8-1-2010

A Generic attack on CubeHash, a SHA-3 candidate

Philip Doughty Jr.

A Generic Attack on CubeHash, a SHA-3 Candidate

by

Philip C. Doughty, Jr.

A Thesis Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Science
in Computer Engineering

Supervised by

Prof. Marcin Łukowiak & Prof. Alan Kaminsky
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
August 2010

Approved by:

Prof. Marcin Łukowiak
Thesis Advisor, Department of Computer Engineering

Prof. Alan Kaminsky
Department of Computer Science

Prof. Stanisław P. Radziszowski
Department of Computer Science

Prof. Andres Kwasinski
Department of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

A Generic Attack on CubeHash, a SHA-3 Candidate

I, Philip C. Doughty, Jr., hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Philip C. Doughty, Jr.

Date

Dedication

To Kathryn, the love of my life.

Acknowledgments

I would like to acknowledge and thank my Lord and Savior, Jesus Christ for giving me perseverance. I would like to acknowledge Prof. Alan Kaminsky for helping me to shape the idea for this thesis. I would like to acknowledge Nicholas Palladino for his help with making the FPGA cluster work. And I would also like to acknowledge my wife, Kathryn, for help with creating the images.

Abstract

A Generic Attack on CubeHash, a SHA-3 Candidate

Philip C. Doughty, Jr.

Supervising Professors:

Prof. Marcin Łukowiak & Prof. Alan Kaminsky

A secure cryptographic hashing function should be resistant to three different scenarios: First, a cryptographic hashing function must be *preimage resistant*, that is, it should be infeasible for an attacker to construct a message such that it produces a known hash output value. Second, a cryptographic hashing function must be *second preimage resistant*, or it should be infeasible for an attacker to construct a message such that it has the same hash output value as another known message. Third, a cryptographic hashing function must be *collision resistant*, which means that it should be infeasible for an attacker to find any two different messages such that their hash output values are the same.

The current Secure Hash Algorithm (SHA) family, namely SHA-1 and SHA-2, were designed by the National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST). Recent advances in cryptanalysis of hash functions have led to concerns about the *collision resistance* in the SHA family. To address these concerns, NIST has opened a public worldwide competition known as the SHA-3 competition to find the new hash function, which will become SHA-3. Each candidate hash function is scrutinized by the public, and candidates with found weaknesses are dropped from advancing to the next rounds of the competition. The goal is that the strongest hash function will emerge at the end of the competition, and this hash function will be free for everyone to use.

This thesis implemented a generic attack against the *collision resistance* of small variants of one candidate in the SHA-3 competition, CubeHash.

A unique hash-chaining approach was used to find the collisions, and the parallelization of several FPGAs lead to parallelization measurements and analysis to see if a linear speedup could be obtained.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Often in computing it is necessary to convert an arbitrary piece of data into a fixed length, typically small, digest. The algorithm which takes an arbitrary piece of data and transforms it into a fixed length digest is usually referred to as a *hash function*. A hash function will always produce the same output if given the same input. It is usually desirable that the hash function has a high level of uniformity in mapping an input to an output; each possible output digest should have an equal probability of occurring. It is also desirable that the hash function have a high throughput; in other words, it is desirable for a hash function to be able to process a large amount of data in a short time.

Bit Index	**0 – 15**	**16 – 31**
0	Source Port Number	Destination Port Number
32	Length	*Checksum*
64	Data	

Table 1.1: UDP Packet Layout

One application of a hash function is the *checksum*. Checksums can be found in areas where it is possible for data to become corrupted. This data corruption can be found in almost any, if not all, transmissions over a physical medium. On the Internet, for example, there are a number of scenarios where data that is in transmission could have one or more bits flipped to the opposite value. User Data Protocol (UDP) packets [10], outlined in Table 1.1, utilize checksums to detect such corruption. In UDP packets, 16-bit words in the packet are added together in one's complement format to produce a sum. The one's complement of the sum is then stored in the UDP packet in the checksum field. The idea behind this checksum, is that it is

possible for a few bits in the packet to become flipped due to transmission error, but it is unlikely that both the bits in the packet and the bits in the checksum field to be flipped in such a way that when the receiver of the UDP packet computes the checksum, it will match.

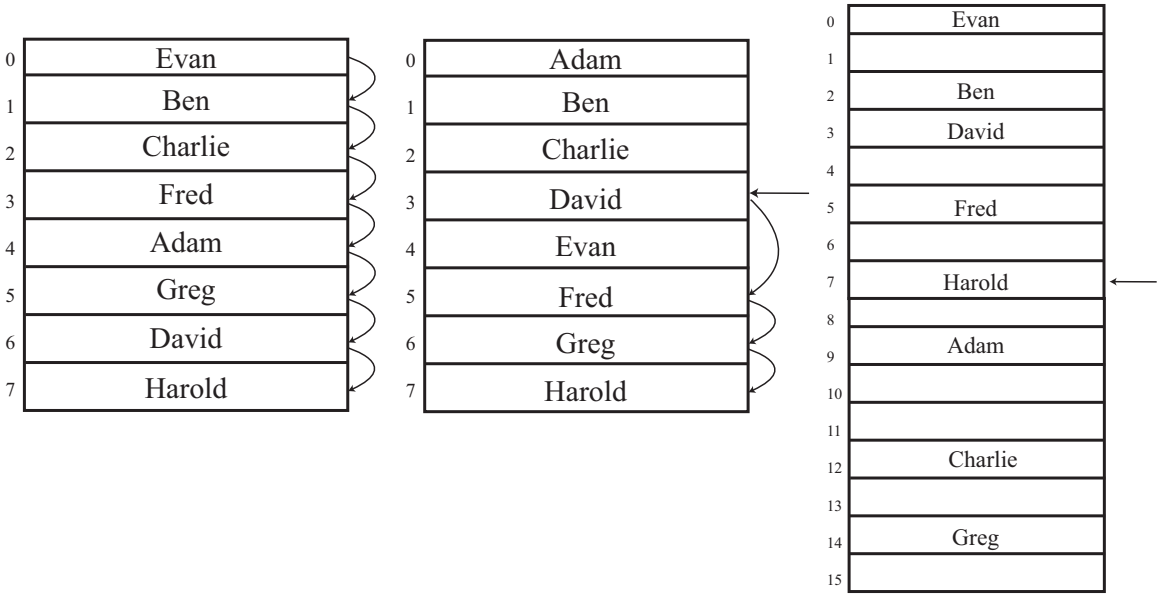

Figure 1.1: Illustration of Linear Search, Binary Search, and a Hash Table

Another use of the hash function is in *hash tables*. The hash table is a construct used to store a large amount of data efficiently. Data is typically stored as a Key-Value pair. The Key is a unique identifier, and the Value could be any arbitrary piece of data. For example, in a school database system, the Key could be the student's name, and the Value could be the student's entire academic history. If these Key-Value pairs were just stored in an unordered array, it would take a very long time to retrieve a student's academic history, because a query would have to go element by element in the array to see if it was the desired student. This algorithm would run in $O(n)$ time, where n is the number of students. This is shown on the left in Figure 1.1 when searching for the student *Harold*. If the Key-Value pairs were stored in an ordered array, it would provide a speedup because a query could then perform a binary search to find a student's academic history in $O(\log n)$ time. This is shown in the center of Figure 1.1; the students are sorted alphabetically by name, so it only takes at most 3 retries

to find any student's academic history. However, a greater speedup still can be obtained by using a hash table. In a hash table, elements are placed into array positions based on the hash digest of the Key, such that the Key with hash digest 0 would be the first element, the Key with hash digest 1 would be the second element, and so on. Since it takes a constant amount of time to compute the hash digest of each key, a query would run in $O(1)$ time, because once the hash digest is found, the index within the array can immediately be found. This is shown on the right in Figure 1.1. In this example, *Harold*'s name produces a hash digest of 7, which immediately points the query to the correct academic history. It should be noted that it is entirely possible for two Keys to produce the same hash digest, especially if modular arithmetic is used to wrap the digest around the size of the array. In practice, there are additional implementation details to handle this problem in hash tables. Figure 1.1 shows one of these details on the left, which is having the table partially empty.

The previous two examples of hash functions, *checksums* and *hash tables*, have alluded to a problem that can, and indeed must, occur in hash functions. That is the problem of a *collision*. A collision occurs when two different data inputs to a hash function produce the same output. Since the hash output is a fixed length, and the hash input could be any length, that means there are an infinite number of such collisions for each hash function. This may be acceptable for the two scenarios previously described, but there are many scenarios where this would have disastrous consequences.

A *cryptographic hash function* is a hash function that also provides three basic security properties. It must be *preimage resistant*, which means that it should be infeasible to construct input data that will match a known hash digest. Secondly, a cryptographic hash function must be *second preimage resistant*, which means that it should be infeasible to construct input data that will have the same hash digest as another known input data. And finally, a cryptographic hash function must be *collision resistant*, which means that it should be infeasible to find any two different messages such that their hash digests are the same.

One place where cryptographic hash functions are used is in storing passwords for user accounts on various operating systems. It would be insecure

to store a password in a file on the hard disk, because then if the hard disk is ever compromised to an attacker, the attacker can easily see what the password was. So, instead, most operating systems will only store a cryptographic hash digest of the password. When the user types his password to log in, the computer will compute the hash digest of that password and see if it matches the value stored on the disk. If it does match, then the user is permitted to log in. If the hash function used meets the requirements of a cryptographic hashing algorithm, then this is a more secure way to store the password on the hard disk. It should be infeasible to find a *preimage* for the hash digest stored on the disk, or it should be infeasible to find another password that, when processed through the cryptographic hash function, produces the same hash digest that is stored on the disk.

Another place where cryptographic hash functions are present is in places where data integrity could maliciously be modified. The previous example of a UDP packet used a checksum which was designed to help prevent accidental data integrity errors in transmission lines. However, in many cases there are attackers who will attempt to purposely corrupt data. One example of this could be a large file distribution center that has a number of volunteer mirrors. The main website could give copies of a file to the mirrors, and then publish the hash digest of the file on the main website and direct users to the mirrors for downloading. Users could then compute and check the hash digest of the files they receive to make sure the privately owned mirror did not alter the file, since it is infeasible for an attacker to produce a *preimage* or a data input that matches a hash digest. A similar mechanism is present in most peer-to-peer file sharing networks. It should be noted that if the publisher is typically from only one source, a digital signature approach is more secure than a cryptographic hash function alone.

One of the most commonly discussed places of importance for cryptographic hash functions is in conjunction with digital signatures. In public key cryptography, there are two keys involved. There is a public key which is shared with everyone and anyone, and there is a private key which the user keeps absolutely secret. In public key encryption, the public key is used to transform data such that only the private key can be used to decrypt

the data. Conversely, with digital signatures, the private key is used to transform data into a signature such that anyone with the public key can know that only the owner of the private key could have produced the signature. There is one unwanted side effect to many digital signature schemes which is that the signature is approximately the same length as the input data. For small amounts of data, this may be acceptable, but if the data being signed is a CD or DVD image, then a user downloading the data would then have to download an equally large signature just to check the data. A common compromise is that the data is processed through a cryptographic hash function and then the digital signature is taken from the hash digest, which is typically very small, to produce a signature which is also very small, and the signature is just as valid so long as the hash function is cryptographically secure. However, if the hash function is not cryptographically secure, then there could be disastrous effects with computing a digital signature on the hash digest.

This was the case with MD5 [11] in X.509 certificates [4], in an attack which yielded fraudulent certificates [7]. A common way for websites to secure communications with clients is to use X.509 certificates. In this scheme, each website that wishes to have secure communication with clients will generate a public and private key. The website will then generate a Certificate Signing Request (CSR) based on its public key and some other identifying information such as the URL of the website and the owner's name. The CSR is then passed to a Certificate Authority (CA). The CA's purpose is to investigate and ensure the identity of the website. Once the CA has determined that the identity is legitimate, the CA will then sign the CSR with the CA's own private key to generate a X.509 certificate which is then sent back to the website. The security is ensured based on the fact that most browsers come with several trusted public keys from various CAs, and it is infeasible for an attacker to determine a CA's private key given only the CA's public key. Whenever a user establishes a secure connection with a website, the website sends the user the X.509 certificate, and the user's browser automatically verifies the digital signature on the certificate using the known CA's public keys. If the signature does not match, then the

browser will warn the user of possible fraudulent behavior with the website in question. But, as mentioned previously, digital signatures typically rely on a cryptographic hash function. This is exactly the case with X.509 certificates. X.509 certificates originally used the MD5 cryptographic hash function, but MD5 was later found to lack a desirable level of collision resistance. Therefore, it was possible for an attacker to generate two different CSRs with two different, but valid, public keys and the same MD5 hash digest. The ideal attack would be for one CSR to be a legitimately verifiable CSR, while the other CSR would be for a fraudulent organization. The CA would then perform its due diligence on the legitimate CSR and generate a X.509 certificate. The fraudulent organization could then just copy and paste the digital signature from the legitimate X.509 certificate onto the fraudulent CSR to generate a new X.509 certificate which appeared to be signed by a CA, even though it was not, and no browser could detect the forged signature. Other cryptographic hash functions are now used by CAs to prevent this problem.

The current standard cryptographic hash function is SHA-1 [9]. SHA-1 has been under analysis for some time, because it is used heavily in several areas and it has a relatively short hash digest of 160 bits. With 160 bits, the theoretical number of computations required to find a collision is on the order of $O(2^{80})$ operations, but recently McDonald, Hawkes, and Pieprzyk [8] found that it is possible to find a collision in about $O(2^{52})$ hash operations. If collisions could easily be found in SHA-1, then there could be security problems similar to that of MD5.

There are several successors to SHA-1 known as the SHA-2 family, which claim to be more secure than SHA-1, but in the interest of prudence, the National Institute of Standards and Technology (NIST) initiated a global competition known as the Cryptographic Hash Algorithm Competition on November 2, 2007. In this competition, anyone was allowed to submit an algorithm by October 31, 2008 so long as that algorithm met a certain list of submission requirements. Throughout the course of the competition, the public is continually encouraged to examine each candidate to determine if there are any security vulnerabilities as the primary factor for advancement. Additional factors that will help advance candidates include, but are

not limited to, efficiency, memory usage, and the performance of hardware implementations. There are multiple rounds in which several algorithms become eliminated from the competition. The final sole algorithm remaining will become the new SHA-3 standard.

CubeHash [2] is a candidate developed by Daniel J. Bernstein that was submitted to the Cryptographic Hash Algorithm Competition, and CubeHash has advanced to the second round. During the time that CubeHash has been a candidate, it has been subject to a fair amount of cryptanalysis from the public. Several attacks on CubeHash can be found in Section 2.2.

This thesis utilized CubeHash as an algorithm in a generic attack. The generic attack requires a cryptographic hashing algorithm that can be easily implemented in FPGA hardware. CubeHash is a very good candidate for implementation in hardware.

Chapter 2

Thesis Objectives

A generic collision search on small variants of the CubeHash cryptographic hash function was implemented. The data obtained from this collision search was analyzed to see if multiple processors could provide a parallel speedup that was directly proportional to the number of processors (a *linear* speedup).

Three different variants of CubeHash were used. All variants had the same parameters for bytes of message per set of rounds (32) and number of rounds between XOR operations with the message (16). The variation was in the number of bits used as the hash digest. Experiments with 56, 64, and 72-bit hash lengths were performed.

The work presented here relates to the work done by Oorschot and Wiener [12], which describes the generic hashing attack that forms the foundation of these experiments. It discusses the concept of feeding the output of a hash function back into the input of the hashing function until a certain pattern, called a *distinguished value*, is found, namely a number of leading zeros.

The paper goes further by describing two pitfalls that can occur. One pitfall is that one hash chain will intersect with the starting point of another hash chain (referred to as a *Robin Hood*). To make the probability of this pitfall very rare, the proportion of hash outputs that meet the pattern should be small enough to the point that hash chains are long, which decreases the probability of running into this problem. The second pitfall is that a hash chain loops around onto itself without ever meeting the pattern. This thesis followed the procedure that after a chain reached a length of 20 times the expected average chain length, that the chain was to be abandoned. The average expected chain length can be computed by raising 2 to the power

of the number of static bits in a *distinguished value*. For example, if a *distinguished value* was composed of 25 leading zeros, then the expected average chain length would be equal to $2^{25} = 33{,}554{,}432$.

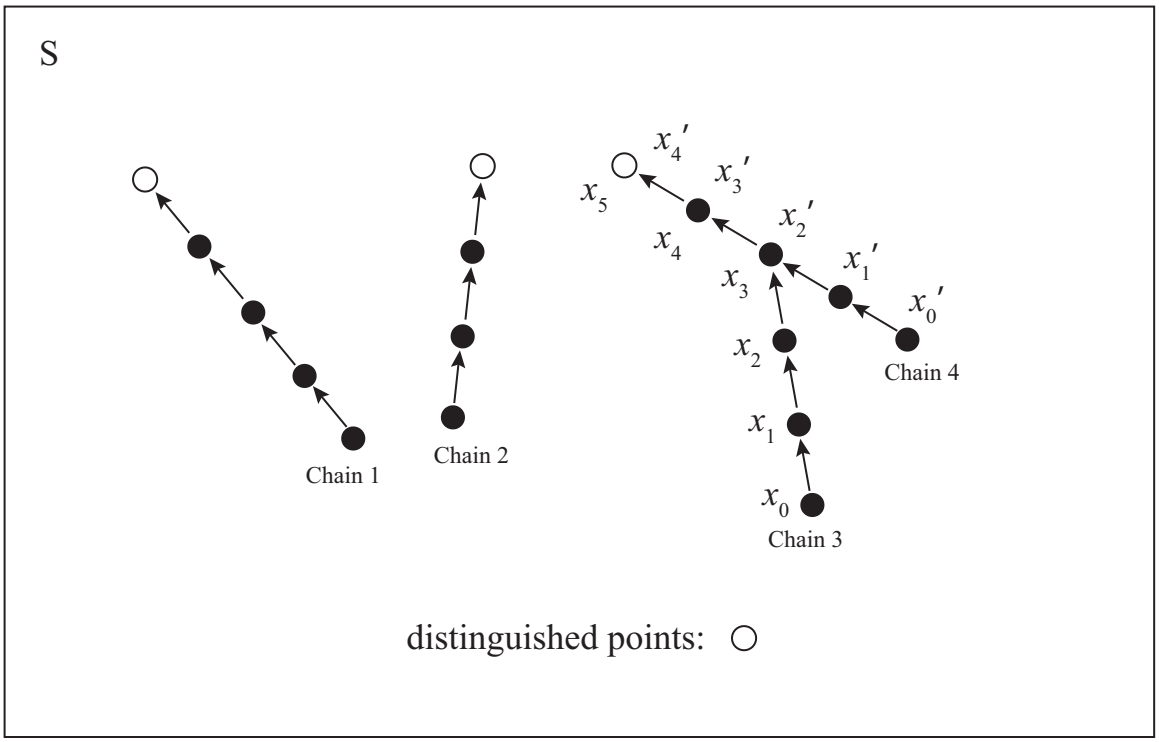

Figure 2.1: Illustration of the Hash Chain Concept

Figure 2.1 illustrates the hash chain concept that was first proposed by Oorschot and Wiener [12]. In this diagram, the points represent values within the hash digest space, *S*, and the arrows represent computing the hash function to arrive at another value in the hash space. The white point represents a distinguished point, or *distinguished value*.

The chains can start at any arbitrary point within *S*, and they must end at a distinguished point. After computing numerous chains, the scenario as depicted by Chain 3 and Chain 4 in Figure 2.1 should occur if the distinguished points are rare enough. In this scenario, Chain 3 and Chain 4 each start at different points but they end up at the same distinguished point. This shows that somewhere along the line the two chains collided. When these chains are reevaluated, it can be shown that point x_2 and point $x_1{}'$ both hash to the same value. This is a hash collision, and this is the goal of the attack.

The novel contribution from Oorschot and Wiener [12] was that this method of finding collisions in hash functions provides a parallel speedup that is directly proportional to the number of processors involved (a *linear* speedup). Previously, the Pollard Rho algorithm (described later in Section 2.3.3) had provided a parallel speedup which was proportional to the square root of the number of processors. This thesis tests through real-world experimentation the hypothesis that the method of computing hash chains actually does yield a *linear* speedup.

2.1 CubeHash Algorithm

There are many variants of the CubeHash [2] algorithm. The particular variant being used depends solely on three parameters, (r, b, h). r is the number of rounds performed after inserting a piece of the message into the algorithm; b is the number of bytes to insert between r rounds; and h is the number of bits to output as the final digest.

The rest of the CubeHash algorithm is composed of these steps:

1. Initialize a 128-byte state consisting of 32 32-bit Little Endian words.
2. Pad the message to be a multiple of 1 or more b-byte blocks.
3. For each b-byte block of the padded message: XOR the block into the state.
4. Transform the state through r identical rounds.
5. Finalize the state.
6. Output the first h/8 bytes of the state.

Initialization is performed by setting the first three 32-bit words to the values h/8, b, and r respectively. The remaining words are set to zero. Then the state is transformed through 10r rounds. It is important to note that this initialization step produces the same output for he same parameters r, b, and h. Therefore, in the interest of saving time, this value can be precomputed beforehand.

To pad the incoming message, a 1 bit must be appended. Then 0 bits are appended until the message is a multiple of b-bytes.

The round transformations are described in Section 2.1.1.

State finalization is performed by performing the XOR operation on the last word of the state with the value 1. In essence, the least significant bit of the last word is flipped. Then the state is transformed through another $10r$ rounds to arrive at the final state.

The message digest is then taken to be the first $h/8$ bytes of the final state.

2.1.1 CubeHash Transformation

State transformation is performed $10r$ times after Initialization, r times after b-byte blocks of the message are inserted into the algorithm, and $10r$ times after Finalization. Each step of the transformation is shown in Figure 2.2.

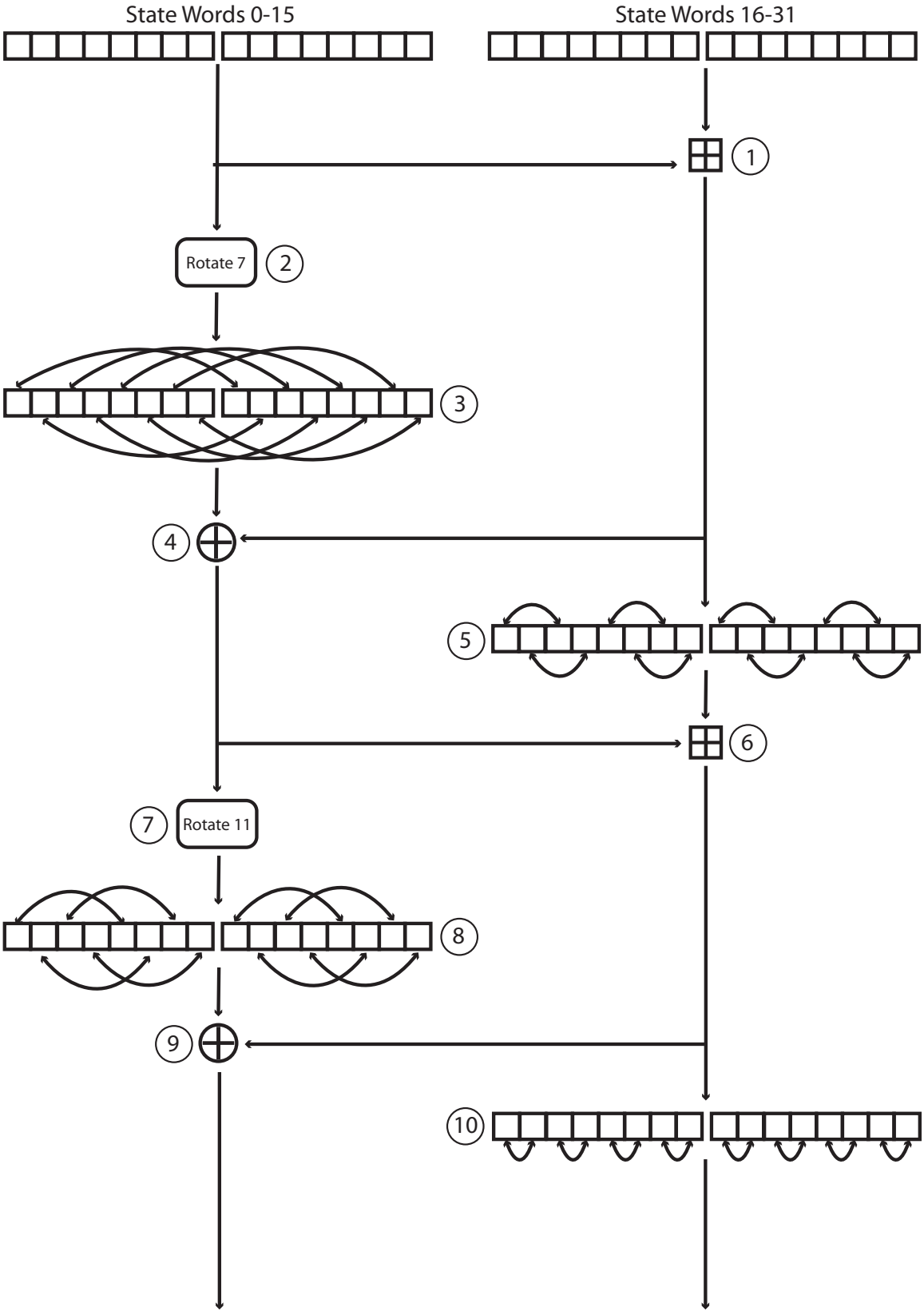

Figure 2.2: One State Transformation in CubeHash. Inspired by [1].

Figure 2.2 shows a single state transformation in CubeHash. This can be evaluated as 10 different steps.

1. The first 16 words are added into the second 16 words.
2. The first 16 words undergo a bit rotation. Each word is rotated upwards by 7 bits.
3. The first 16 words are swapped with each other. Each of these words is swapped with the word that has the same index, but with the fourth bit of that index flipped. Therefore, word 0 is swapped with word 8; word 1 is swapped with word 9; and so on.
4. The second 16 words are XOR'd into the first 16 words.
5. The second 16 words are swapped with each other. Each of these words is swapped with the word that has the same index, but with the second bit of that index flipped. Therefore, word 16 is swapped with word 18; word 17 is swapped with word 19; word 20 is swapped with word 22; and so on.
6. The first 16 words are added into the second 16 words. Same as in Step 1.
7. The first 16 words undergo a bit rotation. Each word is rotated upwards by 11 bits.
8. The first 16 words are swapped with each other. Each of these words is swapped with the word that has the same index, but with the third bit of that index flipped. Therefore, word 0 is swapped with word 4; word 1 is swapped with word 5; and so on.
9. The second 16 words are XOR'd into the first 16 words. Same as in Step 4.
10. The second 16 words are swapped with each other. Each of these words is swapped with the word that has the same index, but with the first bit of that index flipped. Therefore, word 16 is swapped with word 17; word 18 is swapped with word 19; and so on.

2.2 Structural Attacks on CubeHash

Structural attacks on a hash function are typically the most successful types of attacks against a hash function. Structural attacks may find collisions, preimages, or second preimages. However, the distinguishing trait of structural attacks is that they are almost always particular for a specific hash function. This is because structural attacks exploit weaknesses within the hash function itself, such as how state transformations are computed, etc.

2.2.1 Single Block Attacks and Statistical Tests on CubeHash

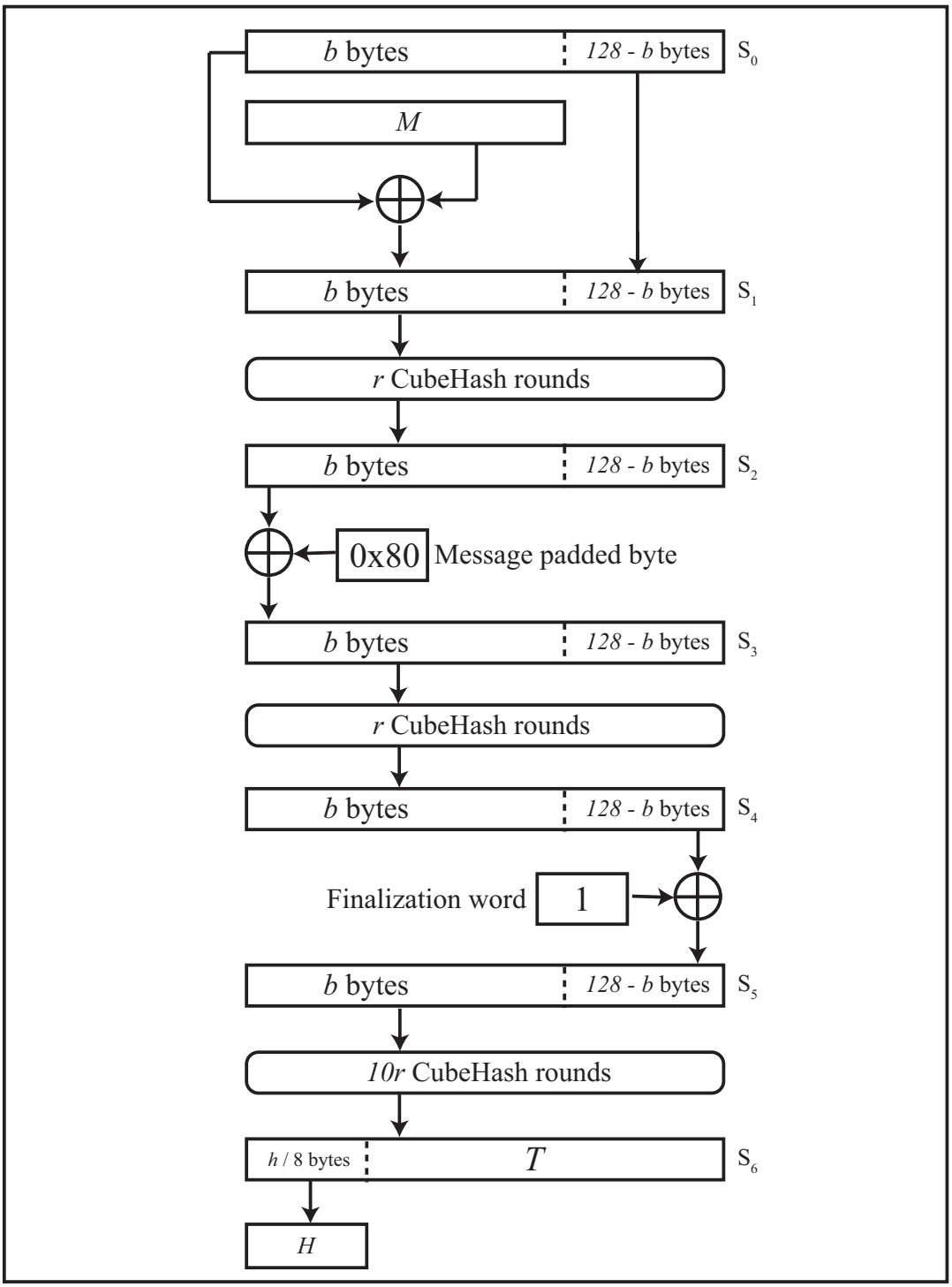

Figure 2.3: Processing a *b*-byte message through CubeHash. Inspired by [3]

Figure 2.3 shows how a *b*-byte message would get processed through CubeHash. This is the foundation of the attack proposed by Bloom and Kaminsky [3]. The state S_0 is the CubeHash internal state following the initialization step, as described in Section 2.1. Then a *b*-byte message block is directly XOR'd into the first *b* bytes of the state, leaving the remaining $(128 - b)$ bytes completely unaltered. This brings the state to S_1. Then *r* rounds are applied bringing the state to S_2, the message padding byte (0x80) is XOR'd into the state bringing the state to S_3, *r* rounds are applied bringing the state to S_4, the finalization word is XOR'd into the state bringing the state to S_5, and lastly $10r$ rounds are applied to bring the state to S_6. The hash digest (labeled as *H*) is taken as the first $h/8$ bytes of S_6 and the remaining $(128 - \frac{h}{8})$ bytes (labeled as *T*) are discarded.

The novel idea proposed by Bloom and Kaminsky is that altering the value of *T* does not in any way change the hash digest. Therefore, if one was able to change the value of *T* and trace backward through the computation and the last $(128-b)$ bytes of S_1 matched the last $(128-b)$ bytes of S_0, then it would be trivial to find a message that would produce a second preimage. Figure 2.4 on Page 17 illustrates this process.

Figure 2.4 shows the process of going backwards through the CubeHash computation by supplying a desired hash digest to which a second preimage should be found and any value of *T* that is not equal to the value of *T* for the first message processed (if the value of *H* and *T* are the same as the first message processed, then the same message will result). An appropriate value for *T* requires a lot of trials, because, as shown in Figure 2.4, the last $(128 - b)$ bytes of S_1 must match the last $(128 - b)$ bytes of S_0. Once a match is found between these bytes, the desired alternate message may be obtained by computing the XOR of the first *b* bytes of S_1 with the first *b* bytes of S_0.

It should be noted that while this attack produces a second preimage in less time than a brute force search, this attack is only feasible with an extremely large value of *b*, well beyond the recommended values. Also, the time required to perform this attack grows linearly with *r*.

The part of this attack that makes it a structural attack is how it relies on the CubeHash method of taking the first *h* bits of the final state as the

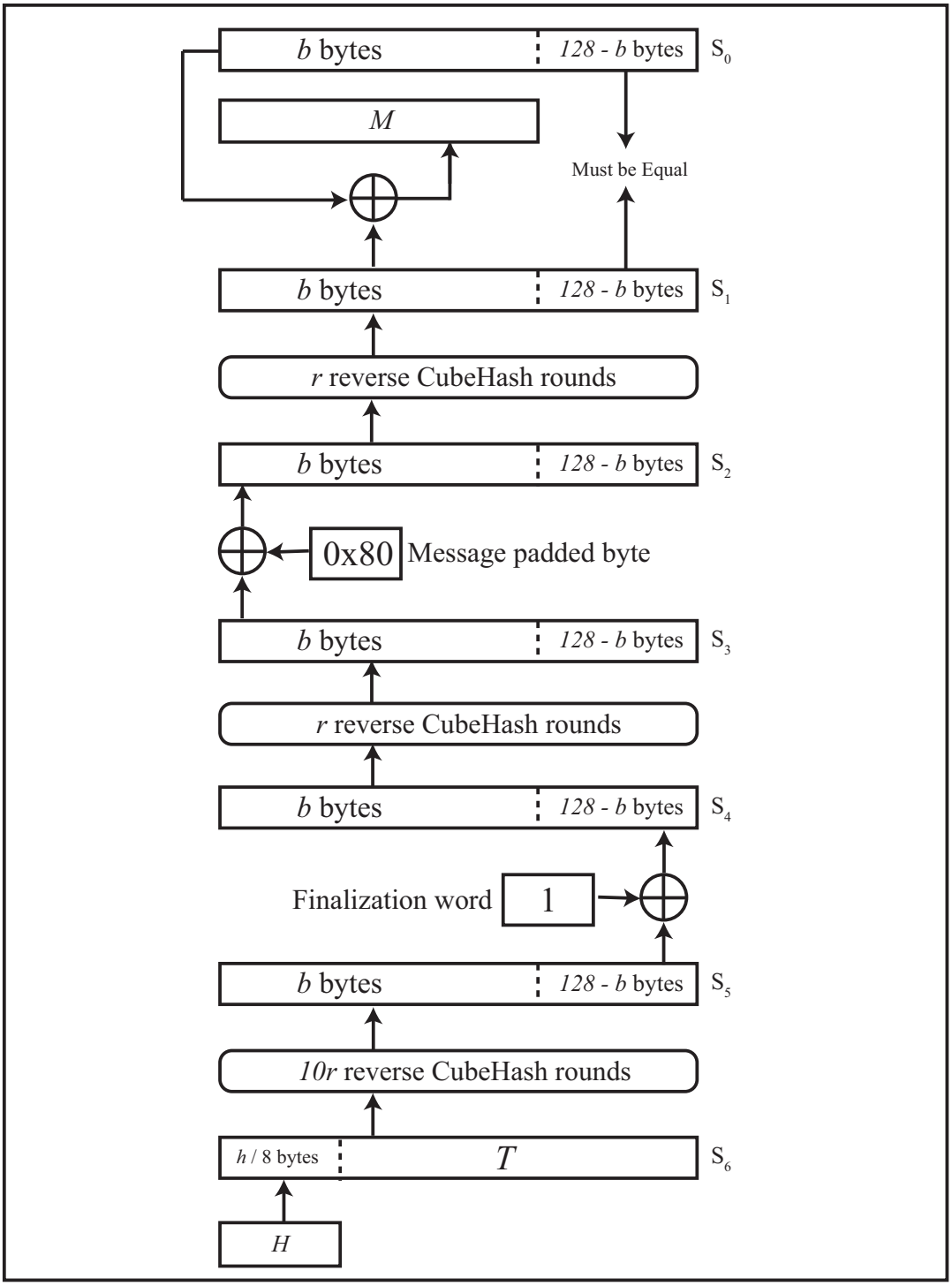

Figure 2.4: Processing a *b*-byte Message in Reverse Through CubeHash. Inspired by [3]

hash digest while discarding the remaining bits, the way that CubeHash processes messages blocks (XOR), and the invertibility of the CubeHash round function. This attack would not work for any other hashing algorithm that does not have these components.

2.2.2 Inside the Hypercube

Class		32-bit words of CubeHash state
C_1	:	AABBCCDD EEFFGGHH IIJJKKLL MMNNOOPP
C_2	:	ABABCDCD EFEFGHGH IJIJKLKL MNMNOPOP
C_3	:	ABBACDDC EFFEGHHG IJJIKLLK MNNMOPPO
C_4	:	ABCDABCD EFGHEFGH IJKLIJKL MNOPMNOP
C_5	:	ABCDBADC EFGHFEHG IJKLJILK MNOPNMPO
C_6	:	ABCDCDAB EFGHGHEF IJKLKLIJ MNOPOPMN
C_7	:	ABCDDCBA EFGHHGFE IJKLLKJI MNOPPONM
C_8	:	ABCDEFGH ABCDEFGH IJKLMNOP IJKLMNOP
C_9	:	ABCDEFGH BADCFEHG IJKLMNOP JILKNMPO
C_{10}	:	ABCDEFGH CDABGHEF IJKLMNOP KLIJOPMN
C_{11}	:	ABCDEFGH DCBAHGFE IJKLMNOP LKJIPONM
C_{12}	:	ABCDEFGH EFGHABCD IJKLMNOP MNOPIJKL
C_{13}	:	ABCDEFGH FEHGBADC IJKLMNOP NMPOJILK
C_{14}	:	ABCDEFGH GHEFCDAB IJKLMNOP OPMNKLIJ
C_{15}	:	ABCDEFGH HGFEDCBA IJKLMNOP PONMLKJI

Table 2.1: Symmetrical Classes within CubeHash

An attack published by Aumasson, Brier, Meier, Naya-Plasencia, and Peyrin [1] exploits the fact that the CubeHash round function, and the inverse of that round function, preserve symmetries within the 128-byte state. The authors found 15 distinct classes of symmetries in CubeHash. They are shown in Table 2.1. On the left side of Table 2.1 is the class index number and on the right are a series of letters. Each letter represents one 32-bit word within CubeHash's 1024-bit internal state. The same letter repeated represents the same 32-bit word.

What happens in each of these classes of symmetry is that whenever the CubeHash round function is processed on a symmetric state, the same symmetry will be preserved, even though the particular words will change.

$C_1 \cap C_2 \cap C_3$	: AAAACCCC EEEEGGGG IIIIKKKK MMMMOOOO

Table 2.2: Example of a Union Between Symmetric Classes

Furthermore, as shown in Table 2.2, the internal state may be composed of multiple classes of symmetry. The example in Table 2.2 shows what the internal state would look like if it were composed of symmetric classes C_1, C_2, and C_3.

The CubeHash algorithm however does have some protection against symmetries. In the finalization step, as described in Section 2.1, the last word of the state has its least significant bit flipped. This act breaks any symmetry within the CubeHash internal state.

But the attack is not completely thwarted by the CubeHash finalization step. The first step in the attack is to construct internal $State_0$, which is defined as the state that follows the initialization step as described in Section 2.1. Then, from $State_0$ process 2^{500} message blocks each through r rounds to arrive at one class of symmetry, as shown in Table 2.1. Likewise, work backwards from the final state, past the finalization step, and then in reverse process 2^{500} message blocks to arrive at one class of symmetry, not necessarily the same class as found previously. Then in the forward and the backward direction process *null messages* (messages with only 0 bits) until both the forward state and the reverse state are in the same set of symmetries. Then, in both directions process message blocks until a collision is obtained. These last two steps are upper-bounded by 2^{256} operations.

So, this attack is capable of finding collisions in approximately 2^{501} computations. However, such a high number is currently infeasible on current technology, and the messages that are processed would be of unauthorized length, over 2^{256} bytes. This makes the attack impractical.

The part of this attack that makes it a structural attack is that only CubeHash has this particular set of symmetries, and only CubeHash preserves this set of symmetries through its round function. This attack could not apply to any other hashing algorithm, because all other hashing algorithms have a different round function (if applicable) and a different internal state (if applicable).

2.2.3 Preimage attack on CubeHash512-r/4 and CubeHash512-r/8

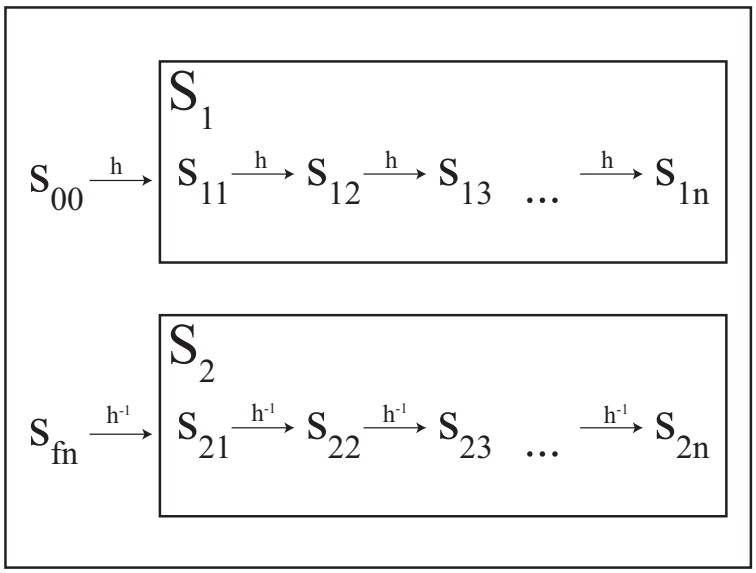

Figure 2.5: Sets for a Meet in the Middle Attack

Another structural attack published by Khovratovich, Nikolic, and Weinmann [6] shows a *meet in the middle* attack. A *meet in the middle* attack divides a problem into two smaller easier pieces and then the process of combining the pieces should be simple enough that it doesn't increase the complexity beyond either of the smaller pieces. Overall, breaking the attack into two pieces should be faster than working on the full problem.

In this scenario, there are two sets, S_1 and S_2. Each set contains multiple CubeHash states, denoted by a lowercase s. s_{00} represents the state that immediately follows initialization, as described in Section 2.1. From there S_1 is filled with a chain of states by processing the round function, presumably with a *null message* (a message containing only zero bits). s_{fn} is the state just prior to the finalization step and the message padding byte. S_2 is filled with a chain of states by processing the inverse round function, again with a *null message*. n, the number of states in each set, is equal to $2^{\frac{1024-8b}{2}}$.

Once the sets are complete, there should be one state in S_1 that matches any state in S_2 in all but the first *b* bytes. The XOR between the first *b* bytes of these states is the message block that will get inserted at this point. So the preimage will be a series of *null message* blocks, the result of the XOR

as one message block, and then another series of *null message* blocks.

The memory requirement for this attack is nontrivial, and the gain over a brute-attack is about 4 times fewer computations. A memoryless modification to this attack is about 3 times more costly, which is roughly the same complexity as a brute-force attack.

This attack is a structural attack, because it relies on the way CubeHash processes *null message* blocks and it relies on invertible round function within CubeHash. This attack would not work on another hash function.

2.3 Other Generic Attacks on Hash Functions

A generic attack, when used in the context of hash functions, is an attack which does not depend on any particular hash function. This contrasts with Section 2.2, because the attacks in Section 2.2 depend on specific implementation details in CubeHash. A generic attack may be applied to any hash function. The success rate of generic attacks depends solely on computational power and memory. Good cryptographic hash functions therefore typically have a hash digest that is large enough to be computationally infeasible for any generic attack.

2.3.1 The Naïve Collision Attack

Figure 2.6 illustrates the naïve collision attack. The dots on the left represent values within the domain of the hash function, *N*, which is typically anything between 0 and 2^{64} bits or more depending on the particular hash function. The dots on the right represent values within the range of the hash function, *S*, which all have a fixed bit length which is also defined by the hash function. The arrows represent computing the hash function and arriving at a hash digest value, which is always within *S*. Messages are processed one at a time through the hash function, usually sequentially but any non-repeating pattern would suffice.

As the hash computations are performed, both the messages and their corresponding hash digest values are stored, usually in a table or database.

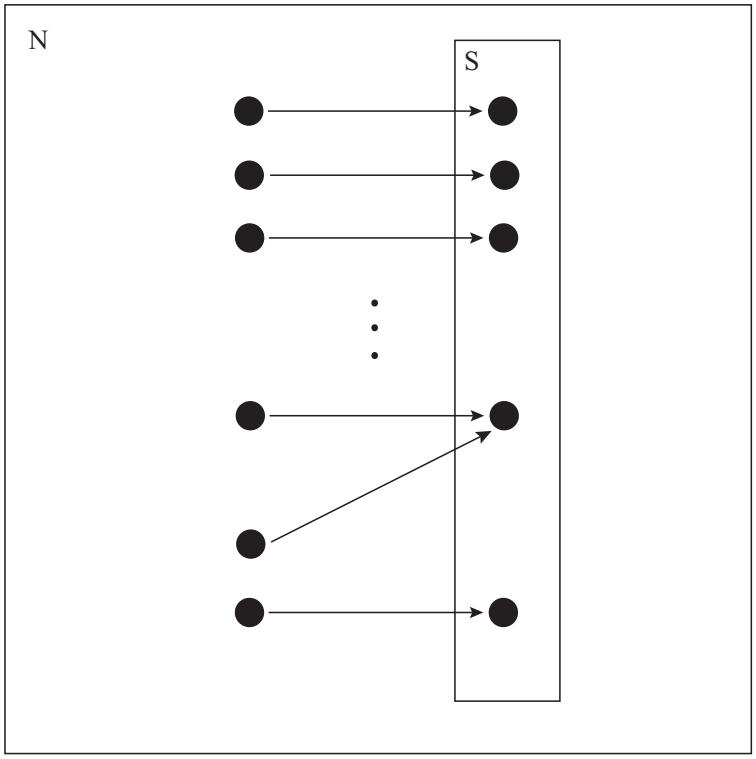

Figure 2.6: Naïve Collision Attack

Since *S* is finite and *N* is much larger than *S*, a collision must occur. This is illustrated in Figure 2.6 with the second and third to last messages. The two arrows point to the same hash digest in *S* but they originate from two different points within *N*.

This attack will find a collision with probability 50% after evaluating the hash function on the order of $\sqrt{2^h}$ times, where *h* is the length of the hash digest in bits. The same number of messages and hash digests must also be stored. This is derived from the well-known problem known as the *Birthday Problem*. It should be noted that the complexity of this attack is nontrivial with modern hardware for hash digests with 128 bits or more, and the SHA-3 competition mandates a minimum of 224 bits. Many candidates recommend hash digests with as many as 512 or 1024 bits.

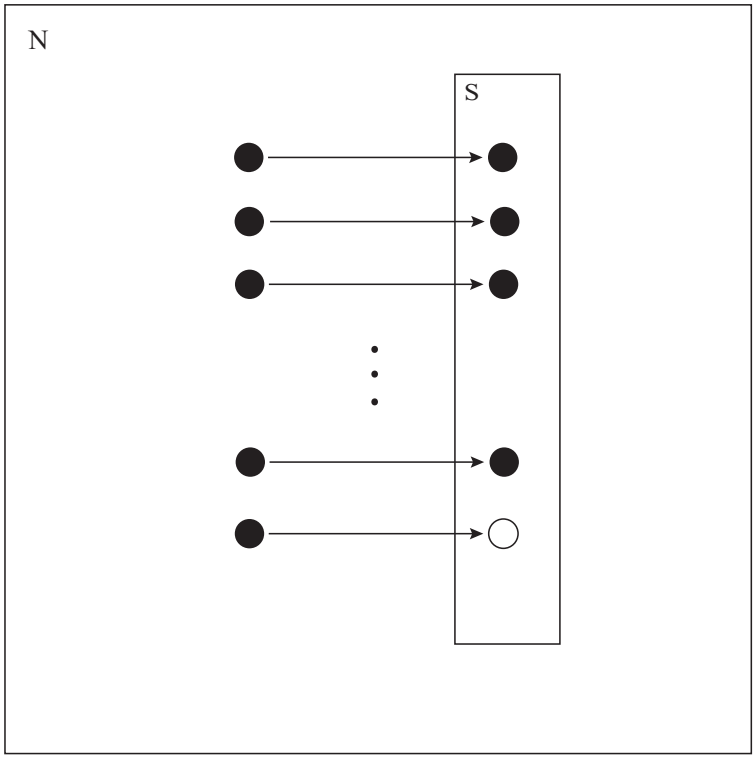

Figure 2.7: Naïve Preimage Attack

2.3.2 The Naïve Preimage Attack

Figure 2.7 illustrates the naïve preimage attack. The dots on the left represent values within the domain of the hash function, *N*, which is typically anything between 0 and 2^{64} bits or more depending on the particular hash function. The dots on the right represent values within the range of the hash function, *S*, which all have a fixed bit length which is also defined by the hash function. The white dot represents the hash digest value which is the target for this attack. The arrows represent computing the hash function and arriving at a hash digest value, which is always within *S*. Messages are processed one at a time through the hash function, usually sequentially but any non-repeating pattern would suffice.

As the hash computations are performed, the resulting hash digest value is compared against the target hash digest value (represented by the white dot in Figure 2.7). If any resulting hash digest value results in the target hash digest value, then the attack is successful.

This attack will find a preimage after evaluating the hash function on the order of 2^h times, where h is the length of the hash digest in bits. The memory requirement is trivial. It should be noted that the complexity of this attack is currently nontrivial with modern hardware for hash digests with 64 bits or more. And, as mentioned, the SHA-3 competition mandates a minimum of 224 bits.

2.3.3 The Pollard Rho Collision Search

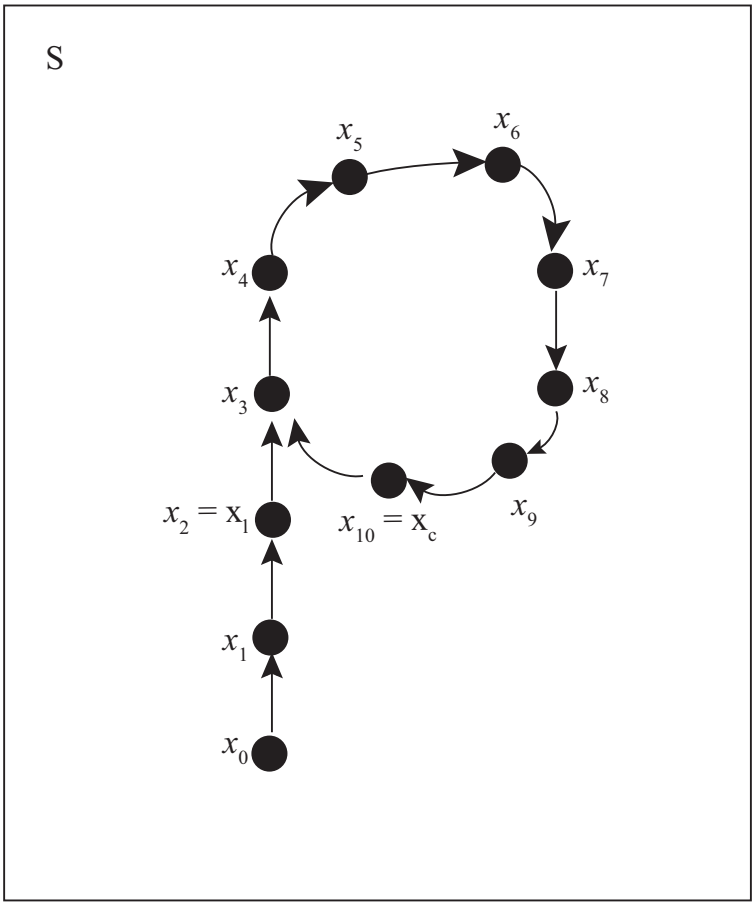

Figure 2.8: Pollard Rho Collision Search

Figure 2.8 illustrates a more sophisticated collision attack that is based on the Pollard Rho algorithm, and is described in [12]. The dots represent values within the range of the hash function, S, which all have a fixed bit

length which is defined by the hash function. The arrows represent computing the hash function.

This attack follows a hash chain until the chain loops back around onto itself, thereby entering a cycle. The first step in the attack is to choose a random point, and call it x_0. Then produce the sequence $x_i = f(x_{i-1})$ where f is the hash function, for $i = 1, 2, \ldots$. Since the space of the hash digest is finite, the sequence must begin to cycle. Let x_l be the point in the sequence that exists just prior to entering the cycle (in Figure 2.8 this is equivalent to x_2). Therefore, x_{l+1} (x_3 in Figure 2.8) must be on the cycle. Let x_c be the point on the cycle that precedes x_{l+1} (x_{10} in Figure 2.8). When $i = c$ the collision is found, because $f(x_l) = f(x_c)$, but $x_l \neq x_c$. It is shown in [12] that the number of points visited is $\sqrt{\pi 2^{h-1}}$, where *h* is the length of the hash digest in bits. The advantage of this method, as stated by the authors of [12], is that the memory requirement is small if a clever method is used to detect a cycle. It should be noted that this method is infeasible for modern-day cryptographic hash functions.

Chapter 3

System Design

3.1 Architecture

The objectives of this thesis required a fast platform capable of varying degrees of parallelism. To meet this requirement, an FPGA cluster was used. Each FPGA is capable of running 25 CubeHash engines, each CubeHash engine running independently from each other. The results from each CubeHash engine were collected in a MySQL database via a Java intermediary server.

Figure 3.1 shows the architecture for the design in this thesis. On the left are four independent Virtex-5 FPGA boards with a built-in PowerPC processor. Each FPGA board ran a relatively recent version of the GNU/Linux operating system, and each FPGA Accelerator could hold a total of 25 CubeHash engines. On the right is a single desktop computer that ran both a custom Java-based server as well as a MySQL database. The Java server and the MySQL database communicated with each other via a MySQL connection (MySQL provides a Java interface to its database software).

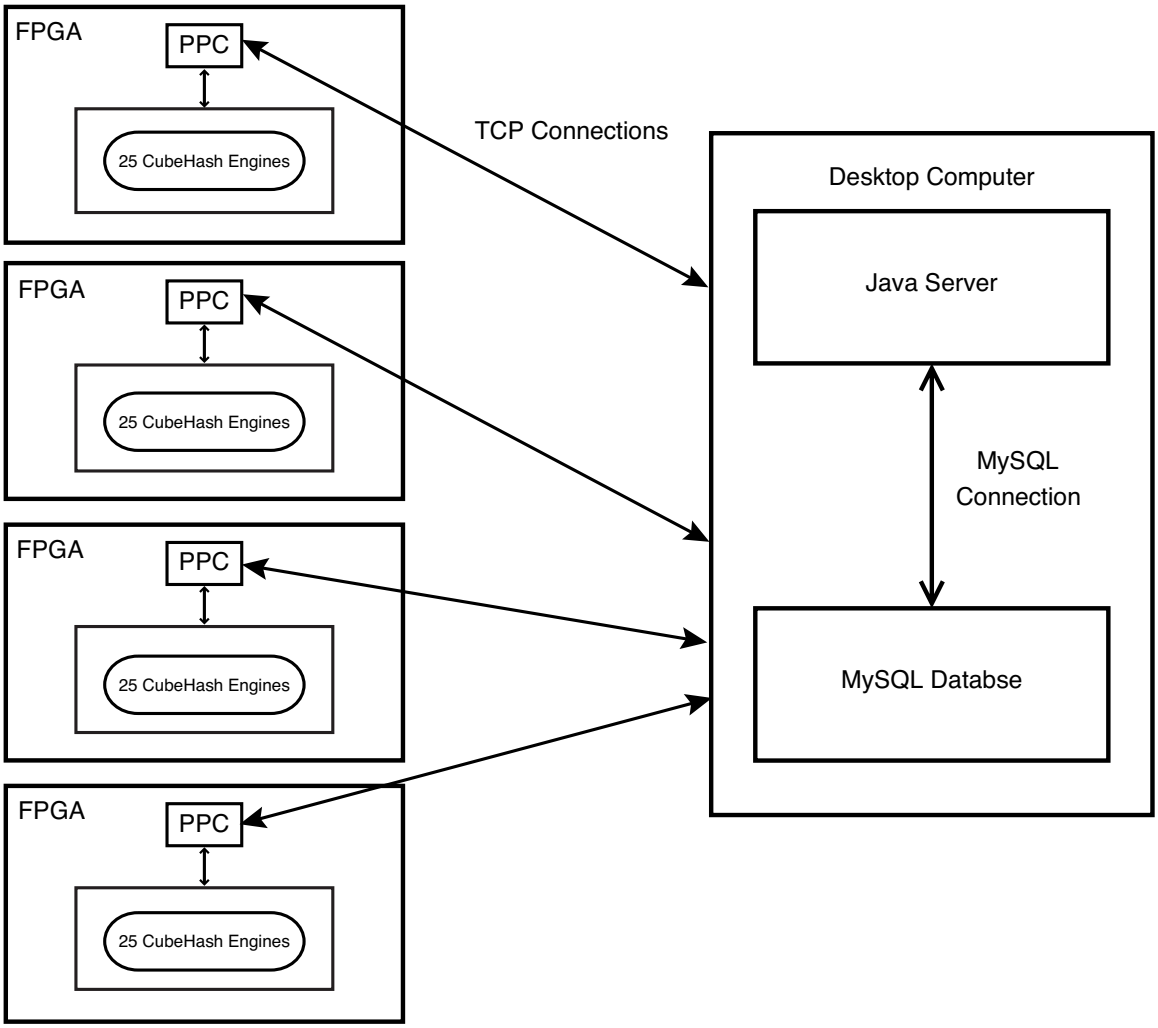

Figure 3.1: System Architecture

The PowerPC processor initiated a TCP connection with the Java server in order to request work. The Java server sent work to each PowerPC processor. To ensure maximum throughput, the Java server guaranteed that no two PowerPC processors will be given the same work. The work that was given to each PowerPC was a starting h-bit integer. The PowerPC communicated the starting integer to a particular CubeHash engine in the FPGA Accelerator via a specialized kernel module that could read and write data in FIFO queues on each CubeHash engine within each FPGA.

Each CubeHash engine then computed the CubeHash hash digest of the starting integer using predefined r, b, and h parameters. If the resulting hash digest did not match a specific pattern (specifically 25 leading zeros), then the hash digest was fed back through the CubeHash hash algorithm to produce a new hash digest. This cycle repeated until the hash digest matched a specific pattern. If no match was found within 20 times the expected chain length, the chain was aborted and a new chain was started. Once the pattern was reached, the CubeHash engine communicated the final hash digest and the number of hash iterations back to the PowerPC via the kernel module. The PowerPC then sent that data to the Java server for storage on the MySQL Database, and the PowerPC requested its next work load.

The Java server did some additional work with the MySQL database. The Java server checked for hash-chains that ended up at the exact same hash digest. These merged chains were stored in a separate MySQL table for post-processing. The Java server also stored resume data so that it could continue from previous results in the event of a power failure or a system shutdown.

3.2 CubeHash Engine

Figure 3.2 shows the overall layout of the hardware in each CubeHash engine (25 CubeHash engines were on each FPGA). At the bottommost layer is the *cubehash2* component which actually computes the CubeHash algorithm on a fixed length input message and produces a fixed-length hash digest output.

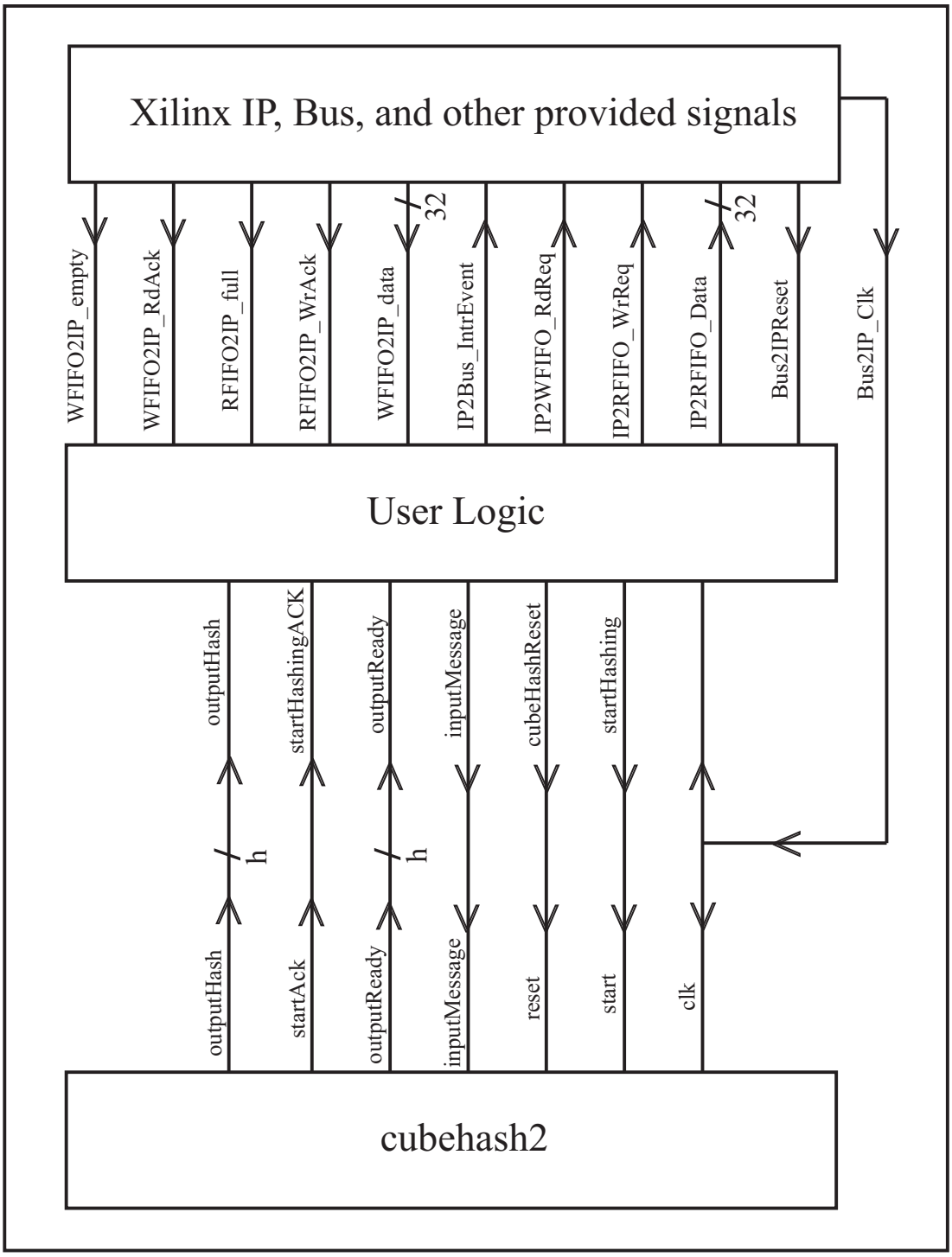

Figure 3.2: Layout of CubeHash Engine Components

Above that layer is the *User Logic*. The *User Logic* component provides an interface between the *Xilinx IP* and the *cubehash2* component. The *User Logic* is responsible for taking data from the FIFO's in the *Xilinx IP* and translating that into a fixed length message for the *cubehash2* component to process, the output of which is sent back to the FIFO's in the *Xilinx IP*.

The *Xilinx IP* is provided by Xilinx so that interaction between the CubeHash engine and the PowerPC is possible. On the PowerPC side there is a special driver, as explained in Section 3.3, that is able to interact with the FIFO's.

3.2.1 CubeHash Implementation

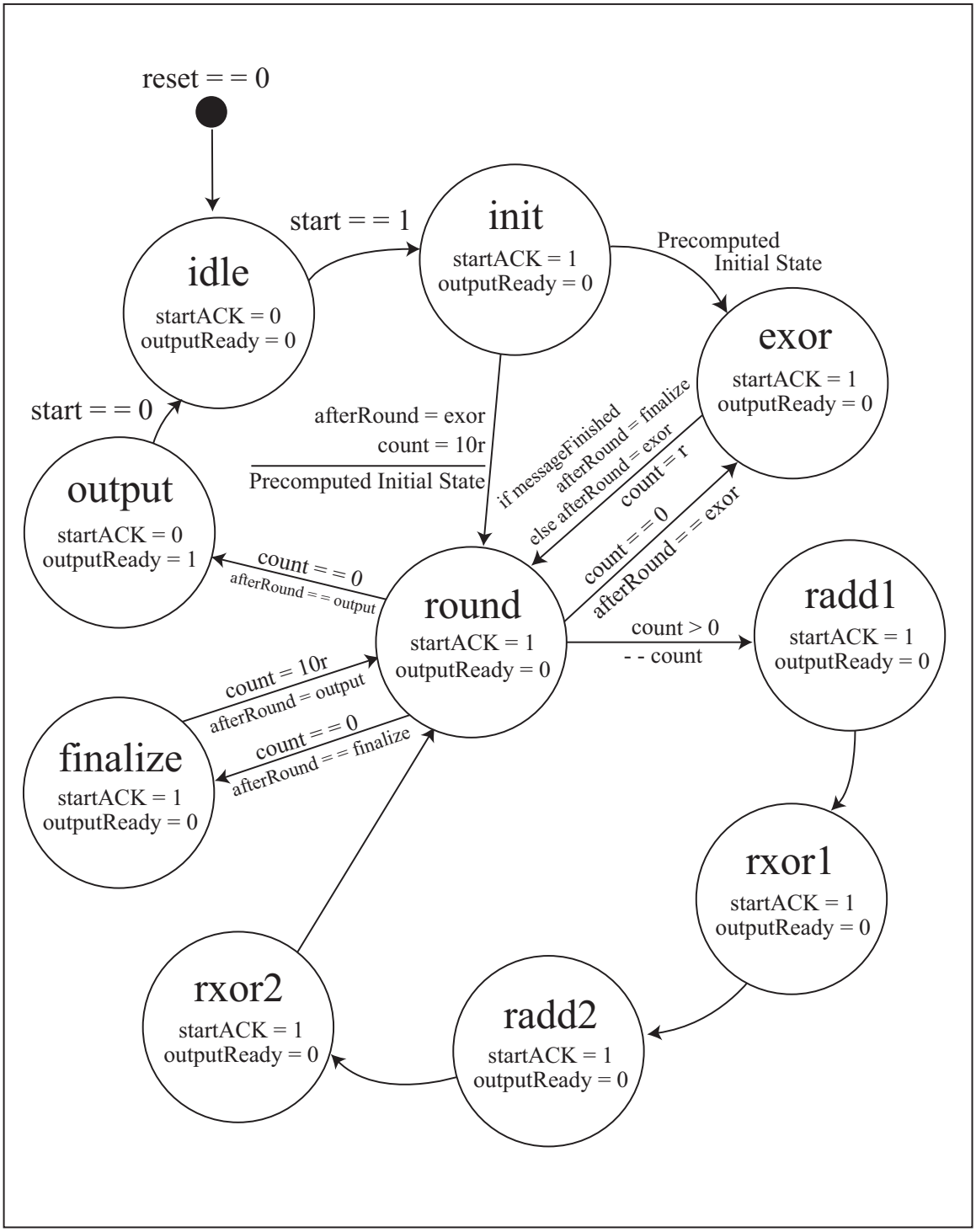

Figure 3.3: CubeHash Implementation State Machine

Figure 3.3 shows the state machine in the *cubehash2* component as shown in Figure 3.2. Whenever the incoming *reset* line becomes 0, the state machine asynchronously returns to the *idle* state. This is the only asynchronous operation in this state machine, all other state changes are synchronous on the rising edge of the incoming *clk* (clock) signal. The *idle* state is where the state machine is to begin. In the *idle* state, the *cubehash2* component does nothing except wait for the incoming *start* line to become 1. In this state, the *outputReady* and *startACK* (start acknowledgment) both output to 0. For all of the other states, *outputReady* is equal to zero except in the *output* state. Similarly, *startACK* is equal to 0 in only the *idle* and *output* states.

Once the incoming *start* line becomes equal to 1, the current state switches to the *init*. The purpose of the *init* state is to initialize the CubeHash internal state (as described in greater detail in Section 2.1). By definition, the internal state initialized by setting the first three 32-bit words to $h/8$, b, and r respectively, and then processing the state through $10r$ rounds. However, since the result of initialization is the same within any particular variant of CubeHash (regardless of the message contents), these values can be precomputed to provide a fair speedup. If the value has been precomputed, then the current state will immediately switch to the *exor* state. If the values have not been precomputed, then the *init* state will set the internal variables *afterRound* to *exor* and *count* to $10r$, and then the current state will switch to the *round* state.

The *round* state, as shown in Figure 3.3, is really the center point for the state machine. The *round* state alters the current state by making one or two comparisons. First, if the internal *count* variable is greater than 0, then the *round* state will always switch into the *radd1* state, and the *round* state will decrement the value of *count* by 1 whenever it does this. Second, if the internal *count* variable is equal to 0, then it will transfer the current state into whichever state is stored in the *afterRound* variable. Using these internal variables allows for a variable number of CubeHash round transformations to be performed, and for control to automatically switch to the appropriate state after all of the round transformations have been completed.

The *radd1*, *rxor1*, *radd2*, and *rxor2* states stand for *First Round Addition*, *First Round XOR*, *Second Round Addition*, and *Second Round XOR*

respectively. These states perform the most intensive work of the CubeHash algorithm. The CubeHash round transformation (as better described in Section 2.1.1) is composed of sixteen parallel 32-bit additions twice, sixteen parallel 32-bit XOR operations twice, and a number of 32-bit word swaps and bit rotations. In FPGA hardware, word swaps and bit rotations do not suffer from any combinational gate delay. Any FPGA could do almost an unlimited number of these operations within 1 clock cycle, because the result is precomputed by the compiler at compile time. However, XOR operations typically suffer from at least 1 gate delay, and 32-bit addition suffers from multiple gate delays, depending on the specific implementation of the adder. Therefore, the CubeHash round transformation was split into each of the 4 tasks that require gate delays, and any preceding swaps or bit rotations are bundled into the operation. The only exception is the very last operation of the CubeHash round transformation which is a swap, in this case the last swap is bundled into the *rxor2* state. Each of these four states take exactly one clock cycle and they are always encountered in succession, preceded by the *round* state, and followed by the *round* state.

The *exor* state stands for *XOR Message*. In this state, the next b bytes of the message are XOR'd into the first b bytes of the internal CubeHash state. The internal *count* variable is always set to r in this state. If the last b bytes of the message have been processed, then the internal *afterRound* state gets set to *finalize*, otherwise it is set to *exor*. There are some endian conversions that occur in this state, because CubeHash requires little endian words, while VHDL addition by default operates on big endian words, but as mentioned in the previous paragraph, swapping bytes does not have any associated time cost since it is all precomputed by the compiler.

The *finalize* state has a very simple task. It XOR's the last word in the CubeHash internal state with the number 1 to essentially flip the least significant bit. Then, it sets the internal variables *afterRound* to *output* and *count* to $10r$ before the current state switches back to *round*.

The *output* state is the only state that sets the *outputReady* signal to 1. As mentioned in Section 2.1, the hash digest is composed of the first h bits of CubeHash internal state. So in the *cubehash2* component (shown in Figure 3.2), the *outputHash* is directly connected to the internal CubeHash

state, and as such, its value is continually changing. Therefore the *output* state allows an external component to know when it can read a valid CubeHash digest from the *outputHash* signal. The *output* state transfers to the *idle* state when *start* becomes equal to 0, although in practice usually the *reset* value is set to 0 for one clock cycle instead.

3.2.2 User Logic Implementation

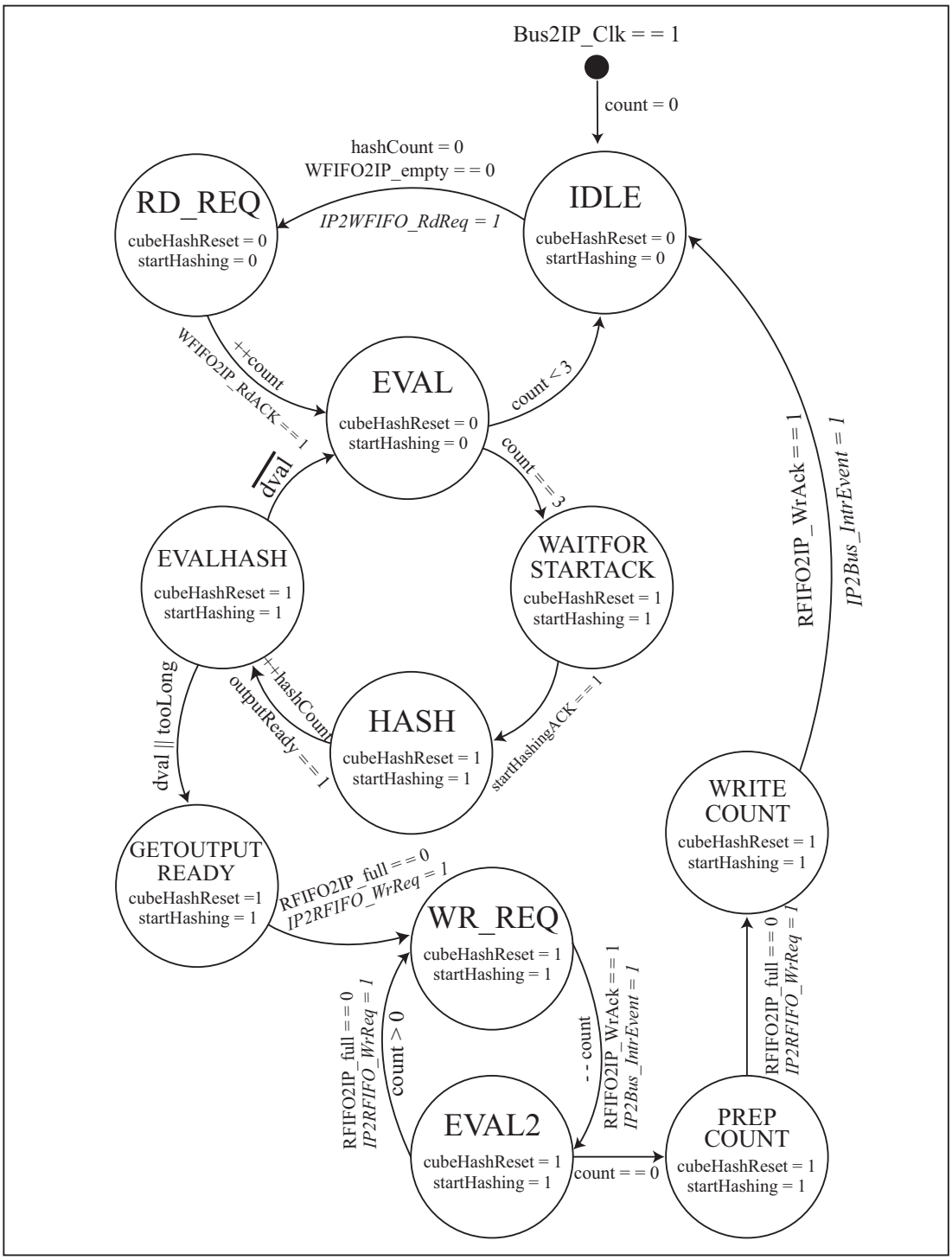

Figure 3.4: User Logic State Machine

Figure 3.4 on Page 35 shows the state machine that operates within the *User Logic* component, as shown in Figure 3.2. The purpose of the *User Logic* component is to act as an intermediary between the *cubehash2* component and the *Xilinx IP*. Its functions are to take incoming *starting words* from the PowerPC via the FIFO's in the *Xilinx IP*. Then it starts the *cubehash2* component to find the hash digest. If the hash digest is not a *distinguished value*, then the hash digest is fed back into the input of the *cubehash2* component, and the cycle repeats until a *distinguished value* is obtained. Once this happens, the *User Logic* component is then responsible for writing the *distinguished value* and the number of hash iterations back to the PowerPC via FIFO's in the *Xilinx IP*.

The state machine in Figure 3.4 starts with a synchronous reset whenever the *Bus2IP_Clk* becomes 1. This places the state machine into the *IDLE* state, and it resets the internal *count* variable to 0. In the *IDLE* state, the state machine waits for a new incoming word from the PowerPC. When a new incoming word is ready for input from the write FIFO, the *WFIFO2IP_empty* signal will become equal 0. When this happens, the *IDLE* state will set the internal *hashCount* variable to 0 and set the signal *IP2WFIFO_RdReq* to 1, and the state transfers into the *RD_REQ* state. (It should be noted that this variable, *IP2WFIFO_RdReq*, is in *italic* in Figure 3.4. This convention means that the value is only held for 1 clock cycle, and then in the next clock cycle it is reset to 0.) The *hashCount* variable stores the number of values in a hash chain, and the *IP2WFIFO_RdReq* signals tells the *Xilinx IP* that the CubeHash engine is ready to read a 32-bit value from write FIFO. In the *IDLE*, the *RD_REQ*, and the *EVAL* states *cubeHashReset* and *startHashing* are equal to 0; in all other states they are equal to 1.

The purpose of the *RD_REQ* state is to wait for acknowledgment from the *Xilinx IP* that the 32-bit value has been received. The 32-bit value is shifted in to a signal that will become the input value for the *cubehash2* component, but the shift only occurs once each time the state enters the *RD_REQ* state and only after a successful acknowledgment has been received. Once the *WFIFO2IP_RdACK* signal becomes equal to 1, that means the 32-bit value has been received, and the state will change into the *EVAL* state. During this state transition, the value of the internal *count* variable is incremented.

The *EVAL* state determines whether the next state should be the *IDLE* state or the *WAITFORSTARTACK* state. It makes this decision based on the value of the internal *count* variable. If *count* is equal to 3 (meaning three 32-bit words have been shifted in), then the next state will be the *WAITFORSTARTACK* state, otherwise it will be the *IDLE* state. The *EVAL* state is part of a second cycle, as shown in Figure 3.4, and in that cycle it acts as a way to reset the *cubehash2* component, because the *cubeHashReset* signal is set to 0 in the *EVAL* state.

The *WAITFORSTARTACK* state waits for the *startHashingACK* signal to become 1. This only occurs once the *cubehash2* component has started computing the hash function. Once *startHashingACK* becomes 1, the state machine transfers into the *HASH* state.

In the *HASH* state, the state machine waits for the *outputReady* signal to become 1. This occurs only when the *cubehash2* component has completed computing the hash function and has a valid output hash digest ready. Once *outputReady* becomes 1, the state machine transfers into the *EVALHASH* state, and the internal *hashCount* variable is incremented.

The *EVALHASH* state is responsible for determining whether or not the output hash digest is a distinguished value. If the output hash is a distinguished value, then the state machine transfers to the *GETOUTPUTREADY* state. If the maximum number of points in a hash chain has been reached (determined by the value of *hashCount*), then the output hash is turned into all 1's and the state machine transfers into *GETOUTPUTREADY* anyway; this is to prevent unending hash chains from wasting FPGA cycles. If neither of the first two cases hold true, then the state machine transfers into the *EVAL* state and the output hash digest is routed back to the input of the *cubehash2* component.

The *GETOUTPUTREADY* state sets the first 32-bit word of the output hash (prefixed with 0 bits if necessary) ready for output, and once the *RFIFO2IP_full* signal becomes zero, the state signals the *Xilinx IP* to transfer the 32-bit value into the read FIFO by setting the *IP2RFIFO_WrReq* signal to 1 and transferring to the *WR_REQ* state.

The *WR_REQ* state waits for an acknowledgment on the *RFIFO2IP_WrAck* signal, and when it receives that acknowledgment, it signals an interrupt on

the *IP2Bus_IntrEvent* line which tells the PowerPC driver (Section 3.3) that a 32-bit word is available on the read FIFO. The state then decrements the internal *count* variable and transfers to the *EVAL2* state.

In the *EVAL2* state, a decision is made whether to transfer back into the *WR_REQ* state or to transfer to the *PREPCOUNT* state. If the internal *count* variable is equal to 0, then that means three 32-bit words have been transfered to the PowerPC, so the state machine transfers into the *PREPCOUNT* state. Otherwise, the *EVAL2* state waits for the *RFIFO2IP_full* signal to become equal to 0, and then sets *IP2RFIFO_WrReq* equal to 1 while transferring to the *WR_REQ* state.

The *PREPCOUNT* state places the internal *hashCount* variable onto the 32-bit read FIFO and then waits until *RFIFO2IP_full* equals 0, at which point it sets *IP2RFIFO_WrReq* equal to 1 and transfers to the *WRITECOUNT* state.

The *WRITECOUNT* state waits for an acknowledgment that the *hashCount* variable has been written to the read FIFO. When the *RFIFO2IP_WrAck* line equals 1, the state sets *IP2Bus_IntrEvent* equal to 1 and transfers back to the *IDLE* state.

3.3 PowerPC Software

The PowerPC processors that were coupled with the Xilinx Virtex 5 FPGAs ran an embedded Linux distribution. Each Xilinx Virtex 5 FPGA could hold 25 CubeHash engines. Embedded systems rarely have support for high-level programming languages, such as Java, C++, and Python, due to the limited storage space, limited memory, and limited processing efficiency. Therefore, the C programming language was used to program the PowerPC software.

The overall purpose of the PowerPC software is to act as an intermediary between the CubeHash engines and the Java server. The operations of the Java server (as explained in Section 3.4) require strong multithreading support and communication with a MySQL server. It may have been possible to incorporate some of this functionality into the PowerPC's, but there would likely have been synchronization issues and there could have been problems accessing MySQL from such a limited environment. In any case, if the functionality of the Java server were to be solely placed on the PowerPC's, then there is no doubt that the PowerPC's would become less efficient.

On the other side of the PowerPC is the FPGA. After given a starting value, the CubeHash engines compute the hash chain until either a distinguished value is found or a chain length grows too large, in which case a result was given. These particular FPGA boards have support for networking, so in theory, they could have communicated directly with the Java server. However, each CubeHash engine would have to share the Ethernet port with the other hardware implementations and the PowerPC. The logic involved in both operating the Ethernet port and sharing it would have consumed a fair-sized portion of the FPGA hardware, thereby limiting the amount of CubeHash engines that could be placed on each FPGA.

So, the PowerPC software fills the place between the CubeHash engines and the Java server. The functions of the Java server are too high-level to be replaced by the PowerPC, and the functions of the CubeHash engine are too low-level to be able to communicate directly with the Java server. Since the C programming language is easily capable of communicating on

TCP sockets and working with file I/O, the PowerPC software can therefore easily transfer messages between the Java server and the CubeHash engines.

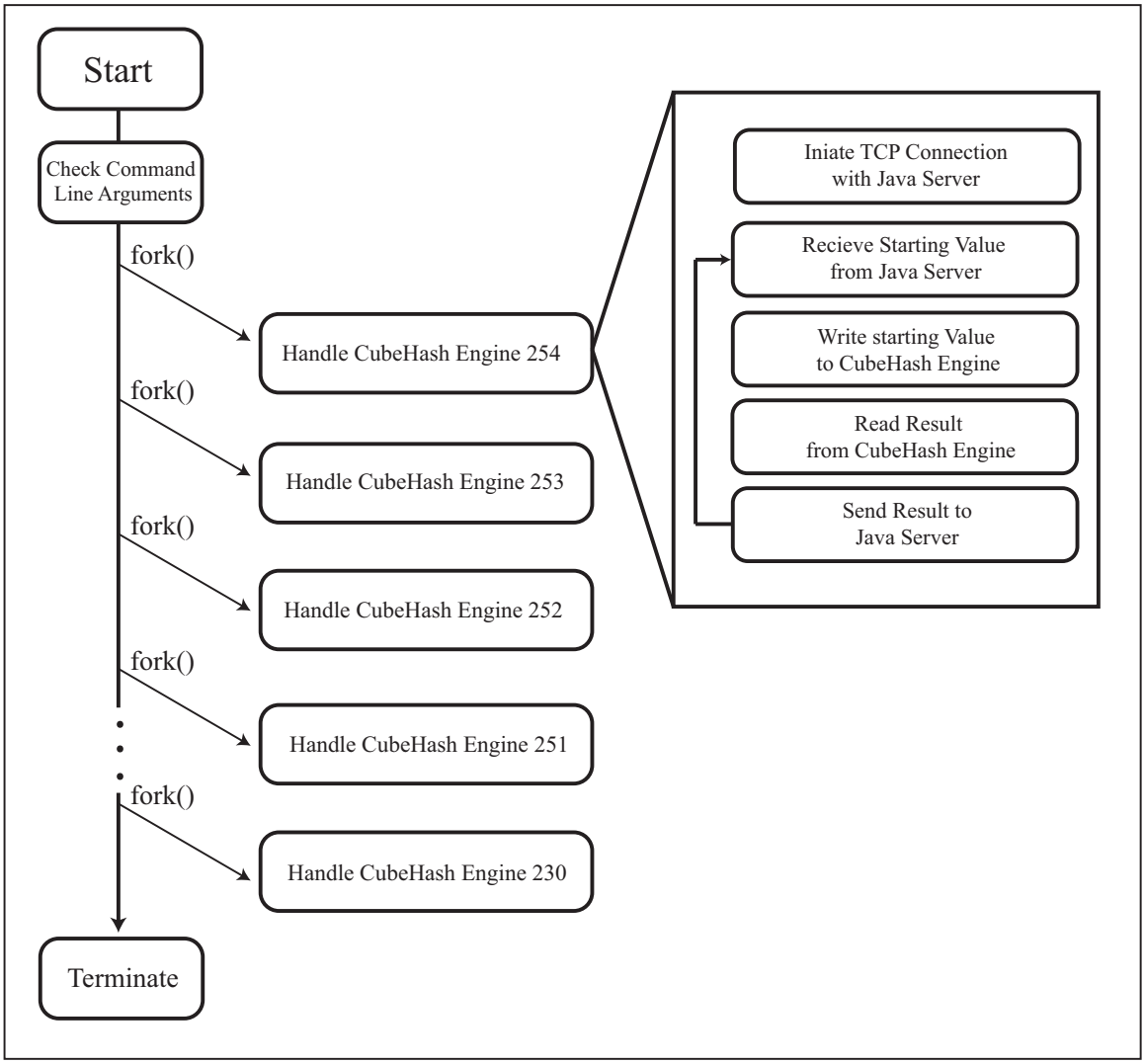

Figure 3.5: PowerPC Software Overview

Figure 3.5 shows an overview of how the PowerPC software operates. The PowerPC software first checks the command line arguments. If the command line arguments are valid the PowerPC software then forks a single child process for each CubeHash engine within the FPGA as specified by the command line arguments. While the child processes are being created, the parent process generates a Linux shell script that is able to terminate each child process by process ID (PID). This shell script is useful in

performing parallelization experiments, because multiple experiments can be started and stopped on the FPGA. A sample shell script is shown in Listing 3.1. Each experiment has exclusive access to a set of CubeHash engines, so they will not interfere with each other. The only possible interference could be caused by process switching in the operating system, but even that would be very minimal seeing as the child processes spend most of their lives blocked waiting on I/O.

```
1  kill -9 3852
2  kill -9 3854
3  kill -9 3855
4  kill -9 3856
5  kill -9 3857
6  kill -9 3861
7  kill -9 3862
8  kill -9 3863
9  kill -9 3864
10 kill -9 3865
```

Listing 3.1: Sample Shell Script to Terminate Child Processes

As shown in Figure 3.5, after all of the child processes are created, the parent process terminates. Each child process executes the Linux *mknod* command to create a character device file to use for communication with each CubeHash engine. This thesis uses a driver create by Jeremy Espenshade in his thesis [5]. The default build setup for the FPGA hardware causes the major device numbers for each CubeHash engine to start at 254 and decrement from there. The minor device numbers are not used. Since there are only at most 25 CubeHash engines per PowerPC processor, the lowest major number used is 230.

After the particular character device file is created, the child process then initiates a TCP connection to the Java server. The child process then blocks waiting to read three 32-bit words from the TCP connection with the Java server. After those three words are read from the TCP connection, they are written to the character device file, which places them on the incoming FIFO to that particular CubeHash engine on the FPGA. The child process then

blocks reading four 32-bit words from that particular CubeHash engine's outgoing FIFO via the character device file. The first three words represent the result of the computation, and the last word represents the number of steps taken to reach that result. Once the four words are received from the CubeHash engine, the child process sends them over the TCP connection to the Java server. The child process then repeats the cycle, as shown in Figure 3.5, by blocking on a read of the next three 32-bit words from the TCP connection with the Java server.

3.4 Java Software

Java was chosen as the language for the server for several reasons. Java has support for SQL databases, and MySQL, a popular free and open source SQL database, provides a Java connector package which allows Java programs to use a MySQL database. Java has strong support for multithreaded applications. Java provides a LinkedList class which provides add and remove operations in constant time, which is very sufficient for passing data between threads. And Java has a relatively simple networking API.

The overall purpose of the Java server is to provide a connection between the PowerPC software (Section 3.3) and the MySQL database, and to provide the PowerPC software with unique starting values.

Figure 3.6 shows an overview of the threads and their tasks within the Java server. The initial thread of execution in the Java server checks the command line arguments, and then performs a number of setup functions. It initializes the connection with the MySQL database, it checks to make sure the required tables are present or creates them if necessary, it loads experiment resume data if applicable, and then it launches three new threads which run throughout the course of the experiment before the initial thread terminates.

The first thread that is launched is called the *OutputThread*. Throughout the server's execution, it reports a number of events which are currently going on, such as new PowerPC processes connecting over TCP, results returned, collision found, etc. This thread's sole purpose is to check a LinkedList for string data, and if there is data in the LinkedList, it should be removed from the LinkedList and printed to the output stream. The idea behind the *OutputThread* is that the other threads in the server should run as fast as possible, and writing to the output stream is a slow task compared to adding a string to a LinkedList (which is a constant time operation), therefore if the time-critical threads could just write a string to a LinkedList and a non-time-critical thread (*OutputThread*) took the time to write the strings to the output stream, then the time-critical threads could perhaps obtain a speedup.

The second thread that is launched is called the *MySQLHandler* which

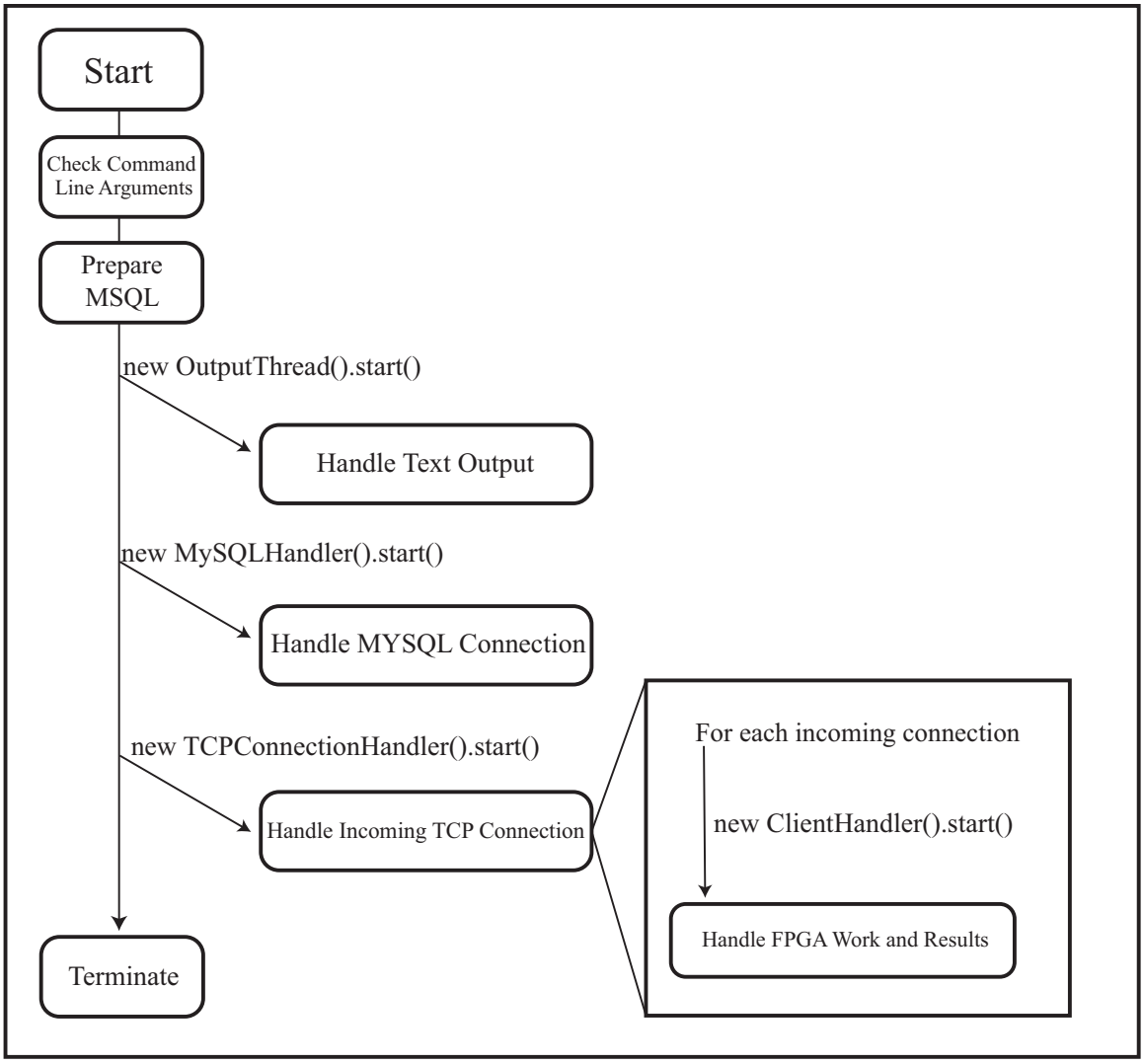

Figure 3.6: Overview of the Java Server Threads

handles the connection with the MySQL database. Nearly all of the MySQL operations in an experiment are insertion operations into the same table. It does not make sense to have multiple threads compete with each other for access to the Java MySQL connector when the MySQL database itself would have to provide some kind of thread synchronization anyway. Therefore, multiple threads that need to insert data into the MySQL database simply add that data to a LinkedList (in constant time) and the *MySQLHandler* thread removes that data and inserts it into the MySQL database. The *MySQLHandler* has sole access to the Java MySQL connector after it

is launched.

One special event that can, and does, happen during the experiments is when a duplicate distinguished value is inserted into the same table. The table is specially constructed within MySQL to only allow unique distinguished values, so the MySQL database will respond with an error message if a duplicate value is entered. The *MySQLHandler* handles this error by reading the current row containing the duplicate distinguished value and inserting the distinguished value and both sets of start and step data into a special table for storage of such information.

The third thread that is launched is called the *TCPConnectionHandler*. This thread simply listens on a TCP port and launches a new thread (*ClientHandler*) to handle each new incoming connection. The listen backlog is set to 100, which is equal to the maximum number of CubeHash engines on the current hardware. The thread can only accept one connection at a time, so the backlog is the number of pending requests that may await acceptance. If the number of incoming connections exceed the space available in the backlog, then each connection exceeding the backlog will be refused. In this experiment all connections should be accepted, which is why the backlog is set to 100 (at any point in time there are at most 100 active CubeHash engines).

There is one *ClientHandler* thread for each incoming connection. Connections come from the PowerPC software as explained in Section 3.3. The *ClientHandler* thread will send a unique starting value on its TCP connection and then wait to receive a distinguished value and the number of steps it took to reach that distinguished value. Then, the starting value, the distinguished value, and the number of steps is passed off to the *MySQLHandler* thread via a LinkedList, and the process repeats from the beginning.

3.5 MySQL Server

MySQL Server is an open source relational database software which is freely available for personal use. In this thesis, a MySQL server was used to store data very efficiently and to be able to quickly detect colliding hash chains. How this is accomplished is beyond the scope of this thesis. MySQL was simply used as a black-box tool. There were 4 tables that were used during the course of the experiment. The length of the binary fields were dependent upon the desired hash length in the experiment. Also, within the MySQL database, each experiment (with a set of 4 tables) was given a unique prefix so that data from one experiment was not mixed with data from another. This had an added benefit of allowing multiple experiments to run simultaneously.

Field	**Type**	**Key**
dval	binary	primary
start	binary	*None*
steps	int	*None*

Table 3.1: Data Table

Table 3.1 describes the data table. The data table stores every hash chain computed by the CubeHash engines. There are 3 fields. The first field is called *dval* which stands for *distinguished value*. This field stores the distinguished value at the end of the hash chain. This field is also the primary key for this table. That means that MySQL will automatically disallow any two rows from containing the same *dval*. That is useful, because in the Java server, as described in Section 3.4, whenever duplicate *dval*'s are placed in this table, MySQL will report an error back to the Java server. This error is evidence that two hash chains have collided. The second field in this table is called *start* which stands for *starting value*. This is the value that was given to a CubeHash engine to begin hash computations. This value is necessary for post-processing, that is, stepping through two hash chains that are known to collide in order to find the exact two points which hash to the same value. The *start* field is not a keyed field, because the Java server ensures that no two chains will start from the same starting value. The third field is

called *steps* which stands for *number of steps in the chain*. This field is also necessary for post-processing.

Field	Type	Key
dval	binary	*None*
start1	binary	*None*
steps1	int	*None*
start2	binary	*None*
steps2	int	*None*
message1	binary	*None*
message2	binary	*None*
commonHash	binary	*None*

Table 3.2: Duplicate Table

Table 3.2 describes the duplicate table. It is very similar to the data table as described by Table 3.1. Whenever two chains produce the same *distinguished value*, it is impossible to store them in the data table due to the primary key. So, both chains are stored in a row of the duplicate table. If two chains are going to collide, the first chain to be computed is stored in the data table. Then, after the second chain is finished computing, the Java server will attempt (and fail) to store it in the data table. The Java server will notice that the failure is due to a duplicate *distinguished value* and will store both chains in a row of the duplicate table. It should be noted that data is never removed from the data table, so the same chain will exist in both the data and duplicate tables. This is intentional, because the data table stores all unique *distinguished values* that have been found. It is also intentional that the duplicate table does not have a primary key, because a single *distinguished value* may be reached by more than just two chains, and when this does happen, there are duplicate *dval*, *start1*, and *steps1* fields. When a chain is copied from the data table into the duplicate table, its fields in the data table are placed in the *start1* and *steps1* fields. The new chain that ends with the same distinguished value has its fields stored in the *start2* and *steps2* fields of the duplicate table.

The last three fields in the duplicate table are only used for post-processing. In fact, the Java server does not store any data in these fields. An alternate program uses these fields to step through the hash chains and find the two

different messages which hash to the same value. The duplicate table is an ideal place to store this information. *message1* and *message2* are the two different messages, and *commonHash* is the hash digest of both messages.

Field	Type	Key
start	binary	*None*

Table 3.3: Resume Table

Table 3.3 describes the resume table. This table's purpose is to allow the search for hash collisions to continue in the event of a power failure or some other disruption. This table was not used during performance testing. There is only one field in this table and that is the *start* field. Whenever a new hash chain began computation, its *starting value* was stored in this table. Whenever a hash chain was completed and successfully stored in either the data or duplicate table, its *starting value* would be removed from the resume table. If the experiment were suddenly stopped and restarted (such as may be the case in a power failure) then the Java server would read in the *starting values* stored in this table and reissue them to CubeHash engines as they became available. The count would continue from the value with the largest magnitude.

Field	Type	Key
start	binary	*None*

Table 3.4: Loop Table

Table 3.4 describes the loop table. This is another table whose sole purpose was for resuming computations. Again, this table was not used during performance testing. There is only one field in this table and that is the *start* field. There is a threshold to the maximum length of a hash chain, and that was equal to the anticipated chain length times 20. If a CubeHash engine exceeded that chain length, it would abandon the chain and request a new one. Any *starting value* that led to a CubeHash engine abandoning the chain was stored in the loop table. The name of the loop table comes from the situation where a chain actually loops back around on itself causing a loop such as in the Pollard-Rho case. If, on that loop, no *distinguished values* were present then the chain computation would continue indefinitely. Therefore,

in an effort to reduce wasted computation time, whenever the experiment is suddenly stopped and restarted (such as may be the case in a power failure), the Java server will read the *starting values* stored in this table and know to not reissue them.

Chapter 4

Experimental Results

All hash chains terminated when there were 25 leading zeros. Initially hash lengths of 40, 48, 56, 64, and 72 bits were to be tested, but it was later found that when using 40 and 48-bit hash lengths, the hash chains did not terminate within the maximum allowable chain length (20×2^{25}), and therefore measuring parallel speedup with these hash bit lengths would not have produced any meaningful results. So this thesis continues with only 56, 64, and 72-bit hash lengths.

4.1 Procedure

As explained in Section 3.1, there were a total of four FPGA nodes. Each FPGA node consisted of a single FPGA and a single PowerPC processor. Also, as mentioned in Section 3.3, the PowerPC ran an embedded Linux environment which would talk to each CubeHash engine on the FPGA (described in Section 3.2). There were a total of 25 CubeHash engines on each FPGA. This resulted in a total of 100 CubeHash engines that could all perform CubeHash computations in perfect parallel.

The experiments were restricted such that each of the FPGA nodes could only be programmed for one particular hash variant. For example, Node 1 could be programmed for the 64-bit implementation of CubeHash, and Node 2 could be programmed for the 72-bit implementation, but Node 1 could not be programmed for both the 72-bit implementation and 64-bit implementation simultaneously.

This allowed for some experiments to run simultaneously. Multiple experiments running simultaneously has the potential to cause one experiment to impede another through competition for hardware access. For example, running two (or more) processes simultaneously on a desktop computer with a single processor may cause the processes to compete with each other for access to the processor, network card, and other hardware components. However, this was determined to not be a problem for these particular experiments due to two reasons. The first reason is that both the desktop computer and each of the PowerPC nodes (the only two components where this may occur) use a modern Linux operating system which is designed to fairly distribute limited hardware resources such as the processor, network card, etc. between multiple processes. The second reason is that all of the software processes involved in this thesis spend the majority of their time blocked on I/O (the PowerPC waiting for a starting value from the Java server, the PowerPC waiting for a result from a CubeHash engine, or the Java server waiting for a result from the PowerPC), and whenever processes are blocked on I/O, the Linux kernel is advanced enough to serve other processes that require more immediate attention. So, due to the fact that all of the processes spent most of their time blocked on I/O, it was very unlikely for multiple processes to request access to the same hardware at the same time. Even if multiple processes happened to request access to the same hardware at the same time, the Linux kernel could service the multiple processes in a time that is on the order of perhaps no more than a few hundred milliseconds, which is very insignificant compared to the size of the results, which was on the order of hundreds of seconds at least.

To start an experiment, the desired nodes would be programmed with the specific CubeHash variant. Programming a node involved placing an *ACE* file on the *FAT* partition of a *CompactFlash* memory card. The *ACE* file itself was composed of two different files. The first was the bitstream which is a file generated by the Xilinx tools. This file contained all of the information necessary for programming the FPGA. However, the other necessity in the *ACE* file was the Linux kernel. The kernel had to be configured to be compatible with the particular bitstream, otherwise the CubeHash engines could not be utilized by the PowerPC. The kernel also held information

(namely the Ethernet MAC address) which would allow the node to come up and receive the proper IP address from the local network. Once the bitstream and the kernel were available, the *ACE* file was generated and the *scp* program was used to transfer the *ACE* file to the *FAT* partition of the *CompactFlash* memory card. Then the node was rebooted used the *reboot* command, and upon startup the FPGA would be programmed. (If the node did not reboot correctly, typically due to a faulty kernel, the *CompactFlash* card would have to be manually ejected from the node and reprogrammed using a different computer.)

Once the node was programmed, the Java server had to be launched. The Java server took a number of command line arguments, including the number of leading zeros in a *distinguished value*; the CubeHash parameters r, b, and h (Section 2.1; the port which it would should listen on to receive incoming TCP connections; and the number of processors that were to be used in this experiment. These parameters helped to identify the unique set of MySQL tables (Section 3.5) that should be used for the experiment. Since the experiments were typically run remotely, and for long periods of time, the *GNU screen* program was used to ensure that the experiment would remain running even in the event of an *SSH* disconnect. Once the Java server was launched, it would run indefinitely, listening for incoming TCP connections and handling them appropriately.

The next step was to launch the PowerPC software. The PowerPC software took a number of command line arguments as well. It took the hash bit length, h; the largest device major number for the CubeHash engine; the smallest device major number; the Java server IP or DNS address; the port number that was listening on the Java server; and the name of shell script to create to terminate the experiment (See Listing 3.1). The PowerPC software was also executed in the *GNU screen* program to prevent accidental termination. Once the PowerPC software was launched, the experiment would continue until it was manually terminated.

Experiments could be run in parallel by using different processes for the Java server, each listening to a different port and using different MySQL tables. Also, the PowerPC software could use a different range of major device numbers to utilize different CubeHash engines.

The number of CubeHash engines chosen to be utilized in each experiment was determined by using powers of the $\sqrt{2}$ until the maximum number of CubeHash engines (100) was used. This method was chosen because it would produce a uniform horizontal spacing on a logarithmic scale, and there would be 14 data points per hash bit length. Each data point is the average interarrival time between finding the first 3 collisions.

The procedure repeated for each of the 14 data points for each of the 3 hash bit lengths: 56, 64, and 72-bits.

4.2 Results

P	56-Bit Hash			64-Bit Hash			72-Bit Hash		
	R	S	E	R	S	E	R	S	E
1	780	1.000	1.000	15226	1.000	1.000	375741	1.000	1.000
2	468	1.667	0.833	7428	2.050	1.025	187703	2.002	1.001
3	248	3.145	1.048	4881	3.119	1.040	125017	3.006	1.002
4	237	3.291	0.823	3659	4.161	1.040	93785	4.006	1.002
6	134	5.821	0.970	2410	6.318	1.053	62395	6.022	1.004
8	264	2.955	0.369	1777	8.568	1.071	46686	8.048	1.006
11	200	3.900	0.355	1209	12.594	1.145	33834	11.105	1.010
16	191	4.084	0.255	895	17.012	1.063	23192	16.201	1.013
23	54	14.444	0.628	598	25.462	1.107	16041	23.424	1.018
32	37	21.081	0.659	367	41.488	1.296	11462	32.781	1.024
45	51	15.294	0.340	196	77.684	1.726	8287	45.341	1.008
64	34	22.941	0.358	88	173.023	2.703	5662	66.362	1.037
91	23	33.913	0.373	180	84.589	0.930	3959	94.908	1.043
100	28	27.857	0.279	218	69.844	0.698	3590	104.663	1.047

Table 4.1: Table of Measurements

Table 4.1 shows the measured results from the experiments. *P* is the number of active processors (CubeHash engines). *R* is the average interarrival time between finding a collisions, expressed in seconds. *S* is the speedup which is defined as the average interarrival time for 1 proccessor divided by the average interarrival time for *p* processors $(\frac{R_1}{R_p})$. And *E* is the efficiency which is defined as the speedup divided by the number of processors $(\frac{S}{P})$.

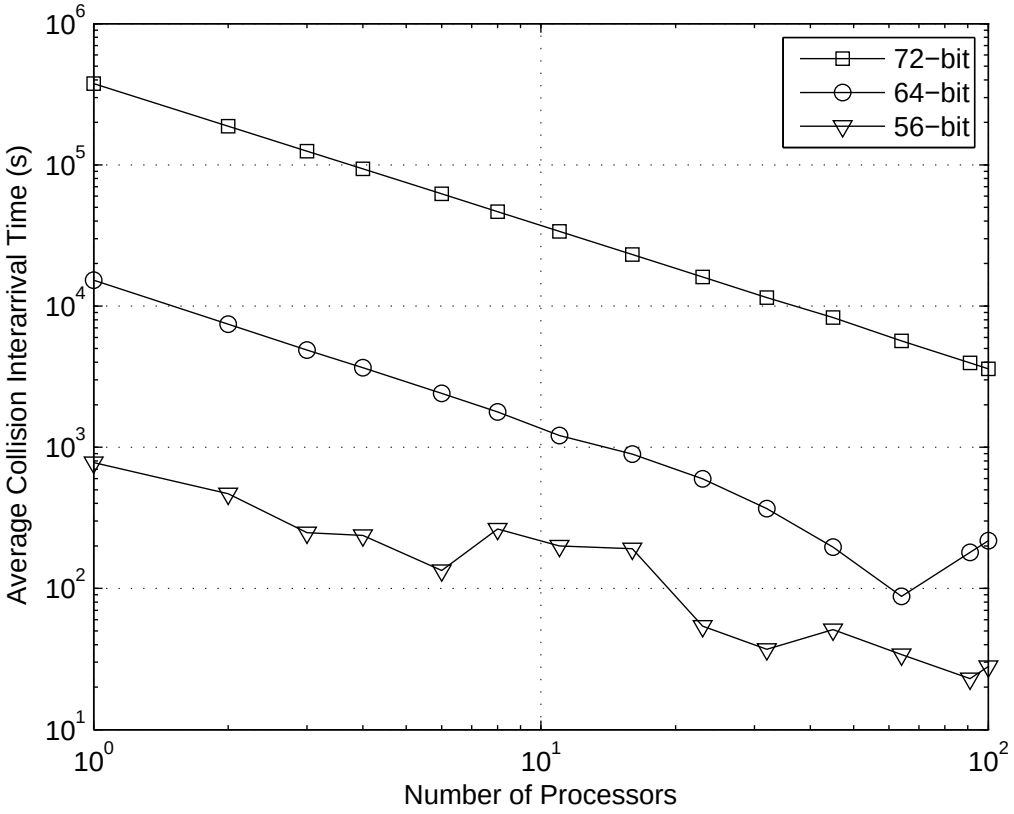

Figure 4.1: Average Interarrival Time (s) vs Number of Processors

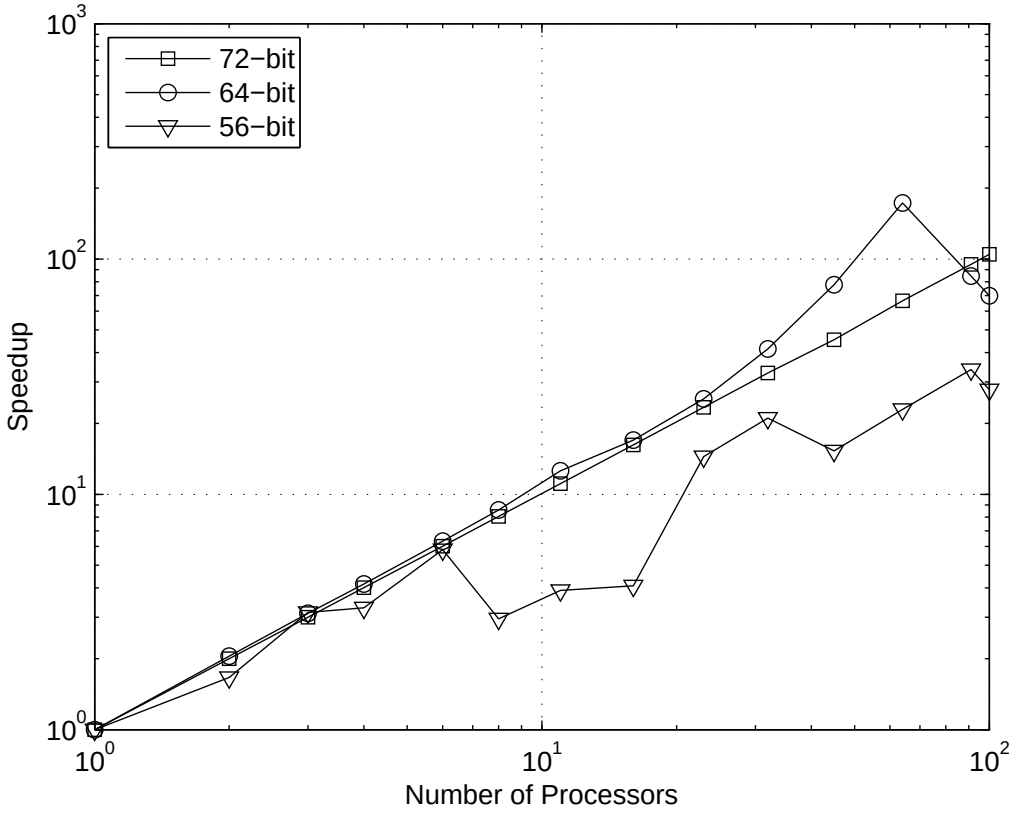

Figure 4.2: Speedup vs Number of Processors

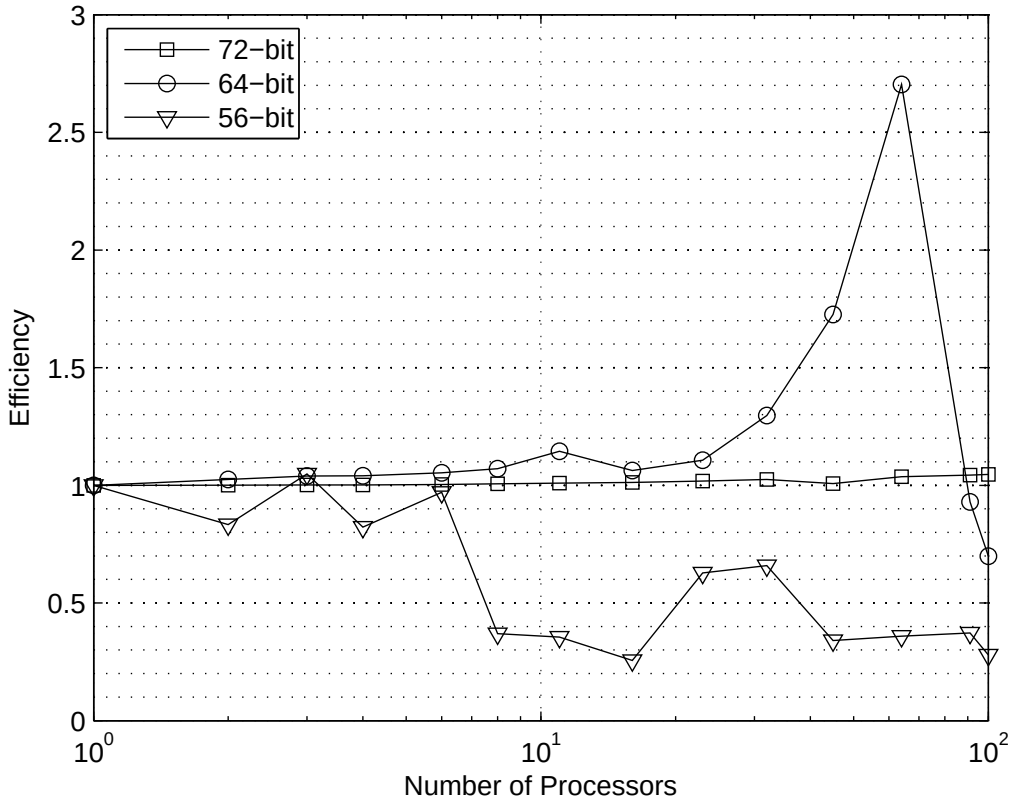

Figure 4.3: Efficiency vs Number of Processors

Hash Bit Length h	Computations to First Collision	Theoretical Computations to First Collision $\sqrt{2^h}$	Percent Difference
56	341,409,769	268,435,456	$+27.15\%$
64	1,688,899,546	4,294,967,296	-60.68%
72	64,432,211,152	68,719,476,736	-6.24%

Table 4.2: Measured vs Theoretical CubeHash Computations to First Collision

Hash Bit Length h	Chains until Three Collisions are Found
56	14
64	136
72	4,158

Table 4.3: Minimum Number of Chains Before Three Collisions are Found

4.3 Analysis of Results

The primary objective of this thesis was to determine if this generic attack on CubeHash could yield a linear speedup by adding more processors. The plots in Figures 4.1, 4.2, and 4.3 are constructed using the data from Table 4.1.

The plot in Figure 4.1 shows the average interarrival time of collisions vs the number of processors. This plot is useful in that it shows there is a general trend for each hash variant tested to gain some kind of speedup as more processors are added. The 72-bit hash variant shows this very well, but the 64-bit hash variant has more deviation, and the 56-bit hash variant has even more deviation.

The plot in Figure 4.2 shows the speedup vs the number of processors. Here it can be seen that there is a consistent speedup obtained with the 72-bit hash variant, however again the 64 and 56-bit hash variants are showing more deviation.

The plot in Figure 4.3 shows the efficiency vs the number of processors. Here it is clearly visible that a linear speedup is obtained in the 72-bit hash variant, because the 72-bit hash variant sticks very close to an efficiency of 1, and an efficiency of 1 means that the task is evenly divided among all of

the processors, and the work is completed at the theoretical limit. The 56 and 64-bit hash variants start off close to an efficiency of 1, but they quickly move away from an efficiency of 1 as the number of processors increases. The 64-bit hash variant's efficiency grows to nearly 3. This is the result of a *speedup anomaly*, which occurs when the ordering of work, not necessarily the processing power, causes a speedup.

Tables 4.2 and 4.3 show why 72-bit hash variant was able to exhibit an efficiency of 1, but the 56 and 64-bit hash variants moved away from an efficiency of 1. Table 4.2 shows the number of CubeHash computations that were performed in order to arrive at the first collision for each hash variant, and it shows what the theoretical number of hash computations is as well as the percent difference in the two numbers. Here it is shown that the 72-bit hash variant had approximately the same number of computations until the first hash collision as theory would predict, exhibiting only a 6.24% decrease. The 64-bit hash variant was able to find the first collision in many fewer computations than theory predicted, exhibiting a 60.68% decrease. This sharp decrease in the number of computations until the first collision is obtained and the spike in efficiency for the 64-bit hash variant as the number of processors increased may mean that using a distinguished value of 25 leading zero bits perfectly suited the 64-bit hash variant. The 56-bit hash variant however actually reported an increase in the number of hash computations that it had to perform compared to theory's prediction, exhibiting a 27.15% increase. This increase implies that each hash chain may have been too large to obtain a high efficiency. When there are 25 leading zero bits in a distinguished value, the average hash chain length is expected to be 2^{25} or $33{,}554{,}432$. Fewer leading zero bits may have provided a better efficiency for the 56-bit hash variant. Table 4.3 further supports the reasons stated why the 64 and 56-bit hash variants did not exhibit the same efficiency as the 72-bit hash variant. Table 4.3 shows the minimum number of chains that must have been computed in order to find the first 3 collisions. In the 72-bit hash variant, $4{,}158$ hash chains had to be computed. This is a large enough number that even the maximum of 100 processes could all be evenly loaded, because each hash chain is computed by 1 CubeHash engine. However, the 64-bit hash variant required only 136 chains to be computed before the first

3 collisions could be found. This does not provide for even loading, because, in the case of 100 processors, each processor would compute at least 1 chain, but only 36 had to compute a second chain. The uneven loading between processors is likely the cause of the deviation in speedup and efficiency. Also, the 56-bit hash variant only had to compute 14 chains before all 3 collisions were found. This explains why the efficiency dropped as more processors were added; the extra processors didn't perform any meaningful work since only 14 were required.

Chapter 5

Conclusion

In conclusion, this thesis was a success. The objective, which was to determine if the hash chain concept proposed by [12] actually did linearly increase with the number of additional processors, was proven to be true for the 72-bit hash variant. The 56 and 64-bit hash variants did not exhibit an efficiency of 1 as the number of processors increased, but using a fewer number of leading zeros or searching for a larger number of collisions may improve the efficiency of parallelizing these hash variants in this algorithm.

It should be noted that although collisions were easily found in these hash variants, the number of processors required to find a collision in a full-scale implementation of CubeHash is still too numerous for this attack to be feasible. Also, the memory requirement would certainly be a nontrivial factor in finding collisions in the full-scale CubeHash implementation.

Chapter 6

Future Work

A few interesting phenomena were observed throughout the course of this thesis. The first of which was that the number of leading zeros in a *distinguished value* could not be chosen to be arbitrarily high. In the case of the proposed 40 and 48-bit hash digests, 25 leading zeros was too large. There were no hash chains found of those bit lengths that led to a *distinguished value* with 25 bits. It was beyond the scope of this thesis to adjust the number of leading zeros in a *distinguished value*.

Another interesting occurrence was that in long runs, longer than the experiments recorded in this thesis, hash chains would hit the same *distinguished value* many times. This could be an anomaly in the CubeHash algorithm, or it could be a natural observation for a mapping into a finite space.

Bibliography

[1] Jean-Philippe Aumasson, Eric Brier, Willi Meier, Mara Naya-Plasencia, and Thomas Peyrin. Inside the Hypercube. In Colin Boyd and Juan Manuel González Nieto, editors, *ACISP*, volume 5594 of *LNCS*, pages 202–213. Springer, 2009.

[2] Daniel J. Bernstein. CubeHash specification (2.B.1). Submission to NIST (Round 2), 2009.

[3] Benjamin Bloom and Alan Kaminsky. Single Block Attacks and Statistical Tests on CubeHash. Cryptology ePrint Archive, Report 2009/407, 2009.

[4] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008.

[5] Jeremy K. Espenshade. Scalable Framework for Heterogeneous Clustering of Commodity FPGAs. Master's thesis, Rochester Institute of Technology, 2009.

[6] Dmitry Khovratovich, Ivica Nikolic', and Ralf-Philipp Weinmann. Preimage attack on CubeHash512-r/4 and CubeHash512-r/8. Available online, 2008.

[7] Arjen Lenstra, Xiaoyun Wang, and Benne de Weger. Colliding X.509 Certificates. Cryptology ePrint Archive, Report 2005/067, 2005.

[8] Cameron McDonald, Philip Hawkes, and Josef Pieprzyk. Differential Path for SHA-1 with complexity $O(2^{52})$. Cryptology ePrint Archive, Report 2009/259, 2009.

[9] National Institute of Standards and Technology. FIPS 180-2, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-2. Technical report, Department of Commerce, August 2002.

[10] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[11] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992.

[12] Paul C. van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *J. Cryptology*, 12(1):1–28, 1999.

Appendix A

Collisions Found

Message 1	Message 2	Common Hash Digest
07CFBC9EE2658C	A2495F1EE73118	A587A4CD3A1DAB
9D7E396E732612	C5FD3BA7E9F456	A33DDE4D03A214
77D68F6B34B6C6	005A94257053B4	9C82A32CD45BF4
92F067A002A2D7	35C7E3DBABFDE0	5C29ED91F684DB
C1D76B8DE9DE60	AE3B24519C5F62	3AD0DC26E9BF7C
9A70A63012A83E	2FEDF338BD3D57	7B58BE6D3FF52E
1DD02D92D03C65	68A84F530B2233	A6CA2573A7855C
C8E2FA6565C8CB	16649BAB0A9156	75924BA6ED05AE
D614686A1F9D86	C6AB8B5E195D3D	E49DD139B7CA2F
472CB9DF7F62E1	14B04ED7ED8FE0	268444FBBC031D

Table A.1: Collisions Found in CubeHash16/32-56

Message 1	Message 2	Common Hash Digest
859D1C1F9109B538	31B9F5ED983B0276	5A0A8BDF0585694D
13708A21364CD0AF	45C52B022BEE05C1	8E1392E57B7B1B2F
E00924082FDE4F13	4BB2E9D67A271D02	B453B73B0E31CF76
8B65E96E6BD720D9	92658FD26ED56B18	7F8EEBE90D9C6566
E00924082FDE4F13	4BB2E9D67A271D02	B453B73B0E31CF76
46D65E040D318C67	72D5214A817555D1	CCD562E35A95E872
589F9C3433111485	430B96FD4746E66D	F15752760D6221A6
422B747B7BC912CC	3FF05900A5887E4A	FFCCE3C3922EF699
EE813CF60821C081	8D1339D5511F0D47	276E09C3256F4964
B61448BC6C51A87F	C36A5591EA1A79AC	4BB808ABC629BE81

Table A.2: Collisions Found in CubeHash16/32-64

Message 1	Message 2	Common Hash Digest
63AF7BB5D1D47D4314	8D908ABD05AFC4F8C7	8BE130B58218BFF8BD
95421019743121AD63	CA1E1ABED52E84CD06	16D626AD23F3794156
04070D26AFBB6577F0	4ACEAF6E0399196751	E8C20E3666989FF754

Table A.3: Collisions Found in CubeHash16/32-72