum over all the possible keys, as this is a trivial task for $n = 3$, and thus the BLR tests hold with probabilty 1.

(a) Let $k = 3$, and $I = I_0 = \{1, 2, 3\}$. We compute the cube sums:

$\boldsymbol{x}$	$(0,0,0)$	$(1,0,0)$	$(0,1,0)$	$(1,1,0)$	$(0,0,1)$	$(1,0,1)$	$(0,1,1)$	$(1,1,1)$
$p_{C_{I_0}}(\boldsymbol{x})$	0	0	0	0	0	0	0	0

Since the output is constant regardless of the key, I is too large and, thus, we must remove a variable from it.

(b) Alternatively, suppose we start with $k = 1$, and $I = I_1 = \{3\}$. First, we set $v_1 = v_2 = 0$ and then compute the different-key cube sums over C_{I_1}:

$\boldsymbol{x}$	$(0,0,0)$	$(1,0,0)$	$(0,1,0)$	$(1,1,0)$	$(0,0,1)$	$(1,0,1)$	$(0,1,1)$	$(1,1,1)$
$p_{C_{I_1}}(\boldsymbol{x})$	0	0	0	1	1	1	1	0

In this case, the output is not constant, but simply taking the first three keys (and doing the BLR test) we can verify the non-linearity of the superpoly:

$$p_{C_{I_0}}(0,0,0) + p_{C_{I_0}}(1,0,0) + p_{C_{I_0}}(0,1,0) \neq p_{C_{I_0}}(1,1,0)$$
$$0+0+0 \neq 1.$$

Thus, we need to add more terms to I.

(c) Suppose we add v_2, so that $I = I_2 = \{2,3\}$. Setting $v_1 = 0$, we compute the cube sums:

$\boldsymbol{x}$	$(0,0,0)$	$(1,0,0)$	$(0,1,0)$	$(1,1,0)$	$(0,0,1)$	$(1,0,1)$	$(0,1,1)$	$(1,1,1)$
$p_{C_{I_2}}(\boldsymbol{x})$	0	0	0	0	1	1	1	1

We can easily verify that the BLR test passes and confirm that t_{I_2} is a maxterm. Consequently, we also have an estimate of p's degree $d \approx k+1 = 3$.

Continuing as above, we find that, in addition to I_2, the index-sets $I_3 = \{1,2\}$ and $I_4 = \{1\}$ correspond to maxterms that have linear superpolys. Note that it is not required for different maxterm index-sets to be disjont, e.g., $I_2 \cap I_3 = \{2\}$ indicates that both maxterm index-sets I_2 and I_3 contain the common variable v_2.

Denoting $GF(2)$ addition by $\oplus$, a random sampling by $\xleftarrow{R}$, and a sum over cube C_I with private variables set to k by $p_{C_I}(k)$, we present a concrete algorithm for finding maxterms in Algorithm 4.1. Given T maximum number of tries per term and a bounded dimension m_d, the algorithm returns a maxterm I or `reject` if no maxterm is found. The number of BLR tests per maxterm is $3N$ and thus the probability that a returned maxterm is linear is $1 - 2^{-N}$; once a maxterm is found additional BLR tests can be performed to further increase the confidence level in the term. The costliest parts of the algorithm are the four cube sums computed

for the BLR tests, lines 7, 13, 14, 15. We estimate the worst-case complexity of this algorithm to be $O(T \cdot ((9N+1) \cdot 2^{m_d}))$, and if the degree of p is known, the complexity finding (and confirming) l maxterms is roughly $O(\ell \cdot ((9N+1) \cdot 2^{d-1}))$.

4.2.2 Superpoly Reconstruction

Given a maxterm t_I in master polynomial $p(x_1, \ldots, x_n)$, we wish to reconstruct the ANF of the linear superpoly $p_{S(I)}$. Note that in this case we again do not explicitly distinguish between public and private variables. Instead, let $J \subset \{1, \ldots, n\}$ be the set of indexes of the (private variable) terms in $p_{S(I)}$, so that $I \cap J = \varnothing$, and

$$p_{S(I)} = a_0 + \sum_{i=1}^{n} a_i x_i, \tag{4.5}$$

where $a_0 \in \{0, 1\}$ is the superpoly's constant term, and $a_i = 1$ iff $i \in J$. In other words, J is the index-set of the private linear variables in the superpoly. Hence, reconstructing $p_{S(I)}$ can then be reformulated to the problem of finding $\{a_0 \cup J\}$.

Theorem 4.2 (Theorem 2 [46])**.** *The constant term a_0 can be computed by summing over the cube C_I, setting all remaining variables to zero.*

Proof. Using (4.3) and setting the undetermined variables of $v \in C_I$ to zero (i.e., $\forall i \notin I : x_i = 0$), the theorem states that $a_0 = p_I = \sum_{v \in C_I} p_{|v}$. Following (4.5),

$$p_I = p_{S(I)} = a_0 + \sum_{i=1}^{n} a_i x_i = a_0 + \sum_{i \in J} x_i = a_0,$$

as $a_i = 1$ iff $i \in J$, $x_i = 0$ for all $i \notin I$, and $I \cap J = \varnothing$. □

Theorem 4.3 (Theorem 2 [46])**.** *The coefficient a_j in superpoly $p_{S(I)}$ is computed by the summing over cube C_I, setting all but j-th variable to zero, and adding the result to the constant term a_0.*

Algorithm 4.1: Finding a maxterm

Input : Master polynomial $p(x_1, \ldots, x_n, v_1, \ldots, v_m)$.
Maximum dimensions m_d.
Maximum tries T per term.
Output: Maxterm index-set I or `reject`.

begin
  $k \xleftarrow{R} \{1, \ldots, \min(m, m_d)\}$;// Random (bounded) dimension
  // Start with random k-dimensional term:
  for $i \leftarrow 1$ **to** k **do** $I_i \xleftarrow{R} \{1, \ldots, m\} \backslash I$; // Create random index-set
  for $i \leftarrow 1$ **to** T **do**
    $n_0 \leftarrow 0$;
    $n_1 \leftarrow 0$;
    $p_0 \leftarrow p_{C_I}(0)$;
    if $p_0 = 1$ **then** $n_1 \leftarrow n_1 + 1$;
    else $n_0 \leftarrow n_0 + 1$;
    for $i \leftarrow 1$ **to** $3N$ **do**
      $k_1 \xleftarrow{R} \{0,1\}^n$;// Random key 1
      $k_2 \xleftarrow{R} \{0,1\}^n$;// Random key 2
      $p_1 \leftarrow p_{C_I}(k_1)$;
      $p_2 \leftarrow p_{C_I}(k_2)$;
      $p_{1,2} \leftarrow p_{C_I}(k_1 \oplus k_2)$;
      // Update counters:
      if $p_1 = 1$ **then** $n_1 \leftarrow n_1 + 1$;
      else $n_0 \leftarrow n_0 + 1$;
      if $p_2 = 1$ **then** $n_1 \leftarrow n_1 + 1$;
      else $n_0 \leftarrow n_0 + 1$;
      // Linearity test:
      if $p_1 \oplus p_2 \neq p_{1,2}$ **and** $p_0 \oplus p_1 \oplus p_2 \neq p_{1,2}$ **then**
        // Non-linear, add term:
        $k \leftarrow k + 1$;
        $I_k \xleftarrow{R} \{1, \ldots, m\} \backslash I$;
        break;
    if $n_0 = 3N$ **or** $n_1 = 3N$ **then**
      // Constant, remove term:
      $j \xleftarrow{R} I$;
      $I \leftarrow I \backslash j$;
      $k \leftarrow k - 1$;
    else
      // Linear term:
      return I;
  return `reject`;

Proof. Similar to the previous theorem, we set all but the j-th undetermined variables to zero; in this case, $x_j = 1$. The theorem states $a_j = a_0 + p_I = a_0 + \sum_{v \in C_I} p_{|v}$. As before,

$$p_I = p_{S(I)} = a_0 + \sum_{i=1}^{n} a_i x_i = a_0 + a_j,$$

since $x_i = 0$ for all $i \notin \{I \cup j\}$. Hence, $p_I + a_0 = a_j$. □

Given a maxterm t_I and using the results of Theorem 4.2 and Theorem 4.3, we have a direct algorithm for reconstructing the maxterm's superpoly by only querying the black box. We present the superpoly reconstruction, explicitly distinguishing the private variables from public variables, in Algorithm 4.2. Note that the costliest parts of the algorithm are, as in Algorithm 4.1, the cube sums, which are explicity computed on lines 6 and 13; hence, given n private variables, the complexity of Algorithm 4.2 is $(n+1)2^{|I|}$. Using this result, we can further estimate the reconstruction time of l superpolys given the master polynomial p of degree d to require $O(l(n+1)2^{d-1})$ black box queries (i.e., polynomial evaluations).

Example 4.5. Let us consider the reconstruction of the superpoly for maxterm $t_{I_3} : I_3 = \{1, 2\}$ of Example 4.4. First, we set the undertermined $v_3 = 0$ and sum over the cube (the cube variables are underlined) to find the constant term:

$$a_0 = p(0,0,0,\underline{0},\underline{0},0) + p(0,0,0,\underline{1},\underline{0},0) + p(0,0,0,\underline{0},\underline{1},0) + p(0,0,0,\underline{1},\underline{1},0) = 0.$$

Then we calculate the remaining terms (setting only $x_j = 1$ when computing a_j):

$$a_1 = a_0 + p(1,0,0,\underline{0},\underline{0},0) + p(1,0,0,\underline{1},\underline{0},0) + p(1,0,0,\underline{0},\underline{1},0) + p(1,0,0,\underline{1},\underline{1},0) = 1,$$

$$a_2 = a_0 + p(0,1,0,\underline{0},\underline{0},0) + p(0,1,0,\underline{1},\underline{0},0) + p(0,1,0,\underline{0},\underline{1},0) + p(0,1,0,\underline{1},\underline{1},0) = 1,$$

$$a_3 = a_0 + p(0,0,1,\underline{0},\underline{0},0) + p(0,0,1,\underline{1},\underline{0},0) + p(0,0,1,\underline{0},\underline{1},0) + p(0,0,1,\underline{1},\underline{1},0) = 0.$$

Algorithm 4.2: Superpoly reconstruction

Input : Master polynomial $p(x_1, \ldots, x_n, v_1, \ldots, v_m)$.
Maxterm index-set I.
Output: Reconstructed superpoly $p_{S(I)}$.

```
1  begin
2      J ← ∅ // Index of coefficients in superpoly
       // Compute the constant term:
3      for i ← 1 to n do x_i ← 0;// Set key to zero
4      foreach i ∉ I do v_i ← 0;// Set all undetermined variables to zero
5      p_I ← 0;
6      foreach v ∈ C_I do p_I ← p_I ⊕ p_|v;
7      a_0 ← p_I;
       // Compute the remaining terms:
8      for j ← 1 to n do
9          for i ← 1 to n do
10             if i=j then x_i ← 1;
11             else x_i ← 0;
12         p_I ← 0;
13         foreach v ∈ C_I do p_I ← p_I ⊕ p_|v;
14         a_j ← a_0 ⊕ p_I;
15         if a_j = 1 then
16             J ← {J ∪ j};
17     return p_S(I) = a_0 + Σ_{j∈J} x_j; // Return symbolic superpoly
```

Given the coefficients, the reconstucted superpoly for $t_{I_3} = v_1 v_2$ is $p_{S(I_3)} = x_1 + x_2$. It is important to observe that, without any knowledge of p's form, we recovered t_{I_3}'s superpoly. This simple result can be confirmed algebraically using the definition of p from Example 4.4.

4.3 Online Attack

In the online phase, the goal of the attacker is to fully or partially recover the secret key. Thus, during this phase, the attacker is only allowed to adaptively modify the public variables when querying the black box cryptosystem. Figure 4.2 highlights

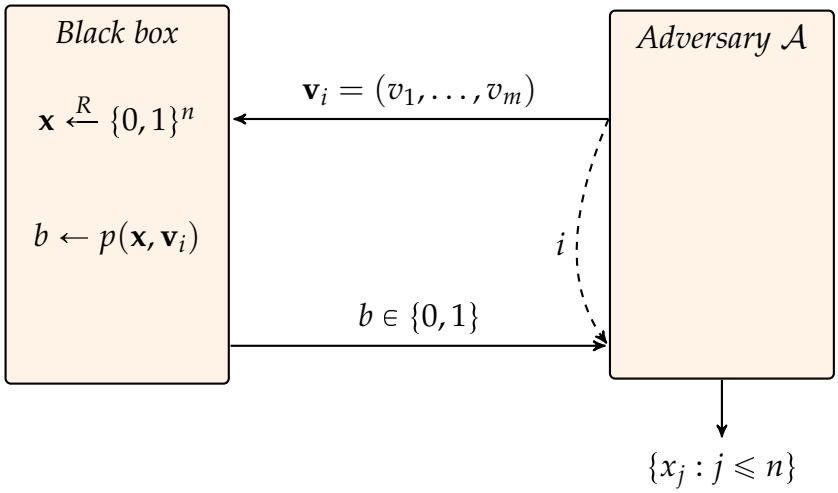

Figure 4.2 Online phase of the cube attack

the attack scenario, in which adversary $\mathcal{A}$ performs multiple queries to the black box, the results of which are used to recover bits $\{x_j : j \leqslant n\}$ of the secret/random key $\boldsymbol{x}$.

Assuming that $\ell \leqslant n$ linearly independent superpolys were found during the preprocessing phase, the online phase consists of:

1. Performing black box queries to compute the sum $p_{I_i} \in \{0,1\}$ over cube C_{I_i}, for each maxterm $t_{I_i} : 1 \leqslant i \leqslant \ell$.

2. Taking the p_{I_i}'s and the reconstructed superpolys from the preprocessing phase to solve a system of linear equations in the private variables.

More explicitly, after the preprocessing phase, the attacker precomputes the

$l \times n$ binary matrix of superpolys:

$$\boldsymbol{A} = \begin{pmatrix} p_{S(I_1)} \\ p_{S(I_2)} \\ \vdots \\ p_{S(I_\ell)} \end{pmatrix}.$$

Then, during the online phase, $\mathcal{A}$ computes the vector of cube sums

$$\boldsymbol{p}^\top = (p_{I_1}, p_{I_2}, \cdots, p_{I_\ell}),$$

and solves $A\boldsymbol{x}^\top = \boldsymbol{p}$ for $\boldsymbol{x}$, the secret key. For a full key-recover, i.e., $\ell = n$, the adversary may precompute the inverse of square matrix A, and simply compute $\boldsymbol{x}^\top = A^{-1}\boldsymbol{p}$. This reduces the complexity of the equation-solving from $O(n^3)$ to $O(n^2)$, as explained in [46],

We emphasize that the preprocessing phase is done only once for a cryptosystem, while the online phase is performed individually for each key. Hence, a higher complexity for the preprocessing stage is tolerated in the hopes of lowering the complexity of the online attack. Given a master polynomial of maximum degree d the complexity of the online phase is $O(n^2 + n \cdot 2^{d-1})$. Moreover, as we showed in the previous section, the overall complexity of the preprocessing in only slightly larger, linear in n. Thus, for cryptosystems with low degree master polynomials, the cube attack becomes a very attractive and efficient key-recovery attack.

Chapter 5

Linear Differential Cryptanalysis

Differential cryptanalysis is a general technique that has been widely applied to the analysis of many cryptosystems, including block ciphers, stream ciphers, and hash functions. Informally, differential cryptanalysis is concerned with the propagation of input (XOR) differences to a function, and their effects the output. More specifically, given a function $f(x)$, one is concerned with finding *differentials* (or *differential trails*) $\Delta_i \xrightarrow{(f,p)} \Delta_o$ such that $f(x) \oplus f(x \oplus \Delta_i) = \Delta_o$ holds with probability p for a random input x. For example, in the case of block ciphers or stream ciphers, the attacker commonly wishes to construct differentials with initial difference Δ_i in plaintext (or initial value) in order to observe a known Δ_o that is a function of the (sub)key. Correspondingly, in the case of hash functions, an attacker may seek to find colliding message pairs $(x_1, x_2) : x_1 \neq x_2$ and $f(x_1) = f(x_2)$. Using differential cryptanalysis, this problem is reformulated to that of finding high-probability trails with $\Delta_i \neq 0$ and $\Delta_o = 0$, along with *conforming* input messages x (see Section 5.2).

Although differential cryptanalysis originally appeared in the study of block ciphers, specifically the Data Encryption Standard (DES) block cipher [26], we

only focus on the technique's application to hash functions, in this thesis. More specifically, we focus on finding collisions in hash functions. Chabaud and Joux [38] presented the first *linear* differential collision attack on SHA-0. Using a linearized model of the hash function, they found trails (with input differences in the message) that lead to a collision of the original hash function with a higher probability than the birthday bound[1] Similar strategies were later used by Rijmen and Oswald [101] on SHA-1 and by Indesteege and Preneel [64] on EnRUPT. In [99], Pramstaller *et al.* related the problem of finding high-probability linear differentials to the problem of finding low-weight codewords of a linear code. The recent work of Brier *et al.* [34] further analyzed this relation for hash functions whose operations consist only of additions, XORs, and rotates; hence, these functions are referred to as AXR hash functions.

In brief, this framework reformulates the problem of finding message pairs that conform to a linear differential trail to that of finding preimages of zero of a *condition function*. The search for preimages is accelerated by using a *dependency table*, which implicitly takes advantage of message modification techniques [23, 36, 69, 114–116]. Moreover, given a linear differential trail, these concepts further allows for complexity estimation of the corresponding collision attack. Below, we introduce Brier *et al.*'s generalized framework [34, 35] and our extensions [67].

[1] For a hash function $f : \{0,1\}^* \rightarrow \{0,1\}^n$, the birthday attack entails the evaluation of f with random inputs x_1, x_2 until a collision is found, i.e., $f(x_1) = f(x_2)$. According to the birthday problem it will take approximately $2^{n/2}$ (the birthday bound) function evaluations for a collision to be found; a technique requiring fewer function evaluations than the birthday bound is an attack on f.

5.1 Constructing Differential Trails

As explained in [34, 67], we can attribute a fixed-input-length compression function $\mathsf{Compress} : \{0,1\}^m \times \{0,1\}^v \rightarrow \{0,1\}^h$, where $m \geqslant h$, to any hash function, regardless of its design, This compression function maps an m-bit message and v-bit IV to an h-bit output. This 'construction' is created to allow one to find collisions for the compression function that directly translate to collisions for the hash function. We note that this notion of a compression function does not necessarily coincide with the frequently used compression function in the context of Merkle-Damgård, HAIFA, and other iterated constructions. For example, we can simply restrict the domain of the hash function and consider this as the hash function's compression function.

In this thesis, we focus on a differential cryptanalysis approach to finiding collisions. Specifically, we are interested in finding differences Δ such that there is a high probability of a random message pair $(M, M \oplus \Delta)$ and random (but public) initial vector (IV) V leading to a collision for the compresion function:

$$\mathsf{Compress}(M, V) \oplus \mathsf{Compress}(M \oplus \Delta, V) = 0. \tag{5.1}$$

Since the IV is common to both compression functions, and thus has no effect in the differential setting, we simplify the notation of the compression function to $\mathsf{Compress}(M)$. Furthermore, we assume that $\mathsf{Compress}$ is an AXR function and hence search for differentials by linearizing the compression function. We denote the linearized compression function by $\mathsf{Compress}_{\mathsf{lin}}$ and search for differences Δ such that

$$\mathsf{Compress}_{\mathsf{lin}}(M, V) \oplus \mathsf{Compress}_{\mathsf{lin}}(M \oplus \Delta, V) = 0. \tag{5.2}$$

Since the difference is independent of the message M and IV V, we further simplify the notation to $\mathsf{Compress}_{\mathsf{lin}}(\Delta)$. Once linear differentials are found, we pro-

ceed to search for random messages M such that the compression function differences conform to the linear ones:

$$\mathsf{Compress}(M) \oplus \mathsf{Compress}(M \oplus \Delta) = \mathsf{Compress}_{\mathsf{lin}}(\Delta) = 0. \tag{5.3}$$

Below, we establish the notation used in the remaining sections and then introduce three methods for finding differential trails.

5.1.1 Notation

For any input M to the compression function, we denote $\boldsymbol{A}(M)$ and $\boldsymbol{B}(M)$ as the concatenation of all left and, respectively, right addends that are added in the course of computing $\mathsf{Compress}(M)$. Likewise, we define $\boldsymbol{C}(M)$ as the concatenation of the corresponding carry words. Thus, if n_a is the number of w-bit additions affected in the course of one evaluation of the compression function, each of $\boldsymbol{A}(M), \boldsymbol{B}(M)$ and $\boldsymbol{C}(M)$ contains $n_a w$ bits. Analogously, for any input Δ to the linearized compression function, we denote $\boldsymbol{\alpha}(\Delta)$ and $\boldsymbol{\beta}(\Delta)$ the concatenation of the left and, respectively, right addends that are XORed in the course of computing $\mathsf{Compress}_{\mathsf{lin}}(\Delta)$, setting their most significant bits to zero[2]. We denote the individual addends $i = 0, \ldots, n_a - 1$ with a superscript, e.g., $\boldsymbol{A}^i(M)$ denotes the i-th left addend in the compression function, and $\boldsymbol{C}^i(M) = (\boldsymbol{A}^i(M) + \boldsymbol{B}^i(M)) \oplus \boldsymbol{A}^i(M) \oplus \boldsymbol{B}^i(M)$.

5.1.2 Raw Probability

From (5.2), it is clear that searching for a differential is equivalent to the problem of finding preimages of zero of $\mathsf{Compress}_{\mathsf{lin}}$. However, since our goal is to find

[2]The most significant bits of each addition are linear.

collisions in the non-linear compression function, we are interested in finding differentials such that, for a random message M and random IV V, (5.3) holds with a high probabilty. However, a random preimage might not be the best solution, i.e., it might not be the differential with the highest probability, and so we must compute the *raw probability*, p_Δ, for each candidate Δ:

$$p_\Delta = \Pr[\mathsf{Compress}(M) \oplus \mathsf{Compress}(M \oplus \Delta) = \mathsf{Compress}_{\mathsf{lin}}(\Delta)]. \tag{5.4}$$

Lemma 5.1 (Lemma 1 [34])**.** *For any random w-bit addends A and B, the probability that addition is linear (in $GF(2)$) in the difference of $A + B$ and its perturbed version $(A \oplus \Delta_A) + (B \oplus \Delta_B)$ is given by:*

$$\Pr[((A \oplus \Delta_A) + (B \oplus \Delta_B)) \oplus (A + B) = \Delta_A \oplus \Delta_B] = 2^{-\mathrm{wt}((\Delta_A \vee \Delta_B) \wedge (2^{w-1}-1))},$$

where Δ_a and Δ_b are constants, $\mathrm{wt}(\cdot)$ denotes the Hamming weight, and $\vee$ is the bitwise-OR operation.

Note that the second equality, involving the term $\Delta_A \oplus \Delta_B$, in Lemma 5.1 comes directly from the properties of the XOR:

$$((A \oplus \Delta_A) \oplus (B \oplus \Delta_B)) \oplus (A \oplus B) = \Delta_A \oplus \Delta_B.$$

Furthermore, we point out that the most signifiant bits, which are linear, are cleared with the $2^{w-1} - 1$ mask, and thus have no effect on the probability.

To illustrate the significance of Lemma 5.1, consider a trivial compression function Compress with a single (non-linear) w-bit addition (among other linear operations). Suppose we also have a differece Δ in the kernel of $\mathsf{Compress}_{\mathsf{lin}}$, i.e., $\mathsf{Compress}_{\mathsf{lin}}(\Delta) = 0$. Using the lemma, we can compute the probability that the difference $\mathsf{Compress}(M) \oplus \mathsf{Compress}(M \oplus \Delta)$ behaves like $\mathsf{Compress}_{\mathsf{lin}}(\Delta)$, for a random message and initial value, also random. Then, among the various candidate

Δ's, we can find the trail that will most likely lead to a collision, i.e., the trail with the highest raw probability.

Lemma 5.2 (Lemma 2 [34])**.** *For any random m-bit message and v-bit initial value, the probability that the difference* $\mathsf{Compress}(M) \oplus \mathsf{Compress}(M \oplus \Delta)$ *behaves like the difference of its linear equivalent,* $\mathsf{Compress}_{lin}(\Delta)$*, is given by:*

$$p_\Delta = \Pr[\mathsf{Compress}(M) \oplus \mathsf{Compress}(M \oplus \Delta) = \mathsf{Compress}_{lin}(\Delta)] = 2^{-\mathrm{wt}(\boldsymbol{\alpha}(\Delta) \vee \boldsymbol{\beta}(\Delta))}.$$

Since we are only interested in differentials for which Δ is in the kernel of $\mathsf{Compress}_{\mathsf{lin}}$, we can use Lemma 5.2 to directly compute the (raw) lower bound probability of finding a collision for any such Δ.

5.1.3 Forward Differential Trails

As described previously, we linearize the compression function of a hash function to find message differences that can be used for a collision attack. The most direct approach to finding good differentials is to then search for differences with high raw probability. We call these trails *forward differentials*.

Using the canonical bases, $\mathsf{Compress}_{\mathsf{lin}}$ can be written as a $m \times h$ matrix $\mathcal{H}$, such that $\mathsf{Compress}_{\mathsf{lin}}(\Delta) = \mathcal{H}\Delta$. Similarly, we let $\mathcal{A}\Delta = \boldsymbol{\alpha}(\Delta)$, and $\mathcal{B}\Delta = \boldsymbol{\beta}(\Delta)$. Thus, as explained in [34], the problem of finding differential trails is similar to that of finding low-weight codewords, i.e., given *parity check matrix* $\mathcal{H}$ of a linear code we are searching for codewords $\Delta : \mathcal{H}\Delta = 0$. In the search for low-weight codewords we simply need to find a Δ that is of low Hamming weight. Conversely, in the differential trail search problem, we search for codewords Δ that have low Hamming weight $\mathcal{A}\Delta \vee \mathcal{B}\Delta$. Although the latter problem is more difficult (as it generalizes the first), low-weight codewords commonly have high

raw probabilities, i.e., low-weight $\mathcal{A}\Delta \vee \mathcal{B}\Delta$, and hence we approach the search with low-weight Δ's as candidates.

For many hash function parameters (e.g., in CubeHash varying r and b), the matrix $\mathcal{H}$ does not have full rank, as noted in [34], thus one can find differences with high raw probability in the set of linear combinations of at most λ kernel basis vectors. Letting τ be the dimension of $\mathcal{H}$'s kernel, the number of kernel basis $\lambda \geqslant 1$ is chosen such that a subset of combinations can be exhaustively searched over all $\sum_{i=1}^{\lambda} \binom{\tau}{i}$ possible choices. In this thesis, we extend [34] by parallelizing this search, which in turn allows us to consider a larger subset of elements—here, we set $\lambda = 4$.

5.1.4 Reverse Differential Trails

In Section 5.2, we introduce a method of finding messages that satisfy (5.3) for a given Δ. The basic observation is that a differential Δ imposes a number of conditions on the addends which, for a collision attack, must be satisfied by modifying the input message M. Depending on Δ, these conditions can appear at various stages of the compression function. As previously observed in [23, 38, 85, 99, 116], conditions in early steps of the computation can be more easily satisfied than those in later steps. This is due to message modifications, (probabilistic) neutral bits, submarine modifications and other *freedom degrees use* (see e.g., [23]). Hence, this motivates the search for differences Δ such that $\boldsymbol{\alpha}(\Delta) \vee \boldsymbol{\beta}(\Delta)$ is sparse at the end of the trail. As observed in [67], this is generally not the case for forward differential trails found using the method of Section 5.1.3; most forward differential trails are sparse at the beginning and dense at the end, due to the diffusion of the linearized compression function.

Assuming the transformations of the hash function are invertible, we can de-

fine a reverse linear compression funtion, $\mathsf{Compress}^{\mathsf{r}}_{\mathsf{lin}}$. The function $\mathsf{Compress}^{\mathsf{r}}_{\mathsf{lin}}$ is defined in the same way as $\mathsf{Compress}_{\mathsf{lin}}$, except with inverse linearized round transformations, i.e., the steps of $\mathsf{Compress}_{\mathsf{lin}}$ are reversed. Further, suppose that the m-bit input message to the compression function is actually composed of t m'-bit message blocks to the hash function. Consequently, a differential Δ is composed of t (sub)differentials $\Delta = \Delta_0 \| \cdots \| \Delta_{t-1}$. Following this, we define a *reverse differential* as a difference $\Delta' = \Delta'_0 \| \cdots \| \Delta'_{t-1}$ that lies in the kernel of $\mathsf{Compress}^{\mathsf{r}}_{\mathsf{lin}}$. Like forward differentials, a reverse differential is sparse at the beginning of the trail and dense at the end. However, the 'reverse' of this differential $\Delta'' = \Delta'_{t-1} \| \cdots \| \Delta'_0$ has the property of being dense in the beginning and sparse at the end. More importantly, because $\mathsf{Compress}^{\mathsf{r}}_{\mathsf{lin}}$ is the reverse of $\mathsf{Compress}_{\mathsf{lin}}$, Δ'' lies in the kernel of $\mathsf{Compress}_{\mathsf{lin}}$. In Section 7.2.2, we apply the new reverse differential search to finding collisions for CubeHash-5/96.

Finally, we note that the search for reverse differentials is mostly identical to the exhaustive search described in previous section for forward differentials. The only difference is the linear-code for which the parity check matrix is defined; in this case, $\mathcal{H}$ is defined so that $\mathcal{H}\Delta = \mathsf{Compress}^{\mathsf{r}}_{\mathsf{lin}}(\Delta) = 0$.

5.1.5 Randomized Differential Trails

Note that the kernel of matrix $\mathcal{H}$ contains 2^τ different elements. The above methods find the best differences out of a subset of $\sum_{i=1}^{\lambda} \binom{\tau}{i}$ elements. We may find better results by increasing λ or by repeating the search for another choice of basis vectors. Using ideas from [99], we propose an alternative, randomized, search algorithm that does not depend on the choice of the kernel basis, as done in [67].

Let $\Delta^0, \ldots, \Delta^{\tau-1}$ be a kernel basis of $\mathsf{Compress}_{\mathsf{lin}}$, and denote $\mathcal{G}$ as the matrix whose τ rows consist of the binary vectors $\Delta^i \| \boldsymbol{\alpha}(\Delta^i) \| \boldsymbol{\beta}(\Delta^i)$ for $0 \leqslant i \leqslant \tau - 1$.

Elementary row operations on $\mathcal{G}$ preserve this structure; that is, the rows always have the form $\Delta \| \boldsymbol{\alpha}(\Delta) \| \boldsymbol{\beta}(\Delta)$, where Δ lies in the kernel of $\mathsf{Compress}_{\mathsf{lin}}$ and has raw probability $\mathrm{wt}(\boldsymbol{\alpha}(\Delta) \vee \boldsymbol{\beta}(\Delta))$. We call this the *row raw probability*. The task of finding a more optimal differential is equivalent to finding a linear combination of the rows that leads to the highest row raw probability. Starting with the index of the row with the highest row raw probability, denoted $i_{\max}$, we iterate through the following steps:

1. Randomly choose a column index j and let i be the smallest row index such that $\mathcal{G}_{i,j} = 1$. If no such i exists, choose a different column index.

2. For all row indices $k = i+1, \ldots, \tau - 1$ such that $\mathcal{G}_{k,j} = 1$:

   - add row i to row k,
   - set $i_{\max} = k$ if row k has a higher raw probability than row $i_{\max}$.

3. Move row i to the bottom of $\mathcal{G}$, shifting rows $i+1, \ldots, \tau - 1$ up by one.

Using this algorithm, we are able to find differentials that are at least as good as the previous two methods with considerably less computational effort, as shown [67]. In Section 7.2.2, we restate these results for CubeHash, comparing them to the best exhaustive search method for $\lambda = 4$.

5.2 Finding Collisions Using Condition Functions

Sections 5.1.3- 5.1.5 introduced three approaches to finding linear differential trails for a compression function Compress. In this section, we introduce a method for finding messages M which, given a differential Δ, leads to a collision attack (and/or second preimage attack) on the compression function.

Definition 5.1 (Definition 1 [35])**.** Following the notation established in Section 5.1.1, and the results of Lemma 5.1 and Lemma 5.2, a message M is said to *conform* to the trail of Δ if and only if:

$$((A^i(M) \oplus \boldsymbol{\alpha}^i(\Delta)) + (B^i(M) \oplus \boldsymbol{\beta}^i(\Delta))) \oplus (A^i(M) + B^i(M)) = \boldsymbol{\alpha}^i(\Delta) \oplus \boldsymbol{\beta}^i(\Delta).$$

Thus, the raw probability p_Δ of Lemma 5.2 corresponds to the probability that a random message M conforms to the trail of Δ, where, a message is conforming if and only if (5.3) holds. Therefore, the problem of finding a colliding message pair $(M, M \oplus \Delta)$ for the compression function can be reformulated to the problem of finding messages that conform to the trail of Δ. For every differential, we define a condition function which, in turn, is used to find such conforming messages. However, we first restate Lemma 3 of [34], from which the definition of the condition function follows directly.

Lemma 5.3 (Lemma 3 [34])**.** *Let $\delta = 2^i$, and A, B, and C be three w-bit words, where $C = (A + B) \oplus A \oplus B$ and subscript i denotes the i-th bit of a word. Then, for any $i : 0 \leqslant i < w - 1$,*

$$((A \oplus \delta) + (B \oplus \delta)) \oplus (A + B) = \delta \oplus \delta \Leftrightarrow A_i \oplus B_i \oplus 1 = 0, \tag{5.5}$$

$$(A + (B \oplus \delta)) \oplus (A + B) = 0 \oplus \delta \Leftrightarrow A_i \oplus C_i = 0, \tag{5.6}$$

$$((A \oplus \delta) + B) \oplus (A + B) = \delta \oplus 0 \Leftrightarrow B_i \oplus C_i = 0. \tag{5.7}$$

Proof. We first prove (5.5) and then (5.6) from which (5.7) directly follows.

(a) Let $S = A + B$, and similarly $S' = A' + B'$, where $A' = A \oplus \delta$ and $B' = B \oplus \delta$. Since δ is introduced at the i-th bit, the sum bits (S_i, S'_i) and generated carries (C_{i+1}, C'_{i+1}) must be equal for the equality to hold. First, we compute the

sum bits:

$$\begin{aligned} S_i &= A_i \oplus B_i \oplus C_i, \\ S_i' &= A_i' \oplus B_i' \oplus C_i' = (A_i \oplus 1) \oplus (B_i \oplus 1) \oplus C_i = A_i \oplus B_i \oplus C_i, \\ \therefore \quad & S_i = S_i'. \end{aligned}$$

Similarly, for the carry bits:

$$\begin{aligned} C_{i+1} &= (A_i \wedge B_i) \vee (C_i \wedge (A_i \oplus B_i)), \\ C_{i+1}' &= (A_i' \wedge B_i') \vee (C_i' \wedge (A_i' \oplus B_i')) \\ &= ((A_i \oplus 1) \wedge (B_i \oplus 1)) \vee (C_i \wedge ((A_i \oplus 1) \oplus (B_i \oplus 1))) \\ &= ((A_i \oplus 1) \wedge (B_i \oplus 1)) \vee (C_i \wedge (A_i \oplus B_i))). \end{aligned}$$

Thus, in the forward direction:

$$\begin{aligned} C_{i+1} = C_{i+1}' \quad & \Rightarrow (A_i \wedge B_i) = ((A_i \oplus 1) \wedge (B_i \oplus 1)) = (\neg A_i \wedge \neg B_i) \\ & \Rightarrow A_i \neq B_i \\ & \Rightarrow A_i \oplus B_i \oplus 1 = 0. \end{aligned}$$

In the reverse direction:

$$\begin{aligned} (A_i \oplus B_i) = 1 \quad & \Rightarrow (\neg A_i \wedge B_i) \vee (A_i \wedge \neg B_i) = 1 \\ & \Rightarrow ((\neg A_i \wedge B_i) \vee A_i) \wedge ((\neg A_i \wedge B_i) \vee \neg B_i) = 1 \\ & \Rightarrow ((\neg A_i \vee A_i) \wedge (A_i \vee B_i)) = ((\neg A_i \vee \neg B_i) \wedge (B_i \vee \neg B_i)) \\ & \Rightarrow (A_i \vee B_i) = (\neg B_i \vee \neg A_i) \\ & \Rightarrow \neg(\neg A_i \wedge \neg B_i) = \neg(B_i \wedge A_i) \\ & \Rightarrow (\neg A_i \wedge \neg B_i) = (B_i \wedge A_i) \\ & \Rightarrow C_{i+1} = C_{i+1}'. \end{aligned}$$

This completes the proof for (5.5).

(b) As above, let $S = A + B$, $B' = B \oplus \delta$, and $S' = A + B'$. For the sum bits:

$$\begin{aligned} S_i &= A_i \oplus B_i \oplus C_i, \\ S'_i &= A_i \oplus B'_i \oplus C'_i = A_i \oplus (B_i \oplus 1) \oplus C_i = A_i \oplus B_i \oplus C_i \oplus 1, \\ \therefore \quad & S_i \oplus S'_i = 1. \end{aligned}$$

Assuming $C_{i+1} = C'_{i+1}$, the latter $S_i \oplus S'_i = 1 \Rightarrow S \oplus S' = \delta$. Hence, we are again left to prove the relationship between equality of the carry bits and the condition (in this case, $A_i \oplus C_i = 0$). As before:

$$\begin{aligned} C_{i+1} &= (A_i \wedge B_i) \vee (C_i \wedge (A_i \oplus B_i)), \\ C'_{i+1} &= (A_i \wedge B'_i) \vee (C'_i \wedge (A'_i \oplus B'_i)) \\ &= (A_i \wedge (B_i \oplus 1)) \vee (C_i \wedge (A_i \oplus (B_i \oplus 1))). \end{aligned}$$

Using some algebraic manipulation and given that $p \oplus q = (p \wedge \neg q) \vee (\neg p \wedge q)$, we can rewrite the carry bits as:

$$\begin{aligned} C_{i+1} &= ((A_i \wedge B_i) \vee C_i) \wedge (A_i \vee B_i), \\ C'_{i+1} &= ((A_i \wedge B'_i) \vee C_i) \wedge (A_i \vee B'_i) \\ &= ((A_i \wedge \neg B_i) \vee C_i) \wedge (A_i \vee \neg B_i). \end{aligned}$$

Then, for the forward case, $C_{i+1} = C'_{i+1} \Rightarrow A_i = C_i$, we consider two cases. First, letting $B_i = 0$:

$$\begin{aligned} C_{i+1} &= ((A_i \wedge 0) \vee C_i) \wedge (A_i \vee 0) \\ &= C_i \wedge A_i, \\ C'_{i+1} &= ((A_i \wedge 1) \vee C_i) \wedge (A_i \vee 1) \\ &= A_i \vee C_i, \\ \therefore \quad & C_{i+1} = C'_{i+1} \Rightarrow C_i \wedge A_i = C_i \vee A_i \Rightarrow A_i = C_i. \end{aligned}$$

Correspondingly, letting $B_i = 1$:

$$\begin{aligned} C_{i+1} &= ((A_i \wedge 1) \vee C_i) \wedge (A_i \vee 1) \\ &= A_i \vee C_i, \\ C'_{i+1} &= ((A_i \wedge 0) \vee C_i) \wedge (A_i \vee 0) \\ &= C_i \wedge A_i, \\ \therefore \quad & C_{i+1} = C'_{i+1} \Rightarrow A_i \vee C_i = C_i \wedge A_i \Rightarrow A_i = C_i. \end{aligned}$$

For the reverse case, we let $A_i = C_i$, and compute the carries:

$$\begin{aligned} C_{i+1} &= ((A_i \wedge B_i) \vee A_i) \wedge (A_i \vee B_i) \\ &= (A_i \wedge (A_i \vee B_i)) \wedge (A_i \vee B_i) \\ &= A_i \wedge (A_i \vee B_i) \\ &= A_i, \\ C'_{i+1} &= ((A_i \wedge \neg B_i) \vee A_i) \wedge (A_i \vee \neg B_i) \\ &= (A_i \wedge (A_i \vee \neg B_i)) \wedge (A_i \vee \neg B_i) \\ &= A_i \wedge (A_i \vee \neg B_i) \\ &= A_i, \\ \therefore \quad & A_i = C_i \Rightarrow C_{i+1} = C'_{i+1}. \end{aligned}$$

This completes the proof for (5.6).

(c) Since addition is commutative, the proof for (5.7) is the same as that of (5.6), except A and B are swapped.

□

In Lemma 5.1, we estimated the probability that modular addition in a differential context behaves like an XOR, i.e., given two constant differences Δ_a and Δ_b to (random) addends A and B, respectively, we estimated the probability of addition being linear in $((A \oplus \Delta_A) + (B \oplus \Delta_B)) \oplus (A + B)$. Lemma 5.3, in turn,

imposes conditions on the inputs A and B (right hand sides of (5.5), (5.6), and (5.7)) necessary to make the additions linear. We now extend this lemma to the whole compression function.

Definition 5.2. Given a Δ in the kernel of the linearized compression function, $\mathsf{Compress}_{\mathsf{lin}}$, we define a *condition function* $Y = \mathsf{Condition}_{\Delta}(M, V)$ whose domain is the same as that of the compression function, but has an output Y of length $y = -\log_2 p_{\Delta} = \mathrm{wt}(\boldsymbol{\alpha}(\Delta) \vee \boldsymbol{\beta}(\Delta))$. Let $i_0, \ldots, i_{y-1}$ be the bit positions of the y non-zero bits in $\boldsymbol{\alpha}(\Delta) \vee \boldsymbol{\beta}(\Delta)$. For $j = 0, \ldots, y-1$, the condition function is defined as:

$$Y_j = \begin{cases} \boldsymbol{A}_{i_j}(\Delta) \oplus \boldsymbol{B}_{i_j}(\Delta) \oplus 1 & \text{if } (\boldsymbol{\alpha}_{i_j}(\Delta), \boldsymbol{\beta}_{i_j}(\Delta)) = (1,1), \\ \boldsymbol{A}_{i_j}(\Delta) \oplus \boldsymbol{C}_{i_j}(\Delta) & \text{if } (\boldsymbol{\alpha}_{i_j}(\Delta), \boldsymbol{\beta}_{i_j}(\Delta)) = (0,1), \\ \boldsymbol{B}_{i_j}(\Delta) \oplus \boldsymbol{C}_{i_j}(\Delta) & \text{if } (\boldsymbol{\alpha}_{i_j}(\Delta), \boldsymbol{\beta}_{i_j}(\Delta)) = (1,0). \end{cases}$$

By Proposition 1 in [34], and intuitively from Lemma 5.3, the problem of finding a message M that conforms to the trail of Δ is equivalent to the problem of finding a preimage M of zero of the condition function, i.e., $M : \mathsf{Condition}_{\Delta}(M) = 0$.

In the previous section, we established the preliminaries necessary to find differentials. Subsequently, in this section, we have introduced a method of finding messages that conform to the trails of these differentials. Using these concepts, we now have a method to carry out and estimate second preimage attacks. Specifically, given a compression function with an h-bit output and a differential path Δ, if the number of bit conditions $y < h$, i.e., $p_{\Delta} > 2^{-h}$, we can consider this as a (theoretical) second preimage attack. Moreover, if $1/p_{\Delta}$ condition function evalauations is pratically attainable, we may find the preimages by randomly trying approximately $1/p_{\Delta}$ random messages as inputs to the condition function before finding a preimage of zero.

5.3 Freedom Degrees Use: Dependency Table

Previously, neutral bits [23], message modification techniques [116], and probabilistic neutral bits [4], among other, have been introduced as *freedom degrees use* to accelerate hash function collision searches. However, in this thesis, we use a more-recent freedom degrees use technique called the *dependency table* [34]. The dependency table technique, which implicitly uses probabilistic neutral bits, takes advantage of a function's limited diffusion properties. The observation of freedom degrees use techniques is that a (target) function does not always mix its input bits perfectly[3], and statistical biases can be observed in the output bits. The non-random relationship between the inputs and outputs can then be used to modify the inputs, with a certain degree of freedom, to attain a desired output. In the case of the condition function, we use these techniques to modify the input message until the desired output, zero, is attained. To summarize, we use the dependency table to construct a partition of the input bits and, correspondingly, the output bits such that modifying certain bits in the input only affects a subset of the output bits. Using the partitions, we then create an algorithm to efficiently find preimages (of zero) of the condition function.

Recall that we are working with a condition function $\mathsf{Condition}_\Delta(M, V)$ that maps an m-bit message and a v-bit initial value to an y-bit output. If $\mathsf{Condition}_\Delta$ mixed its input well then then it would take roughly 2^m function evaluations to find a preimage of zero. However, as in [34], we suppose an ideal situation where $\mathsf{Condition}_\Delta$ does not mix its input bit well and we are given (input) partitions $\bigcup_{i=1}^{\ell} \mathcal{M}_i = \{0, \ldots, m-1\}$ and (output) partitions $\bigcup_{i=0}^{\ell} \mathcal{Y}_i = \{0, \ldots, y-1\}$ such that for $j = 0, \ldots, \ell$ the output bits with position indices in $\mathcal{Y}_j$ only depend on

[3]Informally, the output of a function that mixes its inputs well cannot be statistically distinguished from a random sequence, regardless of the chosen input.

the input bits with position indices in $\bigcup_{i=1}^{j} \mathcal{M}_i$. We construct the segments (or blocks) $\mathcal{M}_i$ and $\mathcal{Y}_i$ using the dependecy table.

To create the dependency table, we, first create a *probabilistic-effect table* P of size $y \times m$, where each entry $P_{j,k}$ is a measure of the effect (or influence) an input message bit k has on output bit j. Then, given a threshold $\gamma : 0 \leqslant \gamma < \frac{1}{2}$ and table P, we create the binary dependecy table T such that each entry $T_{j,k} = 1$ if $P_{j,k} > \gamma$ and $T_{j,k} = 0$ otherwise. Informally, entry $T_{j,k}$ is set to 1 only if the input bit k highly affects output bit j, i.e., changing the k-th input bit likely changes the j-th output bit. For a given number of test iterations N (e.g., $N = 100,000$), Algorithm 5.1 presents an algorithm for creating the probabilistic-effect and dependacy tables.

Given the dependency table, we construct the input and output partition using Algorithm 2 of [34]. For completeness, we present a slightly modified version of this algorithm in Algorithm 5.2, where we denote row j of the dependency table T by $T_{j,*}$ and column k by $T_{*,k}$. In lines 8–11, we construct a bulk $\mathcal{M}_i$ consisting of input message bits that have the most effect on the (remaining) output bits. Similarly, in lines 4–6 we construct a bulk $\mathcal{Y}_i$ consisting of output bits that are not affected by the remaining message bits, i.e., these were only affected by $\bigcup_{j=1}^{i} \mathcal{M}_j$.

In Algorithm 5.3, we present a tree-based backtracking algorithm that, given the input and output partitions of $\mathsf{Condition}_\Delta$, sequentially modifies the input blocks $\mathcal{M}_i$ until all output blocks are null. The basis for the algorithm, as discussed in [34], is to first find an initial value V that makes (at least) $\mathcal{Y}_0$ null. Then, at each later step (or level) i, we find a random message bulk for M_i making (at least) $\mathcal{Y}_i$ null, leaving $\mathcal{Y}_j$ for $j < i$ unaffected, i.e., to null. As explained in [34], the partitioning is not always ideal and in practice $\mathcal{M}_1, \ldots, \mathcal{M}_i$ might not fully influence $\mathcal{Y}_i$. Thus, if we are level i and after $2^{\min(|\mathcal{M}_i|,|\mathcal{Y}_i|)}$ we have not found a random bulk $\mathcal{M}_i$ that makes $\mathcal{Y}_i$ null, we backtrack to step $i - r_i$, find an alternative $\mathcal{M}_{i-r_i}$

Algorithm 5.1: Calculating the probabilistic-effect and dependancy tables

Input : Condition function $\mathsf{Condition}_\Delta$.
Threshold γ.
Output: Probabilistic-effect table P and dependancy table T.

begin
  `// Clear table:`
  for $k \leftarrow 0$ **to** $m-1$ **do**
    for $j \leftarrow 0$ **to** $y-1$ **do** $P_{j,k} \leftarrow 0$;
  `// Compute the probabilistic-effect table:`
  for $k \leftarrow 0$ **to** $m-1$ **do**
    for $n \leftarrow 0$ **to** $N-1$ **do**
      $M \xleftarrow{R} \{0,1\}^m$;
      $Y^0 \leftarrow \mathsf{Condition}_\Delta(M, V)$;
      $M_k \leftarrow M_k \oplus 1$;
      $Y^1 \leftarrow \mathsf{Condition}_\Delta(M, V)$;
      for $j \leftarrow 0$ **to** $y-1$ **do**
        if $Y^0_j \neq Y^1_j$ **then** $P_{j,k} \leftarrow P_{j,k} + \frac{1}{N}$;
  `// Compute the dependancy table:`
  for $k \leftarrow 0$ **to** $m-1$ **do**
    for $j \leftarrow 0$ **to** $y-1$ **do**
      if $P_{j,k} > \gamma$ **then** $T_{j,k} \leftarrow 1$;
      else $T_{j,k} \leftarrow 0$;
  return (P, T);

that makes $\mathcal{Y}_{i-r_i}$ null, and then proceed forward. Unlike the algorithm in [34], where the backtracking steps are manually set by the cryptanalyst, our modified algorithm backtracks r_i steps at level i, having estimated $\mathcal{M}_{i-r_i}$ to have the most effect on $\mathcal{Y}_i$. We use Algorithm 5.4 to compute the backtrack steps, where a max limit of $r_{\max}$ steps is imposed.

To estimate the complexity of Algorithm 5.3 we use induction on the number of condition function evaluations at level i. Suppose at level i we examine 2^{q_i} random message bulks for $\mathcal{M}_i$ to make $\mathcal{Y}_i$ null. Each of the 2^{q_i} random message bulks make $\mathcal{Y}_i$ null with probability $2^{-|\mathcal{Y}_i|}$. Thus, we expect approximately $2^{q'_{i+1}} =$

Algorithm 5.2: Creating input and output partitions of a condition function

Input : Dependency table T.
Output: Paritions $\bigcup_{i=1}^{\ell} \mathcal{M}_i = \{0, \ldots, m-1\}$ and $\bigcup_{i=0}^{\ell} \mathcal{Y}_i = \{0, \ldots, y-1\}$.

begin
  $i \leftarrow 0$;
  while $T \neq \varnothing$ **do**
    foreach $j : \sum_t T_{j,t} = 0$ **do**
      `// All-zero row, no message bit strongly affects output` j
      $\mathcal{Y}_\ell \leftarrow \{\mathcal{Y}_\ell \cup j\}$;
      `RemoveRow`$(T_{j,*})$;
    $i \leftarrow i + 1$;
    while $T \neq \varnothing$ **and** $\nexists j : \sum_t T_{j,t} = 0$ **do**
      $k \leftarrow \underset{k}{\operatorname{argmax}} \sum_t T_{k,t}$; `// Column with highest number of 1's`
      `// Message bit` k `affects the most number of output bits`
      $\mathcal{M}_i \leftarrow \{\mathcal{M}_i \cup k\}$;
      `RemoveColumn`$(T_{*,k})$;
  $\mathcal{Y}_i \leftarrow \{0, \ldots, y-1\} \setminus \bigcup_{k=0}^{i-1} \mathcal{Y}_k$;
  $\ell \leftarrow i$;
  return $(\{\mathcal{M}_1, \ldots, \mathcal{M}_\ell\}, \{\mathcal{Y}_0, \ldots, \mathcal{M}_\ell\})$;

$2^{q_i - |\mathcal{Y}_i|}$ surviving candidates, i.e., $2^{q'_{i+1}}$ message bulks make $\mathcal{Y}_i$ null. At step $i+1$, we take the $2^{q'_{i+1}}$ candidates, and for each, we only need test at most $2^{|\mathcal{M}_{i+1}|}$ message blocks to make $\mathcal{Y}_{i+1}$ null, since we have $|\mathcal{M}_{i+1}|$ bits of freedom. Hence, at level $i+1$ we examine at most a total of $2^{q'_{i+1}+|\mathcal{M}_{i+1}|} = 2^{q_{i+1}}$ messages, of which $2^{q'_{i+2}}$ will make $\mathcal{Y}_{i+1}$ null. Since $q'_{i+1} = q_i + |\mathcal{Y}_i|$ and, from the induction, $q_{i+1} - q'_{i+1} \leqslant |\mathcal{M}_{i+1}|$. Furthermore, for the algorithm to succeed, the number of surviving candidates $q'_{i+1} \geqslant 0$ for $i = 0, \ldots, \ell$. Using these results, we compute the algorithm parameters, $q_i = |\mathcal{Y}_i| + \max(0, q_{i+1} - |\mathcal{M}_{i+1}|)$, for $i = \ell - 1, \ldots, 0$ and $q_\ell = |\mathcal{Y}_\ell|$. By definition of q_i, we evaluate 2^{q_i} condition functions at each step i. Thus, the *theoretical complexity*, c_Δ of finding a preimage of zero for $\mathsf{Condition}_\Delta$

Algorithm 5.3: Tree-based backtracking preimage search

Input : Paritions $\bigcup_{i=1}^{\ell} \mathcal{M}_i = \{0, \ldots, m-1\}$ and $\bigcup_{i=0}^{\ell} \mathcal{Y}_i = \{0, \ldots, y-1\}$.
Output: Preimage (M, V) of $\mathsf{Condition}_\Delta(M, V) = 0$.

```
1  begin
2      M ←R {0,1}^m;
3      repeat
4          V ←R {0,1}^v;
5          Y ← Condition_Δ(M, V);
6      until ∃i ∈ Y_0 : Y_i ≠ 0;
7      for d ← 1 to ℓ do
8          c ← 0; // Counter for number of function evaluations
9          repeat
10             foreach k ∈ M_d do M_k ←R {0,1};
11             Y ← Condition_Δ(M, V);
12             c ← c + 1;
13         until ∃i ∈ ⋃_{j=0}^{d} Y_j : Y_i ≠ 0 or c ⩾ 2^min(|M_d|,|Y_d|);
14         if ∃i ∈ ⋃_{j=0}^{d} Y_j : Y_i ≠ 0 then
               // Backtrack:
15             d ← d − r_d;
16             if d = 0 then goto 1;
17     return (M, V);
```

using Algorithm 5.3 is givn by

$$c_\Delta = \sum_{i=0}^{\ell} 2^{q_i}. \tag{5.8}$$

We point out that the parameters and algorithm complexity were previously derived in [34], and our explanation deviates only slightly. Additionally, theoretical analysis of the complexity of the adaptive backtracking is not an easy task, and is considered to be future work. However, as in [34], we do account for the case of non-ideal partitions, where segments $\mathcal{M}_{i+1}, \ldots, \mathcal{M}_\ell$ have influence on $\mathcal{Y}_j$ for $j \leqslant i$. For this, we let 2^{-p_i} denote the probabilty that $\mathcal{M}_i$ has no influence on $\mathcal{Y}_j$ for $0 \leqslant j < i$, and thus will not make an already-null bulk non-null. Hence, the num-

Algorithm 5.4: Computing adaptive backtrack steps

Input : Probabilistic-effect table P.
Maximum backtrack step $r_{\max}$.
Output: Backtrack steps $r_d : d = 0, \dots \ell$.

```
1  begin
2      r_0 ← 0;
3      r_1 ← 1;
4      for d ← 2 to ℓ do
5          for i ← max(0, d − r_max) to i do
               // Compute normalized influence of M_i on Y_d:
6              s_i ← 0;
7              foreach k ∈ M_i do
8                  foreach j ∈ Y_d do  s_i ← s_i + P_{j,k};
9              s_i ← s_i / (|M_i| + |Y_d|);
10             r_d ← d − argmax_i s_i;
11     return {r_d : 0 ⩽ d ⩽ ℓ};
```

ber of surviving candidates at step i is $2^{q'_{i+1}} = 2^{q_i} \cdot 2^{-|\mathcal{Y}_i|} \cdot 2^{-p_i}$ and so the non-ideal parameters are simply $q_i = p_i + |\mathcal{Y}_i| + \max(0, q_{i+1} - |\mathcal{M}_{i+1}|)$, for $i = \ell - 1, \dots, 0$ and $q_\ell = |\mathcal{Y}_\ell|$. Using the new parameters, the theoretical complexity c_Δ can then be recomputed using (5.8).

Chapter 6

Cryptography and Cryptanalysis on GPUs

In this chapter we present our approach to implementing the eSTREAM ciphers, SHA-3 hash functions, and the cube-attack on NVIDIA graphics processing units. In porting algorithms to the GPU we have developed a number of *patterns* that are common to the stream ciphers, hash functions and cryptanalytic algorithm. Below we present the challenge in porting code to the GPU and summarize the main optimizations. Though we only present the application of these patterns to stream ciphers and hash functions, we believe that they are applicable to other symmetric cryptographic algorithms. An example of these patterns applied in the block cipher setting is presented in [95].

The GPU parallel thread execution (PTX[1]) ISA [91] has a very limited instruction set, when compared to other high-performance gaming processors such as the Cell B.E. With respect to integer arithmetic operations, programmers have

[1]We note that the PTX is an intermediate description and not the actual GPU ISA. The latter is not publicly available.

access to 32-bit bitwise operations (`and`, `or`, `xor`, etc.), left/right shifts, 32-bit additions (with carry-in and carry-out), and 32-bit multiplication (sometimes implemented using several 24-bit multiplication instructions).

Given the simplicity of PTX, to gain the most speedup from the raw computational power, it is imperative that the kernels be very compact (especially with respect to register utilization and shared memory allocation). Compact and non-divergent kernels allow for the execution of more simultaneous threads, and can thus increase the performance of the target algorithm. Thus, when implementing common stream cipher and hash function building blocks, a simple approach is also usually the most optimal. For example, a rotation of a 32-bit word is implemented using two shifts (`shl` and `shr`), and an `or` instruction. Furthermore, for many of the primitives we can store the full internal state, and sometimes even the input message block, in registers. Although this limits the number of simultaneous threads per SM, it also lowers the copies to and from (shared) memory and thereby contributes to a faster implementation, overall. Additionally, when possible, we manually unroll loops, such as the compression functions of hash functions, since branching on the SMs can lead to a degradation in performance when threads of a common thread block take divergent paths and execution is serialized. Moreover, conditional statements consisting of a small number of operations in each basic block are implemented using predicate instructions, instead of branches. This is because PTX allows for the predication of almost all instructions. Nevertheless, when branching is necessary (e.g., the compression function of Skein-512), the thread execution is synchronized (at a barrier near the branch) and the branch instruction is executed uniformly by all the threads.

When considering algorithms using 64-bit operations, the number of registers and instructions usually doubles. For example, a 64-bit addition is performed

using two additions with carry (`add.cc`). Similarly, rotations by $x \not\equiv 0 \bmod 32$ are implemented using 4 `shift` and 2 `or` 32-bit instructions. For these algorithms, rather than using expensive registers to cache chain values or message blocks, we resort to using shared memory for caching. We, again, stress that the restriction on shared memory bank access applies to all our algorithms, and thus a 64-bit cache value requires 2 (non-conflicting) memory accesses per 64-bit word.

6.1 GPU Implementation of eSTREAM Ciphers

In this section we present uniform GPU API for stream cipher implementations following the ECRYPT eSTREAM API [47]. We then present estimates for the eSTREAM profile using instruction counts for both a generic modern architecture and NVIDIA GPUs followed by actual implementation results of all but one of the stream ciphers.

6.1.1 gSTREAM Framework

Compared to the SHA-3 candidates, see Section 6.2, many of the eSTREAM stream ciphers have simple designs. Hence, when porting the original 32-bit CPU code, which is usually already well-suited for GPUs, to the GPU it is more appropriate to devote effort to the design and optimization of a generic stream cipher GPU interface than minute cipher details.

We designed the GPU stream cipher framework (gSTREAM) to be easily adoptable by programmers using the ECRYPT eSTREAM API; for most eSTREAM functions, we provide a counterpart gSTREAM function that handles the GPU details transparently. Figure 6.1 shows a typical program flow using the gSTREAM framework. To the programmer using the API, with the exception of specifying

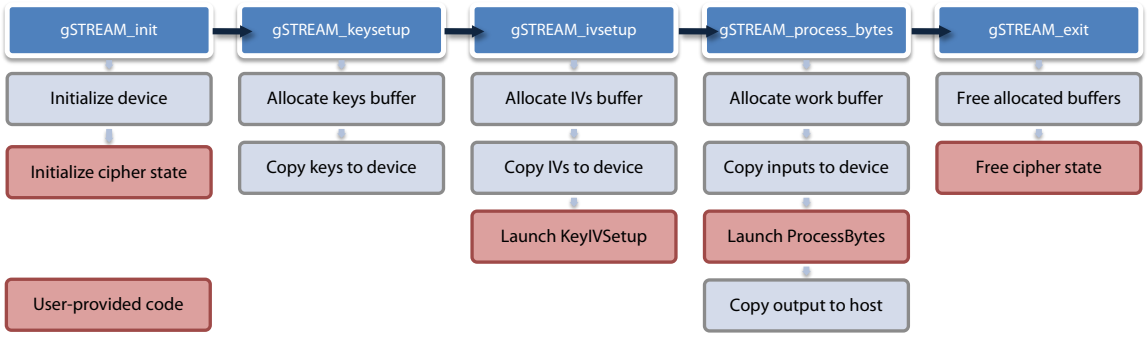

Figure 6.1 Program flow using the gSTREAM framework.

the device and number of streams (number of threads $\times$ number of blocks) the GPU is transparent. Moreover, even these details can be further abstracted by providing a dual CPU implementation which is used in cases where the programmer does not have access to a GPU. Similarly, the GPU stream cipher implementor is only required to provide the implementation details of the stream cipher: defining the key/iv setup function, and the process bytes (i.e., generate keystream) function. Boiler-plate code for converting row-based buffers, containing keys or plaintexts, to the GPU stream-based (i.e., column-based) buffers and device to/from host copies and allocations is provided. Below we present the details of the main functions; the full API, test code, and example implementations are presented in Appendix D.

Initialization

```
1 void gSTREAM_init(gSTREAM_ctx* ctx, int device, int nr_threads, int nr_blocks);
```

Given the device number, and grid dimensions, the function initializes the device, setting the appropriate properties (e.g., pinned memory), and executes any user-provided cipher-state initialization code.

Key and IV setup

```
void gSTREAM_keysetup(gSTREAM_ctx* ctx, u8* keys, u32 keysize, u32 ivsize);
```

Given an initialized context, column-aligned keys and key- and iv-size, the function allocates the multi-stream key buffers, transforms the input keys and copies them to the device.

```
void gSTREAM_ivsetup(gSTREAM_ctx* ctx, u8* ivs);
```

Given an already key-setup context, and column-aligned IVs, the function allocates they multi-stream IV buffers, transforms the input IVs and copies them to the device. The function then launches a user-defined kernel that does the actual key-IV setup.

Keystream generation

```
void gSTREAM_process_bytes(gSTREAM_action action, gSTREAM_ctx* ctx,
                           u8* inputs, u8* outputs, u32 length);
```

Given an action (encrypt, decrypt, or generate keystream), a column-aligned input buffer, an output buffer and input length (in bytes), the function allocates working buffers on the device, and transforms column-aligned input to stream-aligned device input. The function then launches the user-defined kernel that does the actual byte processing, and copies the device stream-aligned output to the column-aligned output buffer.

```
void gSTREAM_keystream_bytes(gSTREAM_ctx* ctx, u8* keystreams, u32 length);
```

Given the length (in bytes), the function generates the desired keystream output.

```
void gSTREAM_exit(gSTREAM_ctx* ctx);
```

Deallocates any dynamically allocated buffers, both on the device and host, and destroys context. The function then executes any user-defined code that deallocates the cipher state.

6.1.2 Implementation of eSTEAM Ciphers

We ported the eSTREAM ciphers to the gSTREAM framework, benchmarking the keystream generators. Similar to the work in [32] and Section 6.2, we estimate the performance on a hypothetical 32-bit (modern) architecture.

The estimates for all the eSTREAM ciphers are shown in Table 6.1 where `cxor` denotes a conditional `xor`. These *raw* instruction counts are obtained from the optimized implementations as submitted to ECRYPT, except for the Grain steam cipher. We note that only the number of instructions in the keystream generator are considered. Since load and store operations are hard to predict (due to possible cache misses), and may be incompatible between platforms, only arithmetic instructions are taken into account (i.e., the required moves, loads/stores, including all the possible table-lookups, are ignored).

We would like to stress that the performance figures presented in Table 6.1 are estimates for a hypothetical 32-bit architecture, the instruction set of which includes all the operations shown in the columns of Table 6.1. Moreover, we assume that such a machine can dispatch one instruction per clock cycle. Estimating the actual performance number on modern platforms is considerably more difficult because they often have access to a separate SIMD unit, which is ignored by our estimates. The motivation for providing cycle/byte estimates is to have a rough estimates that can be used as a starting point to create more accurate platform-specific speed estimations, e.g., the Cell and GPU.

To estimate the rough performance of the stream ciphers on a single GPU of

Cipher	b	and	or	xor	cxor	shft	rot	add	mul	cmp	C/B
Trivium	4	3	15	11	0	30	0	30	0	0	22.25
MICKEY v2	1	17	0	30	8	27	0	0	0	0	656
Grain	1	312	48	344	0	504	0	0	0	0	1212
Rabbit	16	8	0	16	0	32	12	40	32	8	9.25
Salsa20	64	0	0	320	0	0	320	338	0	0	15.28
HC-128	4	1	0	2	0	0	3	13	0	9	7.0
SOSEMANUK	80	45	10	190	0	100	20	60	20	0	5.6

Table 6.1 Performance estimates, in cycles/byte (C/B), for eSTREAM ciphers based on the number of 32- arithmetic instructions used in the keystream generator (which processes b bytes). We assume that all operations stated in the columns are single instruction operations.

the GTX 295 graphics card, we divide the cycles/byte figure of Table 6.1 by a factor of 240. A factor of 240 is used because a single GTX 295 GPU contains contains 30 SMs, for a total of 240 SPs, each dispatching an instruction each cycle. The GPU estimates are presented in Table 6.1.2, along with actual implementation results. Note that the GPU estimated performance results do not account for message memory-register copies or moves. Additionally, they do not account for kernel launch overhead, host-to/from-device copies, or possible table-setup timings (e.g., copying a table to shared memory). More importantly, we note that with the exception of MICKEY v2 and Grain, the estimates are overly optimistic since they also do not account for the limited PCIe bandwidth. A 16 lane PCIe v2 card would have a raw bandwidth of 64Gb/s, without account for operating system overhead or even the direct 20% overhead of the 8b/10b encoding scheme.

For most of the stream ciphers the measured performance is, as expected, lower than the estimated figures. Despite this, we note that for all but HC-128,

Stream cipher	Estimate		Measured	
	C/B	Gb/s	C/B	Gb/s
Trivium	0.09	106.8	0.23	43.2
MICKEY v2	2.73	3.68	2.97	3.35
Grain	5.05	1.97	4.09	2.43
Rabbit	0.05	225.8	0.23	42.5
Salsa20	0.11	94.32	0.23	42.4
HC-128	0.04	280.5	4.39	2.26
SOSEMANUK	0.03	393.3	—	—

Table 6.2 GPU performance results and estimates for eSTREAM ciphers. The GPU implementations process 680 blocks of 256 threads on a single NVIDIA GTX 295 GPU.

the measured performance either closely matches the estimate or peaks at nearly 43 Gb/s, indicating a bandwidth limitation. We provide two example implementations of MICKEY v2 and Trivium in Section D.2 and D.3, respectively. Furthermore, for completeness, we discuss the performance of the ciphers with a less-direct implementation, specifically Grain, HC-128, and SOSEMANUK.

As Grain is a hardware-oriented stream cipher, the original submission code did not aim to optimize the code for CPU implementations [59, 60]. Hence, we provide an alternative implementation that is well suited for software, without the use of 64-bit instructions (or registers) or SIMD units. The code is presented in Appendix C, and Figure 6.2 shows the performance comparison with the original submission. When generating keystreams of length between 1KB and 128KB, our implementation is, on average, 6.6 times faster than the original implementation. Hence, we use this implementation when estimating the performance of and

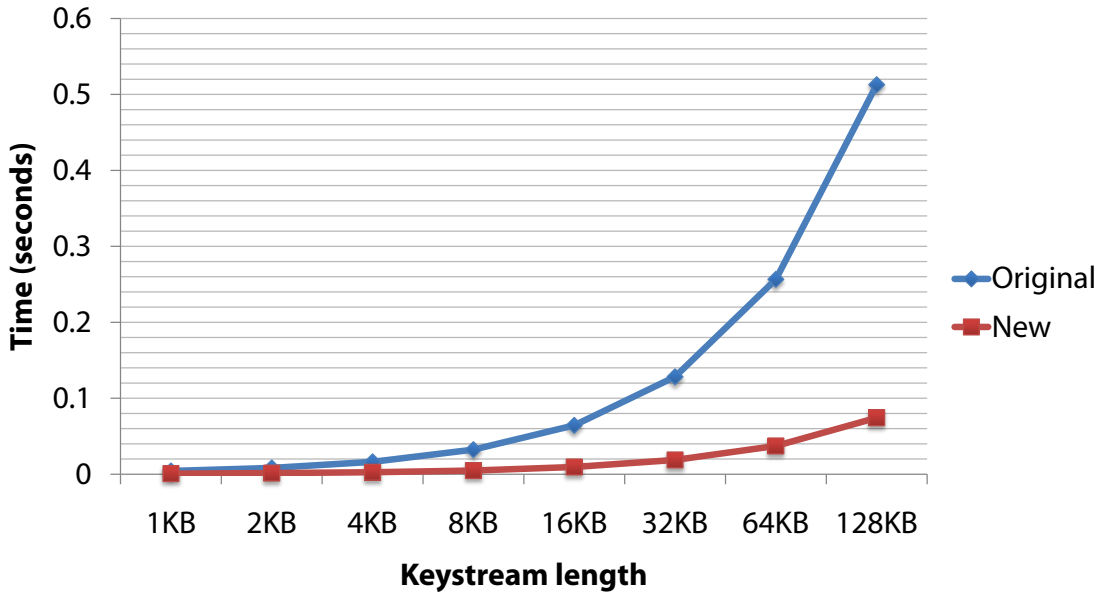

Figure 6.2 32-bit Grain Core i7 920 (2.8GHz) benchmarking results of original eSTREAM, unoptimized submission, and new software-oriented Grain implementation. The average speedup factor of 6.6 is achieved without use of any SIMD or 64-bit instructions.

porting the algorithm to the GPU. As shown in Table the measured performance of Grain closely matches its estimate.

HC-128 is a software-oriented stream cipher with a keystream generator that depends on the modification of two 2KB secret tables, P and Q (see [117]). Importantly, at each iteration step an element of the tables is updated [117], and so the memory access relative to the arithmetic instruction count is quite high. When porting to the GPU, it is important to remember that each SM is limited to 16KB of shared memory, and so it would not be possible to launch a grid with thread blocks consisting of more than 4 threads that store the P and Q tables in shared memory (each thread would need *at least* 4KB for the tables alone). In addition, a thread block of 4 threads would give rise to many scheduling inefficiencies and so, as to launch a larger number of threads, in our implementation we store the

secret tables in global memory. As mentioned in Chapter 3, global memory transfer is also very costly; this performance penalty is observed in HC-128's very low throughput, as shown in Table 6.1.2.

SOSEMANUK is also a software-oriented stream cipher, using design principles from from the SNOW 2.0 stream cipher, and SERPENT block cipher [15]. Compared to the other eSTREAM cipher, SOSEMANUK is considerably more complex and, though appropriate for CPU implementations, not well-suitable for current GPUs [2]. The SOSEMANUK design uses multiplication and division of elements in $\mathbf{F}_{2^{32}}$ by basis element α. In the actual implementation[3], this is accomplished using two lookup tables the combined size of which would exceed not only shared memory (16KB), but the constant memory cache (32KB). Hence, we leave its implementation to future work and GPUs with larger caches.

6.2 GPU Implementation of SHA-3 Candidates

In this section we present estimates for the SHA-3 candidates using instruction counts for both a generic modern architecture and NVIDIA GPUs followed by actual implementations of all the non-AES hash functions. This chapter has been presented in the form of a conference paper at the *Workshop on Cryptographic Hardware and Embedded Systems 2010*, see [32].

In addition to the general porting patterns mentioned in the chapter introduction, some hash function-specific optimizations were used in implementing the SHA-3 candidates. Specifically, for algorithms with small-to-medium sized chain values (e.g., 256- or 512-bits), we buffer the chain values in registers. To avoid

[2]Based on our implementations, only Trivium, Rabbit and Salsa20 are deemed suitable for current GPUs.

[3]Available at http://www.ecrypt.eu.org/stream/sosemanukp3.html

multiple kernel launches, each thread processes multiple message blocks. This, in conjunction with the caching of the chaining values, not only simplifies the multi-block hashing, but also results in a faster implementation (than, for example, executing multiple kernels and having to read/write chain values from/to global memory). For algorithms with larger-sized chain values or internal states, we cache the chain values in shared memory. In implementing algorithms that use shared memory, we require that the thread block size always be a multiple of 16 threads (usually at least 64 threads) and further (implicitly) assert that the n-th thread (counting from 0) loads/stores any shared memory cached values from/to bank n mod 16, as to avoid bank conflicts.

6.2.1 AES-Inspired SHA-3 Candidates

A popular design choice of the SHA-3 hash function designers was to use AES-like byte oriented operations (and, in some cases the AES round function itself) as building blocks in the compression function of their hash function. The second-round SHA-3 candidates following this paradigm include ECHO [13], Fugue [56], Grøstl [54], and SHAvite-3 [25]. The motivation for using AES-like operations is mainly because AES has successfully withstood much cryptanalytic effort and, moreover, one can exploit the high capabilities of AES-like functions on a wide variety of architectures. Moreover, many of the design teams have pointed out the new Intel AES instruction set and claimed several performance figures outperforming the other candidates (for a more detailed analysis, cf. [14]). Considering the possible widespread use of these processors in the future, these designs will likely have a clear advantage.

Although several optimization methods for these hash functions are possible on particular processors, such as using the Intel AES instruction set, we ana-

lyze the performance of AES-inspired candidates in a more generic setting. More precisely, we simply count the number of 'AES-like' operations required for the compression function of each candidate, as this gives an intuition of how these designs behave in architectures without native AES-instructions, such as the PowerPC, SPARC, and most low-power microcontrollers. Table 6.4 provides these rough estimates. Note that since the operations may differ per candidate, we clearly differentiate all possibilities, particularly the variants of the 'Mix-Column' (MC) operation used in AES.

The estimates given in Table 6.4 provide a good indication on the performance of the AES-inspired candidates, especially for hashing extremely long messages, where we simply focus on the compression functions. It should, however, be noted that the techniques used to implement the MC operations used by these candidates account for the largest performance loss/gain. Typically, the MC operation is implemented using a number of `xor` operations and the `XTIME` function. The latter treats a byte-value as a polynomial in the finite field $\mathbf{F}_{2^8}$ and performs modular multiplication by a fixed modulus and multiplier. In practice, `XTIME` can be implemented using a `shift` and a conditional `xor`. An upper bound on the required MC-operations, working on single byte-values, is given in Table 6.3. First, the double and quadruple of the X elements are computed in MCX for $X \in \{8, 16\}$; the octuple for MC16 is not needed since all the constants in Fugue are below 8. We note that these require $2 \cdot X$ `XTIME` operations, and that the number of required `xor` operations depend on the constants. Counting the latter, for MC4 in AES and MC8 in Grøstl, there are at most $4 \times 5 - 4 = 16$ and $14 \times 8 - 8 = 104$ `xor` instructions, since the rows are simply rotations of each other. Similarly, in Fugue there are $4 \times (10 + 8 + 14 + 9 - 4) = 148$ `xor` instructions, corresponding to its constants. We stress that these (naive) estimates should be treated as an upper

		XTIME	xor (byte)	size of table(s) in bytes	xor	rotate
MC4	(AES)	4	16	1,024	3	3
				4,096	3	0
MC8	(Grøstl)	16	104	2,048	7	7
				16,384	7	0
MC16	(Fugue)	32	148	4,096	15	15
				65,536	15	0

Table 6.3 Straight-forward estimates for the different mix-column operations without (left) and with (right) the use of T-tables. Note that the `xor` and `rotate` instruction counts for the T-table approach in MCX operate on $(8 \cdot X)$-bit values.

bound; as illustrated by the implementation of MC4 in [95], the number of times `XTIME` and `xor` are required is lower: 3 and 15, respectively.

Following the "T-table" approach [40], the MC and substitution steps can be performed by using lookup tables on 32-bit (and larger) processors. The use of T-tables can greatly reduce the number of required operations; estimates of the cost of the different MC steps using a varying number of T-tables (as the different tables are simply rotations of each other) are also stated in Table 6.3. The MCX T-table variants require $X - 1$ `xor`, and 0 or $X - 1$ `rotate` instructions (depending on the number of tables used) operating on X-byte values. The use of T-tables is, however, not always favorable where, for example, in memory constraint environments, the tables might be too big.

Among the four AES-inspired second-round SHA-3 candidates, ECHO and SHAvite-3 make use of the AES round itself and can highly benefit from Intel AES instruction set. Therefore, it is relatively easy to infer the speed estimates for

these two hash functions once we have those for AES. We use the recent work by Osvik et al. [95] on AES to obtain estimates for our target platforms. Based on their results, the corresponding workload required to implement the compression function of the AES-inspired candidates is given in Table 6.4. As an example of how SHAvite-3 performs under this result (given the estimates of Table 6.4), one requires 52 AES round function evaluations plus 1280 8-bit `xor`s to perform one compression function invocation of SHAvite-3, compressing a 64 byte message block. From [95] we learn that one AES round can be implemented in 78600 cycles on a single GTX 295 GPU when hashing 600 blocks of 256 streams simultaneously. Hence, SHAvite-3 is estimated to achieve performance of $\frac{52 \cdot 78600 + 1280}{64 \cdot 256 \cdot 600} = 0.42$ cycles/byte on the GPU.

We note that the performance estimates given in Table 6.4 for Grøstl and Fugue are conservative. This is because the naive estimates for MC8 and MC16 use the estimate from Table 6.3, leaving room for significant optimizations. These numbers can be further improved on platforms where a T-table approach is faster than computing the Mix-Column operation. For example, on the GPU, placing the smaller (2KB) table in shared memory, Grøstl would require two 32-bit lookups in addition to the 7 `xor` and 7 `rotate` (64-bit) instructions.

6.2.2 Other SHA-3 Candidates

The non-AES based SHA-3 candidates use a variety of techniques and ideas in their hash function designs. As in the stream cipher case, from a performance perspective, it is interesting to have an indication of the number of required instructions per byte. An approximation of this is given in Table 6.5. We note that operations ending with a '`c`' indicate that one of the input parameters is complemented before use, `eqv` denotes bitwise equivalence (i.e., `xorc`) and `csub` denotes

Hash function	b	(R)	SB	MC4	MC8	MC16	xor	C/B	Gb/sec
AES-128 [95]	16	10	—	—	—	—	16	0.32	30.9
ECHO-256	192	256	—	512	—	—	448	0.85	11.7
Fugue-256	4	—	32	—	—	2	60	0.62	16.1
Grøstl-256	64	—	1280	—	160	—	1472	1.23	8.1
SHAvite-3-256	64	52	—	—	—	—	1280	0.42	23.7

Table 6.4 The number of AES-like operations per b bytes for all AES-inspired candidates and the performance estimation on a single GTX 295 GPU. (R): One AES encryption round, SB: Substitution operation, MCX: Mix-Column operation over X bytes (i.e., X=4 is identical to the one used in AES). Note that Shift-Row operations are ignored because it can be dispatched through the Mix-Column operation. The xor count is per-byte.

conditional subtraction. These *raw* instruction counts are obtained from the optimized implementations as submitted to NIST and only the number of instructions in the compression function are considered; we only account for arithmetic instructions.

We stress that the performance figures presented in Table 6.5 are estimates for a hypothetical single-dispatch 32-bit architecture, the instruction set of which includes all the operations shown in the columns of Table 6.5. Estimating the actual performance number on modern platforms is considerably more difficult because they often have access to a separate SIMD unit, which is ignored by our estimates. However, these estimates can be used as a starting point to create more accurate platform-specific speed estimations, for instance for the Cell and GPU architectures. Note that while the multiplications by the candidate SIMD operate on 16-bit operands, the multiplications in Shabal are by one of the constants {3, 5}. Each of the latter multiplications can be converted into a shift and addition, if

cheaper than native multiplication.

As noted before, the PTX ISA is quite limited, and therefore some of the instructions in Table 6.5 will have to be implemented by multiple, simpler, instructions. For example, each `rotate` is implemented using two `shift` instruction and an `or`; each `andc` is implemented using a `not` and an `and`, etc. Taking the implementation of these non-native instructions into account, as in the stream cipher case, we divide the (slightly higher) instruction count of Table 6.5 by a factor of 240. These estimates are presented in Table 6.6, along with actual implementation results.

The estimated performance results do not account for message memory-register copies or moves, kernel launch overhead, etc. For fair comparison, we, however, do account for the chain value copies to/from registers and global memory; this rough figure was measured for the different sizes using a kernel that simply copied the state to registers and back to global memory. Nevertheless, our GPU estimates are certainly optimistic and implementation results, measuring the full hash function, are higher. Additionally, for algorithms with huge internal states or expanded messages, e.g., SIMD, the use of local storage might not be easily avoided and the implementation results are expected to be much worse than the estimates.

Along with considering the general port patters and optimization techniques when implementing the candidates, we further emphasize the details of Keccak and Hamsi. Since using large tables on the GPU is prohibited, we estimate and implement Keccak with on-the-fly interleaving (Keccak-256† in Table 6.6) and divide the execution of Hamsi into two kernels. The latter requires the use of a very large 32KB table (which is larger than all the fast memories on the SMs) for the message expansion, and, thus, necessitates a less direct implementation approach.

Hash function	b	add	sub *csub*	mul	and	nand *andc*	eqv	or *orc*	rotate	shift	xor	C/B
Hash functions operating on 32-bit words												
BLAKE-32	64	480	—	—	—	—	—	—	320	—	508	20.4
BMW-256	64	296	58	—	—	—	—	—	212	144	277	15.4
CubeHash-16/1	1	512	—	—	—	—	—	—	512	—	512	1536.0
CubeHash-16/32	32	512	—	—	—	—	—	—	512	—	512	48.0
Hamsi-256	4	—	—	—	24	12	—	24	72	24	287	110.8
JH-256	64	—	—	—	1792	*1152*	288	688	—	800	4024	136.6
Keccak-256	136	—	—	—	756	384	—	624	1248	360	4224	55.9
Luffa-256	32	—	—	—	144	—	96	96	392	—	756	46.4
Shabal-256	64	52	16	96	—	*48*	48	—	112	—	242	9.6
SIMD-256	64	817	901 *256*	419	852	—	—	256	288	804	176	74.5
Hash functions operating on 64-bit words												
Skein-512	64	497	-	-	1	-	-	-	288	-	305	17.0

Table 6.5 Performance estimates for all non-AES inspired SHA-3 candidates based on the number of 32- and 64-bit arithmetic instructions used in the various compression functions (which process b bytes). We assume that all operations stated in the columns are single instruction operations.

The proposed two-part approach requires: (i) a kernel in which 16 threads expand the 32-bit message to 256-bits (each using 2 $1KB$ tables and an atomic `xor`), and (ii) a kernel implementing the actual compression function. Because the message expansion requires random access reads and uses atomic instructions (to global memory), estimates without considering the effects of memory operations are expected to diverge.

As expected, we observe that the actual performance numbers in Table 6.6 are slightly higher than the corresponding estimated figures. In most cases, however, the performance overhead is a result of the memory copies (host-to-device and global memory-to-registers). We confirmed this conjecture by measuring the throughput of the compression functions working on a single message block, the results of which are shown [in brackets] in Table 6.6. We note that the implementation result of SIMD does not, however, agree with our estimated figure. We attribute the extremely low performance to using local memory for the message expansion (4096 bits) and having a single thread do the full compression; splitting the compression function across multiple threads would likely improve SIMD's performance. Additionally, we highlight the Shabal implementation, for which we heavily used the optimized reference code, required the use of a non-inline function in the permutations as to address a compiler optimization bug; the fully-inlined, but buggy, implementation is twice as fast.

6.3 Multi-GPU Implementation of the Cube Attack

In this section we present some of the implementation details for the cube attack, as describe in Chapter 4. Our implementation is limited to to the offline phase: the preprocessing stage. Specifically, we implement algorithms for finding max-

Algorithm	Estimate		Measured		
	C/B	Gb/s	C/B	C/B	Gb/s
BLAKE-32	(0.13)	(76.4)	[0.13]	0.27	36.8
BMW-256	(0.10)	(99.4)	[0.27]	0.27	36.8
CubeHash-16/1	(10.9)	(0.91)	[11.0]	11.1	0.90
CubeHash-16/32	(0.34)	(29.2)	[0.35]	0.36	27.6
Hamsi-256	(0.64)	(15.5)	[0.66]	5.19	1.91
JH-256	(0.67)	(14.8)	[0.75]	0.76	13.1
Keccak-256	(0.31)	(32.1)	[0.56]	0.56	17.7
Luffa-256	(0.32)	(31.1)	[0.34]	0.35	28.4
Shabal-256	(0.07)	(141.9)	[0.56]	0.69	14.4
SIMD-256	(0.43)	(23.1)	[0.65]	3.60	2.76
Skein-512	(0.22)	(45.2)	[0.29]	0.46	22.1

Table 6.6 Performance results and estimates for the non-AES based SHA-3 candidates for the GPU architecture. The implementations process 680 blocks of 64 threads on a single NVIDIA GTX 295 GPU. Measurements of only the compression function are shown in [brackets].

terms, testing the linearity of such maxterms, and superpoly reconstruction given a maxterm. Although our implementation can easily be modified to include the online attack, we do not implement it as there are many tools well-suited for this task (recall that the main part of the online phase consists of solving a set of linear equations over $GF(2)$). Moreover, one of the main advantages of the cube attack is the trade off in online time/complexity for a longer/more complex preprocessing stage.

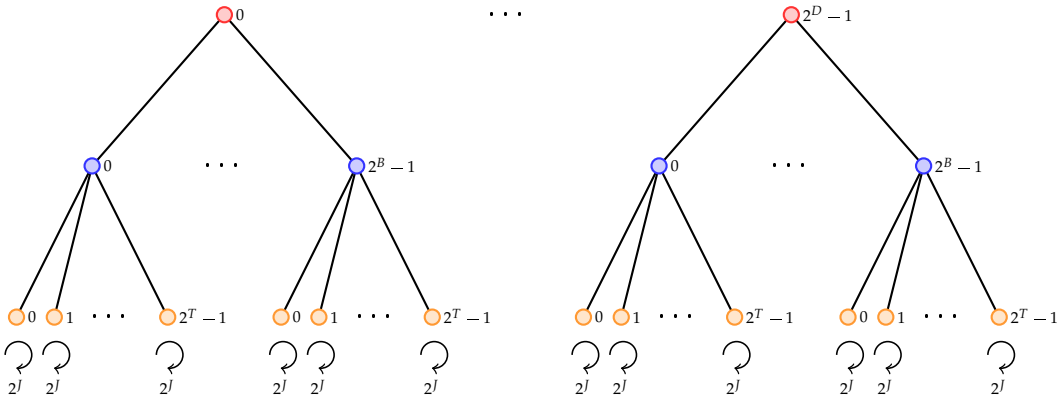

Figure 6.3 Finding a maxterm for high-dimensional cube. In this setting every thread across 2^D devices (each containing 2^B blocks) computes the sum over a sub-cube of dimension J.

6.3.1 Finding Maxterms

In Chapter 4 we presented a concrete algorithm for finding maxterms. We implemented this algorithm (Algorithm 4.1) in the context of a single-threaded CPU, using C, and multi-GPU, using CUDA. The C implementation is used when the degree of the black box polynomial is assumed to be low, in addition to serving as a reference for the multi-GPU implementation.

One of the key-observations to efficiently implementing the cube attack on multiple GPUs is that the sum over a cube can be divided among different threads. For example, for a very high-dimensional cube I of degree k we can can compute p_{C_I} by delegating sub-cube summations to different threads on different devices, as shown in Figure 6.3. In this scenario every thread on the 2^D devices is independently computing a sum over a sub-cube of size J. Each thread-block is in turn reducing the individual thread summations into a sub-cube of degree $T + J$, since there are T threads per block. Finally, $B = |I| - (D + T + J)$ thread blocks are scheduled on each device. We note that our implementation uses 2^D

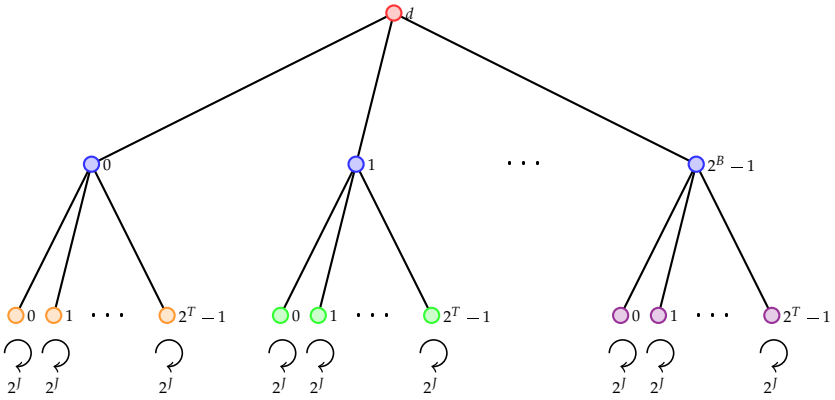

Figure 6.4 Finding a maxterm for medium-dimensional cube. In this setting every thread block computes a full cube sum by delegating 2^T threads a sub-cube of dimension J. 2^B different maxterms are analysed in parallel on a single device d.

'virtual' devices which are mapped to physical devices in a round-robin fashion. For example, in a system consisting of 3 physical devices, we define $D = 2$, i.e., 2^2 virtual devices, in which the fourth virtual device maps to the first physical device.

Though the previous implementation approach is viable, it is limited in several ways. First, the algorithms only searches for a single maxterm at any one time. Second, reducing the 2^B sub-cube summations on each devices requires the use of a global memory cache, and similarly reducing the 2^D sub-cube summations requires the use of a CPU thread. Third, and worse yet, multiple kernel launches are required to fully implement Algorithm 4.1. This is a direct result of implementing the BLR test which requires the summation over the same cube multiple times. Though our initial implementation included this approach, our benchmarking results showed that it is, indeed, quite inefficient and its use is only applicable to very high dimensional cubes (where the kernel launches and global-memory access are no longer the primary bottlenecks).

We also implemented Algorithm 4.1 by delegating the sub-cube summations according to Figure 6.4. In this setup, each thread block, on device d, searches for a maxterm. Hence, on each device we search for 2^B maxterms with cubes of degree k, in parallel, which multiplicatively increases according to the number of physical devices in the system. Each thread block then computes the full cube sum among T threads, which, in turn, compute the sum of a sub-cube of degree J, such that $T + J = k$. We note that the thread blocks in our actual implementation are not restricted to working with a cube of degree k, rather they are only restricted to a dimension range and generate the dimension and index-set randomly. Additionally, since the SPs operate on 32-bit words, our implementations are optimized to compute 32 ciphertext bits in parallel (a form of bitslicing). We now present some additional implementation-specific details for this approach.

Upon launching a kernel, the first thread of each thread block chooses the dimension k of the cube and then randomly chooses an index-set I of such dimension. The index-set I is then written to shared memory, along with other values necessary for coherence (e.g. random number generator seed and the counters n_0, n_1 of Algorithm 4.1); we primarily use the shared memory as a cache, updating the thread local values on every synchronization point. Addressing the need of a random number generator we implemented a 32-bit XOR-Shift random number generator [77], the seed of which is cached in shared memory. The latter point is necessary so that the threads in the thread block generate the same thread-local keys used in the BLR tests. An alternative approach would entail 2 threads generating random keys that are then cached in shared memory; this approach is, however, less efficient than generating the keys on-the-fly primarily because of the communication overhead in reading the keys from the cache and having the majority of the threads stall, waiting for the keys to be generated. Similar to the

initial point of generating I, the first thread is used to write I to global memory, if the BLR tests confirm the linearity of its superpoly. Otherwise, the thread block finishes with no (successful) results. Taking the aforementioned details into consideration, the implementation of Algorithm 4.1 follows directly. We note that for our tests try each maxterm 20 times (i.e., $T = 20$ following Algorithm 4.1), and perform the BLR tests $3N = 128$ times. Furthermore, we have a simple algorithm setting J, the sub-cube size each thread sums over, and 2^T, the number of threads per block, according to the bound of the cube dimension; the number of threads rounded to the nearest multiple of 64.

6.3.2 Superpoly Reconstruction

The implementation of the superpoly reconstruction algorithm given in Algorithm 4.2 follows directly from our maxterm search implementation detailed above. We, again, use a thread block to do a full cube-sum, each thread performing a sum over a sub-cube of size J. Unlike the maxterm search implementation, however, we reconstruct a single maxterm using all the devices. Specifically, every thread block computes a single coefficient. Following the kernel execution, the coefficients are copied from the device to the host (CPU) and the constant coefficient is added (XOR) to all the coefficients.

6.3.3 Performance Measurements

Our implementation targets any black-box polynomial implementing a set of macros, and is thus not specific to any of the ciphers we cryptanalyzed in this work. To measure the performance of our GPU implementation, relative to the CPU counterpart, we benchmarked the both the maxterm search and superpoly

k	CPU	1 GPU	2 GPUs	3 GPUs	4 GPUs
12	49.5	8.1	7.55	7.38	4.75
13	217.81	28.8	27.82	14.6	17.87
14	1159.43	83.87	55.98	29.8	32.15
15	3988.91	176.1	116.4	60.04	62.67
16-17	25322.29	718.61	468.84	241.35	236.03

Table 6.7 Performance measurements of the maxterm search algorithm in seconds on the simplified version of Trivium. In the final case the dimension was bounded between 16 and 17. Our CPU implementation executes on an Intel Core i7 920 (2.67GHz). Our GPU implementations execute on GTX 295 graphics cards, varying the number of GPUs used in the search. Each run was limited to searching for 80 terms, with the number of tries set to $T = 20$, and number of BLR tests set to $3N = 128$.

reconstruction algorithms. As the maxterm search depends on randomness and as such is not easily benchmarked, we note that our results serve to only give a rough comparison. The code will be available publicly and, as such, benchmarking by third parties is recommended to confirm our results. Table 6.7 summarizes our benchmarking results for the maxterm search algorithm, while Table 6.8 summarizes the performance of the superpoly reconstruction implementation.

As our results show, the maxterm search algorithm is well-suited for a multi-GPU setting. Increasing the number of GPUs almost directly results in a greater speedup. Compared to the CPU implementation, we measured the multi-GPU implementations to be considerably faster, with factors of up to 107. Note, however, that the overhead of the threads library is not negligible when increasing the number of GPUs in the system. For example, the 3 GPUs performance is comparable to the 4 GPUs, sometimes even faster.

In the superpoly reconstruction case, a single GPU also outperforms the single

k	CPU	1 GPU	2 GPUs	3 GPUs	4 GPUs
12	2.54	0.4	0.75	0.74	4.74
13	5.09	0.47	0.76	0.74	4.74
14	10.2	0.61	0.82	0.73	4.76
15	20.43	0.93	1.02	1.05	5.13
16	40.84	1.52	1.33	1.20	5.44
17	81.67	2.71	2.45	1.92	5.76

Table 6.8 Performance measurements of the superpoly reconstruction algorithm in seconds on the simplified version of Trivium. Our CPU implementation executes on an Intel Core i7 920 (2.67GHz). Our GPU implementations execute on GTX 295 graphics cards, varying the number of GPUs used in the search.

CPU implementation by up to a factor of 42. As expected, increasing the dimension size results in the GPU outperforming the CPU since the memory copies and kernel launches are minimal when compared to the actual computation time. As before, we note that increasing the number of GPUs in the system does not always result in a speedup. This is primarily because of the threads library setup, memory device to/from host copies, cache misses, and kernel launches dominate the computation. Compared to the maxterm search algorithm, the superpoly reconstruction algorithm is less complex and thus such issues arise. Nevertheless, the results of the single GPU implementation highlight that the algorithm is well-suited for this architecture and we expect it to outperform the CPU implementation with larger factors than the current 42×, for higher-dimensional cubes.

For completeness and since this is the first implementation of the XOR-Shift random number generators on GPUs, we measure the performance of the implementation given in Appendix E. Using a 256 thread blocks, each block consisting

of 256 threads, which in turn generate 100,000 random numbers we measured the performance of XOR-Shift to be 0.0094 cycles/byte on a single GPU of the GTX 295 graphics card. This figure closely agrees with the estimated 0.0063 cycles/byte estimate, which we compute by dividing the number of cycles (6) required to compute a 32-bit sample using 240 SPs. We further note that this random number generator grossly outperforms other common, but GPU-inefficient, random number generators, such as the widely-used Mersenne Twister [78, 98].

Chapter 7

Cryptanalysis Results

In this chapter we present our analysis results. We analyze MICKEY and Trivium using the cube attack, and, dually, BLAKE and CubeHash using the linear differential cryptanalysis framework. We further present implementation details and performance measurements of the linear differential cryptanalysis framework; the corresponding details for the cube attack were presented in Section 6.3.

7.1 Applying the Cube Attack

Our implementation of the cube attack is designed to target any black box polynomial. We do not take advantage of the internal details of the analyzed stream ciphers. This is primarily because we are interested in providing a general framework that can easily be used to analyse many algorithms, though still allowing for simple cipher-specific extensions. To verify the correctness of our implementation we apply the cube attack to the Trivium stream cipher, confirming the results of [45]. Given the confidence in our implementation we further analyze the MICKEY stream cipher.

7.1.1 Trivium

As in [45], we analyse a Trivium (see Section 2.1.1) by reducing the number of initialization rounds to 672, i.e., $N_{\text{pre}} = 672$. Our analysis primarily serves to verify our implementation, rather than undertake variants of Trivium with initialization rounds beyond those of [45]. Since our implementation produces 32-bits, in parallel, we present the maxterms for out bit index 672 to 703. Table 7.1 shows several maxterms and their corresponding superpolys our maxterm search algorithm found. These maxterms were previously presented in [45], affirming the confidence in our framework. In addition, we found some new maxterms which we present in Table 7.2. Although our analysis led to considerably more maxterms than we show, we only present the smallest maxterm necessary in the recovery of a particular bit. Note that our results can be used to carry out an online partial-key recovery attack on this simplified variant of Trivium. We believe that further analysis could lead to a full-key recovery attack, though an extension to the full Trivium is not direct.

7.1.2 MICKEY

As with Trivium, we use the multi-GPU cube attack framework to analyze a simplified version of MICKEY stream cipher detailed in Section 2.1.2. We analyzed MICKEY with no initial vector and no state updates during the Key and IV setup, i.e., $N_{\text{pre}} = 0$. Varying the cube dimension sizes to up to 20, we found no terms with a linear superpoly. Although increasing the cube dimension to a larger, but still manageable size (e.g., 32), or leveraging the internal details of the simplified cipher could potentially reveal maxterms, we believe that the full MICKEY cipher is resistant to such black box.

Maxterm, I	Output bit #	Superpoly
{3,13,18,26,38,40,47,49,55,57,66,79}	672	$1 + x_{01} + x_{25}$
{2,5,7,10,14,24,27,39,49,56,57,61}	672	$1 + x_{14}$
{3,5,14,16,18,20,33,56,57,65,73,75}	672	$1 + x_{24}$
{2,12,17,25,37,39,46,48,54,56,65,78}	673	$1 + x_{00} + x_{24}$
{0,5,8,11,13,21,22,26,36,38,53,79}	673	x_{12}
{0,5,8,11,13,22,26,36,37,38,53,79}	673	x_{13}
{10,13,15,17,30,37,39,42,47,57,73,79}	673	$1 + x_{21} + x_{66}$
{3,14,21,25,38,43,44,47,54,56,58,68}	674	$1 + x_{01} + x_{10} + x_{51}$
{8,11,13,17,23,25,35,45,47,54,70,79}	674	$1 + x_{39} + x_{57} + x_{66}$
{2,13,20,24,37,42,43,46,53,55,57,67}	675	$1 + x_{00} + x_{09} + x_{50}$
{11,18,20,33,45,47,53,60,61,63,69,78}	675	x_{04}
{1,3,6,7,12,18,22,38,47,58,67,74}	675	x_{07}
{1,12,19,23,36,41,42,45,52,54,56,66}	676	$1 + x_{08} + x_{49} + x_{68}$
{6,11,14,19,33,39,44,52,58,60,74,79}	676	$1 + x_{28}$
{0,6,10,16,19,31,43,50,66,69,77,79}	676	$x_{40} + x_{58} + x_{64}$
{1,6,8,19,22,33,39,44,60,68,74,79}	677	$1 + x_{03} + x_{35} + x_{63}$
{7,14,16,18,27,31,37,43,48,55,63,78}	677	x_{05}
{1,5,7,18,21,32,38,43,59,67,73,78}	678	$1 + x_{02} + x_{34} + x_{62}$

Table 7.1 Trivium analysis results: maxterms presented in [45] and confirmed using our cube attack framework

Maxterm, I	Output bit #	Superpoly
{ 1, 3, 14, 20, 23, 40, 43, 46, 52, 53, 55, 57, 65}	672	x_{57}
{ 0, 2, 15, 19, 23, 27, 30, 35, 36, 45, 46, 50, 60, 65, 72}	672	x_{65}
{ 6, 7, 10, 23, 27, 31, 34, 47, 54, 57, 63, 67, 72, 79}	672	x_{56}
{ 1, 23, 30, 36, 40, 49, 52, 53, 55, 56, 57, 60, 64, 67, 71, 77}	672	x_{64}
{ 0, 18, 22, 27, 35, 38, 45, 46, 56, 61, 66, 71, 74, 77}	673	x_{63}
{ 8, 15, 17, 20, 22, 23, 30, 35, 47, 53, 54, 66, 68, 71, 75}	673	x_{22}
{ 2, 5, 7, 10, 18, 20, 26, 33, 45, 47, 64, 67, 70, 73, 75, 77}	673	x_{62}
{ 5, 8, 13, 16, 21, 26, 30, 38, 39, 40, 43, 44, 46, 52, 62, 64}	673	$1 + x_{66}$
{ 0, 8, 10, 12, 19, 36, 38, 40, 44, 45, 69, 71}	673	$x_{27} + x_{54}$
{ 0, 2, 14, 19, 20, 22, 29, 30, 36, 53, 58, 64, 70, 72, 79}	674	x_{55}
{ 1, 6, 11, 12, 22, 30, 33, 40, 46, 63, 68, 71, 79}	674	x_{60}
{ 1, 7, 14, 21, 31, 47, 53, 57, 61, 62, 67, 71, 72, 73, 79}	674	$x_{21} + x_{51}$
{ 1, 12, 19, 33, 34, 35, 46, 58, 65, 70, 71, 72, 78}	674	x_{54}
{ 2, 3, 9, 10, 12, 24, 26, 33, 39, 40, 49, 51, 58, 62, 63, 66, 70}	674	x_{59}
{ 1, 14, 18, 19, 25, 26, 42, 43, 45, 47, 50, 53, 55, 65, 69, 70, 79}	674	x_{51}
{ 1, 5, 11, 14, 15, 17, 18, 21, 27, 29, 30, 39, 41, 58, 70, 72}	674	x_{41}
{ 3, 4, 7, 15, 21, 22, 23, 36, 38, 47, 50, 52, 63, 74, 79}	675	x_{61}
{ 4, 8, 12, 15, 18, 22, 23, 36, 44, 54, 72, 79}	676	$x_{62} + x_{68}$
{ 1, 5, 14, 19, 22, 35, 37, 40, 51, 53, 56, 75, 78}	676	$1 + x_{67}$
{ 6, 18, 19, 20, 22, 26, 34, 36, 40, 43, 51, 55, 60, 78}	676	x_{20}
{ 0, 17, 19, 28, 31, 35, 38, 39, 43, 50, 58, 62, 65, 78}	678	x_{15}
{ 8, 14, 16, 20, 23, 31, 41, 45, 46, 51, 55, 64, 69, 71, 75, 76, 78}	678	x_{58}
{ 2, 7, 10, 15, 16, 19, 29, 33, 38, 49, 57, 61, 65, 70, 77}	679	x_{16}
{ 1, 6, 17, 24, 27, 28, 33, 36, 39, 41, 61, 67, 69, 71, 72, 73, 75}	693	x_{35}
{ 3, 10, 18, 20, 25, 27, 28, 29, 30, 33, 42, 49, 60, 64, 66, 67, 78}	696	x_{29}

Table 7.2 Trivium analysis results: new maxterms found with our cube attack framework

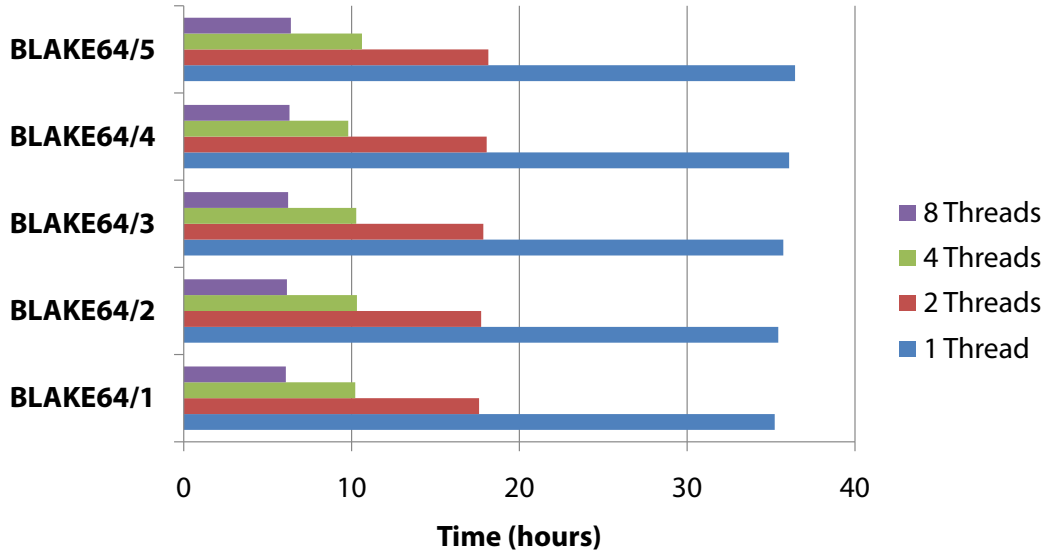

Figure 7.1 Framework performance comparison varying the number of threads when searching for trails on the full BLAKE64 hash function.

7.2 Applying Linear Differential Cryptanalysis

We extend the implementation of the linear differential cryptanalysis framework presented by Brier *et al.* [34]. In Chapter 5 we detail some of our extensions, including the addition of the adaptive backtracking algorithm, the fine-grained message modification algorithms, extension to a larger kernel basis, and a generalization to arbitrary hash functions. The generalization allows us to easily analyze hash functions other than CubeHash [34], and to demonstrate this we present our analysis results on variants of the BLAKE hash function.

Using the OpenMP optimizations mentioned in Chapter 3 we further parallelize the framework. As most of the computation time is spent executing the compression functions, we take a coarse grained approach and optimize the trail search algorithm by performing N number of searches in parallel, where N is the number of CPUs in the system. To evaluate the speedup, we measured the perfor-

r	BLAKE32	BLOKE32	FLAKE32	BLAZE32	BRAKE32
1	813	813	805	**70**	**70**
2	1876	1858	1856	1286	774
3	2933	2907	2942	2139	2262
4	4010	3993	4005	3309	3321
5	5090	5060	**508**	4345	4344

Table 7.3 The minimal number of conditions y found using the forward search algorithm for BLAKE32 variants, the conditions corresponding to the raw probability 2^{-y}.

mance of the framework in searching for paths on BLAKE64 varying the number of rounds and number of threads; Figure 7.1 presents these results, from which we observe an almost linear speedup. Using our modified parallel implementation we present our analysis of BLAKE and CubeHash below.

7.2.1 BLAKE

We analyze the BLAKE hash function family, including BLAKE32 and BLAKE64 presented in Section 2.2.1. As BLAKE relies on an internal compression function, we limit our analysis to this function, given in Algorithm 2.5. An attack on the compression function is sufficient to indicate design weaknesses in the full hash function. For our analysis, we define the linearized compression function by replacing every addition in Algorithm 2.5 with an XOR. As we did not expect to find an attack on the full BLAKE, we varied the number of rounds and also analyzed the toy versions presented in Section 2.2.1.

Our results are summarized in Table 7.3, and Table 7.4 for BLAKE32 and BLAKE64 variants, respectively. The **bold** entries indicate a second preimage at-

r	BLAKE64	BLOKE64	FLAKE64	BLAZE64	BRAKE64
1	1820	1820	1815	**107**	**429**
2	3988	3979	3975	2696	2747
3	6194	6145	6197	4820	4803
4	8394	8346	8400	6848	6911
5	10596	10521	10614	8960	8973

Table 7.4 The minimal number of conditions y found using the forward search algorithm for BLAKE64 variants, the conditions corresponding to the raw probability 2^{-y}.

tack on the compression function with a 512-bit output, i.e., any differential path with a raw probability 2^{-y} above 2^{-512} is considered a theoretical attack. Unlike our analysis of CubeHash (see Section 7.2.2), our analysis of BLAKE has revealed only 5 theoretical attacks. For these attacks we present the differential trails in Appendix F.1. We note that our attacks are on round-reduced toy versions, which do not extend to the full BLAKE. Moreover, as our results confirm, BLAKE is resistant to generic linear differential attacks.

7.2.2 CubeHash

As with the BLAKE hash function, we analyze round-reduced versions of the CubeHash hash function presented in Section 2.2.2. Here we give our results, previously presented in [67], in addition to several new results not previously considered.

Unlike BLAKE, CubeHash is not built by iterating a compression function, and as such we must first define it. Following [34, 67] we define the fixed-input-length compression function Compress. In this work, Compress is parametrized

by a 1024-bit initial value V and compresses t ($t \geqslant 1$) b-byte message blocks $M = M^0 \| \cdots \| M^{t-1}$. The output $H = \mathsf{Compress}(M, V)$ consists of the last $1024 - 8b$ bits of the internal state after tr round transformations processing M.

A colliding message pair $(M, M \oplus \Delta)$ for Compress directly extends to a collision of CubeHash by appending a pair of message blocks $(M^t, M^t \oplus \Delta^t)$ such that Δ^t erases the difference in the first $8b$ bits of the internal state. The difference Δ^t is called the *erasing block difference*. We note that when searching for collisions of Compress, the parameter V is not restricted to be the initial value of CubeHash. Specifically, V can be the state after processing some message prefix M^{pre}. Thus, a pair of colliding messages for the hash function then has the general form

$$(M^{\text{pre}} \| M \| M^t \| M^{\text{suff}}, M^{\text{pre}} \| M \oplus \Delta \| M^t \oplus \Delta^t \| M^{\text{suff}})$$

for an arbitrary message suffix M^{suff}.

We linearize the compression function of CubeHash, i.e. $\mathsf{Compress}_{\mathsf{lin}}$, to find message differences that can be used for a collision attack as described in Chapter 5. Let $\mathsf{Compress}_{\mathsf{lin}}$ be the linearization of Compress obtained by replacing all modular additions in the round transformation, Algorithm 2.7, with XORs and setting $V = 0$. Using the canonical bases, $\mathsf{Compress}_{\mathsf{lin}}$ can be written as a matrix $\mathcal{H}$ of dimension $(1024 - 8b) \times 8bt$. Let τ be the dimension of its kernel. As noted in [34] and discussed in Chapter 5, the matrix $\mathcal{H}$ does not have full rank for many parameters r/b and t, and one can find differences with high a raw probability (imposing a small number of conditions) in the set of linear combinations of at most λ kernel basis vectors, where $\lambda \geqslant 1$ is chosen such that the set can be searched exhaustively.

The success of a second preimage or collision attack heavily depend on the choice of the kernel basis. Table 7.5 compares the minimal number of conditions

b/r	4	5	6	7	8	16
32	156	1244	400	1748	830	2150
64	130	205	351	447	637	1728
96	62	127	142	251	266	878
32	189	1952	700	2428	830	2150
64	189	1514	700	1864	637	1728
96	67	128	165	652	329	928

Table 7.5 Minimal number of conditions found for $\lambda = 3$ using two different algorithms to determine the kernel bases.

for $\lambda = 3$ for two different choices of the kernel basis. The results in the first three rows are obtained using the same algorithm as in [34] to determine the bases. The results in the last three rows are obtained using the more standard procedure implemented for example in the Number Theory Library of Shoup [105].

As discussed in Chapter 5, the inverse raw probability is an upper bound of the theoretical complexity, and as such, we expect that differences with a high raw probability have a low theoretic complexity. However, a higher raw probability does not always imply a lower theoretic complexity. There are differences with lower raw probability that lead to a lower theoretic complexity than that of a difference with a higher raw probability. Hence, when searching for minimal complexity of the collision attack, simply considering the number of conditions imposed by a difference is not sufficient. The examples in Table 7.5 serve to highlight our motivation for implementing the reverse and randomized differential trail search methods.

We note that the linearized round transformation of CubeHash is invertible and let $\mathsf{Compress}^{\mathsf{r}}_{\mathsf{lin}}$ be defined in the same way as $\mathsf{Compress}_{\mathsf{lin}}$ but with inverse

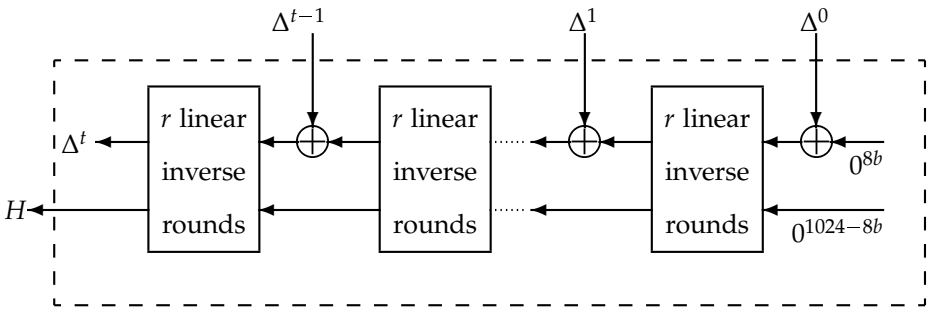

Figure 7.2 Computation of $\mathsf{Compress}^r_{\mathsf{lin}}$ on input $\Delta' = \Delta^0 \| \cdots \| \Delta^{t-1}$. If Δ' lies in the kernel, $H = 0$ and $\Delta = \Delta^t \| \cdots \| \Delta^1$ lies in the kernel of $\mathsf{Compress}_{\mathsf{lin}}$.

linearized round transformations. Suppose that $\Delta' = \Delta^0 \| \cdots \| \Delta^{t-1}$ lies in the kernel of $\mathsf{Compress}^r_{\mathsf{lin}}$ and Δ^t equals the (discarded) first $8b$ bits of the state after the tr linear inverse round transformations processing Δ' as shown in Fig. 7.2. Then, the difference $\Delta = \Delta^t \| \cdots \| \Delta^1$ lies in the kernel of $\mathsf{Compress}_{\mathsf{lin}}$ and Δ^0 is the corresponding erasing block difference. As for the linearized compression function we determine a basis of the kernel of $\mathsf{Compress}^r_{\mathsf{lin}}$ and exhaustively search for linear combinations of at most λ kernel basis vectors of high raw probability. Due to the diffusion of the inverse transformation, these trails tend to be dense at the beginning and sparse at the end.

b/r	4	5	6	7	8	16
32	156	1244	**394**	1748	830	2150
64	130	205	**309**	447	637	1728
96	**38**	127	**90**	251	**151**	**709**

Table 7.6 Minimal number of conditions y found with the randomized search. Values in **bold** improve over the values in Table 7.5.

Table 7.6 shows the best found raw probabilities after 200 trials of 600 iterations

using the randomized search algorithm detailed in Section 5.1.5 and previously presented in [67]. Estimating the corresponding theoretic complexities using the condition function described in Section 5.2 yields the improved collision attacks presented in Table 7.7.

b/r	4	5	6	7	8	16
32	–	–	180	–	–	–
64	–	–	132	–	–	–
96	7	–	51	–	80	–

Table 7.7 Logarithmic theoretical complexities c_Δ of improved collision attacks.

For CubeHash-r/b there is a generic collision attack with complexity of about 2^{512-4b}. For $b > 64$ this is faster than the generic birthday attack on hash functions with output length $h = 512$. For $b = 96$, specifically, the generic attack has a complexity of about 2^{128}. Our attacks clearly improve over this bound.

In addition to the aforementioned analysis, which we originally presented in [67], we also consider CubeHash variants with the number of rounds r and bytes per message, b, not previously considered in [34, 67]. Table 7.8 shows the number of iterations t and conditions y found using the algorithms of Chapter 5 for the new variants. The **bold** entries correspond successful theoretical second preimage attacks on CubeHash. For example, our trail for CubeHash-4/10 with 189 condition bits indicates a theoretical second preimage attack on the algorithm that can be carried out successfully using a single function evaluation, with probability of 2^{-189}. In other words, using a single function evaluation we can produce a second preimage with probability 2^{-189}. We give the differential trails for these attacks in Appendix F.2.

r/b	10	20	24	36	96
1	$(\mathbf{4},\mathbf{32})$	$(\mathbf{4},\mathbf{30})$	$(\mathbf{4},\mathbf{30})$	$(\mathbf{3},\mathbf{13})$	$(\mathbf{1},\mathbf{0})$
2	$(\mathbf{2},\mathbf{32})$	$(\mathbf{2},\mathbf{30})$	$(\mathbf{2},\mathbf{30})$	$(\mathbf{2},\mathbf{28})$	$(\mathbf{1},\mathbf{18})$
3	$(\mathbf{4},\mathbf{478})$	$(\mathbf{4},\mathbf{394})$	$(\mathbf{4},\mathbf{394})$	$(\mathbf{3},\mathbf{343})$	$(\mathbf{1},\mathbf{54})$
4	$(\mathbf{2},\mathbf{189})$	$(\mathbf{2},\mathbf{156})$	$(\mathbf{2},\mathbf{156})$	$(\mathbf{2},\mathbf{130})$	$(\mathbf{1},\mathbf{40})$
5	$(4,1517)$	$(4,1244)$	$(4,1244)$	$(4,965)$	$(\mathbf{1},\mathbf{127})$
6	$(\mathbf{2},\mathbf{478})$	$(\mathbf{2},\mathbf{394})$	$(\mathbf{2},\mathbf{394})$	$(\mathbf{2},\mathbf{309})$	$(\mathbf{1},\mathbf{93})$
7	$(4,2124)$	$(4,1748)$	$(4,1748)$	$(4,1418)$	$(\mathbf{1},\mathbf{251})$
8	$(2,1009)$	$(2,830)$	$(2,830)$	$(2,637)$	$(\mathbf{1},\mathbf{191})$
9	$(2,2025)$	$(2,1788)$	$(2,1788)$	$(2,1713)$	$(\mathbf{1},\mathbf{428})$
10	$(2,1517)$	$(2,1244)$	$(2,1244)$	$(2,965)$	$(\mathbf{1},\mathbf{360})$

Table 7.8 Cubehash (t, y)

As with the previous cases, we use the condition function approach, describe in Section 5.2, to analyse the algorithm's susceptability to collision attacks. Tables 7.9, 7.10, 7.11, and 7.12 show the reduced time complexities of collision attacks corresponding to the trails given in Table 7.8, using a 1-, 2-, 4- and 8-bit message modification level, respectively. Considering a 512-bit output length, the **bold** entries in the table highlight the successful theoretical collision attacks, with complexities 2^{c_Δ} less than the birthday bound, i.e., $2^{c_\Delta} < 2^{256}$.

We note these attacks are new and complement the previous results of [34,67]. Additionally, this is also the first implementation considering sub-byte message modification techniques. Although [34] suggests that using a 1-, 2-, or 4-bit message modification technique, i.e., using bit-level dependency table, is likely to reduce the complexity of a collision attack our results suggest a minimal improve-

r/b	10	20	24	36	96
1	**6.05**	**5.93**	**5.93**	**4.75**	–
2	**6.14**	**6.08**	**6.04**	**5.83**	**5.17**
3	263.9	**100.4**	**87.8**	**65.1**	**6.90**
4	**101.9**	**71.8**	**62.5**	**45.9**	**6.47**
5	1244	801.8	738.4	338.5	**82.2**
6	364.4	**205.5**	**192.6**	**146.6**	**57.6**
7	1853	1233	1135	1233	**211.6**
8	953.5	630.7	589.5	340.1	**150.2**
9	1884	1496	1459	1311	404.3
10	1404	1053	1024	643.3	323.0

Table 7.9 CubeHash logarithmic collision complexities c_Δ with 1-bit modifications.

r/b	10	20	24	36	96
1	**6.05**	**5.93**	**5.93**	**4.75**	–
2	**6.21**	**6.13**	**6.01**	**5.83**	**5.17**
3	262.3	**95.5**	**80.6**	**35.1**	**6.94**
4	**99.8**	**61.5**	**51.1**	**37.1**	**6.61**
5	1242	801.6	735.7	331.3	**70.1**
6	361.3	**198.5**	**181.6**	**142.2**	**54.5**
7	1851	1228	1130	1228	**197.4**
8	949.4	626.8	592.5	336.0	**144.7**
9	1881	1491	1450	1290	374.2
10	1401	1046	1011	631.6	315.7

Table 7.10 CubeHash logarithmic collision complexities c_Δ with 2-bit modifications.

r/b	10	20	24	36	96
1	**6.17**	**6.02**	**6.02**	**4.76**	–
2	**6.29**	**6.30**	**6.04**	**5.83**	**5.17**
3	262.1	**87.9**	**67.0**	**28.3**	**7.11**
4	**97.7**	**62.6**	**54.3**	**35.7**	**7.14**
5	1241	795.9	732.4	324.7	**66.5**
6	357.0	**195.6**	**178.5**	**139.7**	**56.0**
7	1850	1226	1130	1226	**192.9**
8	951.1	624.1	589.7	327.6	**142.4**
9	1876	1486	1443	1268	365.0
10	1398	1042	1009	629.5	311.4

Table 7.11 CubeHash logarithmic collision complexities c_Δ with 4-bit modifications.

r/b	10	20	24	36	96
1	**6.21**	**6.15**	**6.02**	**4.75**	–
2	**6.61**	**6.51**	**6.30**	**5.90**	**5.17**
3	260.8	**82.3**	**66.9**	**67.1**	**7.98**
4	**98.2**	**60.3**	**55.7**	**39.1**	**8.34**
5	1239	798.5	732.7	322.3	**70.6**
6	357.1	**196.8**	**185.1**	**141.6**	**60.3**
7	1845	1229	1131	1229	**194.5**
8	949.6	629.8	593.2	330.4	**150.5**
9	1875	1488	1446	1260	366.0
10	1397	1042	1008	635.8	319.5

Table 7.12 CubeHash logarithmic collision complexities c_Δ with 8-bit modifications.

ment, in many cases none. Moreover, we find that the sub-byte implementations are considerably slower often are less efficient in finding actual collisions. We evaluated the latter point by searching for collisions in the variants with the number of rounds $r < 3$.

For completeness we illustrates the significance of our proposed reverse differential trail search algorithm by presenting a collision for CubeHash-5/96, the trail complexity of which was presented in Table 7.5.

We consider two linear differences found using the forward and reverse trail the methods of Section 5.1.3 and 5.1.4 respectively. Both consist of two 96-byte blocks, a first block that lies in the kernel of the linearized compression function and a second one that is the corresponding erasing block difference. They are given by

$$\begin{aligned}
\Delta_a^0 = \ & \texttt{40000000 00000000 40000000 00000000 00000000 00000000} \\
& \texttt{00000000 00000000 00200000 00000000 00000000 00000000} \\
& \texttt{00000000 00000000 00000000 00000000 00000000 00000040} \\
& \texttt{00000000 00000040 00000000 00020000 00000000 00000000}, \\
\Delta_a^1 = \ & \texttt{01000111 01000111 00000000 00000000 8008002A 00000000} \\
& \texttt{08000022 00000000 00000000 00000000 00000000 00000000} \\
& \texttt{00000000 00000000 11040000 00000000 40000101 01000111} \\
& \texttt{00000000 00000000 00002208 00000000 08002000 00000000}
\end{aligned}$$

and

$$\begin{aligned}
\Delta_b^0 = \ & \texttt{08000208 08000208 00000000 00000000 40000100 00000000} \\
& \texttt{00400110 00000000 00000000 00000000 00000000 00000000} \\
& \texttt{00000000 00000000 0800A000 00000000 08000888 08000208} \\
& \texttt{00000000 00000000 40011000 00000000 00451040 00000000,} \\
\Delta_b^1 = \ & \texttt{80000000 00000000 80000000 00000000 00000000 00000000} \\
& \texttt{00000000 00000000 00400000 00000000 00000000 00000000} \\
& \texttt{00000000 00000000 00000000 00000000 00000000 00000080} \\
& \texttt{00000000 00000080 00000000 00040000 00000000 00000000.}
\end{aligned}$$

The number of conditions imposed by Δ_a and Δ_b are 127 and 134, respectively. Despite its lower raw probability, Δ_a has a theoretical complexity of $2^{31.9}$ compression function calls, which is much less than $2^{68.7}$ for Δ_b. As discussed above, this is due to the different structure of $\boldsymbol{\alpha} \vee \boldsymbol{\beta}$. We recall that y denotes the Hamming weight of $\boldsymbol{\alpha} \vee \boldsymbol{\beta}$ and let

$$y_i = \sum_{k=32i}^{32(i+1)} \mathrm{wt}(\alpha^k \vee \beta^k).$$

That is, y_i is the number of conditions imposed by a difference at round i. Table 7.13 compares the values y_i for Δ_a and Δ_b. The conditions imposed in the first two rounds can easily be satisfied by appropriate message modifications, and thus, do not significantly increase the complexity of the attack — contrary to the conditions imposed in the last two rounds.

Due to its low theoretical complexity, we can use Δ_b to find a collision and to confirm empirically the estimated theoretical complexity. As mentioned before, we can construct a collision for the hash function out of a collision for the compression function. Using a dependency table at byte-level, we obtained a partition

	y_1	y_2	y_3	y_4	y_5	y	$\log_2(c_\Delta)$
a	14	17	23	30	43	127	68.7
b	44	36	25	17	12	134	31.9

Table 7.13 Number of conditions per round theoretical complexities.

of the condition function attributed to Δ_b^0 using Algorithm 5.2 (see Table 7.14). Then, using the tree-based backtracking algorithm given in Algorithm 5.3, we found several collisions after $2^{22.41}$ to $2^{32.25}$ condition function calls. One of them, found after $2^{29.1}$ condition function calls, is given by

$$
\begin{aligned}
M^{\text{pre}} = {} & \texttt{F06BB068 487C5FE1 CCCABA70 0A989262 801EDC3A 69292196} \\
& \texttt{8848F445 B8608777 C037795A 10D5D799 FD16C037 A52D0B51} \\
& \texttt{63A74C97 FD858EEF 7809480F 43EB264C D6631863 2A8CCFE2} \\
& \texttt{EA22B139 D99E4888 8CA844FB ECCE3295 150CA98E B16B0B92,} \\
M^0 = {} & \texttt{3DB4D4EE 02958F57 8EFF307A 5BE9975B 4D0A669E E6025663} \\
& \texttt{8DDB6421 BAD8F1E4 384FE128 4EBB7E2A 72E16587 1E44C51B} \\
& \texttt{DA607FD9 1DDAD41F 4180297A 1607F902 2463D259 2B73F829} \\
& \texttt{C79E766D 0F672ECC 084E841B FC700F05 3095E865 8EEB85D5.}
\end{aligned}
$$

For $M^1 = 0$, the messages $M^{\text{pre}} \| M^0 \| M^1$ and $M^{\text{pre}} \| M^0 \oplus \Delta_b^0 \| M^1 \oplus \Delta_b^1$ collide to the

same digest

$$H = \begin{array}{l} \texttt{C2E51517 C503746E 46ECD6AD 5936EC9B} \\ \texttt{FF9B74F9 2CEA4506 624F2B0B FE584D2C} \\ \texttt{56CD3E0E 18853BA8 4A9D6D38 F1F8E45F} \\ \texttt{2129C678 CB3636D4 D865DE13 410E966C} \end{array}$$

for CubeHash-5/96. Instead of $M^1 = 0$, any other M^1 can be chosen and, moreover, the colliding messages can be extended by an arbitrary message suffix M^{suff}.

We note, as in our earlier work [67], that our method of backward computation lead to the first practical collision attack on CubeHash-5/96. Additionally, the randomized search yielded new highly probable differential trails which lead to improved collision attacks for up to eight rounds. However, it is important to also note that our analysis did not lead to an attack on the official CubeHash-16/32.

i	$\mathcal{M}_i$	$\mathcal{Y}_i$	q_i
0	$\varnothing$	$\varnothing$	0.00
1	$\{2, 6, 66\}$	$\{1, 2\}$	2.00
2	$\{10, 1, 9, 14, 74, 5, 13, 65, 17, 70\}$	$\{5\}$	1.35
3	$\{73, 7, 16, 19, 18, 78, 25, 37, 41\}$	$\{23, 24\}$	2.00
4	$\{69, 77, 24, 33\}$	$\{21, 22\}$	2.00
5	$\{50, 89\}$	$\{12, 13\}$	2.00
6	$\{20, 27, 45, 88\}$	$\{11\}$	1.15
7	$\{57, 4\}$	$\{38\}$	1.00
8	$\{80\}$	$\{7, 8\}$	2.00
9	$\{38, 40, 81, 3, 28, 32\}$	$\{34\}$	1.24
10	$\{49\}$	$\{41\}$	1.00
11	$\{58\}$	$\{19, 20, 42, 43\}$	4.00
12	$\{91\}$	$\{16, 17\}$	2.00
13	$\{23, 34, 44, 83\}$	$\{29, 30\}$	2.07
14	$\{90\}$	$\{14\}$	1.07
15	$\{15, 26\}$	$\{15\}$	1.07
16	$\{36\}$	$\{37, 55\}$	2.31
17	$\{42, 46, 48\}$	$\{25, 26\}$	2.12
18	$\{56\}$	$\{18, 31, 40\}$	3.01
19	$\{59\}$	$\{48, 79\}$	2.00
20	$\{84, 92, 0\}$	$\{35\}$	1.00
21	$\{82\}$	$\{9, 10, 27, 28, 32, 33\}$	6.04
22	$\{31, 51\}$	$\{44, 56, 64\}$	3.03
23	$\{71\}$	$\{6\}$	1.00
24	$\{11, 54, 67\}$	$\{3\}$	1.00
25	$\{75\}$	$\{78\}$	1.00
26	$\{21, 55\}$	$\{46, 59\}$	2.00
27	$\{63\}$	$\{50\}$	1.00
28	$\{79\}$	$\{45, 49, 65, 70\}$	4.00
29	$\{12\}$	$\{71\}$	1.06
30	$\{22\}$	$\{58, 67, 81, 82, 83\}$	5.00
31	$\{29, 62\}$	$\{63\}$	1.03
32	$\{87, 95\}$	$\{53, 54, 74, 76, 85\}$	5.01
33	$\{39, 47\}$	$\{39\}$	1.01
34	$\{53, 8\}$	$\{69, 88, 89\}$	3.30
35	$\{30\}$	$\{77, 86, 94, 98\}$	5.04
36	$\{60, 61\}$	$\{62, 91, 101, 102\}$	4.35
37	$\{35, 52\}$	$\{61, 90, 103\}$	4.22
38	$\{43\}$	$\{36, 57, 60, 104, 111\}$	5.77
39	$\{64\}$	$\{0\}$	1.33
40	$\{68\}$	$\{4\}$	2.03
41	$\{72\}$	$\{97, 100, 121\}$	8.79
42	$\{76\}$	$\{66, 80, 92, 93\}$	13.39
43	$\{85\}$	$\{47, 112\}$	16.92
44	$\{93\}$	$\{51, 52, 68, 72, 75, 87, 95\}$	22.91
45	$\{86, 94\}$	$\{73, 84, 96, 99, 105, \ldots, 110, 113, \ldots, 132, 133\}$	31.87

Table 7.14 Partition sets corresponding to the trail Δ_b for CubeHash-5/96. Numbers in $\mathcal{M}_i$ are byte indices, whereas numbers in $\mathcal{Y}_i$ are bit indices.

Chapter 8

Conclusion

Efficiency of stream cipher and hash function algorithms is a very important design criterion, almost parallel with security. This work presents a generic framework for analyzing and evaluating the performance of such algorithms; specifically, we estimate the performance of the eSTREAM ciphers, and second-round SHA-3 candidates in the ongoing competition to establish a new cryptographic hash standard, SHA-3. Using this framework as a base, we then take advantage of platform-specific optimization techniques to provide more precise performance estimates for NVIDIA Graphics Processing Units. We further support our analysis by presenting multi-stream implementation results of all but one of the eSTREAM ciphers and all non-AES based SHA-3 candidates.

Simultaneously, we present the first (to our knowledge) open source multi-GPU implementation of the cube attack. The framework can be used to analyze arbitrary black box polynomials, while also providing speedup factors an order of magnitude greater than corresponding CPU implementations. To verify the soundness of our implementation we analyze the Trivium stream cipher, confirming previously found results, in addition to several new equations for finding key

bits of the round reduced variant. We further analyze the MICKEY v2 stream cipher, the results of which lead us to conclude that the cipher is not susceptible to such algebraic attacks, given our limitation to cubes of dimensions up to 20. However, we note that the main motivation behind the framework implementation is to provide for a means for third parties to verify cryptanalysis results, a tool we believe the cryptanalysis community has been lacking.

Finally, we present two methods for finding improved linear differential trails for CubeHash and extend the linear differential cryptanalysis framework of Brier *et al.* to allow for parallel trail searches, finer-grained dependency tables and a more generic interface. Our method of backward computation lead to the first practical collision attack on CubeHash-5/96, while the randomized search yielded new highly probable differential trails applicable to finding improved collision attacks for up to eight rounds. Furthermore, our extended, generalized framework implementation was used to analyze toy versions of the BLAKE hash function family and several new CubeHash variants not previously considered. Though our analysis did not lead to an attack on the official CubeHash-16/32 or BLAKE hash functions, to our knowledge, our results show the best attacks on simplified versions of these two hash functions. We believe that these results will motivate further analysis which we hope will reach the goal of understanding the strengths and weaknesses of the full algorithms.

Appendix A

BLAKE Constants

For completeness we give the initial values h_i^0's, permutation functions $\sigma_r(i)$'s, and constants c_i's of the BLAKE family, as detailed in [8]; these are shown in Table A.1, Table A.2, and Table A.3, respectively. Literals starting with `0x` are in hexadecimal form.

	BLAKE-28	**BLAKE-**32	**BLAKE-**48	**BLAKE-**64
h_0^0	0xC1059ED8	0x6A09E667	0xCBBB9D5DC1059ED8	0x6A09E667F3BCC908
h_1^0	0x367CD507	0xBB67AE85	0x629A292A367CD507	0xBB67AE8584CAA73B
h_2^0	0x3070DD17	0x3C6EF372	0x9159015A3070DD17	0x3C6EF372FE94F82B
h_3^0	0xF70E5939	0xA54FF53A	0x152FECD8F70E5939	0xA54FF53A5F1D36F1
h_4^0	0xFFC00B31	0x510E527F	0x67332667FFC00B31	0x510E527FADE682D1
h_5^0	0x68581511	0x9B05688C	0x8EB44A8768581511	0x9B05688C2B3E6C1F
h_6^0	0x64F98FA7	0x1F83D9AB	0xDB0C2E0D64F98FA7	0x1F83D9ABFB41BD6B
h_7^0	0xBEFA4FA4	0x5BE0CD19	0x47B5481DBEFA4FA4	0x5BE0CD19137E2179

Table A.1 BLAKE intial values

Note that since there are only 10 permuation functions, for BLAKE-$\{48, 64\}$,

the permutation function used for rounds 9 and above are simply $\sigma_{(r \mod 10)}(i)$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma_0(i)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma_1(i)$	14	10	4	8	9	15	13	6	1	12	0	2	11	7	5	3
$\sigma_2(i)$	11	8	12	0	5	2	15	13	10	14	3	6	7	1	9	4
$\sigma_3(i)$	7	9	3	1	13	12	11	14	2	6	5	10	4	0	15	8
$\sigma_4(i)$	9	0	5	7	2	4	10	15	14	1	11	12	6	8	3	13
$\sigma_5(i)$	2	12	6	10	0	11	8	3	4	13	7	5	15	14	1	9
$\sigma_6(i)$	12	5	1	15	14	13	4	10	0	7	6	3	9	2	8	11
$\sigma_7(i)$	13	11	7	14	12	1	3	9	5	0	15	4	8	6	2	10
$\sigma_8(i)$	6	15	14	9	11	3	0	8	12	2	13	7	1	4	10	5
$\sigma_9(i)$	10	2	8	4	7	6	1	5	15	11	9	14	3	12	13	0

Table A.2 BLAKE permutation function

	BLAKE-$\{28, 32\}$	**BLAKE-**$\{48, 64\}$
c_0	0x243F6A88	0x243F6A8885A308D3
c_1	0x85A308D3	0x13198A2E03707344
c_2	0x13198A2E	0xA4093822299F31D0
c_3	0x03707344	0x082EFA98EC4E6C89
c_4	0xA4093822	0x452821E638D01377
c_5	0x299F31D0	0xBE5466CF34E90C6C
c_6	0x082EFA98	0xC0AC29B7C97C50DD
c_7	0xEC4E6C89	0x3F84D5B5B5470917
c_8	0x452821E6	0x9216D5D98979FB1B
c_9	0x38D01377	0xD1310BA698DFB5AC
c_{10}	0xBE5466CF	0x2FFD72DBD01ADFB7
c_{11}	0x34E90C6C	0xB8E1AFED6A267E96
c_{12}	0xC0AC29B7	0xBA7C9045F12C7F99
c_{13}	0xC97C50DD	0x24A19947B3916CF7
c_{14}	0x3F84D5B5	0x0801F2E2858EFC16
c_{15}	0xB5470917	0x636920D871574E69

Table A.3 BLAKE constant values

Appendix B

MICKEY v2 Constants

The tap vector T for register r is defined below:

$$\begin{aligned} T = (&1,1,0,1,1,1,1,0,0,1,0,0,1,1,0,0,1,0,0,1,1,1,1,0,0,\\ &1,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0,0,1,1,0,0,1,1,0,0,0,\\ &1,0,1,0,1,0,1,0,1,0,1,1,0,1,1,1,1,1,0,0,0,1,1,0,0,\\ &0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,1,1,0,1,1,1,1,0,0). \end{aligned}$$

Similarly, the bit vectors used by $\texttt{Clock}_s$ to update the s register are:

$$\begin{aligned} C_0 = (&0,0,0,0,1,1,0,0,0,1,0,1,1,1,1,0,1,0,0,1,0,1,0,1,0,\\ &1,0,1,0,1,1,0,1,0,0,1,0,0,0,0,0,0,0,1,0,1,0,1,0,1,\\ &0,0,0,0,1,0,1,0,0,1,1,1,1,0,0,1,0,1,0,1,1,1,1,1,1,\\ &1,1,1,0,1,0,1,1,1,1,1,1,0,1,0,1,0,0,0,0,0,0,1,1,0), \end{aligned}$$

$$C_1 = (0,1,0,1,1,0,0,1,0,1,1,1,1,0,0,1,0,1,0,0,0,1,1,0,1,$$
$$0,1,1,1,0,1,1,1,1,0,0,0,1,1,0,1,0,1,1,1,0,0,0,0,1,$$
$$0,0,0,1,0,1,1,1,0,0,0,1,1,1,1,1,1,0,1,0,1,1,1,0,1,$$
$$1,1,1,0,0,0,1,0,0,0,0,1,1,1,0,0,0,1,0,0,1,1,0,0,0),$$

$$F_0 = (1,1,1,1,0,1,0,1,1,1,1,1,1,1,1,0,0,1,0,1,1,1,1,1,1,$$
$$1,1,1,1,0,0,1,1,0,0,0,0,0,0,1,1,1,0,0,1,0,0,1,0,1,$$
$$0,1,0,0,1,0,1,1,1,1,0,1,0,1,0,1,0,0,0,0,0,0,0,0,0,$$
$$1,1,0,1,0,0,0,1,1,0,1,1,1,0,0,1,1,1,0,0,1,1,0,0,0),$$

$$F_1 = (1,1,1,0,1,1,1,0,0,0,0,1,1,1,0,1,0,0,1,1,0,0,0,1,0,$$
$$0,1,1,0,0,1,0,1,1,0,0,0,1,1,0,0,0,0,0,1,1,0,1,1,0,$$
$$0,0,1,0,0,0,1,0,0,1,0,0,1,0,1,1,0,1,0,1,0,0,1,0,1,$$
$$0,0,0,1,1,1,1,0,1,1,1,1,1,0,0,0,0,0,0,1,0,0,0,0,1).$$

Appendix C

Software Implementation of Grain

In Chapter 6 we presented general-purpose CPU benchmarking results for a 32-bit software implementation of, the Grain stream cipher. As mentioned, unlike most of the eSTREAM cipher submissions the Grain submission was not optimized for software. Simply basing the GPU code on the submission code would have resulted in a wholly inefficient implementation, and, thus we first optimized Grain (following the ECRYPT API) for software. Using the original submitted source code[1], we only modified the C functions implementing the keystream generator and IV setup.

The modifications to the original Grain submission are four-fold:

1. Below, we provide the modified data structure containing the internal state of the cipher, defined in the standard `ecrypt-sync.h` header file (the reader unfamiliar with the ECRYPT eSTREAM API is referred to the original Grain submission source code).

```
1 typedef struct {
2     u32 s[3];
3     u32 b[3];
```

[1] Available at http://www.ecrypt.eu.org/stream/grainpf.html

```
    const u8* p_key;
    u32 keysize;
    u32 ivsize;
} ECRYPT_ctx;
```

2. The tables used in the original Grain implementation, as defined in `grain.h`, were removed as deemed unnecessary.

3. The new keystream generator and supporting macros are given below. We note that the keystream generator is a direct implementation of Grain and refer to [59,60] for a description of the cipher.

```
#define S0 (ctx->s[0])
#define S1 (ctx->s[1])
#define S2 (ctx->s[2])

#define B0 (ctx->b[0])
#define B1 (ctx->b[1])
#define B2 (ctx->b[2])

/* Helper macros for retrieving specific bits of a shift register */
#define get64(S) (S##2)
#define get63(S) (S##1>>31)
#define get62(S) (S##1>>30)
#define get60(S) (S##1>>28)
#define get56(S) (S##1>>24)
#define get52(S) (S##1>>20)
#define get51(S) (S##1>>19)
#define get46(S) (S##1>>14)
#define get45(S) (S##1>>13)
#define get43(S) (S##1>>11)
#define get38(S) (S##1>> 6)
#define get37(S) (S##1>> 5)
#define get33(S) (S##1>> 1)
#define get31(S) (S##0>>31)
#define get28(S) (S##0>>28)
#define get25(S) (S##0>>25)
#define get23(S) (S##0>>23)
#define get21(S) (S##0>>21)
#define get15(S) (S##0>>15)
#define get14(S) (S##0>>14)
#define get13(S) (S##0>>13)
#define get10(S) (S##0>>10)
```

```
32 #define get9(S) (S##0>> 9)
33 #define get4(S) (S##0>> 4)
34 #define get3(S) (S##0>> 3)
35 #define get2(S) (S##0>> 2)
36 #define get1(S) (S##0>> 1)
37 #define get0(S) (S##0)
38
39 /* Helper macros for setting specific bits of a shift register */
40 #define set79(S,bit) (S##2=(S##2&(~(1<<15)))|((bit&1)<<15))
41
42 /* Helpter macro for XORing bits into the shift register */
43 #define xor79(S,bit) (S##2^=((bit&1)<<15))
44
45
46 /* The nonlinear function as defined in the article */
47 #define h(x0,x1,x2,x3,x4)\
48     ((x1)^(x4)^((x0)&(x3))^((x2)&(x3))^((x3)&(x4))^((x0)&(x1)&(x2))\
49     ^((x0)&(x2)&(x3))^((x0)&(x2)&(x4))^((x1)&(x2)&(x4))^((x2)&(x3)&(x4)))
50
51 /* Helper macro for shifting 3 32-bit registers */
52 #define SHIFT_FSR(S)\
53         do {\
54             S##0=(S##0>>1)|(((S##1)&1)<<31);\
55             S##1=(S##1>>1)|(((S##2)&1)<<31);\
56             S##2=(S##2>>1);\
57         } while(0)
58
59
60 /*
61  * Function: grain_keystream
62  *
63  * Synopsis
64  * Generates a new bit and updates the internal state of the cipher.
65  */
66 u8 grain_keystream(ECRYPT_ctx* ctx) {
67     u32 x0 = get3(S),
68         x1 = get25(S),
69         x2 = get46(S),
70         x3 = get64(S),
71         x4 = get63(B);
72
73     /* Compute output bit */
74     u32 Z = get1(B) ^ get2(B) ^ get4(B) ^ get10(B) ^ get31(B) ^ get43(B) ^ get56(B)
75               ^ h(x0,x1,x2,x3,x4);
76
```

```
77      /* Compute linear feedback bit */
78      u32 S80 = get62(S) ^ get51(S) ^ get38(S) ^ get23(S) ^ get13(S) ^ get0(S);
79
80      /* Compute nonlinear feedback bit */
81  #if !defined(COMBINE_TERMS)
82
83      u32 B80 = (get0(S)) ^ (get62(B)) ^ (get60(B)) ^ (get52(B)) ^ (get45(B))
84                ^ (get37(B)) ^ (get33(B)) ^ (get28(B)) ^ (get21(B))
85                ^ (get14(B)) ^ (get9(B)) ^ (get0(B)) ^ (get63(B)&get60(B))
86                ^ (get37(B)&get33(B)) ^ (get15(B)&get9(B))
87                ^ (get60(B)&get52(B)&get45(B)) ^ (get33(B)&get28(B)&get21(B))
88                ^ (get63(B)&get45(B)&get28(B)&get9(B))
89                ^ (get60(B)&get52(B)&get37(B)&get33(B))
90                ^ (get63(B)&get60(B)&get21(B)&get15(B))
91                ^ (get63(B)&get60(B)&get52(B)&get45(B)&get37(B))
92                ^ (get33(B)&get28(B)&get21(B)&get15(B)&get9(B))
93                ^ (get52(B)&get45(B)&get37(B)&get33(B)&get28(B)&get21(B));
94
95  #else
96      /* Some terms are used multiple times, 'cache' them.
97          The number in the comments of the following lines denotes the
98          number of occurrences in computing s_{i+80}.
99      */
100
101     u32 B33_28_21 = (get33(B)&get28(B)&get21(B)); /* 3 */
102     u32 B52_45_37 = (get52(B)&get45(B)&get37(B)); /* 2 */
103     u32 B52_37_33 = (get52(B)&get37(B)&get33(B)); /* 2 */
104     u32 B60_52_45 = (get60(B)&get52(B)&get45(B)); /* 2 */
105     u32 B63_60 = (get63(B)&get60(B)); /* 3 */
106     u32 B37_33 = (get37(B)&get33(B)); /* 3 */
107     u32 B45_28 = (get45(B)&get28(B)); /* 2 */
108     u32 B15_9 = (get15(B)&get9(B)); /* 2 */
109     u32 B21_15 = (get21(B)&get15(B)); /* 2 */
110
111     u32 B80 = (get0(S)) ^ (get62(B)) ^ (get60(B)) ^ (get52(B)) ^ (get45(B))
112               ^ (get37(B)) ^ (get33(B)) ^ (get28(B)) ^ (get21(B))
113               ^ (get14(B)) ^ (get9(B)) ^ (get0(B)) ^ (B63_60) ^ (B37_33) ^ (B15_9)
114               ^ (B60_52_45) ^ (B33_28_21) ^ (get63(B)&B45_28&get9(B))
115               ^ (get60(B)&B52_37_33) ^ (B63_60&B21_15)
116               ^ (B63_60&B52_45_37) ^ (B33_28_21&B15_9)
117               ^ (B52_45_37&B33_28_21);
118 #endif
119
120     /* Shift registers */
121     SHIFT_FSR(S);
```

```
    SHIFT_FSR(B);
    /* Feedback bit */
    set79(S,S80);
    set79(B,B80);

    return Z&1;
}
```

4. The modified IV setup function is given below:

```
void ECRYPT_ivsetup(
  ECRYPT_ctx* ctx,
  const u8* iv)
{
    u32 outbit;
    int i;
    u8 *b=(u8*)ctx->b;
    u8 *s=(u8*)ctx->s;

    /* Copy key to context data structure */
    for(i=0;i<10;i++)
        b[i]=ctx->p_key[i];

    /* Copy IV to context data structure */
    for(i=0;i<ctx->ivsize/8;i++)
        s[i]=iv[i];

    /* Set remaining bits to 1, as describe in paper */
    for(i=ctx->ivsize/8;i<10;i++)
        s[i]=0xff;

    /* Do initial clockings */
    for (i=0;i<INITCLOCKS;++i) {
        outbit=grain_keystream(ctx);
        xor79(S,outbit);
        xor79(B,outbit);
    }
}
```

We note that our implementation is not highly optimized and taking advantage of SIMD instructions, 64-bit instructions, or inline assembly would provide for additional speedup. Regardless, this implementation is 6.6 times faster than the reference implementation.

Appendix D

gSTREAM API and Implementations

D.1 gSTREAM API

The header file below provides the API for the gSTREAM framework. The API is very similar to the eSTREAM API, allowing for easier porting of already implemented ciphers that conform to the aforementioned standard. Additionally, the cu file provides cipher-independent code which may be used as a base for implementing the portfolio.

gSTREAM.h:

```
1  #ifndef __GSTREAM_H_
2  #define __GSTREAM_H_
3
4  #define DEBUG
5  #include <stdint.h>
6
7  #define debug(...) \
8      fprintf(stderr, __VA_ARGS__)
9  #else
10 #define debug(...) ;
11 #endif
12
```

```
#define CH_ENDIANESS32(a) \
    (((a)>>24) | (((a)>>8)&0x0000FF00) | (((a)<<8) & 0x00FF0000) | ((a)<<24))


typedef uint8_t u8;
typedef uint16_t u16;
typedef uint32_t u32;
typedef uint64_t u64;

typedef enum { ENCRYPT=0, DECRYPT=1, GEN_KEYSTREAM=2 } gSTREAM_action;


typedef struct {
    int nr_threads; /* per block */
    int nr_blocks;

    u32 *keys_d;
    u32 key_size; /* in bits */
    int allocated_keys;

    u32 *ivs_d;
    u32 iv_size; /* in bits */
    int allocated_ivs;

    u32 *buff_d, *buff_h;
    u32 buff_size; /* in bytes (ceil to nearest 4-bytes) */
    int allocated_buff;

    struct { /* expandable benchmarking struct */
        unsigned timer;
    } bench;

    /* Insert cipher-dependent fields here: */

} gSTREAM_ctx;

/* Initialize device and allocate any state-related buffers.
    device - which device to use,
    nr_threads - number of threads/block,
    nr_blocks - number of blocks/grid
*/
void gSTREAM_init(gSTREAM_ctx* ctx, int device, int nr_threads, int nr_blocks);

/* Do the key setup.
    keys - all the stream keys: key[i][] corresponds to the i-th streams's key,
```

```
58      keysize − size of key in bits,
59      ivsize − size of iv in bits
60  */
61  void gSTREAM_keysetup(gSTREAM_ctx* ctx, u8* keys, u32 keysize, u32 ivsize);
62
63  /* Do the iv setup.
64      ivs − all the stream ivs: iv[i][] corresponds to the i−th streams's iv,
65  */
66  void gSTREAM_ivsetup(gSTREAM_ctx* ctx, u8* ivs);
67
68  /*
69      inputs − all the stream inputs:
70                  input[i][] corresponds to the i−th streams's input,
71      outputs − all the stream outputs:
72                  output[i][] corresponds to the i−th streams's output,
73      length − input/output length in bytes
74   */
75  void gSTREAM_process_bytes(gSTREAM_action action, gSTREAM_ctx* ctx,
76                             u8* inputs, u8* outputs, u32 length);
77
78  /* Generate keystream bytes.
79      keystreams[i] = keystream i
80      length − keystream length in bytes
81   */
82  void gSTREAM_keystream_bytes(gSTREAM_ctx* ctx, u8* keystreams, u32 length);
83
84  /* Free any allocated buffers and destroy context. */
85  void gSTREAM_exit(gSTREAM_ctx* ctx);
86
87  /* Get the measured time elapsed during keystream generation. */
88  double gSTREAM_getTimerValue(gSTREAM_ctx* ctx);
89
90  #endif
```

gSTREAM.cu:

```
1  #include <cutil_inline.h>
2  #include <cuda_runtime_api.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include <errno.h>
6  #include "gSTREAM.h"
7
8  /* include cipher kernel function cu file */
9
```

```
10
11 void gSTREAM_init(gSTREAM_ctx* ctx, int device, int nr_threads, int nr_blocks){
12
13     cudaDeviceProp deviceProp;
14     int nr_streams=nr_threads*nr_blocks;
15
16     /* set device */
17     cudaGetDeviceProperties(&deviceProp, device);
18     cudaSetDevice(device);
19     debug("\nUsing device %d: \"%s\"\n", device, deviceProp.name);
20
21     cutilSafeCall(cudaSetDeviceFlags(cudaDeviceMapHost));
22
23     ctx->nr_threads = nr_threads;
24     ctx->nr_blocks = nr_blocks;
25     ctx->allocated_keys=0;
26     ctx->allocated_ivs=0;
27     ctx->allocated_buff=0;
28
29     cutilCheckError(cutCreateTimer(&(ctx->bench.timer)));
30
31     /* allocate cipher state */
32
33 }
34
35 void gSTREAM_exit(gSTREAM_ctx* ctx) {
36
37     if(ctx->allocated_keys) {
38         cutilSafeCall(cudaFree(ctx->keys_d));
39     }
40
41     if(ctx->allocated_ivs) {
42         cutilSafeCall(cudaFree(ctx->ivs_d));
43     }
44
45     if(ctx->allocated_buff) {
46         cutilSafeCall(cudaFreeHost(ctx->buff_h));
47     }
48
49     cutilCheckError(cutDeleteTimer(ctx->bench.timer));
50
51     /* free cipher state */
52 }
53
54 void gSTREAM_keysetup(gSTREAM_ctx* ctx, u8* keys, u32 keysize, u32 ivsize) {
```

```
55
56  size_t keys_size;
57  int nr_streams=ctx->nr_threads*ctx->nr_blocks;
58  u32* keys_h=NULL;
59  size_t key_size_bytes=sizeof(u8)*(((keysize-1)/(sizeof(u8)*8))+1);
60  size_t key_size_nrwords=(((keysize-1)/(sizeof(u32)*8))+1);
61
62  ctx->key_size=keysize;
63  ctx->iv_size=ivsize;
64
65  /* allocate keys */
66  keys_size=nr_streams*sizeof(u32)*(((keysize-1)/(sizeof(u32)*8))+1);
67  cutilSafeCall(cudaMalloc((void**)&(ctx->keys_d),keys_size));
68  ctx->allocated_keys=1;
69  if(!(keys_h=(u32*)malloc(keys_size))) {
70      fprintf(stderr,"Could not allocate keys_h: %s\n",strerror(errno));
71      exit(-1);
72  }
73
74  /* copy byte-aligned keys to word-stream-aligned keys */
75  {
76      u32 *curr_key;
77      u8* tmp_keys=keys;
78
79      /* allocate a current working key */
80      if(!(curr_key=(u32*)malloc(sizeof(u32)*key_size_nrwords))) {
81          fprintf(stderr,"Could not allocate curr_key: %s\n",strerror(errno));
82          exit(-1);
83      }
84      memset(curr_key,0x00,sizeof(u32)*key_size_nrwords);
85
86      for(int i=0;i<nr_streams;i++) {
87          /* copy one of the keys to current key */
88          memcpy(curr_key,tmp_keys,key_size_bytes);
89          tmp_keys+=key_size_bytes;
90          /* copy current key to stream-aligned one */
91          for(int j=0;j<key_size_nrwords;j++) {
92              keys_h[j*nr_streams+i]=CH_ENDIANESS32(curr_key[j]);
93          }
94      }
95
96      free(curr_key);
97  }
98
99
```

```
100     /* Copy keys to device and free them from host */
101     cutilSafeCall(cudaMemcpy(ctx->keys_d,keys_h,keys_size,
102                                                 cudaMemcpyHostToDevice));
103     free(keys_h);
104
105 }
106
107 void gSTREAM_ivsetup(gSTREAM_ctx* ctx, u8* ivs) {
108
109     int nr_streams=ctx->nr_threads*ctx->nr_blocks;
110     /* initialize the registers to all zeros */
111
112     if(ctx->iv_size>0) {
113         u8* tmp_ivs=ivs;
114         u32* ivs_h=NULL;
115         size_t ivs_size=
116             nr_streams*sizeof(u32)*(((ctx->iv_size-1)/(sizeof(u32)*8))+1);
117
118         u32 *curr_iv;
119         size_t iv_size_bytes=sizeof(u8)*(((ctx->iv_size-1)/(sizeof(u8)*8))+1);
120         size_t iv_size_nrwords=(((ctx->iv_size-1)/(sizeof(u32)*8))+1);
121
122         cutilSafeCall(cudaMalloc((void**)&(ctx->ivs_d),ivs_size));
123         ctx->allocated_ivs=1;
124
125         if(!(ivs_h=(u32*)malloc(ivs_size))) {
126             fprintf(stderr,"Could not allocate ivs_h: %s\n",strerror(errno));
127             exit(-1);
128         }
129
130         /* allocate a current working iv */
131         if(!(curr_iv=(u32*)malloc(sizeof(u32)*iv_size_nrwords))) {
132             fprintf(stderr,"Could not allocate curr_iv: %s\n",strerror(errno));
133             exit(-1);
134         }
135         memset(curr_iv,0x00,sizeof(u32)*iv_size_nrwords);
136
137         for(int i=0;i<nr_streams;i++) {
138             /* copy one of the ivs to current iv */
139             memcpy(curr_iv,tmp_ivs,iv_size_bytes);
140             tmp_ivs+=iv_size_bytes;
141             /* copy current iv to stream-aligned one */
142             for(int j=0;j<iv_size_nrwords;j++) {
143                 ivs_h[j*nr_streams+i]=CH_ENDIANESS32(curr_iv[j]);
144             }
```

```
145         }
146         free(curr_iv);
147
148         /* Copy ivs to device and free them from host */
149         cutilSafeCall(cudaMemcpy(ctx->ivs_d,ivs_h,ivs_size,
150                                                 cudaMemcpyHostToDevice));
151         free(ivs_h);
152     }
153
154     /* Load in iv, key and preclock */
155     /* cipher */_keyivsetup<<<ctx->nr_blocks,ctx->nr_threads>>>(/* cipher state */
156                                                             ,ctx->keys_d
157                                                             ,ctx->key_size
158                                                             ,ctx->ivs_d
159                                                             ,ctx->iv_size);
160     cutilCheckMsg("Kernel execution failed");
161     cudaThreadSynchronize();
162
163 }
164
165 void gSTREAM_keystream_bytes(gSTREAM_ctx* ctx, u8* keystreams, u32 length) {
166     gSTREAM_process_bytes(GEN_KEYSTREAM,ctx,NULL,keystreams,length);
167 }
168
169 void gSTREAM_process_bytes(gSTREAM_action action, gSTREAM_ctx* ctx,
170                                         u8* inputs, u8* outputs, u32 length) {
171     int nr_streams=ctx->nr_blocks*ctx->nr_threads;
172     size_t length_nr_words=(((length-1)/(sizeof(u32)))+1);
173     size_t buff_size=nr_streams*length_nr_words*sizeof(u32);
174     u32* tmp_buffer;
175
176     /* allocate buffer */
177     if((!ctx->allocated_buff)||((length_nr_words*sizeof(u32))>ctx->buff_size)) {
178         if(ctx->allocated_buff) {
179             free(ctx->buff_h); //alocate a large buffer
180         }
181         cutilSafeCall(cudaHostAlloc((void**)&(ctx->buff_h),buff_size,
182                     cudaHostAllocMapped));
183         cutilSafeCall(cudaHostGetDevicePointer((void **)&(ctx->buff_d),
184                     ctx->buff_h,0));
185         ctx->allocated_buff=1;
186         ctx->buff_size=length_nr_words*sizeof(u32);
187     }
188
189     /* allocate a current working buffer */
```

```
190     if(!(tmp_buffer=(u32*)malloc(sizeof(u32)*length_nr_words))) {
191         fprintf(stderr,"Could not allocate tmp_buffer: %s\n",strerror(errno));
192         exit(-1);
193     }
194
195     if(action!=GEN_KEYSTREAM) {
196         for(int i=0;i<nr_streams;i++) {
197             /* copy one of the inputs to current working buffer */
198             memcpy(tmp_buffer,inputs,length);
199             inputs+=length;
200             /* copy current iv to stream-aligned one */
201             for(int j=0;j<length_nr_words;j++) {
202                 ctx->buff_h[j*nr_streams+i]=CH_ENDIANESS32(tmp_buffer[j]);
203             }
204         }
205     }
206
207     /* process bytes */
208     cutilCheckError(cutStartTimer(ctx->bench.timer));
209     /* cipher */_process_bytes<<<ctx->nr_blocks,ctx->nr_threads>>>(action
210                                                                 ,/* cipher state */
211                                                                 ,ctx->buff_d
212                                                                 ,length_nr_words);
213     cutilCheckMsg("Kernel execution failed");
214     cudaThreadSynchronize();
215     cutilCheckError(cutStopTimer(ctx->bench.timer));
216
217     /* copy from working buffer to output buffer */
218     for(int i=0;i<nr_streams;i++) {
219         /* copy one of the keystreams to current keystream */
220         for(int j=0;j<length_nr_words;j++) {
221             tmp_buffer[j]=ctx->buff_h[i+j*nr_streams];
222         }
223         memcpy(outputs,tmp_buffer,length);
224         outputs+=length;
225     }
226
227     free(tmp_buffer);
228 }
229
230 double gSTREAM_getTimerValue(gSTREAM_ctx* ctx) {
231     return cutGetTimerValue(ctx->bench.timer);
232 }
```

In addition to API files we provide a sample program which may be used to test keystream generation, encryption, and decryption for all gSTREAM cipher implementations. The program output may be used to test against eSTREAM reference implementations.

gSTREAM_test.h:

```
#ifndef __GSTREAM_TEST_H_
#define __GSTREAM_TEST_H_

#include <stdlib.h>
#include "gSTREAM.h"

#define PRINT_KEYIV 0
#define PRINT_INPUT 0
#define PRINT_OUTPUT 1

/* Given a seed, the number of streams=nr_threads*nr_blocks, key size,
   iv size, and buffer byte-size generate buff_size_bytes keystream bytes.
   If action is ENCRYPT or DECRYPT, a random buffer of same size is
   created and the respective action (rather than generate keystream)
   is performed.
*/

void do_test(int seed, int dev_no, int nr_threads, int nr_blocks,
             gSTREAM_action action,
             size_t key_size_bytes,
             size_t iv_size_bytes, size_t buff_size_bytes);

#endif
```

gSTREAM_test.cpp:

```
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <cutil_inline.h>
#include <cuda_runtime_api.h>
#include "gSTREAM.h"
#include "gSTREAM_test.h"

```

```
/* Allocate and generate many random keys and random ivs. */
static void gen_rand_keys_ivs(u8 **keys, size_t key_size,
                              u8 **ivs, size_t iv_size,
                              int nr) {
    unsigned i;

    if(!(*keys=(u8*)malloc(sizeof(u8)*key_size*nr))) {
        fprintf(stderr, "Failed to allocate keys: %s\n", strerror(errno));
        exit(-1);
    }

    if(!(*ivs=(u8*)malloc(sizeof(u8)*iv_size*nr))) {
        fprintf(stderr, "Failed to allocate ivs: %s\n", strerror(errno));
        exit(-1);
    }
    for(i=0;i<nr*key_size;i++) { (*keys)[i]=(u8)rand();}
    for(i=0;i<nr*iv_size;i++) { (*ivs)[i]=(u8)rand();}
}

/* Generate random buffer. */
static void gen_rand_buffs(u8 *buffs, size_t buff_size, int nr) {
    unsigned i;
    for(i=0;i<nr*buff_size;i++) {
        buffs[i]=(u8)rand();
    }
}

void do_test(int seed, int dev_no, int nr_threads, int nr_blocks,
             gSTREAM_action action,
             size_t key_size_bytes,
             size_t iv_size_bytes, size_t buff_size_bytes) {


    gSTREAM_ctx ctx;
    u8 *keys, *ivs, *buffs;
    int nr_streams=nr_threads*nr_blocks;
    double ms_time;

    srand(seed);

    gen_rand_keys_ivs(&keys,key_size_bytes,&ivs,iv_size_bytes,nr_streams);

    if(PRINT_KEYIV){
        /* print keys and ivs */
        unsigned i;
```

```
54      printf("Keys:\n");
55      for(i=0;i<nr_streams*key_size_bytes;i++) {
56          printf("0x%02x, ",keys[i]);
57          if(!((i+1)%key_size_bytes)) { printf("\n"); }
58      }
59
60      printf("IVs:\n");
61      for(i=0;i<nr_streams*iv_size_bytes;i++) {
62          printf("0x%02x, ",ivs[i]);
63          if(!((i+1)%iv_size_bytes)) { printf("\n"); }
64      }
65  }
66
67  if(!(buffs=(u8*)malloc(sizeof(u8)*buff_size_bytes*nr_streams))) {
68      fprintf(stderr, "Failed to allocate buffs: %s\n", strerror(errno));
69      exit(-1);
70  }
71
72  /* initialize context */
73  gSTREAM_init(&ctx,dev_no,nr_threads,nr_blocks);
74
75  /* do the key and iv setup */
76  gSTREAM_keysetup(&ctx,(u8*)keys,key_size_bytes*8,iv_size_bytes*8);
77  gSTREAM_ivsetup(&ctx,(u8*)ivs);
78
79  if(action==GEN_KEYSTREAM) {
80      gSTREAM_keystream_bytes(&ctx,(u8*)buffs,buff_size_bytes);
81  } else {
82      gen_rand_buffs(buffs,buff_size_bytes,nr_streams);
83      if(PRINT_INPUT) {
84          /* print input */
85          unsigned i;
86          printf("Input:\n");
87          for(i=0;i<nr_streams*buff_size_bytes;i++) {
88              printf("0x%02x, ",buffs[i]);
89              if(!((i+1)%buff_size_bytes)) { printf("\n"); }
90          }
91      }
92      gSTREAM_process_bytes(ENCRYPT,&ctx,(u8*)buffs,(u8*)buffs,buff_size_bytes);
93  }
94
95
96
97  if(PRINT_OUTPUT){
98      /* print output */
```

```
99          unsigned i;
100         printf("Output:\n");
101         for(i=0;i<nr_streams*buff_size_bytes;i++) {
102             printf("0x%02x, ",buffs[i]);
103             if(!((i+1)%buff_size_bytes)) { printf("\n"); }
104         }
105     }
106
107     ms_time=gSTREAM_getTimerValue(&ctx);
108
109     debug("Elapsed time: %f ms, %f cycles/byte\n",ms_time
110                 ,1242*1000*ms_time/(buff_size_bytes*nr_streams));
111
112     gSTREAM_exit(&ctx);
113
114     free(keys);
115     free(ivs);
116     free(buffs);
117
118 }
```

D.2 MICKEY v2 Example Implementation

Below we present a full gSTREAM implementation of the MICKEY v2 stream cipher. This implementation corresponds to the benchmarking result presented in Chapter 6.

gSTREAM.h:

```
1  #ifndef __GSTREAM_H_
2  #define __GSTREAM_H_
3
4  #define DEBUG
5  #include <stdint.h>
6
7  #ifdef DEBUG
8  #define debug(...) \
9      fprintf(stderr, __VA_ARGS__)
10 #else
11 #define debug(...) ;
12 #endif
```

```
13
14 #define CH_ENDIANESS32(a) \
15     (((a)>>24) | (((a)>>8)&0x0000FF00) | (((a)<<8) & 0x00FF0000) | ((a)<<24))
16
17
18 typedef uint8_t u8;
19 typedef uint16_t u16;
20 typedef uint32_t u32;
21 typedef uint64_t u64;
22
23 typedef enum { ENCRYPT=0, DECRYPT=1, GEN_KEYSTREAM=2 } gSTREAM_action;
24
25 typedef struct {
26     u32 *r_d;
27     u32 *s_d;
28 } MICKEY_ctx;
29
30 typedef struct {
31     int nr_threads; /* per block */
32     int nr_blocks;
33
34     u32 *keys_d;
35     u32 key_size; /* in bits */
36     int allocated_keys;
37
38     u32 *ivs_d;
39     u32 iv_size; /* in bits */
40     int allocated_ivs;
41
42     u32 *buff_d, *buff_h;
43     u32 buff_size; /* in bytes (ceil to nearest 4-bytes) */
44     int allocated_buff;
45
46     struct { /* expandable benchmarking struct */
47         unsigned timer;
48     } bench;
49
50     /* Insert cipher-dependent fields here: */
51     MICKEY_ctx mctx;
52
53 } gSTREAM_ctx;
54
55 /* Initialize device and allocate any state-related buffers.
56     device - which device to use,
57     nr_threads - number of threads/block,
```

```
58      nr_blocks − number of blocks/grid
59  */
60  void gSTREAM_init(gSTREAM_ctx* ctx, int device, int nr_threads, int nr_blocks);
61
62  /* Do the key setup.
63      keys − all the stream keys: key[i][] corresponds to the i−th streams's key,
64      keysize − size of key in bits,
65      ivsize − size of iv in bits
66  */
67  void gSTREAM_keysetup(gSTREAM_ctx* ctx, u8* keys, u32 keysize, u32 ivsize);
68
69  /* Do the iv setup.
70      ivs − all the stream ivs: iv[i][] corresponds to the i−th streams's iv,
71  */
72  void gSTREAM_ivsetup(gSTREAM_ctx* ctx, u8* ivs);
73
74  /*
75      inputs − all the stream inputs:
76                  input[i][] corresponds to the i−th streams's input,
77      outputs − all the stream outputs:
78                  output[i][] corresponds to the i−th streams's output,
79      length − input/output length in bytes
80   */
81  void gSTREAM_process_bytes(gSTREAM_action action, gSTREAM_ctx* ctx,
82                                  u8* inputs, u8* outputs, u32 length);
83
84  /* Generate keystream bytes.
85      keystreams[i] = keystream i
86      length − keystream length in bytes
87   */
88  void gSTREAM_keystream_bytes(gSTREAM_ctx* ctx, u8* keystreams, u32 length);
89
90  /* Free any allocated buffers and destroy context. */
91  void gSTREAM_exit(gSTREAM_ctx* ctx);
92
93  /* Get the measured time elapsed during keystream generation. */
94  double gSTREAM_getTimerValue(gSTREAM_ctx* ctx);
95
96  #endif
```

gSTREAM.cu:

```
1  #include <cutil_inline.h>
2  #include <cuda_runtime_api.h>
3  #include <stdio.h>
```

```
4  #include <string.h>
5  #include <errno.h>
6  #include "gSTREAM.h"
7
8  #include "MICKEY_kernel.cu"
9
10 void gSTREAM_init(gSTREAM_ctx* ctx, int device, int nr_threads, int nr_blocks){
11
12     cudaDeviceProp deviceProp;
13     MICKEY_ctx *mctx=&ctx->mctx;
14     int nr_streams=nr_threads*nr_blocks;
15
16
17
18     /* set device */
19     cudaGetDeviceProperties(&deviceProp, device);
20     cudaSetDevice(device);
21     debug("\nUsing device %d: \"%s\"\n", device, deviceProp.name);
22
23     cutilSafeCall(cudaSetDeviceFlags(cudaDeviceMapHost));
24
25     ctx->nr_threads = nr_threads;
26     ctx->nr_blocks = nr_blocks;
27     ctx->allocated_keys=0;
28     ctx->allocated_ivs=0;
29     ctx->allocated_buff=0;
30
31     cutilCheckError(cutCreateTimer(&(ctx->bench.timer)));
32
33     /* allocate Rabbit state */
34
35     cutilSafeCall(cudaMalloc((void**)&(mctx->r_d),nr_streams*4*sizeof(u32)));
36     cutilSafeCall(cudaMalloc((void**)&(mctx->s_d),nr_streams*4*sizeof(u32)));
37 }
38
39 void gSTREAM_exit(gSTREAM_ctx* ctx) {
40     MICKEY_ctx *mctx=&ctx->mctx;
41
42     if(ctx->allocated_keys) {
43         cutilSafeCall(cudaFree(ctx->keys_d));
44     }
45
46     if(ctx->allocated_ivs) {
47         cutilSafeCall(cudaFree(ctx->ivs_d));
48     }
```

```
49
50      if(ctx->allocated_buff) {
51          cutilSafeCall(cudaFreeHost(ctx->buff_h));
52      }
53
54      cutilCheckError(cutDeleteTimer(ctx->bench.timer));
55
56      cutilSafeCall(cudaFree(mctx->r_d));
57      cutilSafeCall(cudaFree(mctx->s_d));
58  }
59
60  void gSTREAM_keysetup(gSTREAM_ctx* ctx, u8* keys, u32 keysize, u32 ivsize) {
61
62      size_t keys_size;
63      int nr_streams=ctx->nr_threads*ctx->nr_blocks;
64      u32* keys_h=NULL;
65      size_t key_size_bytes=sizeof(u8)*(((keysize-1)/(sizeof(u8)*8))+1);
66      size_t key_size_nrwords=(((keysize-1)/(sizeof(u32)*8))+1);
67
68      ctx->key_size=keysize;
69      ctx->iv_size=ivsize;
70
71      /* allocate keys */
72      keys_size=nr_streams*sizeof(u32)*(((keysize-1)/(sizeof(u32)*8))+1);
73      cutilSafeCall(cudaMalloc((void**)&(ctx->keys_d),keys_size));
74      ctx->allocated_keys=1;
75      if(!(keys_h=(u32*)malloc(keys_size))) {
76          fprintf(stderr,"Could not allocate keys_h: %s\n",strerror(errno));
77          exit(-1);
78      }
79
80      /* copy byte-aligned keys to word-stream-aligned keys */
81      {
82          u32 *curr_key;
83          u8* tmp_keys=keys;
84
85          /* allocate a current working key */
86          if(!(curr_key=(u32*)malloc(sizeof(u32)*key_size_nrwords))) {
87              fprintf(stderr,"Could not allocate curr_key: %s\n",strerror(errno));
88              exit(-1);
89          }
90          memset(curr_key,0x00,sizeof(u32)*key_size_nrwords);
91
92          for(int i=0;i<nr_streams;i++) {
93              /* copy one of the keys to current key */
```

```
94              memcpy(curr_key,tmp_keys,key_size_bytes);
95              tmp_keys+=key_size_bytes;
96              /* copy current key to stream−aligned one */
97              for(int j=0;j<key_size_nrwords;j++) {
98                  keys_h[j*nr_streams+i]=CH_ENDIANESS32(curr_key[j]);
99              }
100         }
101
102         free(curr_key);
103     }
104
105     /* Copy keys to device and free them from host */
106     cutilSafeCall(cudaMemcpy(ctx−>keys_d,keys_h,keys_size,
107                                                 cudaMemcpyHostToDevice));
108         free(keys_h);
109
110 }
111
112 void gSTREAM_ivsetup(gSTREAM_ctx* ctx, u8* ivs) {
113     MICKEY_ctx *mctx=&ctx−>mctx;
114
115     int nr_streams=ctx−>nr_threads*ctx−>nr_blocks;
116     /* initialize the registers to all zeros */
117
118     if(ctx−>iv_size>0) {
119         u8* tmp_ivs=ivs;
120         u32* ivs_h=NULL;
121         size_t ivs_size=
122             nr_streams*sizeof(u32)*(((ctx−>iv_size−1)/(sizeof(u32)*8))+1);
123
124         u32 *curr_iv;
125         size_t iv_size_bytes=sizeof(u8)*(((ctx−>iv_size−1)/(sizeof(u8)*8))+1);
126         size_t iv_size_nrwords=(((ctx−>iv_size−1)/(sizeof(u32)*8))+1);
127
128         cutilSafeCall(cudaMalloc((void**)&(ctx−>ivs_d),ivs_size));
129         ctx−>allocated_ivs=1;
130
131         if(!(ivs_h=(u32*)malloc(ivs_size))) {
132             fprintf(stderr,"Could not allocate ivs_h: %s\n",strerror(errno));
133             exit(−1);
134         }
135
136         /* allocate a current working iv */
137         if(!(curr_iv=(u32*)malloc(sizeof(u32)*iv_size_nrwords))) {
138             fprintf(stderr,"Could not allocate curr_iv: %s\n",strerror(errno));
```

```
139             exit(-1);
140         }
141         memset(curr_iv,0x00,sizeof(u32)*iv_size_nrwords);
142 
143         for(int i=0;i<nr_streams;i++) {
144             /* copy one of the ivs to current iv */
145             memcpy(curr_iv,tmp_ivs,iv_size_bytes);
146             tmp_ivs+=iv_size_bytes;
147             /* copy current iv to stream-aligned one */
148             for(int j=0;j<iv_size_nrwords;j++) {
149                 ivs_h[j*nr_streams+i]=CH_ENDIANESS32(curr_iv[j]);
150             }
151         }
152         free(curr_iv);
153 
154         /* Copy ivs to device and free them from host */
155         cutilSafeCall(cudaMemcpy(ctx->ivs_d,ivs_h,ivs_size,
156                                                 cudaMemcpyHostToDevice));
157         free(ivs_h);
158     }
159 
160     /* Load in iv, key and preclock */
161     MICKEY_keyivsetup<<<ctx->nr_blocks,ctx->nr_threads>>>(mctx->r_d
162                                                           ,mctx->s_d
163                                                           ,ctx->keys_d
164                                                           ,ctx->key_size
165                                                           ,ctx->ivs_d
166                                                           ,ctx->iv_size);
167     cutilCheckMsg("Kernel execution failed");
168     cudaThreadSynchronize();
169 
170 }
171 
172 void gSTREAM_keystream_bytes(gSTREAM_ctx* ctx, u8* keystreams, u32 length) {
173     gSTREAM_process_bytes(GEN_KEYSTREAM,ctx,NULL,keystreams,length);
174 }
175 
176 void gSTREAM_process_bytes(gSTREAM_action action, gSTREAM_ctx* ctx,
177                            u8* inputs, u8* outputs, u32 length) {
178     int nr_streams=ctx->nr_blocks*ctx->nr_threads;
179     size_t length_nr_words=(((length-1)/(sizeof(u32)))+1);
180     size_t buff_size=nr_streams*length_nr_words*sizeof(u32);
181     MICKEY_ctx *mctx=&ctx->mctx;
182     u32* tmp_buffer;
183 
```

```
184     /* allocate buffer */
185     if((!ctx->allocated_buff)||((length_nr_words*sizeof(u32))>ctx->buff_size)) {
186         if(ctx->allocated_buff) {
187             free(ctx->buff_h); //alocate a large buffer
188         }
189         cutilSafeCall(cudaHostAlloc((void**)&(ctx->buff_h),buff_size,
190                 cudaHostAllocMapped));
191         cutilSafeCall(cudaHostGetDevicePointer((void **)&(ctx->buff_d),
192                 ctx->buff_h,0));
193         ctx->allocated_buff=1;
194         ctx->buff_size=length_nr_words*sizeof(u32);
195     }
196
197     /* allocate a current working buffer */
198     if(!(tmp_buffer=(u32*)malloc(sizeof(u32)*length_nr_words))) {
199         fprintf(stderr,"Could not allocate tmp_buffer: %s\n",strerror(errno));
200         exit(-1);
201     }
202
203     if(action!=GEN_KEYSTREAM) {
204         for(int i=0;i<nr_streams;i++) {
205             /* copy one of the inputs to current working buffer */
206             memcpy(tmp_buffer,inputs,length);
207             inputs+=length;
208             /* copy current iv to stream-aligned one */
209             for(int j=0;j<length_nr_words;j++) {
210                 ctx->buff_h[j*nr_streams+i]=CH_ENDIANESS32(tmp_buffer[j]);
211             }
212         }
213     }
214
215     /* process bytes */
216     cutilCheckError(cutStartTimer(ctx->bench.timer));
217     MICKEY_process_bytes<<<ctx->nr_blocks,ctx->nr_threads>>>(action
218                                                              ,mctx->r_d
219                                                              ,mctx->s_d
220                                                              ,ctx->buff_d
221                                                              ,length_nr_words);
222     cutilCheckMsg("Kernel execution failed");
223     cudaThreadSynchronize();
224     cutilCheckError(cutStopTimer(ctx->bench.timer));
225
226     /* copy from working buffer to output buffer */
227     for(int i=0;i<nr_streams;i++) {
228         /* copy one of the keystreams to current keystream */
```

```
229         for(int j=0;j<length_nr_words;j++) {
230             tmp_buffer[j]=ctx->buff_h[i+j*nr_streams];
231         }
232         memcpy(outputs,tmp_buffer,length);
233         outputs+=length;
234     }
235 
236     free(tmp_buffer);
237 }
238 
239 double gSTREAM_getTimerValue(gSTREAM_ctx* ctx) {
240     return cutGetTimerValue(ctx->bench.timer);
241 }
```

MICKEY_kernel.cu:

```
1  #ifndef __MICKEY_KERNEL_CU__
2  #define __MICKEY_KERNEL_CU__
3  
4  #define __mem(mm,i,j,N) ((mm)[(i)+(j)*(N)])
5  #define max(a,b) (((a)>(b))?(a):(b))
6  #define min(a,b) (((a)<(b))?(a):(b))
7  
8  /* Barely modified (form original submission) CLOCK_R function. */
9  __device__ void CLOCK_R(u32 *oR0, u32 *oR1, u32 *oR2, u32 *oR3,
10                          int input_bit, int control_bit) {
11     u32 R0=*oR0, R1=*oR1, R2=*oR2, R3=*oR3;
12     int Feedback_bit;
13     int Carry0, Carry1, Carry2;
14     Feedback_bit = ((R3 >> 3) & 1) ^ input_bit;
15     Carry0 = (R0 >> 31) & 1;
16     Carry1 = (R1 >> 31) & 1;
17     Carry2 = (R2 >> 31) & 1;
18 
19     if (control_bit) {
20         R0 ^= (R0 << 1);
21         R1 ^= (R1 << 1) ^ Carry0;
22         R2 ^= (R2 << 1) ^ Carry1;
23         R3 ^= (R3 << 1) ^ Carry2;
24     } else {
25         R0 = (R0 << 1);
26         R1 = (R1 << 1) ^ Carry0;
27         R2 = (R2 << 1) ^ Carry1;
28         R3 = (R3 << 1) ^ Carry2;
29     }
```

```
30
31      if (Feedback_bit) {
32          R0 ^= 0x1279327b;
33          R1 ^= 0xb5546660;
34          R2 ^= 0xdf87818f;
35          R3 ^= 0x00000003;
36      }
37      *oR0=R0;
38      *oR1=R1;
39      *oR2=R2;
40      *oR3=R3;
41  }
42
43  /* Barely modified (form original submission) CLOCK_S function. */
44  __device__ void CLOCK_S(u32 *oS0, u32 *oS1, u32 *oS2, u32 *oS3,
45                          int input_bit, int control_bit) {
46      u32 S0=*oS0, S1=*oS1, S2=*oS2, S3=*oS3;
47      int Feedback_bit;
48      int Carry0, Carry1, Carry2;
49
50
51      Feedback_bit = ((S3 >> 3) & 1) ^ input_bit;
52      Carry0 = (S0 >> 31) & 1;
53      Carry1 = (S1 >> 31) & 1;
54      Carry2 = (S2 >> 31) & 1;
55
56      S0 = (S0<<1) ^ ((S0^0x6aa97a30) & ((S0>>1) ^ (S1<<31) ^ 0xdd629e9a) & 0xfffffffe);
57      S1 = (S1<<1) ^ ((S1^0x7942a809) & ((S1>>1) ^ (S2<<31) ^ 0xe3a21d63)) ^ Carry0;
58      S2 = (S2<<1) ^ ((S2^0x057ebfea) & ((S2>>1) ^ (S3<<31) ^ 0x91c23dd7)) ^ Carry1;
59      S3 = (S3<<1) ^ ((S3^0x00000006) & ((S3>>1) ^ 0x00000001) & 0x7) ^ Carry2;
60
61      if (Feedback_bit) {
62          if (control_bit) {
63              S0 ^= 0x4c8cb877;
64              S1 ^= 0x4911b063;
65              S2 ^= 0x40fbc52b;
66              S3 ^= 0x00000008;
67          } else {
68              S0 ^= 0x9ffa7faf;
69              S1 ^= 0xaf4a9381;
70              S2 ^= 0x9cec5802;
71              S3 ^= 0x00000001;
72          }
73      }
74      *oS0=S0;
```

```
75      *oS1=S1;
76      *oS2=S2;
77      *oS3=S3;
78  }
79
80  /* Macro version of the CLOCK_KG function provided in submission code */
81  #define CLOCK_KG(Keystream_bit,R0,R1,R2,R3,S0,S1,S2,S3,mixing,input_bit) \
82  do { \
83      int control_bit_r; \
84      int control_bit_s; \
85  \
86      (Keystream_bit) = ((R0) ^ (S0)) & 1; \
87      control_bit_r = (((S1) >> 2) ^ ((R2) >> 3)) & 1; \
88      control_bit_s = (((R1) >> 1) ^ ((S2) >> 3)) & 1; \
89  \
90      if((mixing)) { \
91          CLOCK_R(&(R0), &(R1), &(R2), &(R3), (((S1)>>18)&1)^(input_bit), control_bit_r);\
92      } else { \
93          CLOCK_R(&(R0), &(R1), &(R2), &(R3), (input_bit), control_bit_r); \
94      } \
95      CLOCK_S(&(S0), &(S1), &(S2), &(S3), (input_bit), control_bit_s); \
96  } while(0)
97
98  /* Key and iv setup GPU code. */
99  __global__ void MICKEY_keyivsetup(u32* g_r, u32* g_s,
100                                    u32 *keys, u32 key_size,
101                                    u32 *ivs, u32 iv_size) {
102     u32 tID=blockIdx.x*blockDim.x+threadIdx.x;
103     u32 nr_streams=blockDim.x*gridDim.x;
104
105     u32 r0,r1,r2,r3,
106         s0,s1,s2,s3;
107     int Keystream_bit;
108
109     u32 sub_keyiv;
110     int ivkey_bit;
111     int ivkey_no;
112     int i;
113
114     /* initialize the registers to all-zero */
115     r0=0; s0=0;
116     r1=0; s1=0;
117     r2=0; s2=0;
118     r3=0; s3=0;
119
```

```
120     ivkey_no=0;
121     while(iv_size>0) {
122     /* read in the iv and clock for each bit */
123         sub_keyiv = __mem(ivs,tID,ivkey_no++,nr_streams);
124         for(i=0;i<min(iv_size,32);i++) {
125             ivkey_bit=(sub_keyiv&0x80000000)?1:0;
126             CLOCK_KG(Keystream_bit,r0,r1,r2,r3,s0,s1,s2,s3,1,ivkey_bit);
127             sub_keyiv<<=1;
128         }
129         iv_size=max((int)(iv_size-32),0);
130     }
131
132     ivkey_no=0;
133     while(key_size>0) {
134     /* read in the key and clock for each bit */
135         sub_keyiv = __mem(keys,tID,ivkey_no++,nr_streams);
136         for(i=0;i<min(key_size,32);i++) {
137             ivkey_bit=(sub_keyiv&0x80000000)?1:0;
138             CLOCK_KG(Keystream_bit,r0,r1,r2,r3,s0,s1,s2,s3,1,ivkey_bit);
139             sub_keyiv<<=1;
140         }
141         key_size=max((int)(key_size-32),0);
142     }
143
144     for(i=0;i<100;i++) { /* preclock */
145             CLOCK_KG(Keystream_bit,r0,r1,r2,r3,s0,s1,s2,s3,1,0);
146     }
147     /* write the registers to global state */
148     __mem(g_r,tID,0,nr_streams)=r0; __mem(g_s,tID,0,nr_streams)=s0;
149     __mem(g_r,tID,1,nr_streams)=r1; __mem(g_s,tID,1,nr_streams)=s1;
150     __mem(g_r,tID,2,nr_streams)=r2; __mem(g_s,tID,2,nr_streams)=s2;
151     __mem(g_r,tID,3,nr_streams)=r3; __mem(g_s,tID,3,nr_streams)=s3;
152
153 }
154
155 /* Process (encrypt/decrypt/keystream generate) bytes on the GPU */
156 __global__ void MICKEY_process_bytes(gSTREAM_action act,u32* g_r, u32* g_s,
157                                      u32 *buff, u32 nr_words) {
158     u32 tID=blockIdx.x*blockDim.x+threadIdx.x;
159     u32 nr_streams=blockDim.x*gridDim.x;
160
161     u32 r0,r1,r2,r3,
162         s0,s1,s2,s3;
163
164     int Keystream_bit;
```

```
165
166     /* read in the current state */
167     r0=__mem(g_r,tID,0,nr_streams); s0=__mem(g_s,tID,0,nr_streams);
168     r1=__mem(g_r,tID,1,nr_streams); s1=__mem(g_s,tID,1,nr_streams);
169     r2=__mem(g_r,tID,2,nr_streams); s2=__mem(g_s,tID,2,nr_streams);
170     r3=__mem(g_r,tID,3,nr_streams); s3=__mem(g_s,tID,3,nr_streams);
171
172     for(int w=0;w<nr_words;w++) {
173         u32 output_word=0;
174
175         if(act!=GEN_KEYSTREAM) {
176             /* if encrypting/decrypting XOR with input */
177             output_word=__mem(buff,tID,w,nr_streams);
178         }
179
180 #pragma unroll 32
181         for(int i=0;i<32;i++) {
182         /* generate 4-bytes of keystream */
183             CLOCK_KG(Keystream_bit,r0,r1,r2,r3,s0,s1,s2,s3,0,0);
184             output_word ^= Keystream_bit << (31-i);
185         }
186
187         /* write output to global memory */
188         __mem(buff,tID,w,nr_streams)=CH_ENDIANESS32(output_word);
189     }
190
191     /* write new (local) state to global state */
192     __mem(g_r,tID,0,nr_streams)=r0; __mem(g_s,tID,0,nr_streams)=s0;
193     __mem(g_r,tID,1,nr_streams)=r1; __mem(g_s,tID,1,nr_streams)=s1;
194     __mem(g_r,tID,2,nr_streams)=r2; __mem(g_s,tID,2,nr_streams)=s2;
195     __mem(g_r,tID,3,nr_streams)=r3; __mem(g_s,tID,3,nr_streams)=s3;
196
197 }
198 #endif
```

MICKEY_test.cpp:

```
1  #include "gSTREAM.h"
2  #include "gSTREAM_test.h"
3
4  int main(void) {
5      do_test(0
6              ,2,128,680
7              ,GEN_KEYSTREAM,10,10
8              ,1024);
```

```
    return 0;
}
```

D.3 Trivium Example Implementation

Below we present a full gSTREAM implementation of the Trivium stream cipher. This implementation corresponds to the benchmarking result presented in Chapter 6.

gSTREAM.h:

```
#ifndef __GSTREAM_H_
#define __GSTREAM_H_

#define DEBUG
#include <stdint.h>

#ifdef DEBUG
#define debug(...) \
    fprintf(stderr, __VA_ARGS__)
#else
#define debug(...) ;
#endif

#define CH_ENDIANESS32(a) (a)


typedef uint8_t u8;
typedef uint16_t u16;
typedef uint32_t u32;
typedef uint64_t u64;

typedef enum { ENCRYPT=0, DECRYPT=1, GEN_KEYSTREAM=2 } gSTREAM_action;

typedef struct {
    u32 *s32_d;
} Trivium_ctx;


typedef struct {
    int nr_threads; /* per block */
```

```
31      int nr_blocks;
32
33      u32 *keys_d;
34      u32 key_size; /* in bits */
35      int allocated_keys;
36
37      u32 *ivs_d;
38      u32 iv_size; /* in bits */
39      int allocated_ivs;
40
41      u32 *buff_d, *buff_h;
42      u32 buff_size; /* in bytes (ceil to nearest 4-bytes) */
43      int allocated_buff;
44
45      struct { /* expandable benchmarking struct */
46          unsigned timer;
47      } bench;
48
49      /* Insert cipher-dependent fields here: */
50      Trivium_ctx tctx;
51
52  } gSTREAM_ctx;
53
54  /* Initialize device and allocate any state-related buffers.
55      device - which device to use,
56      nr_threads - number of threads/block,
57      nr_blocks - number of blocks/grid
58  */
59  void gSTREAM_init(gSTREAM_ctx* ctx, int device, int nr_threads, int nr_blocks);
60
61  /* Do the key setup.
62      keys - all the stream keys: key[i][] corresponds to the i-th streams's key,
63      keysize - size of key in bits,
64      ivsize - size of iv in bits
65  */
66  void gSTREAM_keysetup(gSTREAM_ctx* ctx, u8* keys, u32 keysize, u32 ivsize);
67
68  /* Do the iv setup.
69      ivs - all the stream ivs: iv[i][] corresponds to the i-th streams's iv,
70  */
71  void gSTREAM_ivsetup(gSTREAM_ctx* ctx, u8* ivs);
72
73  /*
74      inputs - all the stream inputs:
75                  input[i][] corresponds to the i-th streams's input,
```

```
76     outputs − all the stream outputs:
77               output[i][] corresponds to the i−th streams's output,
78     length − input/output length in bytes
79   */
80  void gSTREAM_process_bytes(gSTREAM_action action, gSTREAM_ctx* ctx,
81                             u8* inputs, u8* outputs, u32 length);
82
83  /* Generate keystream bytes.
84     keystreams[i] = keystream i
85     length − keystream length in bytes
86   */
87  void gSTREAM_keystream_bytes(gSTREAM_ctx* ctx, u8* keystreams, u32 length);
88
89  /* Free any allocated buffers and destroy context. */
90  void gSTREAM_exit(gSTREAM_ctx* ctx);
91
92  /* Get the measured time elapsed during keystream generation. */
93  double gSTREAM_getTimerValue(gSTREAM_ctx* ctx);
94
95  #endif
```

gSTREAM.cu:

```
1   #include <cutil_inline.h>
2   #include <cuda_runtime_api.h>
3   #include <stdio.h>
4   #include <string.h>
5   #include <errno.h>
6   #include "gSTREAM.h"
7
8   #include "Trivium_kernel.cu"
9
10  void gSTREAM_init(gSTREAM_ctx* ctx, int device, int nr_threads, int nr_blocks){
11
12      cudaDeviceProp deviceProp;
13      Trivium_ctx *tctx=&ctx->tctx;
14      int nr_streams=nr_threads*nr_blocks;
15
16
17
18      /* set device */
19      cudaGetDeviceProperties(&deviceProp, device);
20      cudaSetDevice(device);
21      debug("\nUsing device %d: \"%s\"\n", device, deviceProp.name);
22
```

```
23      cutilSafeCall(cudaSetDeviceFlags(cudaDeviceMapHost));
24
25      ctx->nr_threads = nr_threads;
26      ctx->nr_blocks = nr_blocks;
27      ctx->allocated_keys=0;
28      ctx->allocated_ivs=0;
29      ctx->allocated_buff=0;
30
31      cutilCheckError(cutCreateTimer(&(ctx->bench.timer)));
32
33      /* allocate Trivium state */
34
35      cutilSafeCall(cudaMalloc((void**)&(tctx->s32_d),nr_streams*10*sizeof(u32)));
36  }
37
38  void gSTREAM_exit(gSTREAM_ctx* ctx) {
39      Trivium_ctx *tctx=&ctx->tctx;
40
41      if(ctx->allocated_keys) {
42          cutilSafeCall(cudaFree(ctx->keys_d));
43      }
44
45      if(ctx->allocated_ivs) {
46          cutilSafeCall(cudaFree(ctx->ivs_d));
47      }
48
49      if(ctx->allocated_buff) {
50          cutilSafeCall(cudaFreeHost(ctx->buff_h));
51      }
52
53      cutilCheckError(cutDeleteTimer(ctx->bench.timer));
54
55      cutilSafeCall(cudaFree(tctx->s32_d));
56  }
57
58  void gSTREAM_keysetup(gSTREAM_ctx* ctx, u8* keys, u32 keysize, u32 ivsize) {
59
60      size_t keys_size;
61      int nr_streams=ctx->nr_threads*ctx->nr_blocks;
62      u32* keys_h=NULL;
63      size_t key_size_bytes=sizeof(u8)*(((keysize-1)/(sizeof(u8)*8))+1);
64      size_t key_size_nrwords=(((keysize-1)/(sizeof(u32)*8))+1);
65
66      ctx->key_size=keysize;
67      ctx->iv_size=ivsize;
```

```
68
69      /* allocate keys */
70      keys_size=nr_streams*sizeof(u32)*(((keysize-1)/(sizeof(u32)*8))+1);
71      cutilSafeCall(cudaMalloc((void**)&(ctx->keys_d),keys_size));
72      ctx->allocated_keys=1;
73      if(!(keys_h=(u32*)malloc(keys_size))) {
74          fprintf(stderr,"Could not allocate keys_h: %s\n",strerror(errno));
75          exit(-1);
76      }
77
78      /* copy byte-aligned keys to word-stream-aligned keys */
79      {
80          u32 *curr_key;
81          u8* tmp_keys=keys;
82
83          /* allocate a current working key */
84          if(!(curr_key=(u32*)malloc(sizeof(u32)*key_size_nrwords))) {
85              fprintf(stderr,"Could not allocate curr_key: %s\n",strerror(errno));
86              exit(-1);
87          }
88          memset(curr_key,0x00,sizeof(u32)*key_size_nrwords);
89
90          for(int i=0;i<nr_streams;i++) {
91              /* copy one of the keys to current key */
92              memcpy(curr_key,tmp_keys,key_size_bytes);
93              tmp_keys+=key_size_bytes;
94              /* copy current key to stream-aligned one */
95              for(int j=0;j<key_size_nrwords;j++) {
96                  keys_h[j*nr_streams+i]=CH_ENDIANESS32(curr_key[j]);
97              }
98          }
99
100         free(curr_key);
101     }
102
103
104     /* Copy keys to device and free them from host */
105     cutilSafeCall(cudaMemcpy(ctx->keys_d,keys_h,keys_size,
106                                                 cudaMemcpyHostToDevice));
107         free(keys_h);
108
109
110 }
111
112 void gSTREAM_ivsetup(gSTREAM_ctx* ctx, u8* ivs) {
```

```
113     Trivium_ctx *tctx=&ctx->tctx;
114
115     int nr_streams=ctx->nr_threads*ctx->nr_blocks;
116     /* initialize the registers to all zeros */
117
118     if(ctx->iv_size>0) {
119         u8* tmp_ivs=ivs;
120         u32* ivs_h=NULL;
121         size_t ivs_size=
122             nr_streams*sizeof(u32)*(((ctx->iv_size-1)/(sizeof(u32)*8))+1);
123
124         u32 *curr_iv;
125         size_t iv_size_bytes=sizeof(u8)*(((ctx->iv_size-1)/(sizeof(u8)*8))+1);
126         size_t iv_size_nrwords=(((ctx->iv_size-1)/(sizeof(u32)*8))+1);
127
128         cutilSafeCall(cudaMalloc((void**)&(ctx->ivs_d),ivs_size));
129         ctx->allocated_ivs=1;
130
131         if(!(ivs_h=(u32*)malloc(ivs_size))) {
132             fprintf(stderr,"Could not allocate ivs_h: %s\n",strerror(errno));
133             exit(-1);
134         }
135
136         /* allocate a current working iv */
137         if(!(curr_iv=(u32*)malloc(sizeof(u32)*iv_size_nrwords))) {
138             fprintf(stderr,"Could not allocate curr_iv: %s\n",strerror(errno));
139             exit(-1);
140         }
141         memset(curr_iv,0x00,sizeof(u32)*iv_size_nrwords);
142
143         for(int i=0;i<nr_streams;i++) {
144             /* copy one of the ivs to current iv */
145             memcpy(curr_iv,tmp_ivs,iv_size_bytes);
146             tmp_ivs+=iv_size_bytes;
147             /* copy current iv to stream-aligned one */
148             for(int j=0;j<iv_size_nrwords;j++) {
149                 ivs_h[j*nr_streams+i]=CH_ENDIANESS32(curr_iv[j]);
150             }
151         }
152         free(curr_iv);
153
154         /* Copy ivs to device and free them from host */
155         cutilSafeCall(cudaMemcpy(ctx->ivs_d,ivs_h,ivs_size,
156                                                 cudaMemcpyHostToDevice));
157         free(ivs_h);
```

```
158     }
159     /* Load in iv, key and preclock */
160     Trivium_keyivsetup<<<ctx->nr_blocks,ctx->nr_threads>>>(tctx->s32_d
161                                                            ,ctx->keys_d
162                                                            ,ctx->key_size
163                                                            ,ctx->ivs_d
164                                                            ,ctx->iv_size);
165     cutilCheckMsg("Kernel execution failed");
166     cudaThreadSynchronize();
167
168 }
169
170 void gSTREAM_keystream_bytes(gSTREAM_ctx* ctx, u8* keystreams, u32 length) {
171     gSTREAM_process_bytes(GEN_KEYSTREAM,ctx,NULL,keystreams,length);
172 }
173
174 void gSTREAM_process_bytes(gSTREAM_action action, gSTREAM_ctx* ctx,
175                                          u8* inputs, u8* outputs, u32 length) {
176     int nr_streams=ctx->nr_blocks*ctx->nr_threads;
177     size_t length_nr_words=(((length-1)/(sizeof(u32)))+1);
178     size_t buff_size=nr_streams*length_nr_words*sizeof(u32);
179     Trivium_ctx *tctx=&ctx->tctx;
180     u32* tmp_buffer;
181
182     /* allocate buffer */
183     if((!ctx->allocated_buff)||((length_nr_words*sizeof(u32))>ctx->buff_size)) {
184         if(ctx->allocated_buff) {
185             free(ctx->buff_h); //alocate a large buffer
186         }
187         cutilSafeCall(cudaHostAlloc((void**)&(ctx->buff_h),buff_size,
188                 cudaHostAllocMapped));
189         cutilSafeCall(cudaHostGetDevicePointer((void **)&(ctx->buff_d),
190                 ctx->buff_h,0));
191         ctx->allocated_buff=1;
192         ctx->buff_size=length_nr_words*sizeof(u32);
193     }
194
195
196     /* allocate a current working buffer */
197     if(!(tmp_buffer=(u32*)malloc(sizeof(u32)*length_nr_words))) {
198         fprintf(stderr,"Could not allocate tmp_buffer: %s\n",strerror(errno));
199         exit(-1);
200     }
201
202     if(action!=GEN_KEYSTREAM) {
```

```
203         for(int i=0;i<nr_streams;i++) {
204             /* copy one of the inputs to current working buffer */
205             memcpy(tmp_buffer,inputs,length);
206             inputs+=length;
207             /* copy current iv to stream-aligned one */
208             for(int j=0;j<length_nr_words;j++) {
209                 ctx->buff_h[j*nr_streams+i]=CH_ENDIANESS32(tmp_buffer[j]);
210             }
211         }
212     }
213
214     /* process bytes */
215     cutilCheckError(cutStartTimer(ctx->bench.timer));
216     Trivium_process_bytes<<<ctx->nr_blocks,ctx->nr_threads>>>(action
217                                                               ,tctx->s32_d
218                                                               ,ctx->buff_d
219                                                               ,length_nr_words);
220     cutilCheckMsg("Kernel execution failed");
221     cudaThreadSynchronize();
222     cutilCheckError(cutStopTimer(ctx->bench.timer));
223
224     /* copy from working buffer to output buffer */
225     for(int i=0;i<nr_streams;i++) {
226         /* copy one of the keystreams to current keystream */
227         for(int j=0;j<length_nr_words;j++) {
228             tmp_buffer[j]=ctx->buff_h[i+j*nr_streams];
229         }
230         memcpy(outputs,tmp_buffer,length);
231         outputs+=length;
232     }
233
234     free(tmp_buffer);
235 }
236
237 double gSTREAM_getTimerValue(gSTREAM_ctx* ctx) {
238     return cutGetTimerValue(ctx->bench.timer);
239 }
```

Trivium_kernel.cu:

```
1 #ifndef __Trivium_KERNEL_CU__
2 #define __Trivium_KERNEL_CU__
3
4 #define __mem(mm,i,j,N) ((mm)[(i)+(j)*(N)])
5 #define max(a,b) (((a)>(b))?(a):(b))
```

```
6  #define min(a,b) (((a)<(b))?(a):(b))
7
8  #define S(a, n) (s##a##n)
9  #define T(a) (t##a)
10
11 #define S32(a, b) ((S(a, 2) << ( 64 − (b))) | (S(a, 1) >> ((b) − 32)))
12 #define S64(a, b) ((S(a, 3) << ( 96 − (b))) | (S(a, 2) >> ((b) − 64)))
13 #define S96(a, b) ((S(a, 4) << (128 − (b))) | (S(a, 3) >> ((b) − 96)))
14
15 #define UPDATE() \
16   do { \
17     T(1) = S64(1, 66) ^ S64(1, 93); \
18     T(2) = S64(2, 69) ^ S64(2, 84); \
19     T(3) = S64(3, 66) ^ S96(3, 111); \
20                                                                    \
21     Z(T(1) ^ T(2) ^ T(3)); \
22                                                                    \
23     T(1) ^= (S64(1, 91) & S64(1, 92)) ^ S64(2, 78); \
24     T(2) ^= (S64(2, 82) & S64(2, 83)) ^ S64(3, 87); \
25     T(3) ^= (S96(3, 109) & S96(3, 110)) ^ S64(1, 69); \
26   } while (0)
27
28 #define ROTATE() \
29   do { \
30     S(1, 3) = S(1, 2); S(1, 2) = S(1, 1); S(1, 1) = T(3); \
31     S(2, 3) = S(2, 2); S(2, 2) = S(2, 1); S(2, 1) = T(1); \
32     S(3, 4) = S(3, 3); S(3, 3) = S(3, 2); S(3, 2) = S(3, 1); S(3, 1) = T(2); \
33   } while (0)
34
35 #define LOAD(s)\
36 do {\
37    S(1,1) = __mem((s),tID,0,nr_streams); S(2,1) = __mem((s),tID,3,nr_streams);\
38    S(1,2) = __mem((s),tID,1,nr_streams); S(2,2) = __mem((s),tID,4,nr_streams);\
39    S(1,3) = __mem((s),tID,2,nr_streams); S(2,3) = __mem((s),tID,5,nr_streams);\
40 \
41    S(3,1) = __mem((s),tID,6,nr_streams); S(3,3) = __mem((s),tID,8,nr_streams);\
42    S(3,2) = __mem((s),tID,7,nr_streams); S(3,4) = __mem((s),tID,9,nr_streams);\
43 } while(0)
44
45 #define STORE(s)\
46 do {\
47    __mem((s),tID,0,nr_streams) = S(1,1); __mem((s),tID,3,nr_streams) = S(2,1);\
48    __mem((s),tID,1,nr_streams) = S(1,2); __mem((s),tID,4,nr_streams) = S(2,2);\
49    __mem((s),tID,2,nr_streams) = S(1,3); __mem((s),tID,5,nr_streams) = S(2,3);\
50 \
```

```
51      __mem((s),tID,6,nr_streams) = S(3,1); __mem((s),tID,8,nr_streams) = S(3,3);\
52      __mem((s),tID,7,nr_streams) = S(3,2); __mem((s),tID,9,nr_streams) = S(3,4);\
53  } while(0)
54
55  __global__ void Trivium_keyivsetup(u32* g_s,
56                                     u32 *keys, u32 key_size,
57                                     u32 *ivs, u32 iv_size) {
58      u32 tID=blockIdx.x*blockDim.x+threadIdx.x;
59      u32 nr_streams=blockDim.x*gridDim.x;
60
61      u32 s11, s12, s13;
62      u32 s21, s22, s23;
63      u32 s31, s32, s33, s34;
64
65      u32 key0,key1,key2;
66      u32 iv0,iv1,iv2;
67
68      /* read key and iv */
69      /* assuming the 4-byte aligned key/iv is 0'ed out if not a multiple of 4-bytes */
70      key0 = (key_size>0 )?__mem(keys,tID,0,nr_streams):0;
71      key1 = (key_size>32)?__mem(keys,tID,1,nr_streams):0;
72      key2 = (key_size>64)?__mem(keys,tID,2,nr_streams):0;
73
74      iv0 = (iv_size>0 )?__mem(ivs,tID,0,nr_streams):0;
75      iv1 = (iv_size>32)?__mem(ivs,tID,1,nr_streams):0;
76      iv2 = (iv_size>64)?__mem(ivs,tID,2,nr_streams):0;
77
78
79      /* load key and iv */
80      S(1,1)=key0;
81      S(1,2)=key1;
82      S(1,3)=key2&0xffff;
83
84      S(2,1)=iv0;
85      S(2,2)=iv1;
86      S(2,3)=iv2&0xffff;
87
88      S(3,1)=0;
89      S(3,2)=0;
90      S(3,3)=0;
91      S(3,4)=0x00007000;
92
93  #define Z(w)
94      for(int i = 0; i < 4 * 9; ++i) {
95          u32 t1, t2, t3;
```

```
96
97          UPDATE();
98          ROTATE();
99      }
100
101     STORE(g_s);
102
103
104 }
105
106 __global__ void Trivium_process_bytes(gSTREAM_action act, u32* g_s,
107                                        u32 *buff, u32 nr_words) {
108     u32 tID=blockIdx.x*blockDim.x+threadIdx.x;
109     u32 nr_streams=blockDim.x*gridDim.x;
110
111     u32 s11, s12, s13;
112     u32 s21, s22, s23;
113     u32 s31, s32, s33, s34;
114
115     LOAD(g_s);
116
117 #undef Z
118 #define Z(w) (output_word ^= (w))
119     for(int w=0;w<nr_words;w++) {
120         u32 t1, t2, t3;
121         u32 output_word=0;
122
123         if(act!=GEN_KEYSTREAM) {
124             output_word=__mem(buff,tID,w,nr_streams);
125         }
126
127         UPDATE();
128         ROTATE();
129
130         __mem(buff,tID,w,nr_streams)=CH_ENDIANESS32(output_word);
131     }
132
133     STORE(g_s);
134
135 }
136 #endif
```

Trivium_test.cpp:

```
1 #include "gSTREAM.h"
```

```
2  #include "gSTREAM_test.h"
3
4  int main(void) {
5      do_test(0
6              ,2,128,680
7              ,GEN_KEYSTREAM,10,10
8              ,1024);
9      return 0;
10 }
```

Appendix E

XOR-Shift RNG Implementation

Below we present the implementation of Marsaglia's XOR-Shift 32-bit random number generator [77].

xsr_rng.h:

```
#ifndef __XSR_RNG_H__
#define __XSR_RNG_H__
#include <stdint.h>

void xsr_srand_u32(uint32_t seed);
uint32_t xsr_rand_u32(void);

void xsr_srand_u64(uint64_t seed);
uint64_t xsr_rand_u64(void);
int rand_int(int a, int b);

#define inline_xsr_def_u32() \
    u32 __inline_xsr_seed_u32;

#define inline_xsr_srand_u32(seed) \
    { __inline_xsr_seed_u32=seed; }

#define xor_lsh(s,f) \
    ((s)^((s)<<(f)))

#define xor_rsh(s,f) \
```

```
    ((s)^((s)>>(f)))

#define inline_xsr_rand_u32() \
    __inline_xsr_seed_u32= \
    xor_lsh(xor_rsh(xor_lsh(__inline_xsr_seed_u32,13),17),5)

#define inline_rand_int(a,b) \
        (a)+((inline_xsr_rand_u32())%((b)-(a)+1));

#endif
```

xsr_rng.c:

```
#include "xsr_rng.h"
#include <stdio.h>


/* 32-bit version: */

static uint32_t __xsr_seed_u32=2463534242;

void xsr_srand_u32(uint32_t seed) { __xsr_seed_u32=seed; }

uint32_t xsr_rand_u32(void) {
    __xsr_seed_u32^=(__xsr_seed_u32<<13);
    __xsr_seed_u32^=(__xsr_seed_u32>>17);
    __xsr_seed_u32^=(__xsr_seed_u32<<5);
    return __xsr_seed_u32;

}

/* 64-bit version: */

static uint64_t __xsr_seed_u64=88172645463325252LL;

void xsr_srand_u64(uint64_t seed) { __xsr_seed_u64=seed; }

uint64_t xsr_rand_u64(void){
    __xsr_seed_u64^=(__xsr_seed_u64<<13);
    __xsr_seed_u64^=(__xsr_seed_u64>>7);
    __xsr_seed_u64^=(__xsr_seed_u64<<17);
    return __xsr_seed_u64;
}
/* random integer [a,b] */
int rand_int(int a, int b) {
```

```
33      if(sizeof(int)==sizeof(uint64_t)) {
34          return a+(xsr_rand_u64()%(b-a+1));
35      } else {
36          return a+(xsr_rand_u32()%(b-a+1));
37      }
38  }
```

Appendix F

Differential Trails

In Chapter 7 we presented the complexity of numerous differential trails for both CubeHash and BLAKE. Though some explicit trails were given and discussed in previous chapters, for completeness and as a reference we present all the differential trails in this appendix.

F.1 BLAKE differential trails

In this section we provide the differential trails for some of the paths corresponding to second preimage attacks on round-reduced, toy variants of BLAKE, as introduced in Table 7.3, and Table 7.4. Note that in this section we use $/r$ to denote the number of rounds.

BLAZE32/1

For Δ, with $1/p_\Delta = 2^{70}$:

$\Delta =$ `0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,`
`0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,`
`0x81, 0x00, 0x81, 0x00, 0x10, 0x19, 0x10, 0x19, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,`
`0x90, 0x11, 0x90, 0x11, 0x01, 0x08, 0x01, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,`
`0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,`
`0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,`
`0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,`
`0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00`

BRAKE32/1

For Δ, with $1/p_\Delta = 2^{70}$:

$\Delta =$ `0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,`
`0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,`
`0x81, 0x00, 0x81, 0x00, 0x10, 0x19, 0x10, 0x19, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,`
`0x90, 0x11, 0x90, 0x11, 0x01, 0x08, 0x01, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,`
`0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,`
`0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,`
`0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,`
`0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00`

BLAZE64/1

For Δ, with $1/p_\Delta = 2^{107}$:

$\Delta =$ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x80, 0x80, 0x80, 0x80, 0x80, 0x80, 0x80, 0x80, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

BRAKE64/1

For Δ, with $1/p_\Delta = 2^{429}$:

```
Δ = 0x00,0x40,0x80,0x00,0x00,0x40,0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x20,0x95,0x0a,0x91,0x48,0x80,0xff,0x89,0x70,0x25,0xb7,0x39,0x78,
    0x00,0x00,0x00,0x40,0x20,0x50,0x20,0x10,0x20,0x02,0x18,0x84,0x00,0x12,0x38,0x94,
    0x81,0x00,0x85,0x22,0x90,0x48,0x90,0x08,0x98,0x18,0x30,0x97,0xbc,0x12,0x01,0xdd,
    0x00,0x00,0x20,0x10,0x20,0x50,0x00,0x40,0x38,0xc0,0x28,0x06,0x18,0xd0,0x08,0x16,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
```

F.2 CubeHash-∗/{10,20,24,36,96} differential trails

Below we provide the differential tails (including their corresponding erasing blocks) for the paths introduced in Table 7.8. As it is of little interest to present the differentials for paths that are not considered attacks, we only present paths for which the raw-probability $p_\Delta > 2^{-512}$.

CubeHash-1/10

For $\Delta = \Delta_0 \| \cdots \| \Delta_4$, $1/p_\Delta = 2^{32}$:

$$\begin{aligned}
\Delta_0 &= \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x01}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_1 &= \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_2 &= \texttt{0x40}, \texttt{0x40}, \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_3 &= \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_4 &= \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}
\end{aligned}$$

CubeHash-2/10

For $\Delta = \Delta_0 \| \Delta_1 \| \Delta_2$, $1/p_\Delta = 2^{32}$:

$$\begin{aligned}
\Delta_0 &= \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x01}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_1 &= \texttt{0x40}, \texttt{0x40}, \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_2 &= \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}
\end{aligned}$$

CubeHash-3/10

For $\Delta = \Delta_0 \| \cdots \| \Delta_4$, $1/p_\Delta = 2^{478}$:

$$\begin{aligned}
\Delta_0 &= \texttt{0x00}, \texttt{0x00}, \texttt{0x01}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_1 &= \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_2 &= \texttt{0x41}, \texttt{0x40}, \texttt{0x05}, \texttt{0x15}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_3 &= \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_4 &= \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}
\end{aligned}$$

CubeHash-4/10

For $\Delta = \Delta_0 \| \Delta_1 \| \Delta_2$, $1/p_\Delta = 2^{189}$:

$$\begin{aligned}
\Delta_0 = &\ \texttt{0x00}, \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_1 = &\ \texttt{0x01}, \texttt{0x00}, \texttt{0x01}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_2 = &\ \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}
\end{aligned}$$

CubeHash-6/10

For $\Delta = \Delta_0 \| \Delta_1 \| \Delta_2$, $1/p_\Delta = 2^{478}$:

$$\begin{aligned}
\Delta_0 = &\ \texttt{0x00}, \texttt{0x00}, \texttt{0x01}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_1 = &\ \texttt{0x41}, \texttt{0x40}, \texttt{0x05}, \texttt{0x15}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_2 = &\ \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}
\end{aligned}$$

CubeHash-1/20

For $\Delta = \Delta_0 \| \cdots \| \Delta_4$, $1/p_\Delta = 2^{30}$:

$$\begin{aligned}
\Delta_0 = &\ \texttt{0x40}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x40}, \texttt{0x00}, \\
&\ \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_1 = &\ \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
&\ \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_2 = &\ \texttt{0x00}, \texttt{0x04}, \texttt{0x10}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x04}, \\
&\ \texttt{0x10}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_3 = &\ \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
&\ \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_4 = &\ \texttt{0x00}, \texttt{0x04}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x04}, \\
&\ \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}
\end{aligned}$$

CubeHash-2/20

For $\Delta = \Delta_0 \| \Delta_1 \| \Delta_2$, $1/p_\Delta = 2^{30}$:

$$\begin{aligned}
\Delta_0 = \ & \texttt{0x40}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x40}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_1 = \ & \texttt{0x00}, \texttt{0x04}, \texttt{0x10}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x04}, \\
& \texttt{0x10}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_2 = \ & \texttt{0x00}, \texttt{0x04}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x04}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}
\end{aligned}$$

CubeHash-3/20

For $\Delta = \Delta_0 \| \cdots \| \Delta_4$, $1/p_\Delta = 2^{394}$:

$$\begin{aligned}
\Delta_0 = \ & \texttt{0x00}, \texttt{0x00}, \texttt{0x01}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x01}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_1 = \ & \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_2 = \ & \texttt{0x41}, \texttt{0x40}, \texttt{0x05}, \texttt{0x15}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x41}, \texttt{0x40}, \\
& \texttt{0x05}, \texttt{0x15}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_3 = \ & \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_4 = \ & \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}
\end{aligned}$$

CubeHash-4/20

For $\Delta = \Delta_0 \| \Delta_1 \| \Delta_2$, $1/p_\Delta = 2^{156}$:

$$\begin{aligned}
\Delta_0 = \ & \texttt{0x04}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x04}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_1 = \ & \texttt{0x00}, \texttt{0x44}, \texttt{0x00}, \texttt{0x40}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x44}, \\
& \texttt{0x00}, \texttt{0x40}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_2 = \ & \texttt{0x00}, \texttt{0x04}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x04}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}
\end{aligned}$$

CubeHash-6/20

For $\Delta = \Delta_0 \| \Delta_1 \| \Delta_2$, $1/p_\Delta = 2^{394}$:

$$\begin{aligned}
\Delta_0 = \ & \texttt{0x00}, \texttt{0x00}, \texttt{0x01}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x01}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_1 = \ & \texttt{0x41}, \texttt{0x40}, \texttt{0x05}, \texttt{0x15}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x41}, \texttt{0x40}, \\
& \texttt{0x05}, \texttt{0x15}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_2 = \ & \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}
\end{aligned}$$

CubeHash-1/24

For $\Delta = \Delta_0 \| \cdots \| \Delta_4$, $1/p_\Delta = 2^{30}$:

$$\begin{aligned}
\Delta_0 = \ & \texttt{0x40}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x40}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_1 = \ & \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_2 = \ & \texttt{0x00}, \texttt{0x04}, \texttt{0x10}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x04}, \texttt{0x10}, \texttt{0x10}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_3 = \ & \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_4 = \ & \texttt{0x00}, \texttt{0x04}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x04}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}
\end{aligned}$$

CubeHash-2/24

For $\Delta = \Delta_0 \| \Delta_1 \| \Delta_2$, $1/p_\Delta = 2^{30}$:

$$\begin{aligned}
\Delta_0 = \ & \texttt{0x40}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x40}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_1 = \ & \texttt{0x00}, \texttt{0x04}, \texttt{0x10}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x04}, \texttt{0x10}, \texttt{0x10}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_2 = \ & \texttt{0x00}, \texttt{0x04}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x04}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}
\end{aligned}$$

CubeHash-3/24

For $\Delta = \Delta_0 \| \cdots \| \Delta_4$, $1/p_\Delta = 2^{394}$:

$$\begin{aligned}
\Delta_0 = {} & \texttt{0x00}, \texttt{0x00}, \texttt{0x01}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x01}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_1 = {} & \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_2 = {} & \texttt{0x41}, \texttt{0x40}, \texttt{0x05}, \texttt{0x15}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x41}, \texttt{0x40}, \texttt{0x05}, \texttt{0x15}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_3 = {} & \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_4 = {} & \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x10}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}
\end{aligned}$$

CubeHash-4/24

For $\Delta = \Delta_0 \| \Delta_1 \| \Delta_2$, $1/p_\Delta = 2^{156}$:

$$\begin{aligned}
\Delta_0 = {} & \texttt{0x04}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x04}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_1 = {} & \texttt{0x00}, \texttt{0x44}, \texttt{0x00}, \texttt{0x40}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x44}, \texttt{0x00}, \texttt{0x40}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_2 = {} & \texttt{0x00}, \texttt{0x04}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x04}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}
\end{aligned}$$

CubeHash-6/24

For $\Delta = \Delta_0 \| \Delta_1 \| \Delta_2$, $1/p_\Delta = 2^{394}$:

$$\begin{aligned}
\Delta_0 = \ & \texttt{0x00}, \texttt{0x00}, \texttt{0x01}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x01}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_1 = \ & \texttt{0x41}, \texttt{0x40}, \texttt{0x05}, \texttt{0x15}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x41}, \texttt{0x40}, \texttt{0x05}, \texttt{0x15}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_2 = \ & \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x10}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}
\end{aligned}$$

CubeHash-1/36

For $\Delta = \Delta_0 \| \cdots \| \Delta_3$, $1/p_\Delta = 2^{13}$:

$$\begin{aligned}
\Delta_0 = \ & \texttt{0x00}, \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x80}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_1 = \ & \texttt{0x00}, \texttt{0x04}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x10}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x80}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x80}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x02}, \texttt{0x08} \\
\Delta_2 = \ & \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x04}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x80}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x80}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x80}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x80}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\
\Delta_3 = \ & \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\
& \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x02}, \texttt{0x00}
\end{aligned}$$

CubeHash-2/36

For $\Delta = \Delta_0 \| \Delta_1 \| \Delta_2$, $1/p_\Delta = 2^{28}$:

$\Delta_0 =$ 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01

$\Delta_1 =$ 0x80, 0x00, 0x20, 0x80, 0x00, 0x00, 0x00, 0x00, 0x80, 0x00, 0x20, 0x80,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x40, 0x00, 0x10

$\Delta_2 =$ 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10

CubeHash-3/36

For $\Delta = \Delta_0 \| \cdots \| \Delta_3$, $1/p_\Delta = 2^{343}$:

$\Delta_0 =$ 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

$\Delta_1 =$ 0x00, 0x40, 0x01, 0x11, 0x40, 0x00, 0x01, 0x01, 0x40, 0x00, 0x01, 0x01,
0x40, 0x00, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x08, 0x00, 0x02, 0x08,
0x08, 0x00, 0x02, 0x08, 0x00, 0x00, 0x00, 0x00, 0x08, 0x00, 0xa0, 0x00

$\Delta_2 =$ 0x00, 0x04, 0x10, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x11, 0x00, 0x10, 0x10, 0x80, 0x02, 0x02, 0x82, 0x00, 0x02, 0x22, 0x02,
0x00, 0x02, 0x22, 0x02, 0x00, 0x02, 0x22, 0x02, 0x00, 0x00, 0x00, 0x00

$\Delta_3 =$ 0x01, 0x04, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00,
0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00,
0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02

CubeHash-4/36

For $\Delta = \Delta_0 \| \Delta_1 \| \Delta_2$, $1/p_\Delta = 2^{130}$:

$\Delta_0 =$ 0x00, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00

$\Delta_1 =$ 0x00, 0x02, 0x20, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x20, 0x02,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x01, 0x10

$\Delta_2 =$ 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10

CubeHash-5/36

For $\Delta = \Delta_0 \| \cdots \| \Delta_4$, $1/p_\Delta = 2^{965}$, but $c_\Delta < 2^{512}$:

$\Delta_0 =$ 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00

$\Delta_1 =$ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

$\Delta_2 =$ 0x28, 0xa0, 0x02, 0x88, 0x00, 0x00, 0x00, 0x00, 0x28, 0xa0, 0x02, 0x88,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x44, 0x14, 0x50, 0x01

$\Delta_3 =$ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

$\Delta_4 =$ 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10

CubeHash-6/36

For $\Delta = \Delta_0 \| \Delta_1 \| \Delta_2$, $1/p_\Delta = 2^{309}$:

$\Delta_0 =$ 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00

$\Delta_1 =$ 0x80, 0x0a, 0x2a, 0x82, 0x00, 0x00, 0x00, 0x00, 0x80, 0x0a, 0x2a, 0x82,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x41, 0x40, 0x05, 0x15

$\Delta_2 =$ 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10

CubeHash-8/36

For $\Delta = \Delta_0 \| \Delta_1 \| \Delta_2$, $1/p_\Delta = 2^{637}$, but $c_\Delta < 2^{512}$:

$\Delta_0 =$ 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00

$\Delta_1 =$ 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10

$\Delta_2 =$ 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10

CubeHash-1/96

For $\Delta = \Delta_0$, $1/p_\Delta = 2^0$:

$\Delta_0 =$ 0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

$\Delta_1 =$ 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x08, 0x00, 0x20, 0x01,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00,
0x08, 0x00, 0x20, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01

CubeHash-2/96

For $\Delta = \Delta_0$, $1/p_\Delta = 2^{18}$:

$\Delta_0 =$ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x02, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x04, 0x10, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x80,
0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

$\Delta_1 =$ 0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x80, 0x00, 0x00, 0x00, 0x80,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x80,
0x00, 0x00, 0x00, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

CubeHash-3/96

For $\Delta = \Delta_0 \| \Delta_1$, $1/p_\Delta = 2^{54}$:

$\Delta_0 =$ 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00,
0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

$\Delta_1 =$ 0x04, 0x04, 0x00, 0x01, 0x04, 0x04, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x88, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x80, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x10,
0x00, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x04, 0x04, 0x00, 0x01,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x80,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x80, 0x00, 0x00, 0x00, 0x00

CubeHash-4/96

For $\Delta = \Delta_0 \| \Delta_1$, $1/p_\Delta = 2^{40}$:

$\Delta_0 =$ 0x40, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x10, 0x00, 0x40, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x02, 0x80, 0x00, 0x00, 0x00, 0x00

$\Delta_1 =$ 0x04, 0x00, 0x00, 0x01, 0x04, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x08, 0x20, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10,
0x00, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x01,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

CubeHash-5/96

For $\Delta = \Delta_0 \| \Delta_1$, $1/p_\Delta = 2^{127}$:

$\Delta_0 =$ 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00,
0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

$\Delta_1 =$ 0x01, 0x00, 0x01, 0x11, 0x01, 0x00, 0x01, 0x11, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x80, 0x08, 0x00, 0x2a, 0x00, 0x00, 0x00, 0x00,
0x08, 0x00, 0x00, 0x22, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x11, 0x04, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x01, 0x01, 0x01, 0x00, 0x01, 0x11,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x22, 0x08,
0x00, 0x00, 0x00, 0x00, 0x08, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00

CubeHash-6/96

For $\Delta = \Delta_0 \| \Delta_1$, $1/p_\Delta = 2^{93}$:

$\Delta_0 =$ 0x40, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x10, 0x00, 0x40, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x02, 0x80, 0x00, 0x00, 0x00, 0x00

$\Delta_1 =$ 0x00, 0x40, 0x01, 0x10, 0x00, 0x40, 0x01, 0x10, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x88, 0x08, 0x00, 0x08, 0x00, 0x00, 0x00, 0x00,
0x08, 0x08, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x04, 0x04, 0x00,
0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x01, 0x00, 0x00, 0x40, 0x01, 0x10,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x08, 0x00, 0x02, 0x08,
0x00, 0x00, 0x00, 0x00, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

CubeHash-7/96

For $\Delta = \Delta_0 \| \Delta_1$, $1/p_\Delta = 2^{251}$:

$\Delta_0 =$ 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00,
0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

$\Delta_1 =$ 0x51, 0x44, 0x54, 0x40, 0x51, 0x44, 0x54, 0x40, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x0a, 0x28, 0x82, 0x00, 0x00, 0x00, 0x00,
0x02, 0x80, 0x0a, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10, 0x01, 0x40, 0x05,
0x00, 0x00, 0x00, 0x00, 0x40, 0x44, 0x00, 0x40, 0x51, 0x44, 0x54, 0x40,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x80, 0x0a, 0x20, 0x02,
0x00, 0x00, 0x00, 0x00, 0x80, 0x08, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00

CubeHash-8/96

For $\Delta = \Delta_0 \| \Delta_1$, $1/p_\Delta = 2^{191}$:

$\Delta_0 =$ 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x80, 0x00, 0x00, 0x00, 0x80, 0x00, 0x00, 0x02, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x04, 0x00, 0x01, 0x04, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x80, 0x00, 0x00, 0x00, 0x80, 0x00

$\Delta_1 =$ 0x04, 0x44, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x45, 0x44, 0x05, 0x14,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x20, 0x80, 0x22, 0x08, 0x20, 0x80, 0x22, 0x08, 0x00, 0x02, 0x22, 0x02,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x04, 0x04, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00,
0x44, 0x04, 0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0xa0, 0x00, 0x08, 0x20, 0x80, 0x22, 0x08

CubeHash-9/96

For $\Delta = \Delta_0 \| \Delta_1$, $1/p_\Delta = 2^{428}$:

$\Delta_0 =$ 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00,
0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

$\Delta_1 =$ 0x00, 0x00, 0x08, 0x88, 0x00, 0x00, 0x08, 0x88, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x41, 0x41, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00,
0x01, 0x45, 0x10, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xa0, 0x80, 0x22, 0x08,
0x00, 0x00, 0x00, 0x00, 0x0a, 0x28, 0x8a, 0x88, 0x00, 0x00, 0x08, 0x88,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x45, 0x10, 0x40, 0x01,
0x00, 0x00, 0x00, 0x00, 0x01, 0x44, 0x00, 0x50, 0x00, 0x00, 0x00, 0x00

CubeHash-10/96

For $\Delta = \Delta_0 \| \Delta_1$, $1/p_\Delta = 2^{360}$:

$\Delta_0 =$ 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x80, 0x00, 0x00, 0x00, 0x80, 0x00, 0x00, 0x02, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x04, 0x00, 0x01, 0x04, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x80, 0x00, 0x00, 0x00, 0x80, 0x00

$\Delta_1 =$ 0x51, 0x01, 0x45, 0x11, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x11,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x80, 0x08, 0x20, 0x2a, 0x80, 0x08, 0x20, 0x2a, 0x88, 0xa8, 0x80, 0xa2,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x01, 0x11, 0x00, 0x00, 0x00, 0x00,
0x01, 0x45, 0x11, 0x51, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x08, 0x02, 0x08, 0xa0, 0x80, 0x08, 0x20, 0x2a

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 2nd edition, 2006.

[2] AMD. ATI CTM Reference Guide. Technical Reference Manual, 2006.

[3] J. Aumasson, I. Dinur, W. Meier, and A. Shamir. Cube testers and key recovery attacks on reduced-round MD6 and Trivium. In *Fast Software Encryption*, pages 1–22. Springer, 2009.

[4] J. Aumasson, S. Fischer, S. Khazaei, W. Meier, and C. Rechberger. New features of Latin dances: analysis of Salsa, ChaCha, and Rumba. In *Fast Software Encryption*, pages 470–488. Springer, 2008.

[5] J. Aumasson, J. Guo, S. Knellwolf, K. Matusiewicz, and W. Meier. Differential and invertibility properties of BLAKE. In *Fast Software Encryption*, pages 318–332. Springer, 2010.

[6] J. Aumasson, W. Meier, and R. Phan. The hash function family LAKE. In *Fast Software Encryption*, pages 36–53. Springer, 2008.

[7] J. Aumasson, W. Meier, and R. Phan. Toy versions of BLAKE. See http://www.131002.net/blake/toyblake.pdf, 2008.

[8] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan. SHA-3 proposal BLAKE, 2008.

[9] S. Babbage, C. De Cannière, A. Canteaut, C. Cid, H. Gilbert, T. Johansson, M. Parker, B. Preneel, V. Rijmen, and M. Robshaw. The eSTREAM portfolio. *eSTREAM, ECRYPT Stream Cipher Project*, 2008.

[10] S. Babbage, C. De Cannière, A. Canteaut, C. Cid, H. Gilbert, T. Johansson, M. Parker, B. Preneel, V. Rijmen, and M. Robshaw. The eSTREAM Portfolio (rev. 1), 2008.

[11] S. Babbage and M. Dodd. The stream cipher MICKEY-128 2.0. *ECRYPT Stream Cipher Project*, 2006.

[12] M. Bellare, D. Coppersmith, J. Hastad, M. Kiwi, and M. Sudan. Linearity testing in characteristic two. *IEEE Transactions on Information Theory*, 42(6 Part 1):1781–1795, 1996.

[13] R. Benadjila, O. Billet, H. Gilbert, G. Macario-Rat, T. Peyrin, M. Robshaw, and Y. Seurin. SHA-3 Proposal: ECHO, 2009.

[14] R. Benadjila, O. Billet, S. Gueron, and M. J. B. Robshaw. The Intel AES instructions set and the SHA-3 candidates. In *Asiacrypt 2009*, volume 5912 of *LNCS*, pages 162–178, 2009.

[15] C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, et al. Sosemanuk, a fast software-oriented stream cipher. *New Stream Cipher Designs*, pages 98–118, 2008.

[16] D. Bernstein. The Salsa20 family of stream ciphers. *New Stream Cipher Designs*, pages 84–97, 2008.

[17] D. Bernstein. ChaCha, a variant of Salsa20. See http://cr.yp.to/chacha.html, 2009.

[18] D. J. Bernstein. Cubehash. Submission to NIST, 2008.

[19] D. J. Bernstein. Cubehash. Submission to NIST (Round 2), 2009.

[20] D. J. Bernstein. CubeHash specification (2.B.1), 2009.

[21] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Keccak specifications, 2009.

[22] M. Bevand. MD5 Chosen-Prefix Collisions on GPUs. Black Hat, 2009. Whitepaper.

[23] E. Biham and R. Chen. Near-Collisions of SHA-0. In *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 2004.

[24] E. Biham and O. Dunkelman. A framework for iterative hash functions–HAIFA. In *Second NIST Cryptographic Hash Workshop*. Citeseer, 2006.

[25] E. Biham and O. Dunkelman. The SHAvite-3 Hash Function, 2009.

[26] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. *Journal of CRYPTOLOGY*, 4(1):3–72, 1991.

[27] A. Biryukov, A. Shamir, and D. Wagner. Real Time Cryptanalysis of A5/1 on a PC. In *Fast Software Encryption*, pages 37–44. Springer, 2000.

[28] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 73–83. ACM, 1990.

[29] D. Blythe. The Direct3D 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[30] O. A. R. Board. OpenMP Application Program Interface: Version 3.0. The OpenMP API Specification for Parallel Programming, May 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[31] M. Boesgaard, M. Vesterager, T. Christensen, and E. Zenner. The Stream Cipher Rabbit. *ECRYPT Stream Cipher Project Report*, 6, 2005.

[32] J. W. Bos and D. Stefan. Performance analysis of the SHA-3 candidates on exotic multi-core architectures. In *Workshop on Cryptographic Hardware and Embedded Systems, CHES*, volume 6225 of *LNCS*, pages 279–293. Springer, August 2010.

[33] E. Bresson, A. Canteaut, B. Chevallier-Mames, C. Clavier, T. Fuhr, A. Gouget, T. Icart, J.-F. Misarsky, M. Naya-Plasencia, P. Paillier, T. Pornin, J.-R. Reinhard, C. Thuillet, and M. Videau. The Hash Function Shabal, 2008.

[34] E. Brier, S. Khazaei, W. Meier, and T. Peyrin. Linearization Framework for Collision Attacks: Application to CubeHash and MD6. In *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 560–577. Springer, 2009.

[35] E. Brier, S. Khazaei, W. Meier, and T. Peyrin. Linearization Framework for Collision Attacks: Application to CubeHash and MD6 (extended version). Cryptology ePrint Archive, Report 2009/382, 2009. http://eprint.iacr.org.

[36] C. D. Cannière and C. Rechberger. Finding SHA-1 Characteristics: General Results and Applications. In *Advances in Cryptology – ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[37] C. D. Canniere, H. Sato, and D. Watanabe. Hash Function Luffa, 2009.

[38] F. Chabaud and A. Joux. Differential collisions in SHA-0. In *Advances in Cryptology – CRYPTO 98*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 1998.

[39] B. Chapman, G. Jost, R. Van der Pas, and D. Kuck. *Using OpenMP: portable shared memory parallel programming*. The MIT Press, 2007.

[40] J. Daemen and V. Rijmen. *The design of Rijndael*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2002.

[41] C. De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. *Information Security*, pages 171–186, 2006.

[42] C. De Cannière and B. Preneel. Trivium specifications. *eSTREAM, ECRYPT Stream Cipher Project*, 2006.

[43] C. De Cannière and B. Preneel. Trivium. *New Stream Cipher Designs*, pages 244–266, 2008.

[44] C. De Cannière and C. Rechberger. Finding SHA-1 characteristics: General results and applications. *Advances in Cryptology–ASIACRYPT 2006*, pages 1–20, 2006.

[45] I. Dinur and A. Shamir. Cube attacks on tweakable black box polynomials. Cryptology ePrint Archive, Report 2008/385, 2008. http://eprint.iacr.org/.

[46] I. Dinur and A. Shamir. Cube attacks on tweakable black box polynomials. *Advances in Cryptology–EUROCRYPT 2009*, pages 278–299, 2009.

[47] ECRYPT. The eSTREAM Project. http://www.ecrypt.eu.org/stream/, 2008.

[48] H. Englund, T. Johansson, and M. Sönmez Turan. A framework for chosen IV statistical analysis of stream ciphers. *Progress in Cryptology–INDOCRYPT 2007*, pages 268–281, 2007.

[49] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein Hash Function Family, 2009.

[50] E. Ferro and F. Potorti. Bluetooth and Wi-Fi wireless protocols: a survey and a comparison. *IEEE Wireless Communications*, 12(1):12–26, 2005.

[51] E. Filiol. A new statistical testing for symmetric ciphers and hash functions. *Information and Communications Security*, pages 342–353, 2002.

[52] S. Fischer, S. Khazaei, and W. Meier. Chosen IV statistical analysis for key recovery attacks on stream ciphers. *Progress in Cryptology–AFRICACRYPT 2008*, pages 236–245, 2008.

[53] M. P. I. Forum. MPI: A Message-Passing Interface Standard, Version 2.2. MPI Documents, September 2009. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

[54] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen. Grøstl – a SHA-3 candidate, 2008.

[55] D. Gligoroski, V. Klima, S. J. Knapskog, M. El-Hadedy, J. Amundsen, and S. F. Mjolsnes. Cryptographic Hash Function BLUE MIDNIGHT WISH, 2009.

[56] S. Halevi, W. E. Hall, and C. S. Jutla. The Hash Function Fugue, 2009.

[57] S. Harbison and G. Steele. *C, a reference manual*. Prentice Hall, 2002.

[58] O. Harrison and J. Waldron. Practical Symmetric Key Cryptography on Modern Graphics Hardware. In *USENIX Security Symposium*, pages 195–210, 2008.

[59] M. Hell, T. Johansson, and W. Meier. Grain - a stream cipher for constrained environments. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/010, 2005. http://www.ecrypt.eu.org/stream.

[60] M. Hell, T. Johansson, and W. Meier. Grain: a stream cipher for constrained environments. *International Journal of Wireless and Mobile Computing*, 2(1):86–93, 2007.

[61] J. Hennessy, D. Patterson, D. Goldberg, and K. Asanovic. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2003.

[62] J. Hoeflinger. Extending OpenMP to clusters. *White Paper, Intel Corporation*, 2006.

[63] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Proceedings of the 2010 IEEE International Solid-State Circuits Conference*. IEEE, 2010.

[64] S. Indesteege and B. Preneel. Practical Collisions for EnRUPT. In *FSE 2009*, volume 5665 of *Lecture Notes in Computer Science*, pages 246–259. Springer, 2009.

[65] A. Kaminsky. Parallel Cube Tester Analysis of the CubeHash One-Way Hash Function. In *4th SIAM Conference on Parallel Processing for Scientific Computing (PP10)*, 2010.

[66] S. Khazaei. *Neutrality-Based Symmetric Cryptanalysis*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2010.

[67] S. Khazaei, S. Knellwolf, W. Meier, and D. Stefan. Improved linear differential attacks on CubeHash. In *Africacrypt 2010*, LNCS. Springer, 2010. To appear.

[68] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[69] V. Klima. Tunnels in hash functions: Md5 collisions within a minute. Cryptology ePrint Archive, Report 2006/105, 2006. http://eprint.iacr.org/.

[70] O. Küçük. The Hash Function Hamsi, 2009.

[71] J. Lathrop. Cube attacks on cryptographic hash functions, 2009.

[72] A. Lenstra, X. Wang, and B. de Weger. Colliding X. 509 Certificates. In *Proceedings of ACISP*, 2005.

[73] G. Leurent, C. Bouillaguet, and P.-A. Fouque. SIMD Is a Message Digest, 2009.

[74] H. Lipmaa and S. Moriai. Efficient algorithms for computing differential properties of addition. In *Fast Software Encryption*, pages 35–45. Springer, 2001.

[75] S. A. Manavski. CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. In *ICSPC 2007*, pages 65–68. IEEE, November 2007.

[76] S. Marechal. Advances in password cracking. *Journal in Computer Virology*, 4(1):73–81, 2008.

[77] G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8:1–6, 2003.

[78] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[79] A. Maximov and A. Biryukov. Two trivial attacks on trivium. In *Selected Areas in Cryptography*, pages 36–55. Springer, 2007.

[80] G. Moore et al. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[81] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *Solid-State Circuits Newsletter, IEEE*, 20(3):33 –35, September 2006.

[82] P. Mroczkowski and J. Szmidt. The Cube Attack on Stream Cipher Trivium and Quadraticity Tests. *Rump Session. CRYPTO*, 2010.

[83] S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[84] A. Munshi. The OpenCL Specification. *Khronos OpenCL Working Group*, 2009.

[85] Y. Naito, Y. Sasaki, T. Shimoyama, J. Yajima, N. Kunihiro, and K. Ohta. Improved Collision Search for SHA-0. In *Advances in Cryptology – ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2006.

[86] National Institute of Standards and Technology. Secure hash standard. FIPS 180-1, NIST, http://www.itl.nist.gov/fipspubs/fip180-1.htm, April 1995.

[87] National Institute of Standards and Technology. Secure hash standard. FIPS 180-2, NIST, http://www.itl.nist.gov/fipspubs/fip180-2.htm, August 2002.

[88] National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. Technical report, Department of Commerce, http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf, November 2007.

[89] National Institute of Standards and Technology. Cryptographic hash algorithm competition. http://csrc.nist.gov/groups/ST/hash/sha-3/index.html, 2008.

[90] National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard (AES), 2001. http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[91] NVIDIA. NVIDIA Compute. PTX: Parallel Thread Execution, March 2009.

[92] NVIDIA. NVIDIA CUDA Programming Guide 2.3, 2009.

[93] G. L. Osa. Fast Implementation of Two Hash Algorithms on nVidia CUDA GPU. Master's thesis, Norwegian University of Science and Technology, Norway, 2009.

[94] D. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. *Topics in Cryptology–CT-RSA 2006*, pages 1–20, 2006.

[95] D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright. Fast software AES encryption. In *FSE 2010*, volume 6147 of *LNCS*, pages 75–93, 2010.

[96] J. Owens. GPU architecture overview. In *SIGGRAPH 2007*, page 2. ACM, 2007.

[97] J. Parkhurst, J. Darringer, and B. Grundmann. From single core to multi-core: preparing for a new exponential. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, page 72. ACM, 2006.

[98] V. Podlozhnyuk. Parallel mersenne twister. *NVIDIA white paper*, 2007.

[99] N. Pramstaller, C. Rechberger, and V. Rijmen. Exploiting coding theory for collision attacks on SHA-1. In *Cryptography and Coding, IMA Int. Conf. 2005*, volume 3796 of *Lecture Notes in Computer Science*, pages 78–95. Springer, 2005.

[100] B. Preneel, A. Biryukov, C. De Cannière, S. B. Örs, E. Oswald, B. Van Rompay, L. Granboulan, E. Dottax, G. Martinet, S. Murphy, A. Dent, R. Shipsey, C. Swart, J. White, M. Dichtl, S. Pyka, M. Schafheutle, P. Serf, E. Biham, E. Barkan, Y. Braziler, O. Dunkelman, V. Furman, D. Kenigsberg, J. Stolin, J.-J. Quisquater, M. Ciet, F. Sica, H. Raddum, L. Knudsen, and M. Parker. Final report of European project IST-1999-12324:New European Schemes for Signatures, Integrity, and Encryption, 2004.

[101] V. Rijmen and E. Oswald. Update on SHA-1. In *Topics in Cryptology – CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 58–71. Springer, 2005.

[102] R. Rivest. The MD5 message-digest algorithm. RFC 1321, IETF, http://www.ietf.org/rfc/rfc1321.txt, April 1992.

[103] M. Saarinen. Chosen-IV statistical attacks on eStream ciphers. *SECRYPT*, pages 7–10, 2006.

[104] M. Segal and K. Akeley. The OpenGL graphics system: A specification (version 2.0). *Silicon Graphics, Mountain View, CA*, 2004.

[105] V. Shoup. NTL: A Library for doing Number Theory. Version 5.5.2. http://www.shoup.net/ntl.

[106] A. Sotirov, M. Stevens, J. Appelbaum, A. Lenstra, D. Molnar, D. Osvik, and B. de Weger. MD5 considered harmful today. In *Announced at the 25th Chaos Communication Congress. URL: http://www. win. tue. nl/hashclash/rogue-ca*, 2008.

[107] T. O. G. T. S. B. Specifications. Ieee std 1003.1c-1995. amendment 2: Threads. Technical report, IEEE, http://www.unix.org/version3/, 1995.

[108] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In *Crypto 2009*, volume 5677 of *LNCS*, pages 55–69, 2009.

[109] R. Szerwinski and T. Güneysu. Exploiting the power of GPUs for asymmetric cryptography. In *CHES 2008*, volume 5154 of *LNCS*, pages 79–99, 2008.

[110] I. UEA2&UIA. Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2& UIA2. Document 2: SNOW 3G Specifications. Version: 1.1. *ETSI/SAGE Specification*, 2006.

[111] M. Vielhaber. Breaking one.fivium by aida an algebraic iv differential attack. Cryptology ePrint Archive, Report 2007/413, 2007. http://eprint.iacr.org/.

[112] M. Vielhaber. AIDA Breaks BIVIUM (A&B) in 1 Minute Dual Core CPU Time. Cryptology ePrint Archive, Report 2009/402, 2009. http://eprint.iacr.org/.

[113] M. Vielhaber. Speeding up AIDA, the Algebraic IV Differential Attack, by the Fast Reed-Muller Transform. In *International Conference on Intelligent Systems and Knowledge Engineering, ISKE 2009*, 2009.

[114] X. Wang, D. Feng, X. Lai, and H. Yu. Collisions for hash functions md4, md5, haval-128 and ripemd. Cryptology ePrint Archive, Report 2004/199, 2004. http://eprint.iacr.org/.

[115] X. Wang, Y. Yin, and H. Yu. Finding collisions in the full SHA-1. In *Advances in Cryptology–CRYPTO 2005*, pages 17–36. Springer, 2005.

[116] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

[117] H. Wu. The stream cipher HC-128. *New Stream Cipher Designs*, pages 39–47, 2008.

[118] H. Wu. The Hash Function JH, 2009.

[119] J. Yang and J. Goodman. Symmetric Key Cryptography on Modern Graphics Hardware. In *Asiacrypt 2007*, volume 4833 of *LNCS*, pages 249–264, 2007.