Department of Computer Science

Submitted in part fulfilment for the degree of BEng.

Applications of Homomorphic Cryptographic Primitives in Blockchain and The Internet of Things

Tom Borthwick

06 May 2020

Supervisor: Prof. Delaram Kahrobaei

Acknowledgements

I would like to thank my supervisor, Prof. Delaram Kahrobaei, for all the valuable support and guidance she has provided throughout this project.

Contents

List of Figures

List of Tables

1 Executive Summary

As technology standards increase, so does the need for a more secure network. The Blockchain has been a more recent innovative technology which has often been discussed as the future of Finance, especially with the emergence of Cryptocurrencies such as Bitcoin. To design such a Blockchain network, we need to construct the "blocks" which contain the information about the transaction and unique value to identify this block; known as a "Hash".

A Hash is a unique key which points to an individual block in a blockchain. To construct a Hash, we require a Hash Function - a one-way compression function that converts an arbitrary-length message to a fixed-length hash value. The properties of Hash Functions include Pseudo-Randomness, Collision Resistance, Pre-Image Resistance and Second Pre-Image Resistance. In this paper, we discuss Tillich-Zémor's Hash Function implementation and how this Hash Function aims to withhold these properties. The Tillich-Zémor approach has been built upon a sound mathematical basis of Cayley's graph and matrices. This approach is efficient in computation, making it extremely useful for Blockchain solutions where a large amount of Hashs are needed to be computed efficiently. We discover, however, that there have been successful attacks on the Tillich-Zémor approach, and there needs to be further investigation into how we can fully make this approach fully collision-resistant. However, we also acknowledge the fact that reproducing collisions could be useful to us, particularly in implementing Redactable Private Blockchains by utilising a Trapdoor Hash Function.

In this paper, we will be implementing an expanded version of the Tillich-Zémor approach by [1]. Currently, there are very few actual code implementations of the original Tillich-Zémor approach, and later on, we will be creating the code for the expanded Hash Function so that we can investigate the defining bounds of the Hash Function by running experiments.

To achieve this aim, I reviewed the construction of the original and expanded versions of the Tillich-Zémor Hash Function and designed my implementation, which I created using the SageMath library, which utilises Python. After I successfully created the Hash Function, I decided upon specific hypothesis' I wanted to investigate. Before the investigation, I had four main hypotheses' that will either be proven or disproven following my results:

1. As we increase the size of the field, we will encounter a lower collision range.
2. That as the randomly generated polynomials: modulus, f and $\tilde{f}$ have their maximum degree increased, fewer collisions will occur.
3. The proven statement from [1], which states: "there can be no collisions for messages of up to n / δ", will be shown correct in my results.
4. The length of messages may potentially have an impact on collision resistance.

By running our tests against these hypotheses', I discovered that increasing the size of the field (n) and the degree of the polynomial defining the field (p) increases the collision resistance; due to Polynomials being more complex. We also note that increases these values increases the computation time, sometimes significantly. In an actual implementation, this means that we need to look into how to tighten this bound; making the Hash Function as efficient as possible.

A running problem we encounter throughout this paper is that in an ideal world; we would test every string possible, however that is extremely unrealistic, and therefore we utilise a data set of 100,000 strings. This "limited" data set size perhaps means that it is difficult to prove the statement that there can be no collisions for messages of up to n / δ.

We also see that there appears to be no real trend between the length of a message and its impact on collision resistance.

As an additional hypothesis to the one's above, I also investigate the impact of increasing n, p and δ in conjunction, rather than increase them linearly individually. We find from the results that increasing them in conjunction is just as effective as increasing a single value to infinity. This is an important consideration as we would ideally keep these values as low as possible to keep computation as simple, whilst of course maintaining the collision-resistance.

As future work, I suggest further investigation into how we could utilise this hash function that I have implemented into an actual Blockchain implementation. More specifically, I would like to see how we could convert the Hash Function into a Trapdoor Hash Function and utilise this in a Redactable Private Blockchain solution.

2 Introduction

Cryptocurrencies, such as Bitcoin and Ethereum, have often been discussed as the future of Finance. However, the technology behind it, Blockchain, has often been forgotten about as the building blocks of these Cryptocurrencies. A Blockchain is a public distributed ledger of records, which are called blocks. With the modern world now increasing more dependant on technology, the Blockchain needs to be preserved as a secure network.

Bitcoin is the most popular of these cryptocurrencies, and arguably the technology behind it is what brought so much attention to the idea of having a Blockchain network. Bitcoin is a public Blockchain which is immutable; this means that transactions can not be changed once they enter the network - in this sense the network is append-only and this is an essential part to the security and integrity of Bitcoin. This immutability is achieved through Hash Functions, which generate unique "Hashs" which are the identifiers for each block in the Blockchain. However, this immutable, append-only system is not always appropriate for all Blockchain networks, and there may be a need for a redactable Blockchain; where the trusted and authorised user(s) have a way to edit transactions in the Blockchain. A prototype of a redactable Blockchain has recently been developed by Accenture using Hyperledger[2].

There have been concerns that the Internet of Things is being developed rapidly without appropriate response in security standards. The Blockchain originally was a solution to this problem; however, security in this system is vital. Therefore, there has been throughout investigation into where we can improve the Cryptographic hashing algorithms used in Blockchains, such as SHA-1 and SHA-256.

Recently it has been seen by Researchers at Google and CWI Amsterdam [3] that they have produced their first “collision” for the SHA-1 hashing algorithm, which if exposed, leaves systems built on SHA-1 vulnerable. This advancement means that the SHA-1 algorithm is no longer as secure as other alternate methods, and hence signalling the importance of the development of a more secure hashing alternative.

Old implementations of Hash Functions relied on heavy iteration and pseudo-randomness to ensure that hashes generated are unique and unbreakable. However, in practice, this "randomness" could potentially lead

to a collision, where two distinctly different messages that are input into a hash function, may produce the same hash. This has led to the innovation of building hash functions using mathematical foundations, such as with the Cayley's graph as initially implemented by Tillich-Zémor [4] and later adapted further by [5].

However, Tillich-Zémor's approach was proved insufficient, and there was a need to expand on their work when a collision attack proved successful. In [1], they suggested a new Hash Function which used the foundations of the Tillich-Zémor approach.

In this paper, I will be implementing a Hash Function practically using the design discussed in [1]. I will then be using this hash function implemented to investigate its collision-resistance, changing the bounds of n and δ as discussed in their paper, which will change the girth of the Cayley's graph. I hope the results will be able to help us understand what the impact of the values on the collision range is, by producing a clear trend, and therefore produce a computationally efficient collision-resistant hash function which will be sufficient for future blockchain solutions.

This paper is aimed at academics who have limited to little knowledge of the area of blockchain and, more specifically, hash functions. When introducing key concepts and terminology, I will be aiming to make explanations as straightforward as possible to ensure this target area fully understands the direction of this paper.

3 Literature Review

In this chapter, I will be going over some critical literature that will improve understanding of the aims of this paper. I will be going over the necessary foundations of hash functions and hope to highlight the importance of a secure hash function in Blockchain solutions, as well as theories of how this can be achieved using a mathematical basis from innovations from other papers.

3.1 The Background of Blockchain

Blockchain technology has allowed data to be stored in a secure and encrypted environment. Originally invented in 2008 in a paper by Satoshi Nakamoto, the Blockchain was made to act as a public ledger for the Cryptocurrency Bitcoin.

The Blockchain is a distributed database of records or public ledger of all transactions that have been executed and shared among participating entities in the entire blockchain network. This network is built by constructing a record of “blocks”, which will store specific information; this includes the transaction, who is participating and unique hash code.

This hash code is used as a reference or address to identify the block in the overall Blockchain. When a new record is appened into the Blockchain, the last computed hash is then broadcast throughout the network [6] so that the new block can reference it; this constructs the chain structure as shown below in Figure 3.1.

In general, blockchains were designed to be decentralised. This means and all the blockchain network will not be stored on one central server, but instead shared across all the interested parties. In this way, the blockchain is difficult to tamper with.

Blockchains can be public or private. In a public blockchain, everyone on the network can write to it whereas in a private blockchain, the participants must be approved as a trusted party. A private blockchain is therefore, by nature, much more centralised as fewer participants are involved in the network. The advantage of a private blockchain is that since there are fewer entities involved, the speed of the network is much faster, meaning

Figure 3.1: How a hash is used in a blockchain. A diagram from [7].

more transactions can be processed.

A business that is utilising a private blockchain is Walmart, who is integrating with IBM's Hyperledger Fabric-based blockchain [2], this shows that there is a technological urgency towards moving to Blockchain technology.

3.2 The Background of Hash Functions

To create a hash which is secure and "tamper-proof", we require a high-quality Cryptographic Hash Function. A Cryptographic Hash Function is a one way and compression function that converts an arbitrary-length message to a fixed-length hash value, this can be represented as follows:

$$h : \{0,1\}^* \rightarrow \{0,1\}^n$$

The hash function should behave as much as a random function as possible; even a minor input change should produce a massively noticeable difference in the output of the hash function.

There are properties which hash functions must withhold, these include:

Pseudo-Randomness

A hash function should be fully deterministic. This means that given an input, it should always produce the same output.

Collision Resistant

Given two inputs A and B, where outputs are equal to H(A) and H(B); there should be no scenario where H(A) is equal to H(B). A hash output should be a unique code, which means that there should be no two outputs which are the same.

Pre-Image Resistance

Given output H(A) which is the output of the hash function, it is hard to find any message x where H(x) = H(A). This simply means that finding a specific output will be difficult.

Second Pre-Image Resistance

Given output H(A) and input a, it should be hard to find another input which maps to the same H(A).

These properties, however, are challenging to test. Coming up with test cases to aim to meet these properties would be exhaustive, and while if given unlimited resources, possible, the effort and computation expense is not worth it. In cracking the SHA-1 encryption, Google and CWI said it required nine quintillion SHA-1 compressions to generate the collision which leads to it being cracked [3].

Early day hash functions relied heavily on iteration and hypothetical pseudo-randomness to try to create a Hash output that was unique and had no meaning. However, while these hash functions create this "pseudo-randomness" property; they are not entirely collision-resistant. Given the amount of infinite possible messages that could be potentially be hashed, it is highly likely that two distinct messages could produce the same output from the hash function - breaking the collision-resistant property.

Instead, more recent hash functions have been designed on more mathematical foundations. NIST outlines algorithms for Secure Hash Standards, such as SHA-256, SHA-378 and SHA-512 [8], which show how mathematical basis can lead to a secure collision-resistant hash. Of course, these operations are potentially computationally expensive; and there is a trade-off between having a secure hash function and the amount of time it takes to compute that hash.

As we aim to keep raising the bar on what is considered to be a secure hash function (especially after SHA-1 having been cracked [3]), we are considering innovations on a mathematical foundation to help combat this problem. One of the ways to do this is by the use of a Cayley's Graph.

3.3 Use of Cayley's graph to improve hashing

A Cayley hash function is based on the idea of using a pair of semigroup elements A and B to hash to the 0 and 1 bit of a binary representation of a message. From selecting a pair of elements, A and B, there is the prospect that the Cayley graph generation would have a large girth and therefore would be provably collision-resistant.

In [9] we are given a detailed breakdown of how a Cayley's graph hash function is constructed as well as the benefits and drawbacks of them. There is mention to how a Cayley's hash function is efficient since it is computation can be parallelized because of its homomorphic property, such that H(AB) = H(A)H(B). This property means that larger messages passed into the hash function can be split into smaller pieces and be distributed to different computing units. The final hash matrix can then be derived from the final product of all the individual partial results from the computing units.

Tillich-Zémor were the ones to widely implement an idea of using Cayley Hash Functions to construct a Hash Function over the $SL_2(R)$ family [4] where R is a commutative ring defined as R = $\mathbb{F}_2[x]/I$ where I is an ideal generated by a randomly chosen irreducible polynomial of degree n. Since I is irreducible, R = $\mathbb{F}_2[x]/I$ is a field.

An irreducible polynomial is a non-constant polynomial that cannot be factored into the product of two non-constant polynomials. Tillich-Zémor suggested a prime number between 130 $\leq$ n $\leq$ 170 for the degree n of the irreducible polynomial I.

The degree is the highest power in the equation. For example, in $x^{143} + x^2 + x + 1$, the degree is 143.

The group SL_2 is the special linear group of all matrices with a determinant of 1. The choice of the SL_2 was justified by trying to create a high-quality hash function which created fast responses. The algorithm they proposed involved having two irreducible polynomials in the form of two matrices and defining a mapping between the 0 and 1 binary bit. As such:

- Let m be the string we are trying to hash.
- Let m' be the binary representation of m.
- Let X be an irreducible polynomial of prime degree n between 130 - 170.

Let A and B be the following matrices:

A = $\begin{pmatrix} X & 1 \\ 1 & 0 \end{pmatrix}$ B = $\begin{pmatrix} X & X+1 \\ 1 & 1 \end{pmatrix}$

Define the mapping:

$\pi : 0, 1 \rightarrow A, B$

$0 \rightarrow A$

$1 \rightarrow B$

From here, the hash is the product of all the matrices of the individual binary bits from m'.

Hash functions are supposed to be easy to compute and secure. The challenge is finding a way to optimise the parameters as much as possible to ensure an output is produced efficiently without the expense of losing security, such as ensuring the girth is wide enough to ensure that it is still provably collision-resistant. In [1], we see how we can simplify the matrices, and specific examples of matrices created using the Free Generators Theorem which generates a free group in $GL_2(\mathbb{F}_p[x])$. This table is shown in Figure 3.2.

$\{A, B\}$	A	B	b	$\tilde{a}$	$\tilde{b}$
$G_1(f, \tilde{f})$	$\begin{pmatrix} f & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} \tilde{f}+1 & 1-\tilde{f} \\ 1-\tilde{f} & \tilde{f}+1 \end{pmatrix}$	0	1	-1
$G_2(f, \tilde{f})$	$\begin{pmatrix} f & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} \tilde{f}+1 & \tilde{f}-1 \\ \tilde{f}-1 & \tilde{f}+1 \end{pmatrix}$	0	-1	1
$G_3(f, \tilde{f})$	$\begin{pmatrix} f & 0 \\ f-1 & 1 \end{pmatrix}$	$\begin{pmatrix} \tilde{f} & \tilde{f}-1 \\ 0 & 1 \end{pmatrix}$	1	-1	0
$G_4(f, \tilde{f})$	$\begin{pmatrix} f & 0 \\ f-1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1-\tilde{f} \\ 0 & \tilde{f} \end{pmatrix}$	1	0	-1
$G_5(f, \tilde{f})$	$\begin{pmatrix} f & 0 \\ 1-f & 1 \end{pmatrix}$	$\begin{pmatrix} \tilde{f} & 1-\tilde{f} \\ 0 & 1 \end{pmatrix}$	-1	1	0
$G_6(f, \tilde{f})$	$\begin{pmatrix} f & 0 \\ 1-f & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & \tilde{f}-1 \\ 0 & \tilde{f} \end{pmatrix}$	-1	0	1

Figure 3.2: Table from [1] which shows suggestions for matrices A and B generated from the Free Generator Theorem

I will be attempting to initially try one of these combinations to find collision resistance. I will using the first generator G_1 in the table, the parameters are as shown below:

$$G_1(f, \widetilde{f})\text{: A} = \begin{pmatrix} f & 0 \\ 0 & 1 \end{pmatrix} \text{ B} = \begin{pmatrix} \widetilde{f}+1 & 1-\widetilde{f} \\ 1-\widetilde{f} & \widetilde{f}+1 \end{pmatrix}$$

$$b = 0, \widetilde{a} = 1, \widetilde{b} = -1$$

$f, \widetilde{f} \in \mathbb{F}_p[x]$ such that f, $\tilde{f}$ are irreducible polynomials with zero constant terms.

p is a value where p > 2. p is the size of the field which determines which numbers are included [0,1,...,p_n]. For example, a p value of 4 would have the values of [0,1,2,3]. In this case, this will impact what values the coefficients of the polynomial may have.

There are known attacks to the Tillich-Zémor Hash Function initially proposed in [4]. These attacks were outlined by Grassl et all in [10], concluding that if collision resistance was essential, the original Zémor-Tillich Hash Function implementation was redundant. However, in [1], the original implemented was built upon, and they show how an educated choice of generators from the Figure 3.2 can give Hash Functions which are resistant to the known attacks given by [10].

The collision resistance of this incrementally built hash function will be provably resistant up to length n/δ. Where δ = max{deg f, deg $\widetilde{f}$}, in simpler terms; δ will be the maximum degree of a polynomial in matrices A and B. Later on, we will investigate the impact of changing the values of n, p and δ on collision resistance.

3.4 Attacks on The Tillich-Zémor approach

Since its introduction, the Tillich-Zémor approach remained virtually unbreakable for 15 years [11]. However, in a paper by Grassl et al. [10], they discovered attacks on the approach which produced collisions.

They did by observing that a collision will occur for the Tillich-Zémor Hash Function where certain properties of a message are met. They proposed that for a palindrome of even length, there is an efficient algorithm that can create a collision. Further details of this are given in the paper and details on the Euclidean algorithm proposed and used by Mesirov and Sweet. They conclude that the Tillich-Zémor Hash Function should not be used in applications where collision resistance is essential.

This was later expanded on in [11], where they showed how a tiny modification in the Euclidean algorithm would lead to collision and pre-images being recovered. They did this by replacing the matrices of A and B, as described in the previous section.

In the final discussion, they conclude that the proposed adjusted attacks break the Tillich-Zémor's Hash Function, and this makes the hash function insecure. However, they also suggest that the community not give up on it; due to its homomorphic design being an efficient implementation.

In a public blockchain where collision resistance is essential, this means that the implementation is not appropriate. However, these conditions for producing a collision could potentially be useful to us. Recall the earlier analysis of how a redactable private blockchain may be useful as there may be a need to edit data stored within a block. Therefore a collision may need to computed. This idea may be implemented through the use of a "Trapdoor Hash Function".

3.5 Homomorphic Trapdoor Hash Functions

As mentioned earlier, in a public Blockchain, immutability is key to the security and integrity of the whole network. In private Blockchains, entities in the network may have read-only access. However, there may be a need for the administrator of the Blockchain to change some data for whatever reason - this is known as a Redactable Blockchain. To achieve a Redactable Blockchain, we need to be able to compute a collision of a hash so that a block may be modified without damaging the integrity of the Blockchain itself. This can be achieved through the use of a Trapdoor Hash Function.

A Trapdoor Hash Function allows for a collision to a hash to be computed if the "trapdoor" is known. Formally:

For Trapdoor Hash Function H, we have a Trapdoor Key T.

We also have the Second Pre-Image Resistance property of a Hash Function, which states that given output $H(A)$, it is hard to find an x where $H(x) = H(A)$.

If T is unknown, there is no efficient algorithm to produce $H(A)$, and the Second Pre-Image Resistance property holds.

If T is known, there IS an efficient algorithm to produce $H(A)$, meaning there is a viable and efficient way to produce a Second Pre-Image.

To see the usefulness of producing a collision, we will consider the scenario where modification of the Blockchain is needed. A Hash of a block is computed by using the hash of the previous block and the fingerprint of the current block.

In a Homomorphic Hash Function:

Consider 2 sequential blocks in a Blockchain; B_1, B_2

We calculate the hash of a block x as: $H(B_x) = H(B_{x-1})H(F_x)$, where F_x is the fingerprint of the current block.

So in this scenario: $B_2 = H(B_1)H(F_2)$.

Therefore, if we were to edit B_1 and modify the hash, it would ruin the integrity of the whole Blockchain as B_2 has been computed on an out of date Hash. This is where our Trapdoor Hash Function becomes useful, as after we have modified B_1, we will produce a collision for $H(B_1)$ so that the hash of the modified block is not changed and preserving the integrity of the entire Blockchain.

4 Implementation of the Hash Function

In this chapter, I will be giving an account of how I implemented the hash function as given above in my literature review. I will be going over the design and implementation of the hash function. I will then also propose series of investigations on the bounds and parameters that are set in the initialization of the hash function.

4.1 Design and Architecture

As stated earlier, a Hash Function is a one way and compression function that converts an arbitrary-length message to a fixed-length hash value. This means that we will produce a function which has a singular input, the message m and an output hash value h.

Figure 4.1: A simple diagram of where our hash function fits in, with inputs and outputs

Beneath the hood, we need to consider step-by-step how our hash function will be computed. I have broken down the overall task of coding this hash function into small pieces, to help break down the problem further and fully understand how our message m is transformed from a meaningful message to a secure hash value H.

This has been visualised in Figure 4.2.

From the diagram, it can be seen that I have separated the values of n and δ so that they are inputs. I plan to change the value of n, p and δ and measure the amount of collisions that occur for the data set, gaining an

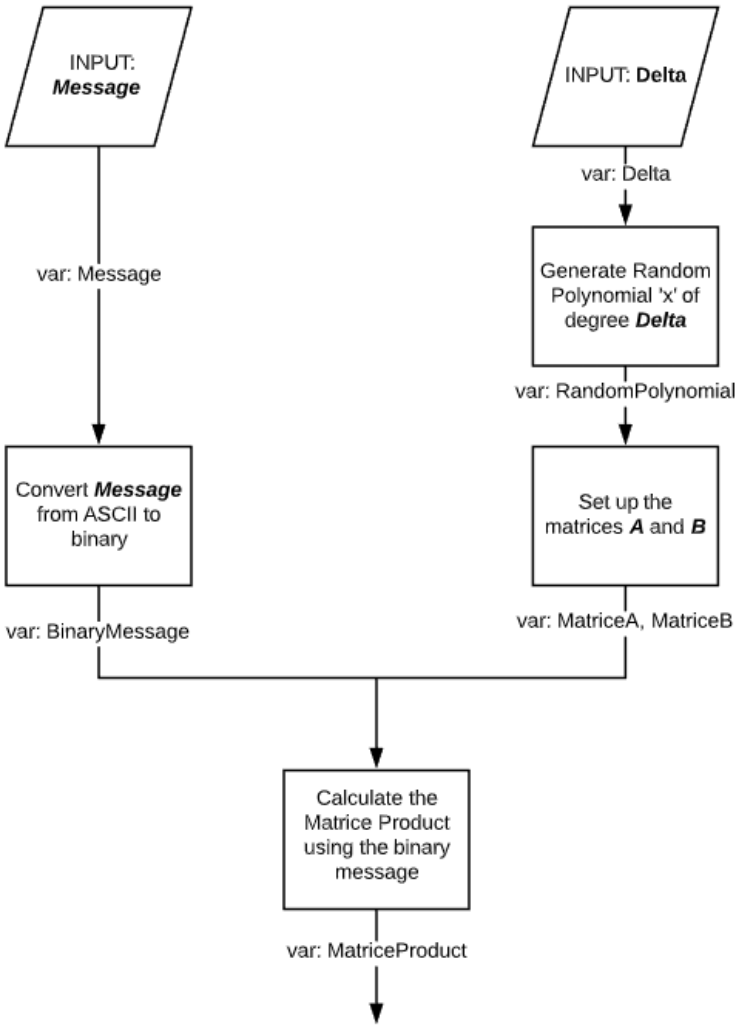

Figure 4.2: A visual representation of how our hash function will calculate the output hash

idea how each of these variables impact collision resistance. Therefore I need to be able to easily change these variables in my code.

As part of this paper, we will inputting a large amount of messages to check for the collision range. The hash function will be called with the message as a parameter.

4.2 Choice of Software and Tools

Since we will be dealing with a large number of messages that will be put into our hash function, we will need a tool which can compute numeric

calculations easily, efficiently and quickly.

Cocalc is a cloud web-based platform that provides a comprehensive selection of software environment and libraries. Cocalc as a tool is advantageous to me as a collaborative tool, easily allowing additional collaborators to view projects. This may be beneficial to me to get a second opinion on the overall logic of my code. In terms of numeric calculation, I am more interested in the SageMath environment, which is offered by CoCalc.

SageMath is a free, open-source math software that supports research in areas such as algebra, number theory and most importantly, matrices and cryptography.

As part of this work, I am going to be doing mathematical equations, including work on Matrices. SageMath offers the ability to define as Matrix as:

matrixA = matrix([[1,0],[1,1]])

From here, I can easily multiply matrices using the inbuilt functions in the Sage library. I will also be performing a large number of computations depending on the size of my data set, and Cocalc gives a generous data allowance to perform these.

I will also be working with types such as Fields and Polynomials. SageMath has an inbuilt package with comprehensive documentation to specifically deal with these more niche types related to Polynomial Rings etc. There is an inbuilt function to produce a random irreducible polynomial, which is a further example of how choosing this package will save me time and logic.

For my data set, I will need to create a text file which contains a random list of strings. For this paper, I would ideally have as many random strings of a certain length as possible. I decided that http://www.unit-conversion.info/texttools/random-string-generator would be a good choice to generate the file. This website allows us to choose what type of characters to allow, what length of strings to create and generates up to 10,000 unique random strings. From what I have seen, most websites charge a premium to generate more than this. I plan to generate 10,000 strings and then generate a further 10,000 and add them to my text file and continue doing so until the capacity of my program is discovered.

While implementing the hash function, I discovered that Cocalc did not provide enough memory to have an efficient capacity above 10,000 strings, as the free tier only provided 1GB of RAM. I ran my program through the SageMath console instead, which computed my program quicker and allowed me to increase my data set size to 100,000. There is always a desire to lift the bar on the size of the data set as we would have meaningful results, however, increasing the data set size puts a strain on the limitations on my hardware and increases the time taken to execute the full program.

For the purposes of this paper, I have outlined the specification of the computer that I am utilising for in Appendix A.8

4.3 Pseudo Code

Given that I am going to be evaluating collision resistance, I have to make sure that I am accounting for that in my design. Below, I have outlined the base Pseudo Code that I will be used to implement my overall program.

As such, I have created two Algorithms in Pseudo Code. The first is related to the actual Hash Function that will convert the message to a hash output, as shown in **Figure 4.2**. The first algorithm is shown in **Algorithm 1**. The second Algorithm relates to how I plan to measure collision resistance and initially set up the conditions to be passed into the actual Hash Function; this includes setting the Field and creating the pair of matrices that will be used as the mapping for the Hash Output. This is shown in **Algorithm 2**.

Algorithm 1: Hash Function

Parameters: $MessageToBeHashed, MatrixA, MatrixB$;
Convert $MessageToBeHashed$ to Binary via ASCII coding;
for *Each binary character in* $MessageToBeHashed$ **do**
  if *character is 0* **then**
    $MatrixProduct = MatrixProduct * MatrixA$;
  else
    $MatrixProduct = MatrixProduct * MatrixB$;
  end
end
Return $MatrixProduct$

With this algorithm, we have 3 parameters; these are:

- $MessageToBeHashed$: String that will be converted to a Hash Output.
- $MatrixA$ and $MatrixB$: The pair of matrices generated through the Free Generator Theorem with the polynomials input as f and $\tilde{f}$

The output will be the $MatrixProduct$, which is the Hashed value of

$MessageToBeHashed$.

Algorithm 2: Algorithm for finding collisions

Initialise Variables: $Degree, n, collisions$;
Set up Field $F_n[x]/I$;
Set up $MatrixA$ and $MatrixB$;
Set up empty list $HashOutputs$;
Set up empty list $InputMessages$;
Read in messages from file into $InputMessages$;
for *Each message in* $InputMessages$ **do**
    $Hash$ = HashFunction($message, matrixA, matrixB$);
    if $Hash$ *is in* $HashsOutputs$ **then**
        $Collisions$ = $Collisions$ + 1;
    else
    end
    Append $Hash$ to $HashOutputs$
end
Print "The total number of collisions were: " + $Collisions$

With this algorithm, we start off by initialising the variables to be utilised in this program. These are:

- $Degree$: The maximum degree of the Irreducible polynomial
- n: The size of the field
- $F_n[x]/I$: The field where n is the size of the field and I is a irreducible polynomial of degree $degree$
- $MatrixA, MatrixB$: The pair of matrices which have been generated by the Free Generator Theorem.
- $HashOutputs$: A list of the Hashs which are generated from the call to the Hash Function in **Algorithm 1**
- $InputMessages$: A list of messages to be hashed

The output of Pseduo Code will print out how many collisions there were throughout the input data set.

4.4 The Input Data Set

To have a fully provably completely reliable collision resistance hash function, we would have to check every single string possible; however, this would require a considerable amount of computation power and a massive input data set which is not possible. This means that we will have to be

dealing with a more realistic data set that we can generate and realistically perform the computation with the memory that we have available to us on CoCalc.

Earlier, in [7], we found that there would be no collisions for messages of length n/δ, where δ is the maximum degree of Matrix A and B. Therefore when investigating the collision range, I want to be able to prove as much as possible that this property stands in my hash function, and I will need an input data set where I have a large number of strings of a certain length.

For this paper, I initially set up a standard text file (.txt) which contains 10,000 messages of length 10 which will be an input into my program.

With Cocalc, and having a data set of much higher length (20,000-30,000), it took a very long time to execute and often ran out of memory before it had fully finished executing. This is because Cocalc offers only 1GB of shared RAM on which is can execute.

To have much more reliable results, I would ideally have a much larger data set than the specified 10,000. Therefore, I have also produced another file of the same format; however, with 100,000 strings. When executing my program from the SageMath command line, we can compute a data set of this size. As stated earlier, however, any way to increase this capacity would produce more meaningful results.

4.5 Implementation of Code

Having considered the architecture, design and choice of tools, I created my program in Cocalc; however I then moved to utilise the SageMath console once I saw the need for a larger data set.

This code and data sets can be viewed on Github at:
https://github.com/tombk24/Hash-Function-Dissertation [12].

In order for the results of my program to be meaningful, I have added logging in order to fully follow all the conditions for which a collision may occur. This includes logging:

- The initial parameters, such as n and δ that are being set at the start
- The irreducible polynomial being set for I.
- f and $\tilde{f}$, which will be input into Matrix A and B.
- Indicators to where the program is at, to highlight where the computation may be hanging.

I also plan to add comments to help describe the logic of the program

I have written, and therefore make it more readable then it currently is. On top of this, I would like to refactor code in some areas in order to improve readability; however, due to time restraints, I was not able to do this refactoring.

In order to reduce execution time, I am sure there are ways to reduce the number of operations in my program. For example, one of the ways I thought to do this was by sorting the hash output list and then checking for duplicate values at this stage. Before implementing a sorting algorithm in my list of hashes, when utilising a higher data set, my program took a noticeable time to execute (with above minimal parameters with a data set of 100,000, the program took around 3-5 minutes) due to the IF statement in Algorithm 2. The reason for this is that after calculating the value of a hash, the program needs to check the entire list of other output hashes to check for any collisions. With the sorting algorithm, the program took around 1-2 minutes in comparison.

4.6 How to use the program

In this section, I will outline how to run my program on the Cocalc environment and your personal environment.

Both $HashFunction.sagews$ and $HashFunction.sage$ follow the same logic, however the $HashFunction.sage$ file implements a different way to count collisions to deal with the larger data set. I implemented the Cocalc version before the version for the personal environment, and therefore there may be some "nice-to-haves" missing from the Cocalc version.

Cocalc environment

To run this program using Cocalc, import the $HashFunction.sagews$ file and a data set such as $randomstrings.txt$ into the files tab of the Cocalc environment.

Note that on the free tier of Cocalc, you should be able to have a data set of around 10,000 strings comfortably.

From here, you will be able to open the $HashFunction.sagews$ file and directly edit the variables of n, p and δ to change the bounds of the Hash Function. You'll also need to change the value of variable $dataLocation$ to the path of where your data set is.

Personal environment

To run the program on your personal computer, you will need to install the *SageMath* library. You will also need to download the $HashFunction.sage$ and a data set such as $randomstrings.txt$ to a known file location.

Once installed, open the SageMath console and change the active directory to where you have stored the downloaded files.

You may edit the variables directly in the code by opening the $HashFunction.sage$ and changing them.

To run the file, go to the SageMath console and execute the command $\%runfile\ HashFunction.sage$. This should start the execution of the program.

4.7 Known problems and future extensions

In this section, I will go over the known problems of my program, which I am aware of. Due to time constraints, I have not been able to correct these fully, and therefore I have highlighted them here just in case.

I will also indicate some areas where there is room for future extension, which would make the overall program of greater value. Yet again, due to time constraints, I have not been able to implement these ideas for expansion.

Known problems

Due to time limitations, I have not been able to fix certain problems that I have identified in my program. I have indicated these, as shown below:

- When computing a large data set (more than 1,000,000 strings), it will take a long time to execute.
- On Line 11 of the hashFunction function, I set A to be the first character in the binary string which can be either 0 or 1. This means we can not be sure if 0 is being mapped to A or B until the moment that it gets converted to a binary string. The reasoning for this lazy was because the program was recognising a 0 or 1 as an integer or string.

Future extensions

As with any program, there are always ways to hopefully improve efficiency to save time and computer resources overall. In this case, we are potentially dealing with a data set of above 100,000 strings and having to check for collisions in a list of 100,000 which is computationally expensive.

4.8 Evaluating collision resistance

In my program, there are multiple variables we can change that would potentially have an impact on collision resistance. These are:

- p: The size of the field.
- I: The modulus of p which is a randomly generated irreducible polynomial.
- n: The prime number that is the maximum degree of I.
- $MatrixA, MatrixB$: The matrices generated by the Free Generator Theorem.
- δ: The maximum degree of entries in MatrixA and MatrixB.
- The messages and output hash for which a collision may occur.

My plan for this section is to execute my hash function while changing variables to find a trend of when collisions may occur and ideally try to find some boundary as to where the hash function is collision-resistant. Having an idea of where this boundary is important with future implementations of a hash function with this mathematical basis, as we gain a better understanding of where the collision-resistant point is, and therefore we can reduce the size of these variables which will save pre-processing computation time. I will be changing and noting down the variables, and then recording down the number of collisions that occur in that data set.

There is a notable difficulty with trying to prove collision resistance with this method. As mentioned in 4.4, we would ideally try every string possible; however, we are testing 100,000 and just because we do not have any collisions in that set does not particularly mean the hash function is collision-resistant with the parameters input at the time.

In order to recreate any outputs, I have included all the possible variable values in each of the raw data sets. I have included this raw data in the Appendix. For this section, I often take away some fields to improve the readability of results, for the full tables; see the aforementioned Appendix.

Before the investigation, I have four main hypotheses' that will either be proven or disproven following my results:

1. As we increase the size of the field, we will encounter a lower collision range; this is because there are more unique values in the overall field and therefore less chance to have a collision.
2. That as the randomly generated polynomials: modulus, f and $\tilde{f}$ have their maximum degree increased, fewer collisions will occur.
3. The proven statement from [1], which states: "there can be no col-

lisions for messages of up to n / δ", will be shown correct in my results.

4. The length of messages may potentially have an impact on collision resistance.

I will be going through each of these hypothesis' and changing values in my hash function so that we can either prove or disprove these theories.

HYPOTHESIS 1: *As we increase the size of the field, we will encounter a lower collision range.*

With this theory, I expect that as the overall size of the field is increased, so does the amount of values from the output hash product H. With a much larger group, we can expect to have less of a chance of a collision, and therefore uphold the second pre-image property of hash functions.

To start, I wanted to find the lowest possible size of the field that we could have. To do this, I minimised all the possible parameters (n, p, δ), and then increase the value of p until we can see some trend.

As we had minimised the degree of f and $\tilde{f}$ to be 1 (and the choice of f and $\tilde{f}$ has a zero constant term), in every circumstance, the value was always x. In trying to notice a trend, keeping these values, the same is useful, as we can now see the direct impact that p has on the overall collision range.

The results of this are shown in Table 1, performed on a file of 100,000 strings of length 10 (Data Set 1).

n value	p value	δ value	Modulus	Collisions
2	2	1	$x^2 + x + 1$	99999
2	3	1	$x^2 + 1$	99280
2	5	1	$x^2 + 3$	70093
2	7	1	$x^2 + 1$	**99928**
2	7	1	$x^2 + 4$	12872
2	11	1	$x^2 + 7x + 7$	**13058**
2	11	1	$x^2 + 3$	553
2	13	1	$x^2 + 8$	231
2	17	1	$x^2 + 11$	62
2	19	1	$x^2 + x + 11$	17
2	21	1	$x^2 + 11$	2

Table 4.1: Seeing the impact increasing p with minimal other values

As can be seen from the results, there is an apparent decline in collisions as the value of p is increased by the next minimal prime number. The conjecture we can gather from this is: For a fixed value of n and δ, as p

increases, the number of collisions decreases to 0.

There are two anomalies to this, however, as shown in bold by the repeated tests when p = 7 and p = 11. There are significant increases in collisions for these values, and when the test is repeated, they fit the downwards trend. As the only variable that changes in each of these tests is the Modulus of the field, this likely is what is causing the increased collisions.

Looking into this further, firstly, when p = 7, we see that there is a varying rate of collision between the two Modulus' of $x^2 + 1$ and $x^2 + 4$. The only real difference between these two polynomials is the constant term, where we see lower collisions with the higher constant value. Concluding that a higher constant value leads to fewer collision values is dis-proven however, by the p = 11 values, where we see that a lower constant term leads to fewer collisions. As such, there are no apparent reasons for these anomalies, and it potentially vital that as future work, we investigate why these polynomials cause more collisions. If this hash function was set up in practice, we need to ensure that our randomly generated polynomial does not meet the same conditions as these collision prone ones.

Despite these anomalies, we can still conclude that generally, as we increase the value of p, the rate of collisions decreases. Despite this generalised conclusion, the question after that is at what point does the value of p make the hash function collision resistant? We can make p a substantial value and have a lower rate of a collision, but generally, the higher the value of p, the longer it takes to compute; and computation time in a blockchain is very important as we are potentially generating a large number of hashes.

It is also worth remembering that we are only testing 100,000 strings; if we had 10,000,000 strings, for example, we might encounter collisions at where we have seen 0 collisions for 100,000. This is worth looking into further by finding a way for our program to deal with a much larger input data set.

HYPOTHESIS 2: *That as the randomly generated polynomials: modulus, f and $\tilde{f}$ have their maximum degree increased, fewer collisions will occur.*

As we increase the value of n, the randomly generated irreducible polynomial I becomes more complex, I found that also led to hash computations taking much longer on average, especially as approaching the suggested value of n = 131 by Tillich-Zémor [4].

For this hypothesis, we increased the value of n as we kept the other variables minimal, as we did in Hypothesis 1. This changed the degree of the random irreducible polynomial generated by the modulus of the field $F_q[x]$.

I expect as the polynomial becomes more complex, we are expanding

the maximum possible values contained in the as Hash output H, and therefore we are decreasing the rate of a collision, potentially to a much greater extent than with just changing the value of p. This is because as we increase n, we get a lot more potential values than we do by increasing p than we do with n (increasing the number of polynomials of differing degrees rather than the coefficients).

n value	p value	δ value	Modulus	Collisions
2	3	1	$x^2 + 2x + 2$	99280
3	3	1	$x^3 + x^2 + 2$	80495
5	3	1	$x^5 + 2x + 2$	360
11	3	1	$x^11 + 2x^6$	0
13	3	1	$x^{13} + x^6 + 2x^4 + 2x^2 + 2$	0

Table 4.2: Seeing the impact increasing n with minimal other values

The results of gradually increasing n can be seen by Table 4.2. As with p, there is a clear decline in collisions as n increases by a much larger degree than previously seen. This shows that the degree of the polynomials involved with the hash function are much more important for complexity than the coefficients.

HYPOTHESIS 3: *The proven statement from [1], which states: "there can be no collisions for messages of up to n / δ", will be shown correct in my results.*

For this hypothesis, I will be aiming to prove that given the conditions for the hash function in [1], we will have a collision-resistant hash function.

My main data set has been 100,000 strings of length 10 with a character set of {a-z, A-Z, 0-9}, and I will be continuing to use this. I have set up the conditions so that n = 131, p = 3 and degree of polynomial to change. According to the theory, the hash function I have implemented will be collision-resistant such that 10 < n/δ. Theoretically, as we are setting n = 131, this means that δ will need to be a value of 13 or lower to satisfy the condition. I then increased the value of the degree so that the hash function *should not* be collision-resistant by setting δ to be 43 so that 10 < 131/43 is FALSE. After this, I decreased the size of strings and investigated the bounds again by having strings of length 5 instead.

As I was curious to see if the polynomial generated would have an impact, I ran each test 3 times and took an average as seen in Table 4.3. For each individual run, I also recorded the modulus and values of f and $\tilde{f}$; however, I have not included them in the presentation of this table. They are included in Appendix A.4.

As seen, by the table, we do not find any collisions as we increase the value of δ. I expected that we would have encountered some collisions

n value	p value	δ value	Data Set	Collisions
131	3	13	1	0
131	3	43	1	0
131	3	26	2	0
131	3	61	2	0

Table 4.3: The averages for the unique degree values from Raw Data 3.1, investigating the bounds of collision resistance

as we approached the n / δ boundary as we increased δ = 26 (which surpasses the boundary for messages of length 10), however, this is not the case. The reasoning for this could be perhaps we do not have a big enough data set to generate any collisions.

I wanted to take this hypothesis a step further, and generated a file of all possible strings of length 3 and repeated the investigation so that the bound was met and not met. My reasoning for this is that we would be testing all possible strings of length 3 and therefore, must be more likely to see a collision. The results can be seen in Table 4.4.

n value	p value	δ value	Data Set	Collisions
131	3	71	4	0
131	3	42	4	0
131	3	20	4	0
131	3	10	4	0

Table 4.4: Testing collision resistance for all strings of length 3

Surprisingly, as seen by the table, we get no collisions at all as the value of δ decreases. According to the theory, that we should get no collisions when n / δ, when n = 131 and δ = 71, we shouldn't see collisions for messages of length 3 however see collisions when δ passes the bound of 42. It is possible that since n is such a big number, we just simply do not have collisions; or perhaps there is an issue with implementation.

With this hypothesis, it is difficult to conclude either way, as we have not seen any collisions at all.

HYPOTHESIS 4: *The length of messages may potentially have an impact on collision resistance.*

The basis of this hypothesis lies behind the mapping of our Hash Function. If we have a string of length 3, there will be 3 mappings, and if we have a string of length 10, there will be 10 mappings. With the latter, this means we will be producing a more complex matrix polynomial product.

For this investigation, I needed some exact test for strings of length 10

where we see a noticeable amount of collisions. From previous runs, I found this to be the conditions where n = 2, p = 11, δ = 1 and there forth 553 collisions, as seen by Table 4.1. I have also set the conditions of the modulus and f and $\tilde{f}$ to consistent so that we truly measure the impact of message size.

I will then decrease the size of messages to 5 and then increase to 20 to see the impact that message size will have on collisions and potentially see a trend. The results of this are seen in Table 4.5.

n value	p value	δ value	Message Length	Collisions
2	11	1	5	580
2	11	1	10	553
2	11	1	20	586

Table 4.5: Table of results when changing the length of messages used in the data set

Whilst we see a varying number of collisions, there appears to be no clear conclusive trend between increasing and decreasing the length of messages that are input into the hash function. This may require further investigation using messages of much larger length or perhaps even a mixed data set of varying length of messages.

ADDITIONAL HYPOTHESIS 5: *A carefully selected choice of n, p and δ is just as effective in reducing collisions as linearly increasing a single one of these.*

I decided to create an additional hypothesis given the investigation I have undertaken. The reason for this is that most of my tests up to now have involved increasing a single variable and observing the number of collisions, therefore seeing a trend of what impact a single variable has on collisions. However, in practice, I believe that a more carefully selected choice of these variables in combination will keep the size of the field down while keeping the around the same range of collisions.

Given the conjecture earlier observed that: "for fixed values of n and δ, as p increases, the number of collisions decreases to 0", I wanted to see the impact of increasing n and δ in combination on the collision range. To do this, I selected a meaningful value from Table 4.1 (n = 2, p = 5, δ = 1, collisions = 70093) and observed the impact on collisions as we slightly increased n and δ. I aimed to keep the modulus of the field the same where possible to reduce the impact of other variables in the code (ie. f and $\tilde{f}$. The results of this are shown in Table 4.6.

As seen by the table, we can see that we can keep the values of n, p and δ to be lower if we increase them in combination, rather than increase a single value on its own. We would want to keep these values as low as

n value	p value	δ value	Modulus	Collisions
2	5	1	$x^2 + 3$	70093
3	5	1	$x^3 + 4x + 3$	1301
3	5	2	$x^3 + 4x + 3$	244
4	5	1	$x^4 + 3$	12
4	5	2	$x^4 + 3$	2
4	5	3	$x^4 + 3$	2
5	5	1	$x^5 + 4x + 2$	0

Table 4.6: Table of results when changing the values of n and δ for fixed p in combination

possible to reduce the complexity of computation, whilst maintaining the collision resistance of the hash function.

The conjecture from these results is that carefully selected values of n, p and δ is just as effective, if not more, than increasing a single variable.

5 Conclusion

To collate this paper, I have created a Hash Function which makes use of Matrices and Cayley's graph as described in [1], which could potentially improve the security of future Blockchain solutions. I then tested this hash function's collision resistance by changing the variables which initialise and define it. I came up for four hypotheses' before conducting experiments to prove and disprove them and have provided evidence as well as suggestions for future works.

In reference to my Hypotheses, I have found that increasing the values of both n and p has increased the collision resistance of the hash function; this is due to polynomials being more complex which leads to a larger output set being possible. Further work will be needed to find an optimal bound at which our hash function is as collision-resistant as possible.

One interesting observation I discovered from gathering my results was that computation time was sometimes very noticeably long. In a blockchain solution, such as Bitcoin, where more than quadrillions of hashes need to be computed per second [13], computation time is an important aspect. As touched on in Section 3.2, a hash function requires quick computation. While we should be aiming to make the hash function as collision-resistant as theoretically possible, increasing the size of the map output field to be a massive degree number is just impractical for trying to compute a high volume of hashes in terms of computation. As discovered in Hypothesis 1 and 2, the larger we make n or p, the longer computation takes.

As touched on many times in this paper, it would be ideal to find a way to increase the capacity of our input data set so that we can find a tighter bound for at which collisions occur. Then we can increase this bound slightly, so we are theoretically collision-resistant. Having generated a file of all possible strings of up to length 3 (including strings of length 2 and 1) and observing no collisions, we have in practice made a collision-resistant hash function for strings of up to length 3 as long as we keep the same conditions as tested (including the modulus, f and $\tilde{f}$). The challenge now lies in increasing the maximum length of strings we can provably put through a hash function and make collision resistant; while keeping his bound of n / δ as low as possible.

We also see that changing the length of messages being input into the Hash Function appear to have no set trend on the impact of collision-

resistance.

As an additional hypothesis, I investigate the impact of changing the values of n, p and δ in conjunction, rather than just linearly increase a single variable as I have been doing. We find that a carefully selected choice of these values is just as effective as increasing, for example, the value of n to infinity whilst keeping fixed p and δ values. In computing hash values, we want to keep the computations as simple as possible and therefore keeping these values as low as possible whilst maintaining collision resistance is an important consideration.

As further work, it may also be a consideration to trying using a different combination for the Matrices A and B. For this paper, I used the first pair of matrices suggested in the paper by [1]. Perhaps a different combination of matrices could yield better collision resistance, and an investigation comparing the different pairs may give results on what leads to better collision resistance given the required values for n, p and δ.

I also believe that actual implementation of a Blockchain solution which utilises this hash function could prove an interesting stepping stone into the future of how we design Blockchains. The hash function itself is computationally efficient as it utilises simple matrix computations which would make it ideal for a Blockchain where the hash rate is important to be as maximal as possible. The collision resistance of this hash function is also another benefit.

We could also take this work further by looking into how we could incorporate a "trapdoor" into the hash function, as described in Section 3.5. The benefit of this would allow private blockchain solutions which require a hash function where a collision may sometimes be required (second pre-image resistance), to utilise this technology.

Bibliography

[1] H. Tomkins, M. Nevins and H. Salmasian, 'New zémor-tillich type hash functions over gl2 (fpn)', *Journal of Mathematical Cryptology*, pp. 1–24, 2020.

[2] HyperLedger, *Walmart case study*. [Online]. Available: https://www.hyperledger.org/resources/publications/walmart-case-study, (accessed: 06.05.2020).

[3] M. Stevens, *Announcing the first sha1 collision*, 2017. [Online]. Available: https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html, (accessed: 06.05.2020).

[4] J.-P. Tillich and G. Zémor, 'Hashing with sl_2', in *CRYPTO 94*, Springer, Annual International Cryptology Conference, 1994, pp. 40–49.

[5] M. Larsen, 'Navigating the cayley graph of sl2 (fp)', *International Mathematics Research Notices*, vol. 2003, no. 27, pp. 1465–1471, 2003.

[6] M. Di Pierro, 'What is the blockchain?', *Computing in Science & Engineering*, vol. 19, no. 5, pp. 92–95, 2017.

[7] RubyGarage, *How a hash is used in a blockchain*. [Online]. Available: https://rubygarage.org/blog/how-blockchain-works, (accessed: 06.05.2020).

[8] Q. H. Dang, 'Secure hash standard', Tech. Rep., 2015.

[9] C. Petit and J.-J. Quisquater, 'Rubik's for cryptographers.', *Notices of the American Mathematical Society*, vol. 60, no. 6, pp. 733–739, Jun. 2013.

[10] M. Grassl, I. Ilić, S. Magliveras and R. Steinwandt, 'Cryptanalysis of the tillich–zémor hash function', *Journal of cryptology*, vol. 24, no. 1, pp. 148–156, 2011.

[11] C. Petit and J.-J. Quisquater, 'Preimages for the tillich-zémor hash function', in *International Workshop on Selected Areas in Cryptography*, Springer, 2010, pp. 282–301.

[12] T. Borthwick, *The code and files for this project*. [Online]. Available: https://github.com/tombk24/Hash-Function-Dissertation/, (accessed: 06.05.2020).

[13] HashGains, *What is hash rate?* [Online]. Available: https://www.hashgains.com/wiki/h/what-is-hash-rate, (accessed: 06.05.2020).

A Appendix

Raw Data has been included here, however some fields (generally, polynomials) have been cut off due to size restrictions, for the full polynomial, view the raw data file at [12].

Data Set	Size	Characters	Length
1	100000	A-Z, a-z, numbers	10
2	100000	A-Z, a-z, numbers	5
3	100000	A-Z, a-z, numbers	20
4	238328	A-Z, a-z, numbers	3

Table A.1: Table of the data sets used for this paper and their properties

n	p	δ	Modulus	f	$\tilde{f}$	Data Set	Collisions
2	2	1	x^2+x+1	x	x	1	99999
2	3	1	x^2+1	x	x	1	99280
2	5	1	x^2+3	x	x	1	70093
2	7	1	x^2+1	x	x	1	99928
2	7	1	x^2+4	x	x	1	12872
2	11	1	x^2+7x+7	x	x	1	13058
2	11	1	x^2+3	x	x	1	553
2	13	1	x^2+8	x	x	1	231
2	17	1	x^2+11	x	x	1	62
2	19	1	x^2+x+11	x	x	1	17
2	23	1	x^2+11	x	x	1	2

Table A.2: RAW DATA 1: Seeing the impact increasing p with minimal other values

n	p	δ	Modulus	f	$\tilde{f}$	Data Set	Collisions
2	3	1	x^2+2x+2	x	x	1	99280
3	3	1	x^3+x^2+2	x	x	1	80495
5	3	1	x^5+2x+2	x	x	1	360
11	3	1	$x^{11}+2x^6$	x	x	1	0
13	3	1	$x^{13}+x^6+2x^4+...$	x	x	1	0

Table A.3: RAW DATA 2: Seeing the impact increasing n with minimal other values

n	p	δ	Modulus	f	$\tilde{f}$	Collisions
131	3	13	$x^{131}+2x^{48}+...$	$x^{10}+x^3+...$	$x^{10}+x^4+...$	0
131	3	13	$x^{131}+x^{41}+...$	$x^{10}+2x^8+...$	$x^{10}+2x^4+...$	0
131	3	13	$x^{131}+2x^{26}+...$	$x^{10}+2x^2$	$x^{10}+2x^5+...$	0
131	3	43	$x^{131}+2x^{18}+...$	$x^{43}+2x^{41}+...$	$x^{43}+2x^{19}+...$	0
131	3	43	$x^{131}+2x^{74}+...$	$x^{43}+2x^{39}+...$	$x^{43}+x^3+...$	0
131	3	43	$x^{131}+2x^{104}+...$	$x^{43}+3x^7+...$	$x^{43}+2x^{24}+...$	0
131	3	26	$x^{131}+2x^{47}+...$	$x^{26}+2x^{19}+...$	$x^{26}+2x^6+...$	0
131	3	26	$x^{131}+2x^{15}+...$	$x^{26}+2x^{19}+...$	$x^{26}+x^{20}+...$	0
131	3	26	$x^{131}+2x^{95}+...$	$x^{26}+2x^2$	$x^{26}+2x^{23}+...$	0
131	3	61	$x^{131}+2x^{109}+...$	$x^{61}+2x^{40}+...$	$x^{61}+2x^{16}+...$	0
131	3	61	$x^{131}+x^{86}+...$	$x^{61}+x^3+...$	$x^{61}+2x^{48}+...$	0
131	3	61	$x^{131}+x^{87}+...$	$x^{61}+2x^{42}+...$	$x^{61}+2x^{38}+...$	0

Table A.4: RAW DATA 3: Testing whether we get any collisions for n / δ

n	p	δ	Modulus	f	$\tilde{f}$	Collisions
131	3	71	$x^{131}+x^{79}+...$	$x^{71}+2x^{50}+...$	$x^{71}+2x^{27}+...$	0
131	3	42	$x^{131}+x^{114}+...$	$x^{42}+x^{18}+...$	$x^{42}+x^{11}+...$	0
131	3	20	$x^{131}+2x^{70}+...$	$x^{20}+x^7+...$	$x^{20}+x^{11}+...$	0
131	3	10	$x^{131}+2x^{121}+...$	$x^{10}+x^5+...$	$x^{10}+2x^2$	0

Table A.5: RAW DATA 4: Testing collision resistance for all strings of up to length 3

n	p	δ	Modulus	f	$\tilde{f}$	Data Set	Collisions
2	11	1	x^2+3	x	x	2	580
2	11	1	x^2+3	x	x	1	553
2	11	1	x^2+3	x	x	3	586

Table A.6: RAW DATA 5: Testing the impact of varying message lengths on collision resistance

n	p	δ	Modulus	f	$\tilde{f}$	Collisions
2	5	1	x^2+3	x	x	70093
3	5	1	x^3+4x+3	x	x	1301
3	5	2	x^3+4x+3	x^2	x^2+x	244
4	5	1	x^4+3	x	x	12
4	5	2	x^4+3	x^2+x	x^2	2
4	5	3	x^4+3	x^3+4x	x^3+3x	2
5	5	1	x^5+4x+2	x	x	0
5	5	2	x^5+4x+2	x^2	x^2+3x	0
5	5	3	x^5+4x+2	x^3+4x	x^3+3x^2	0

Table A.7: RAW DATA 6: Seeing the impact increasing n and delta with fixed p

Computer Specs
Processor: Intel(R) Core(TM) i7-4790K CPU @ 4.00 GHz
Installed Memory (RAM): 16.0 GB
System type: 64-bit Operating System, x64-based processor

Table A.8: Brief specification of the computer used for this paper.