Secure Hash Algorithm 3

Courtney Cook
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
cookx876@morris.umn.edu

ABSTRACT

I discuss the Secure Hash Algorithm (SHA) 3, also known as Keccak. Keccak uses the sponge construction, unlike previous SHAs, and I give an explanation of how both the construction itself and the internal functions of Keccak work. I investigate the security of SHA3 and briefly compare it to that of SHA2 and SHA1.

Keywords

SHA, Sponge Construction, Keccak

1. INTRODUCTION

Nearly everything done online uses hash functions; from password storage to site certification. However, algorithms that take a long time to compute are not feasible in our fast-paced world, no matter how secure they are. SHA3 is the newest of the SHA standards, meaning all government facilities and government contractors will be using at least this level of security, so it is essential it is up to the task, both cryptographically and by its efficiency [6]. Whether it's password storage or the transfer of credit card information, a more widespread use of SHA3 will make the average user's information less likely to be stolen or changed in transit.

In section 2 I'll provide background information necessary to understand the function of SHA3, in section 3 I'll describe the base construction of SHA3, in section 4 I'll explain the details of SHA3's inner functions, in section 5 I'll provide details on the security of SHA3, and in section 6 I'll conclude.

2. BACKGROUND

2.1 Hash Functions

Hash functions are defined as any one-way function that can be used to map data of arbitrary size to data of a fixed size. Thus, the input cannot be calculated from the output, and no matter how long the input is, the output will be a set size dependent on which hash is being used. Hash functions are often used as a way to certify that data has not been changed. They should be collision-free and deterministic, which respectively mean that two different inputs should have different outputs and collisions are hard to find, and hashing the same input several times should result in the same output each time. When a document is run through a hash function, the output is a unique 'fingerprint' of that document, so if the document is ever changed the fingerprint will change as well. [7]

Two important properties of hash functions are diffusion and confusion. In order to implement confusion, a small change in the input should result in a large change in the output, and to implement diffusion that change should be spread out over the output, not concentrated in any one part of the input. This makes it difficult for attackers to guess at the inputs based on the relative outputs.

The National Institute of Standards and Technology (NIST) released a hashing standard in 1993, which was immediately recalled by the National Security Agency. The NSA fixed a few errors and then released Secure Hash Algorithm 1 (SHA1) in 1995. The previous hash was retroactively titled SHA0.

SHA1 uses the Merkle-Damgård construction, which processes each block of the input with a one-way compression function using message digests. It is susceptible to length-extension attacks, which is when an attacker adds more information to a message without changing the hash, so neither the sender nor the receiver realize anything was changed.[9]

SHA2 was released in 2001, also by NIST. SHA2 consists of six different functions which differ only in their digest length. It also uses the Merkle-Damgård construction, making it also somewhat susceptible to length-extension attacks.

An important factor in hash functions is how easily collisions can be found. Collisions are when two different inputs result in the same output. In 2015 a method of finding collisions in SHA1 was published, and in 2017 Google managed to actually find a collision [3]. Theoretically, a similar method could be used to find a collision in SHA2, because of its construction.

Even before methods of finding collisions were known, technology was evolving faster than the SHAs could keep up with. In 2006 NIST sent out a call for submissions for a new SHA to be sent in. Submissions closed in 2008. The entered functions went through rigorous testing and study, and the winner, built by Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, and called Keccak, was officially named SHA3 in 2015.[6]

2.2 Operators

In this section we introduce the necessary notation for the rest of the paper.

UMM CSci Senior Seminar Conference, November 2018 Morris, MN.

Exclusive Or, written XOR and denoted $\oplus$, is adding bits modulo 2. If both bits are 0 or both are 1, the result is 0. If only one bit is 1, the result is 1. It is applied bitwise, without any carrying over. See the examples below.

AND, $\wedge$, is bitwise multiplication. The only way for the result to be 1 is for both factors to be 1, otherwise the result is 0.

When operating on strings of more than 1 bit, line up the strings and work straight down:

$$\begin{array}{rr} 0011 & 0011 \\ \oplus\ 0101 & \wedge\ 0101 \\ = 0110 & = 0001 \end{array}$$

NOT, $\neg$, switches all 1s to 0s and all 0s to 1s. For example, 110010 becomes 001101. This operation only requires one input string, instead of at least two like XOR and AND. [8]

2.3 Notation

$\mathbb{Z}_2^*$ is the set of all strings comprised of 0s and 1s, including the empty string, which is a string with no 0s and no 1s. Strings of 0 and 1 are where all work takes place.

A series of n 1s or 0s will be denoted 1^n or 0^n, therefore $1^3 0^2 1 = 111001$. If the amount is not specified or is variable, we use 1^* or 0^*. This could also mean no 1s or no 0s, and thus be the empty string.

Let the concatenation of strings A and B be denoted A||B. Concatenation is putting strings together consecutively, so $101||001 = 101001$.

Truncating a string M to its first ℓ bits is denoted $\lfloor \mathrm{M} \rfloor_\ell$. Thus $\lfloor 1100010 \rfloor_5 = 11000$.

2.4 Linear Feedback Shift Registers

Shift registers are a set of binary states that shift over as new data is inputted. A linear feedback shift register (LFSR) is a shift register in which the next input value is determined by the previous values. It is expressed as a polynomial, with the highest power denoting the size of the internal state and each subsequent power denoting which values will be XORed together to determine the next input.[7]

For example, $x^3 + x^2 + 1$ means that the internal state is 3 bits long, and the bits at spaces 2 and 0 will be XORed. If we start with a value of 101, the 1 at the right end is outputted and that 1 and the 1 at the left end are XORed to get 0, then shifted to the right, so the next state is 011 and the rightmost bit becomes the output: 1. Then the bits 1 and 0 are XORed for 1 and the 0 is outputted, so the state is 101 and the output so far is 10. You can keep going, concatenating the output bits to the previous output, until the output is as long as needed. LFSRs are often used to make pseudo-random number generators, because if an LFSR has a large unknown polynomial the outputs cannot be predicted.

LFSRs will start to repeat after a period of time that depends on the polynomial, but is at most $2^p - 1$ where p is the highest power. If a long non-cyclic output is needed, a very large-degree polynomial is required.

2.5 Padding

Hash functions require the input length be a multiple of a specific block length r, so padding is concatenated to the end of the input message. It also can be used to obfuscate the input, so padding is added even if the original input is already a multiple of r in length. Let the message and padding together be denoted as M||pad[r]. Padding often needs specific attributes, so we will use the following definitions from [1].

Definition 1. A padding rule is sponge-compliant *if it never results in the empty string and if it satisfies the following criterion:*

$$\forall n \geq 0, \forall M, M' \in \mathbb{Z}_2^* : M \neq M' \implies$$

$$M||pad[r] \neq M'||pad[r]||0^{nr}$$

That is, if two messages are not the same, then the padded version of the messages will not be the same, nor will they only differ by the concatenation of n block-lengths of zeros. Thus sponge-compliant padding schemes cannot be of the form 'pad with 0s until it is x blocks in length.' In addition, some padding must be added to ensure that the empty string is never the input.

Definition 2. Simple padding, *denoted by* pad10*, *appends a single bit 1 followed by the minimum number of bits 0 such that the length of the result is a multiple of the block length.*

Simple padding will pad at least one bit, 1, and at most the number of bits in a block, 10^{x-1}. Here x is the block length. For example, given a string 010110 and a block length 4, simple padding will concatenate 10 to the end for a result of 01011010, which is 8 bits long. This is often not sufficient for security, because XORing a bit with a 0 results in that bit, which is then viewable and can be used to decrypt the rest of the block if enough padding was appended. That is, if the block length is 32 and the input is 32 bits, 10^{31} will be appended, giving a whole block of mostly 0s. The next simplest padding scheme is needed.

Definition 3. Multi-rate padding, *denoted by* pad10*1, *appends a single bit 1 followed by the minimum number of bits 0 followed by a single bit 1 such that the length of the result is a multiple of the block length.*

Multi-rate padding ensures that the last block of input is not composed entirely of zeros. It appends at least two bits, 11, and at most the number of bits in a block plus 1, $10^{x-1}1$. Given the string 0101101 and block length 4, Multi-rate padding will add a 1 then 3 0s, then another 1, for a result of 010110110001, which is 12 bits long.

3. SPONGE CONSTRUCTION

SHA3 is built using a method known as a sponge construction [2]. A sponge construction has variable-length input and output, up to multiples of its block size r. This makes it very versatile and leads to an easier implementation than other functions. [1]

3.1 Overview of Sponge Construction

A sponge construction consists of a fixed-length compression function f, the padding to be used, and the bitrate r, which is the same as the block size. All the work of the construction and function will run in a state, denoted S. The state has two parts: the bitrate and the capacity. Sometimes the capacity c is also specified in the function call, but if not, c is calculated using the given r and the default value of s, which is the length of S. The sponge construction is so

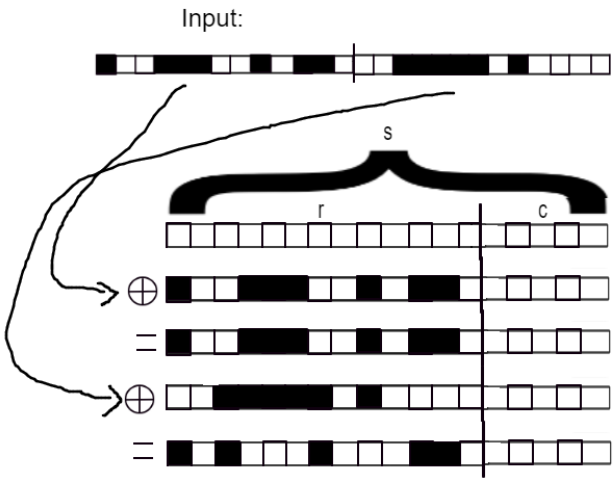

Figure 1: XORing blocks into the state, without f

named because it has two main parts: the absorbing phase and the squeezing phase.

To begin, all the bits in S are initialized to zero. The input message is padded following the rules given in Section 2.5 for multi-rate padding and cut into blocks of length r. The r-bit input blocks are XORed into the first r bits of the state, interleaved with applications of f. This means that the function f is run between every block XORing. This compresses the input down into an s-sized block. Thus, $S^i = f(S^{i-1} \oplus \text{block}_i)$. Figure 1 shows an example of blocks being absorbed into S, without the interleaving of f.

Once all the message blocks have been processed, the state is squeezed: the first r bits of the state are returned as output blocks concatenated with any previous output, interleaved again with applications of f. The number of output blocks is chosen by the user.

The last c bits of the state are never directly affected by the input, nor are they ever directly output by the squeezing. That is, the capacity bits of a state s^i are never output during the squeezing of state s^i, though depending on the compression function some may be outputted later. This makes length-extension attacks impossible under normal operation, because an attacker never knows the entire state at a single time.

3.2 Example of Sponge Construction

Let us call the function sponge[g, 10^*1, 4] where g is the compression function, and g = Circular Left Shift by 1. The bitrate is 4, and the capacity is 3. The input will be 0101101, and we want our output to be 12 bits long.

The state is set to 0000000. The input 0101101 is concatenated with the padding, then split into 4-bit size chunks: 0101 1011 0001. Each chunk is then concatenated with 0s until they are the length of the state, so the capacity bits are not changed: 0101000, 1011000, and 0001000. The state becomes the old state XORed with each chunk and then run through g in rounds until all blocks have been absorbed.

round 1:

S = 0000000 //the state is set to all zeroes

S = 0000000
$\oplus$ 0101000
= 0101000

$S = g(S)$ = 1010000 //the state is g(previous state)

round 2:

S = 1010000
$\oplus$ 1011000
= 0001000

$S = g(S)$ = 0010000

round 3:

S = 0010000
$\oplus$ 0001000
= 0011000

$S = g(S)$ = 0110000

Once we absorb the entire input the squeezing begins. We'll use Z as the output.

$Z = \lfloor S \rfloor_4$ = 0110

round 1:

$S = g(S)$ = 1100000
$Z = Z || \lfloor S \rfloor_4$ = 01101100

round 2:

$S = g(S)$ = 1000001
$Z = Z || \lfloor S \rfloor_4$ = 011011001000

Now our output is the desired length, and we return Z = 011011001000, which is the hash of the input 0101101.

This is a simplified example, with our compression function being a circular shift. The actual compression function used for SHA3 is much more complicated.

4. THE INNER FUNCTION

Most of the work done in Keccak is done by its inner f function, of which there are 7 different permutations, called Keccak-f[b]. Here b=$25 \cdot 2^\ell$ and ℓ goes from 0 to 6, so b ranges from 25 to 1600 and is the width of the permutation. A larger b means a higher security level. Each Keccak-f[b] is a permutation over $\mathbb{Z}_2^b$, the set of all strings composed of 0s and 1s that are of length b.

Each permutation Keccak-f[b] is a series of operations on a state α, which can be best visualized as a three-dimensional array, as opposed to the sponge state S which is thought of as one-dimensional. α has the construction α[5][5][w], with 5 and w being the lengths of each array and w = 2^ℓ. Then, for each x and y between 0 and 4 and z between 0 and w-1, α[x][y][z] corresponds to the position (x, y, z), which is a bit 0 or 1. This can also be written as α[x, y, z]. See Figure 2 for terminology used to describe the state α.

For the purposes of switching between the three-dimensional Keccak state α and the one-dimensional sponge construction state S, α maps to S by

$$\text{S[w(5y+x)+z]} = \alpha\text{[x][y][z]} \quad (1)$$

For example, in Keccak-f[25], α[1][1][1] maps to S[7]. All operations on x or y are done modulo 5, and operations on z are done modulo w.

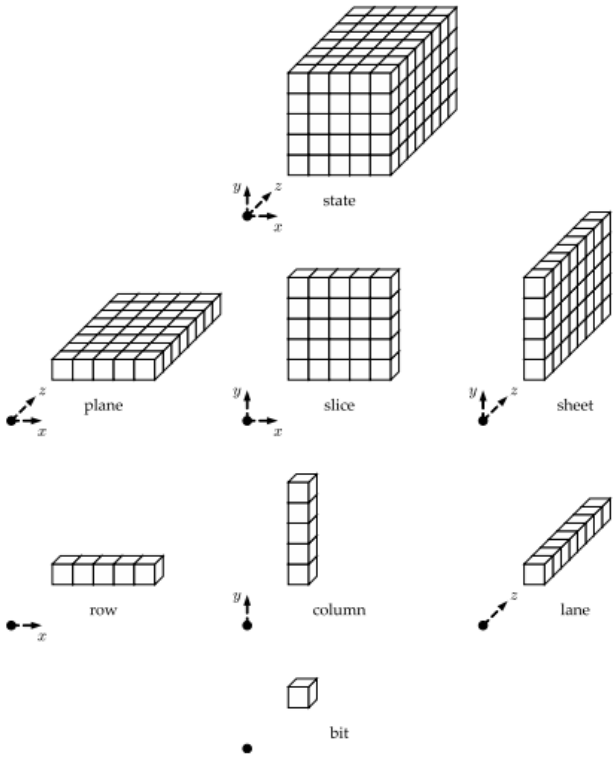

Figure 2: Terminology for the state α [2]

Keccak consists of n_r rounds of R, where $R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$. Each function is defined below. $n_r = 12+2\ell$, and thus changes for each variation of Keccak. The pseudocode in subsequent sections is taken from [2].

4.1 Transformations of Keccak

ROT(a, b) indicates all bits of a being shifted over by b spaces, that is, $a[x] \rightarrow a[x + b \bmod \text{(w-1)}]$, where w is the length of a. Each of the following subsections details the functions of R, starting from the right, which are applied first.

4.1.1 *Transformation θ*

Figure 3: Theta

1: **for** x = 0 to 4 **do**
2: $\quad C[x] = \alpha[x, 0]$
3: $\quad$ **for** y = 1 to 4 **do**
4: $\quad\quad C[x] = C[x] \ \oplus \ \alpha[x, y]$
5: $\quad$ **end for**
6: **end for**
7: **for** x = 0 to 4 **do**
8: $\quad D[x] = C[x-1] \oplus \ ROT(C[x+1], 1)$
9: $\quad$ **for** y = 0 to 4 **do**
10: $\quad\quad \alpha[x, y] = \alpha[x, y] \oplus \ D[x]$
11: $\quad$ **end for**
12: **end for**

As seen in Figure 3, each lane in the bottom row of x is put into a new array C, with one entry for each row. The lanes in the rows above are XORed into the ones below, flattening the state into a 2-dimensional array. Then another array is constructed, D, and its entry in each index is the XOR of the C entries in the previous and next indices, with the next index shifted one bit over first. See Figure 4, where the bit marked with o is the XOR of the bits marked with x. The original state α is then XORed with its corresponding D entry and becomes the new state.

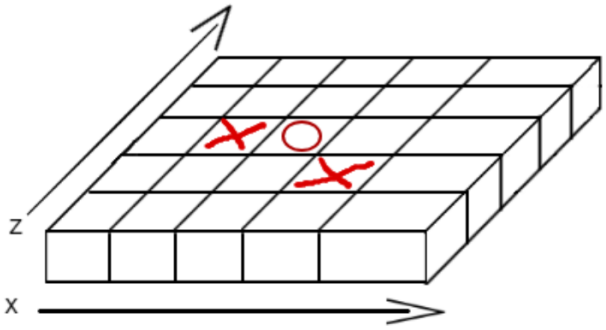

Figure 4: Example of how a bit in D is calculated

θ is run in order to increase the diffusion of Keccak, and is always run first, though the order of the other transformations was arbitrarily chosen.

4.1.2 *Transformation π*

Figure 5: Pi

1: **for** x = 0 to 4 **do**
2: $\quad$ **for** y = 0 to 4 **do**
3: $\quad\quad \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$
4: $\quad\quad \alpha[x, y] = \alpha[a, b]$
5: $\quad$ **end for**
6: **end for**

The state is permuted by lanes according to the matrix multiplication shown in Figure 5, so the lane at [1,1] becomes the lane at [1,0]. Note that [0,0] remains the same. Figure 7 shows how the lanes are permuted, with [0,0] being at the center. π is run to increase long-term diffusion.

4.1.3 *Transformation ρ*

Each lane in the state is shifted by a function of t, given in Figure 6, Line 4. The order in which they are shifted is given by the matrix multiplication, and so the order is also given by Figure 7. Thus, the lane at (0,0) is not shifted. The value of t changes for each lane shift. The function ρ is run for diffusion between the slices.

Figure 6: Rho

1: $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
2: **for** t = 0 to 23 **do**
3: $\quad \alpha[x, y] = ROT(\alpha[x, y], (t+1)(t+2)/2)$
4: $\quad \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$
5: **end for**

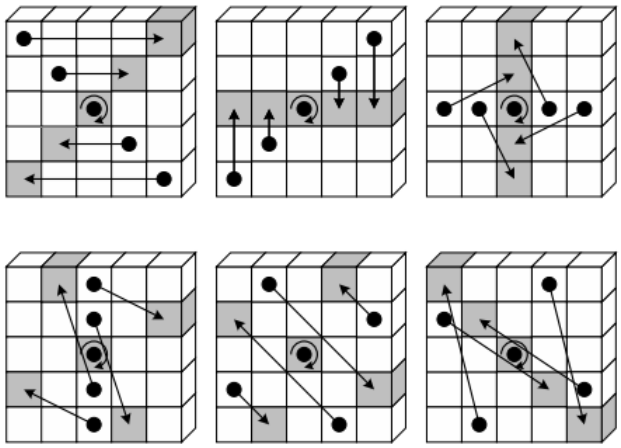

Figure 7: Permutation of rows in π and order of shifts of rows in ρ [2]

4.1.4 Transformation χ

Figure 8: Chi

1: **for** x = 0 to 4 **do**
2: **for** y = 0 to 4 **do**
3: $\alpha[x, y] = \alpha[x, y] \oplus ((\neg\, \alpha[x+1, y]) \wedge \alpha[x+2, y])$
4: **end for**
5: **end for**

The NOT of the next lane, ANDed with the lane two over, is then XORed with the current lane (Figure 8, Line 3). AND is non-linear because it is a bent function, meaning it is as far from a linear function as possible and difficult to approximate. Because XOR is linear but AND is not, χ is the only transformation in Keccak that is non-linear, and is important for this reason.

4.1.5 Transformation ι

Figure 9: Iota

1: $\alpha[0][0] = \alpha[0][0] \oplus RC_{i_r}$

Round Constants are introduced to disrupt symmetry, because symmetry can be exploited. The number of bits that are not equal to 0 is set to $\ell + 1$, so as ℓ increases, so does the asymmetry. The Round Constants are different between rounds, and are determined by a linear feedback shift register:

$$RC[t] = (x^t \bmod x^8 + x^6 + x^5 + x^4 + 1) \bmod x$$

This LFSR means that the exponent x^t is reduced to its congruent polynomial in mod $x^8 + x^6 + x^5 + x^4 + 1$, similar to how integers can be reduced mod an integer. Then that polynomial is reduced mod x, meaning all values of x are set to 0. The resulting bit will be either a 1 or a 0, the coefficient of x^0. Thus $RC - i_r$ are the round constants for round i_r. The Round Constants are XORed with a single lane in the state, α[0,0], but will be propagated to all lanes within a single round by χ and θ.

4.2 Running

To begin, S is initialized to 0^s, which is called the root state. The function is called by Keccak[r,c], where, again, r is the bitrate and c is the capacity, and r+c=s. A high bitrate and lower capacity will be faster, but a lower bitrate and a high capacity will be more secure. The default c value is 576, and the default r value is 1600-c. Therefore if r and c are not specified, r is equal to 1024 and c is equal to 576.

The first block of the input is XORed into S. Then S is mapped to α via the function in Equation 1 and R, the composition of the five functions, is run $n_r = 12 + 2\ell$ times. α is mapped back to S and the next block is XORed in. This continues until the entire message has been absorbed. It is then squeezed, still mapping between S and α and running R n_r times in between each output truncation.

5. CRYPTANALYSIS

Differential fault analysis (DFA) is an attack where attackers inject faults into the internal intermediate variables and then analyze the difference between the outputs to recover the entire state. Faults are when a bit is flipped, either intentionally by an attacker or accidentally. Algebraic fault analysis (AFA) is more powerful than DFA in that it is both more effective and efficient than DFA and also can be automated to a high degree, thus reducing the manpower needed for analysis [4].

Lou, Athanasiou, Fei, and Wahl demonstrated that attackers can insert a byte fault into the penultimate round of SHA3, where $l = 1$. A byte fault is when single byte of a message is changed, some of the bit values switching between 1 and 0 and some remaining the same. Though they have no control over where the fault is inserted in the state nor what the fault is, they can then see both the correct output H and the faulty output H'. Each is a set of equations: H = $\iota^{23} \circ \chi \circ \pi \circ \rho \circ \theta \circ \iota^{22} \circ \chi \circ \pi \circ \rho \circ \theta(\theta_\iota^{22})$ and H' = $\iota^{23} \circ \chi \circ \pi \circ \rho \circ \theta \circ \iota^{22} \circ \chi \circ \pi \circ \rho \circ \theta(\theta_\iota^{22} \oplus \Delta\theta_\iota^{22})$, where $\Delta\theta_\iota^{22}$ is the injected fault. That is, given the input θ_ι^{22} for the penultimate round of SHA3, the output will be H. θ_ι^{22} means the value of theta after the 22nd time round constants are introduced in iota, zero-indexed.

To use AFA, they turn these equations into a boolean satisfiability problem, also called an SAT, and then let their automation of the problem run. SATs are problems asking if there is a way to insert either TRUE or FALSE for each variable and have it evaluate to TRUE. If there is no way for this to happen, such as in "a AND NOT a," the SAT is said to be unsatisfiable.

Using H and H', χ_ι^{22} can be found. Because all the operations in R are reversible, once the entire internal state at any stage has been found, the original message can be found. Finding χ_ι^{22} required 39360 variables and 52160 equations, but was computed in seconds using CryptoMiniSat, a SAT solver. This can be improved by first identifying the fault, than narrowing the search space accordingly [4].

Faults are injected several times and the outputs are compared in order to retrieve the entire χ_ι^{22}. On average, 80 bits can be recovered from each fault, though they often overlap, so more than $|\chi_\iota^{22}|$ / 80 fault injections are needed.

To protect against injected faults and random errors, many cryptological functions either make two copies of each message or use parity checking. Parity bits are non-message bits added into a string, used to check that all bits in a string

were transmitted correctly. Luo, Li, and Fei offer a structure for parity checking in Keccak that is both time- and resource-efficient [5]. They propose that the parity checks for χ and ι can be merged, and the checks for π and ρ can, at worst time efficiency, be merged, at best are unneeded entirely. Thus, only two or three parity checkers are needed to cover all five functions of R.

6. CONCLUSION

A user of the Secure Hash Algorithm 3 has remarkable control over how the function will operate. While hashes in general are used to map data to fixed-size strings, SHA3 offers many different output sizes. Users can adjust to their preferred level of security versus operation time using r and c and by using different permutations of the function. Because SHA3 uses a different construction than SHAs 1 and 2, it is not vulnerable to length-extension attacks nor can a collision be found in the same manner as for SHA1. The functions in R are such that diffusion and confusion are present, so SHA3 is secure. As technology gets better, our protection needs to match. Given that a collision has been found for SHA1, SHA3 is the next step in internet security.

Acknowledgments

Thank you to Elena Machkasova, my advisor and senior seminar professor, for her advice, feedback, patience, and for teaching a cryptography class that I found incredibly enjoyable. Thank you also to my alum reviewer Kevin Arhelger and to my second reader Nic McPhee for their helpful criticism.

7. REFERENCES

[1] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Cryptographic sponge functions. *SHA-3 competition (round 3)*, 2011.

[2] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak reference. *SHA-3 competition (round 3)*, 2011.

[3] Cryptology Group at Centrum Wiskunde Informatica and Google. Shattered. `https://shattered.io/`. [Online; accessed 4-December-2018].

[4] P. Luo, K. Athanasiou, Y. Fei, and T. Wahl. Algebraic fault analysis of SHA-3. *Proceedings of the Conference on Design, Automation Test in Europe*, 2017.

[5] P. Luo, C. Li, and Y. Fei. Concurrent error detection for reliable SHA-3 design. *Proceedings of the 26th edition on Great Lakes Symposium on VLSI*, 2016.

[6] N. I. of Standards and Technology. SHA-3 standard: Permutation-based hash and extendable-output functions. *FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION*, 2015.

[7] C. Paar and J. Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[8] Wikipedia contributors. Bitwise operation — Wikipedia, the free encyclopedia. [Online; accessed 2-November-2018].

[9] Wikipedia contributors. Secure hash algorithms — Wikipedia, the free encyclopedia. [Online; accessed 4-December-2018].