I. Introduction

This written report briefly summarizes the history from the SHA-1 algorithm to the recently declared SHA-3 standard. It begins by defining the symmetric and asymmetric cryptography, defines the hash function and the main differences between the types of encryption and the details on the broken SHA-1 hash function. The second part describes the SHA-3 competition organized by the National Institute of Standards and Technology (NIST) to address this attack. The third part will be SHA-3 encryption in details.

II. Breaking SHA-1

A. Different types of encryption

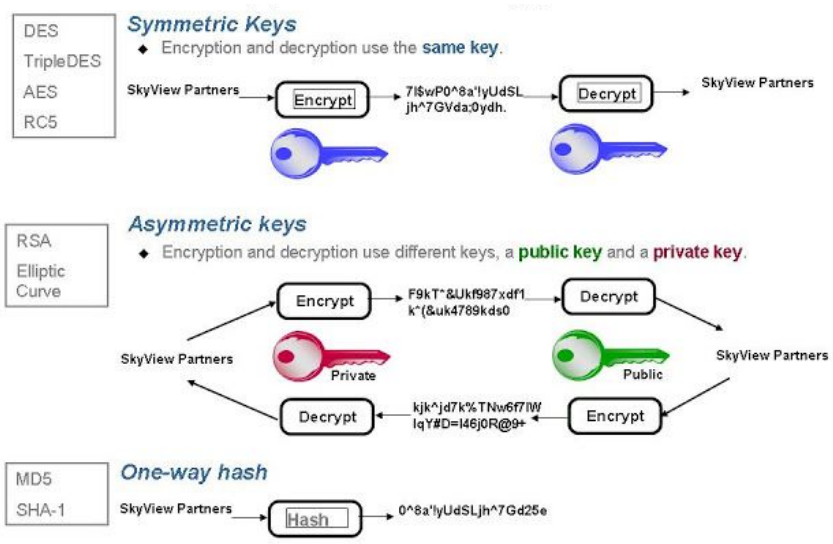

Fig.1 Types of Encryption [s1]

Encryption is a way to hide a message or information using an input and giving an output not readable for those who are not authorized. This is typically divided into three different types. Symmetric encryption takes an input using the same key for encryption and decryption. The main issue with symmetric encryption is the key exchange. Asymmetric encryption uses two keys instead, one for encryption and the other one for decryption, which resolved the key sharing issue but is much slower than symmetric encryption. Lastly, encryption by hashing takes an input to produce a hash value called a message digest that is of fixed length and unique to each input. Hashing is very different from other form of encryption as it takes no key as input.

B. Hash Functions

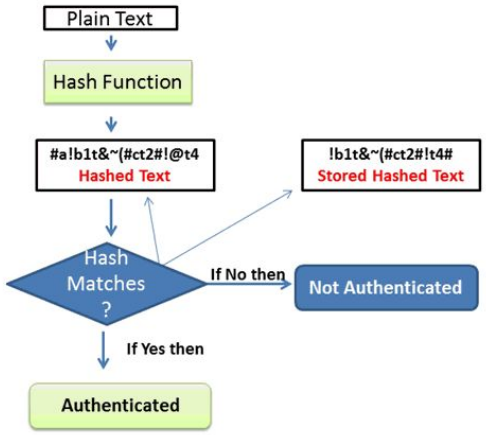

Fig. 2 Flow Chart example of how a hash works [s2]

Cryptographic hash functions are designed to take a string of any length as input and produce a fixed-length hash value. They are used to uniquely identify secret information which means that it should be very hard to find two inputs that will generate the same hash value.

The main function of the hash function is to preserve the integrity of the content by detecting all modifications and thereafter changes to a hash output. Cryptographic hash functions are mostly used for the integrity of the content checks.

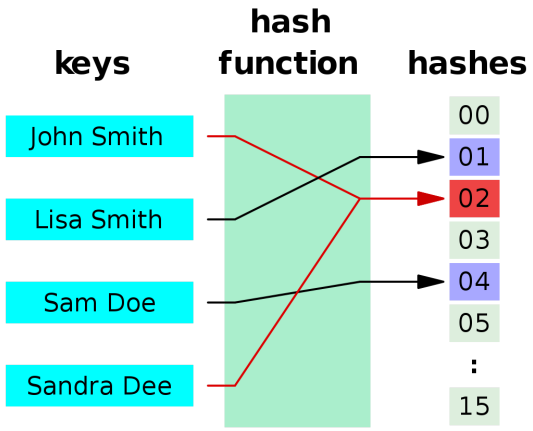

Fig. 3 A hash function that maps keys to the hashes. Collision between John and Sandra [s3]

C. SHA-1

SHA-1 was designed by the United States National Security Agency (NSA) and published by National Institute of Standards and Technology (NIST) as a federal standard [1] in 1995. SHA-1 is the successor of SHA-0 with was quickly deprecated because of flaws identified

by NIST. SHA-1 takes input and produces a fixed size output of 160-bits. Each block is 512 bits and has 80 rounds. SHA-1 used Merkle-Damgård construction with is used in the design of many popular hash algorithms such as MD5, SHA-0, and SHA-2.

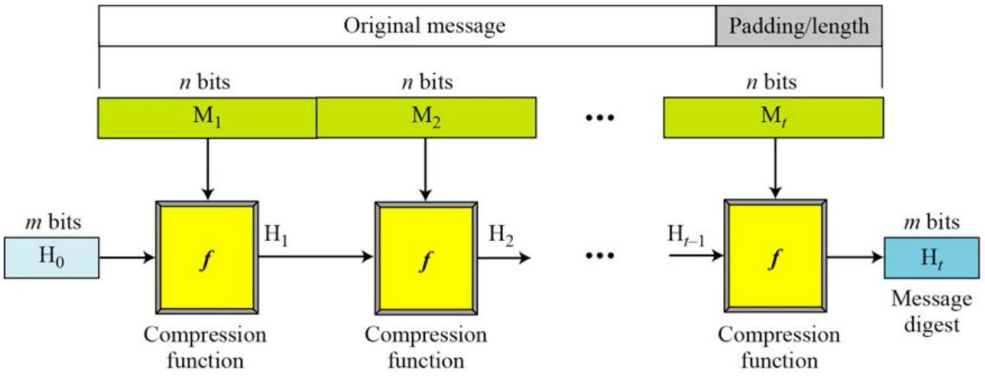

Fig. 4 Merkle-Damgård scheme [s4]

The Merkle–Damgård construction or hash function is a method of building the collision-resistance that a hash function requires from a one-way compression function [s5]. The Merkle–Damgård hash function take the original message (input) and divides it into blocks to feed the compression function in a different round. The output of the first compression function is the input for the second compression function. The second block of the message is used for the second compression function and the block size is the remaining bit of the original message. If the size is not enough to be considered as a block, the remaining bit of the original message is followed by the difference of bits of padding to fill it up. The padding function is used to create an input whose size is a multiple of a fixed number (e.g. 512 or 1024) this is because compression functions cannot handle inputs of arbitrary size. The hash function then breaks the result into blocks of fixed size and processes them one at a time with the compression function, each time combining a block of the input with the output of the previous round. In order to make the construction secure, Merkle and Damgård proposed that messages be padded with a padding that encodes the length of the original message to avoid length extension attack.

D. SHA-1 Collision attack

The hash function is by definition a one way function with strong collision resistance. It should be impossible to find two different inputs that create the same hash value. The weaknesses found on SHA-1 were theoretical until 2013 [2, 3]. The Centrum Wiskunde & Informatica Institute in Amsterdam and Google have successfully developed a practical technique for generating an SHA-1 collision named the SHAttered Attack in 2013. This attack required Nine quintillions (9,223,372,036,854,775,808) SHA1 computations in total 6,500 years of CPU computation to complete the attack first phase and 110 years of GPU computation to complete the second phase. This represents the culmination of two years of research that sprung from a collaboration between the two teams. Even though these numbers are very large, the SHA-1 shattered attack is still more than 130,000 faster than a brute force attack.

The Shattered attack is long term research between the CWI institute and Google's Research security, privacy and anti-abuse group [2]. Marc Stevens and Elie Bursztein started collaborating on making Marc's cryptanalytic attacks against SHA-1 practical using Google infrastructure. Ange Albertini developed the PDF attack, Pierre Karpman worked on the cryptanalysis and the GPU implementation, Yarik Markov took care of the distributed GPU code, Alex Petit Bianco implemented the collision detector to protect Google users and Clement Baisse oversaw the reliability of the computations.

The cryptanalysis team was able to create two different pdf files that share the same SHA-1 hash value. The theoretical attack becomes real. Therefore, systems that validate the authenticity of data would be deceived into accepting a malicious file in place of the genuine file. An example given by the researchers is of a malicious landlord crafting two colliding PDF files containing two identical rental agreements, except one has a vastly higher rent [2]. This attack could be used to obtain a valid signature for the contract with a high rent by having a victim sign the contract stating a lower rent. The SHA-1 collision attack requires significant computational resources, but it is still 130,000 times faster than a brute-force effort. SHAttered Attack is based on an identical-prefix collision attack. However, this attack technique doesn't allow an attacker to generate a collision with an existing file. For example, it's not possible to use this method to generate a malicious executable file which matches the signature of an existing legitimate executable. Thus, it would be possible for an attacker to generate two executable files which have the same SHA-1 hash but perform different actions when they run. Once Google releases the code behind the attack, anyone will be able to create pairs of PDF files that hash to the same SHA-1 sum, with two distinct images and certain preconditions.

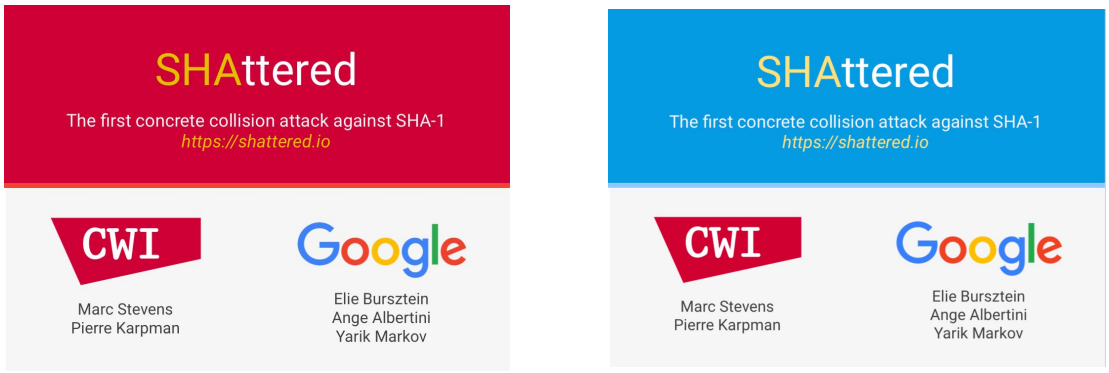

Fig. 5 Collision attack. These two files gives the same SHA-1 hashes [2]

This research certainly proves SHA-1 is broken. According to Google, computing the SHA-1 collision was one of the largest computations ever completed.

i. How does the shattered attack work?

The main idea behind this attack is based on differential cryptanalysis [2, s6]. Having the two inputs t0 and t'0, $\Delta 0$ will be the difference. After process the two inputs through the

function f (the compression function in the case of SHA-1) you get a difference Δ1. The goal here is to find a difference in the input Δ0 such that after some iterations you get Δ2 = 0, in other words, no difference. The Merkle–Damgård construction of SHA-1 property permit to alter the differences between the iterations in order to improve the differences to match his needs. In the case of the SHAttered Attack, they chose an initial prefix (P), then later on the next blocks they introduce a difference (M1(1), M1(2)) and remove it (M2(1) and M2(2)). At this point, they already have their collision. They just need to continue with the same following blocks , leading to a collision on the whole input.

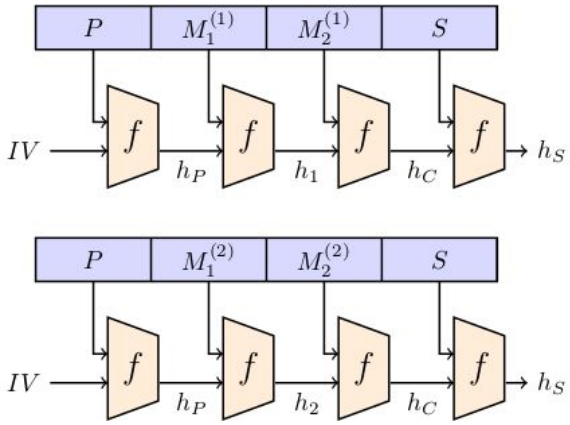

Fig. 6 The SHAttered Attack [s6]

ii. Practical test using shasum

The shasum script provides the easiest and most convenient way to compute SHA message digests. It is very simple to use. The user simply feeds data to the script via the command line. The output or SHA value is in hexadecimal notation. The data can be fed to shasum through files, standard input, or both. In our tests, we will feed the script with a file created by the team and another test using the pdf files released by the team who discovered the Shattered attack (Fig. 7).

```
hal9001@hal9001-MS-7971:~$ shasum shattered-1.pdf
38762cf7f55934b34d179ae6a4c80cadccbb7f0a  shattered-1.pdf
hal9001@hal9001-MS-7971:~$ shasum shattered-2.pdf
38762cf7f55934b34d179ae6a4c80cadccbb7f0a  shattered-2.pdf
hal9001@hal9001-MS-7971:~$
```

Fig. 7 Computing the hash of both PDFs via terminal

III. The NIST Competition

As established in the previous section, SHA-1 has been officially broken by use of the SHAttered attack. Despite this news, SHA-1 remained a very strong algorithm for almost three decades. However, back in 2005, confidence in SHA-1 began to be questioned, and this would eventually prompt NIST to start a competition in order to find the new SHA-3.

There was great fear amongst the cryptographic community that an attack on SHA-1 was imminent [4]. Three Chinese researchers named Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu discovered a way to reduce the amount of computations needed in order to find a SHA-1 collision. Using brute force, finding a collision in SHA-1 requires performing at least 2^80 computations, which was secure enough for practical purposes [4]. These three Chinese Researchers managed to reduce this number to 2^69, which is about 2,000 times faster than brute force [2, 4, 5]. While they did not actually perform the collision, there were concerns that it was only a matter of time before technology reached a level where performing 2^69 computations was feasible within a reasonable time period.

Before going further, the reason for NIST to have a competition in the first place needs to be established. What benefits do these events provide to the cryptographic community? This reason this is done is because it fulfills Kerckoff's Principle. To put it simply, Kerckoff's Principle says that a cryptographic system is only secure if everything about the system is known, except the key. NIST takes this principle to heart by organizing these competitions, since that allows opportunities for algorithms to get rigorously analyzed. A system that has analyzed by many sources inspires more confidence than a system that is kept secret, or has received no analysis. To find a new SHA-3, NIST takes the spirit of Kerckoff's Principle and used it to find a new, truly secure SHA-3.

November 2, 2007	Competition Announced
October 31, 2008	64 submissions enter the competition
December 10, 2008	51 candidates move to the first round
July 29, 2009	14 candidates move on to the second round
December 9, 2010	5 candidates proceed to the final round
October 2, 2012	A winner from the 5 candidates was selected

Fig 8. NIST Competition Timeline

A. The SHA-3 NIST Competition: First Round

Of the 64 submissions, 51 of these made it to the first round. In this part of the competition, NIST was looking for candidates that met the minimum requirements for being "complete and proper submissions" [6].

Candidates had to submit optimized C code implementations, as well at written specifications and intellectual property statements [6]. In addition to these requirements, candidates also needed to submit "known-answer tests" in order to prove that the algorithm in question actually worked [6]. Essentially, they are used to prove the correctness of an algorithm [7]. This is especially important for hash-functions because the output is intended to appear completely random and indecipherable. Candidates would use an output that they know to be correct, and then submit these known-answer tests to prove that the algorithm functions as needed. Candidates had to provide three different types of known-answer tests, which were for short messages, long messages, and extremely long messages [7]. The final requirement, which is one of the most important ones, was to require these algorithms to output message digest sizes of 224, 256, 384, and 512 bits [6]. If a candidate could provide even larger sizes, then that would be even better.

B. The SHA-3 NIST Competition: Second Round

Of the 51 candidates, only 14 were selected to proceed to the second round. The simple requirements of the first round no longer applied, as a more rigorous criteria was used to evaluate the candidates.

Two of the criteria, which are rather straightforward concepts, included "Cost and Performance", and "Algorithm and Implementation Characteristics" [6]. The new SHA-3 would be intended to be used in a wide variety of applications, including mobile devices, smart cards, and RFID [6]. This means that the candidates needed to be cost-worthy and perform well in any given situation, particularly in small devices. If the algorithm required a high degree of computational power, then it would not be useful or practical to implement anywhere, no matter how secure it actually was. The algorithm itself also needed to be flexible, and the designs needed to be simple and elegant [6]. NIST favored this simplicity because that meant the algorithm would be easy to analyze, and thus easier to prove whether it holds up to scrutiny.

The third criteria, which NIST listed as the most important, was simply called "Security" [6]. NIST knew that this term was rather vague as it employs a lot of subjective criteria, so they attempted to further define what it was they were looking for with regards to the competition [6]. Initially, they said that they were looking for an algorithm that comes closest to resembling an oracle in the Random Oracle Model. The model essentially says that any input that enters a box (with an Oracle inside) will produce an output that is truly random. This approach was highly criticized by the cryptographic community, so NIST held a separate workshop that detailed what was expected with regards to Security.

In this conference, the presentation covers how NIST will analyze an algorithm's susceptibility to general attacks [8]. This includes the computational complexity of these attacks,

as attacks that are too complex to be performed can be ignored. This also includes how these attacks affect the algorithm in general. Whether an attack merely wounds or completely breaks an algorithm are different points to consider, for example [8]. Is it possible to launch the attack on weaker variants of the given algorithm, and to what extent does it undermine NIST's confidence? All these points were considered when evaluating a candidate under the "Security" criteria. Lastly, resistance against multi-collision attacks, preimage, and 2nd preimage attacks were viewed favorably [8].

C. The SHA-3 NIST Competition: Third Round

Of the 14 candidates, only 5 of these would proceed to the third and final round. The candidates were evaluated based on the same three criteria as in the second round, however they were applied in an even more stringent way. For example, ECHO was not selected because of its poor performance, despite decent security [9]. It is also worth noting that none of these 14 candidates were broken, although partial attacks on weaker variants were possible [9]. William Burr, the Manager of the Cryptographic Technology Group, had this to say in an email sent to the NIST competition mailing list, sheds some light on how they selected the finalists:

> NIST wanted highly secure algorithms that also performed well. We preferred to be conservative about security, and in some cases did not select algorithms with exceptional performance, largely because something about them made us "nervous," even though we knew of no clear attack against the full algorithm [10].

Due to the strength of all these algorithms, a fourth unofficial criteria started becoming more relevant in order to narrow down the finalists. The amount of cryptanalysis a candidate received started to matter more, since the more analysis that was done on an algorithm, the more it inspired NIST's confidence. For example, one of the main reasons NIST cut Fugue and Luffa from the list is because it only received a small amount of cryptanalysis [9]. After much deliberations, five algorithms were selected as finalists.

D. The SHA-3 NIST Competition: Finalists and Winner

The five finalists of the SHA-3 NIST competition were: BLAKE, Keccak, JH, Skein, and Grostl. Each algorithm had its own clear benefits that set themselves apart from the rest. BLAKE was chosen because of its high security margin, good performance, and simple/clear design [11]. Keccak also had a high security margin and simple design, in addition to high throughput and throughput-to-area ratio [11]. JH was credited with an innovative design, solid security margin, and good performance [11]. Skein's main strength was its speed, and also its high security margin. Finally, Grostl was chosen because of the vast amount of cryptanalysis received, and its similarity to AES.

Keccak was selected as the winner, and is currently the new SHA-3. Keccak won in large part due to its sponge function and authenticated encryption, which will be detailed in the next section.

IV. A Gentle Introduction to Keccak and the SHA-3 Standard

This section is designed to outline a top-down view of the Keccak algorithm, noting specifically the version of Keccak implemented in the SHA-3 standard. It assumes some familiarity with cryptography and binary and modular arithmetic, however, it does not presume any background in programming. For a pseudo-code implementation see [12,13]. For a full implementation see supplemental links [s7,s8].

A. Modes of Operation

Keccak is a family of algorithms based on what is called the "sponge construction" or "sponge function." [12, 13] The "sponge" in sponge construction is so named because it is an algorithm which, in theory, can "absorb" an arbitrary amount of data to "squeeze out" and arbitrary amount of output. The speed at which data is absorbed is defined by *r,* the rate, which is absorbed into an array of width *b* (for *bus*width) and follows the formula where is the bit-length of a register in a CPU (*l*=6 for a 64-bit CPU). Finally, the security bits of the algorithm is defined by the capacity *c* so that

SHA-3 is a family of four hash functions and two extendable output functions (see Table 1 for details). [12] It fixes the Keccak algorithm to a buswidth of 1600, so that the capacity and the rate always add to 1600, and designates 4 output lengths for the hash functions: 224, 256, 384, 512 based on the security levels of 3DES and AES respectively [14]. XOFs are abstractions of hash functions that effectively allow a user to create a hash value of any length. Example hashes of the word "Hello" are printed in Table 2

	r	*c*	Output length (bits)	Security level (bits)
SHAKE128	1344	256	unlimited	128
SHAKE256	1088	512	unlimited	256
SHA3-224	1152	448	224	112
SHA3-256	1088	512	256	128
SHA3-384	832	768	384	192
SHA3-512	576	1024	512	256

Table 1. The SHA-3 Standard with parameters [13]

Algorithm	Output
SHAKE128	4131f8db5745776b48b86caa68d251fa9b19cf46b92b16289bb0c98e57e0e0de
SHAKE256	555796c90bfb8f3256a1cb0d7e574877fd48750e4147cf40aa43da122b4d64dafec00acf1ff59f4c3805f9589ec1ee3b712bc3701ce7d825cba56ca4942f6ffd
SHA3-224	4cf679344af02c2b89e4a902f939f4608bcac0fbf81511da13d7d9b9
SHA3-256	185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969
SHA3-384	df7e26e3d067579481501057c43aea61035c8ffdf12d9ae427ef4038ad7c13266a11c0a3896adef37ad1bc85a2b5bdac
SHA3-512	0b8a44ac991e2b263e8623cfbeefc1cffe8c1c0de57b3e2bf1673b4f35e660e89abd18afb7ac93cf215eba36dd1af67698d6c9ca3fdaaf734ffc4bd5a8e34627

Table 2. SHA-3 Hashes made with https://hashgen.de/

B. The Sponge Function

Keccak is based on the sponge construction which is pictured in Fig. 9. The function is divided into an absorbing and squeezing phase. In the absorbing phase, the state is initialized to *b*-bits of zeros. Then, this state is XORed with the first *r*-bits of the padded message and fed into the f-function, which serves a similar role as the f-function in AES and DES (see section C for more details). The process of taking the XOR of the first *r*-bits of the message with the first *r*-bits of the state is done iteratively until all the data has been absorbed. Note that the last *c*-bits of the state do not directly rely on the message, only the results of the f-function. This is a critical part of Keccak's security.

Once the data has been absorbed, it moves onto the squeezing phase. In the simplest case of a hash function, this phase involves simply truncating the first r-bits of the state to the desired value. If an output greater than *r* is requested (such as in the case of XOFs), the state is fed into the f-function again and the first *r*-bits are added to the output iteratively until the desired length is reached.

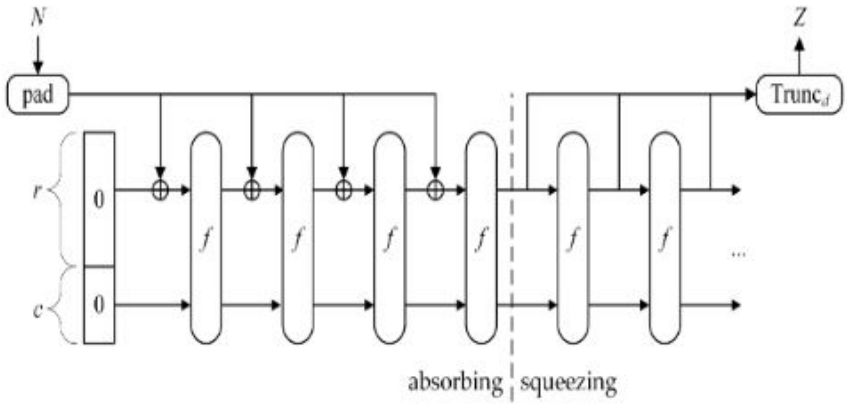

Fig. 9 The Sponge Construction [12]

To ensure data of length *M* can be absorbed into even chunks of *r*-bits, padding rules are applied. Table 3 shows the types of padding applied where *q* is the difference of *r* and *M*. For

any padding over 2 bytes, a number of zero bytes is added to the middle until the length of the padded message is:

$$\frac{r}{8} - \frac{M}{8} \bmod 8$$

Type of SHA-3 Function	Number of Padding Bytes	Padded Message
Hash	$q=1$	$M \| 0x86$
Hash	$q=2$	$M \| 0x0680$
Hash	$q>2$	$M \| 0x06 \| 0x00 \ldots \| 0x80$

Table 3. Example padding for a message *M* [12]

C. The f-function

Keccak-f provides the core of Keccak's cryptographic capability. It is designed to run for 24 rounds (based on the formula) and divided into 5 steps. All functions operate on the bits of the state organized into a 5x5x array (pictured in the Fig. 10).

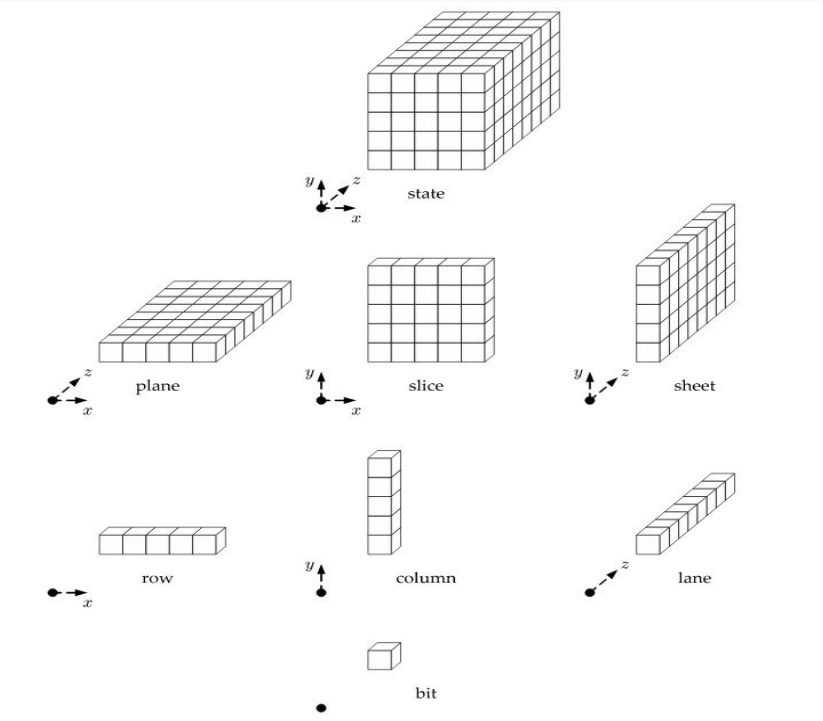

Fig. 10 Organization of the state-array and terminology [12]

The first, and perhaps most complicated is the, *theta* step. It involves taking the XOR sum of 11 bits in the pattern illustrated in Fig. 11. For a given bit *x* in the state array, *x* is assigned the result of x XOR the bits in column *x*-1 XOR the bits in column x+1, y+1 in modulus 5.

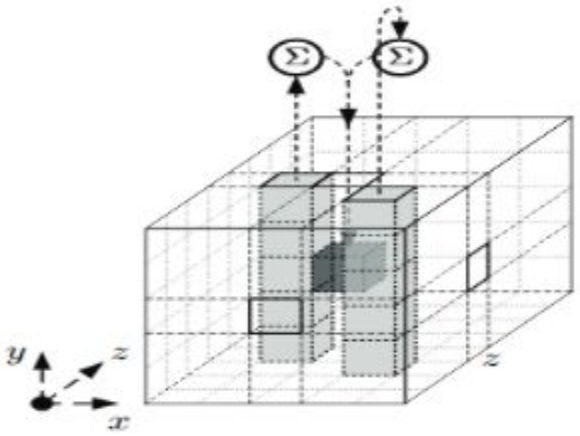

Fig. 11 Theta illustrated [14]

The second step is called *rho* which works together with the third step *pi*. For every lane in the state array (*x*,*y*), *rho* bit rotates the lane to the left (e.g. "1011" becomes "0111") by a fixed constant (*x*,*y*) specified in Table 4. Then, each lane (*x*,*y*) is swapped with the lane at position (*2x+3y mod 5*, *x*).

	x = 3	x = 4	x = 0	x = 1	x = 2
y=2	25	39	3	10	43
y=1	55	20	36	44	6
y=0	28	27	0	1	62
y=4	56	14	18	2	61
y=3	21	8	41	45	15

Table 4. Row offsets [1]

The fourth step is *chi* (pronounced "kai") which applies a logical function on each lane according to the following formula:

$$A[x,y] = B[x,y] \oplus ((\bar{B}[x+1,y]) \wedge B[x+2,y]) \quad , \quad x,y = 0,1,2,3,4$$

where A[x,y] is a given lane in the state from the *theta* function and B[x,y] is a lane in the state after the *pi* function, and B-bar denoted the inverse of the lane [14].

The last step is *iota* which simply XORs a constant with the lane at (0,0) according to the round and Table 5.

Table 1: The round constants RC[i]

RC[0]	0x0000000000000001	RC[12]	0x000000008000808B
RC[1]	0x0000000000008082	RC[13]	0x800000000000008B
RC[2]	0x800000000000808A	RC[14]	0x8000000000008089
RC[3]	0x8000000080008000	RC[15]	0x8000000000008003
RC[4]	0x000000000000808B	RC[16]	0x8000000000008002
RC[5]	0x0000000080000001	RC[17]	0x8000000000000080
RC[6]	0x8000000080008081	RC[18]	0x000000000000800A
RC[7]	0x8000000000008009	RC[19]	0x800000008000000A
RC[8]	0x000000000000008A	RC[20]	0x8000000080008081
RC[9]	0x0000000000000088	RC[21]	0x8000000000008080
RC[10]	0x0000000080008009	RC[22]	0x0000000080000001
RC[11]	0x000000008000000A	RC[23]	0x8000000080008008

Table 5. Round constants [13]

D. The Future of Keccak: Keyak

One reason Keccak ousted it's competition, as cited by NIST, was it's flexibility as an algorithm, specifically, as a building block for an authenticated encryption algorithm [11]. Authenticated encryption is a process which preserves integrity and availability in addition to confidentiality compared to traditional encryption. In addition to a plaintext and key, authenticated encyrption provides a means of validating a message with hashing via a "message authentication code" (MAC). Though there are various approaches to achieving this, the schema proven to be secure [15] is the Encrypt-then-Authenticate model (Fig. 12)

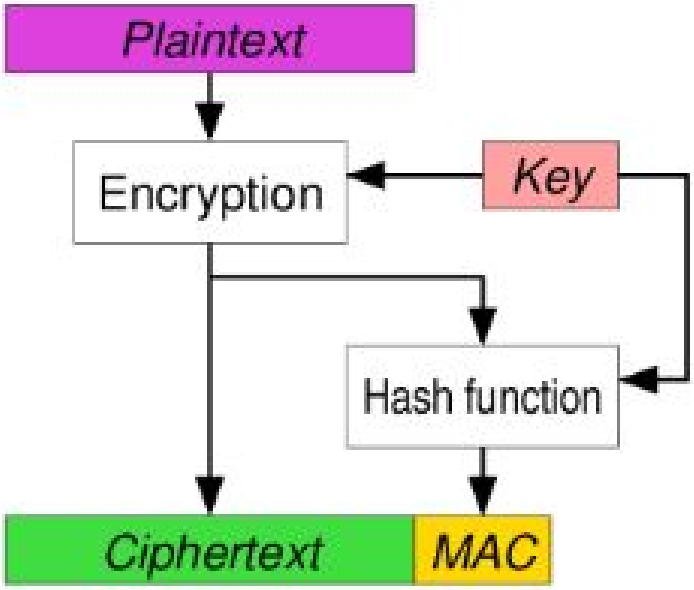

Fig. 12 Encrypt-then-Authenticate: Use key to encrypt, use key and cipher text to generate a MAC [15]

Traditionally, implementing a scheme such as the above requires two separate algorithms: one for encryption, one for hashing. However, due to the unique qualities of Keccak, specifically its ability to output an arbitrary length of bits, it can serve both functions. This involves manipulating the sponge function into what is called a Full-State Keyed Duplex Sponge construction (FKDS) [15].

A good starting point would be to ask, where does the key go in a function that only has one input? Recall from Section B that the sponge construction state starts with all bits set to zero. But this purely arbitrary. If we want to add a key to the mix, we just need to substitute the key in the initial state. Note that if the key is less than *c*-bits, it can be placed in the part of state which is the least related to the plain text input. Recall also that with Keccak we can request as much output as we need to generate a cipher text.

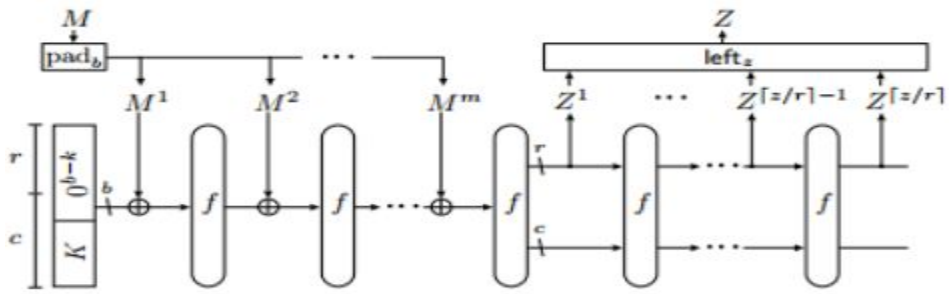

Fig. 13 Full-State Keyed Sponge Construction [15]

A Duplex Sponge Construction is an implementation of Keccak sponge that allows one to switch input and output blocks for each pass of the f-function. This is equivalent to how a block-cipher like AES or DES encrypt one block at a time.

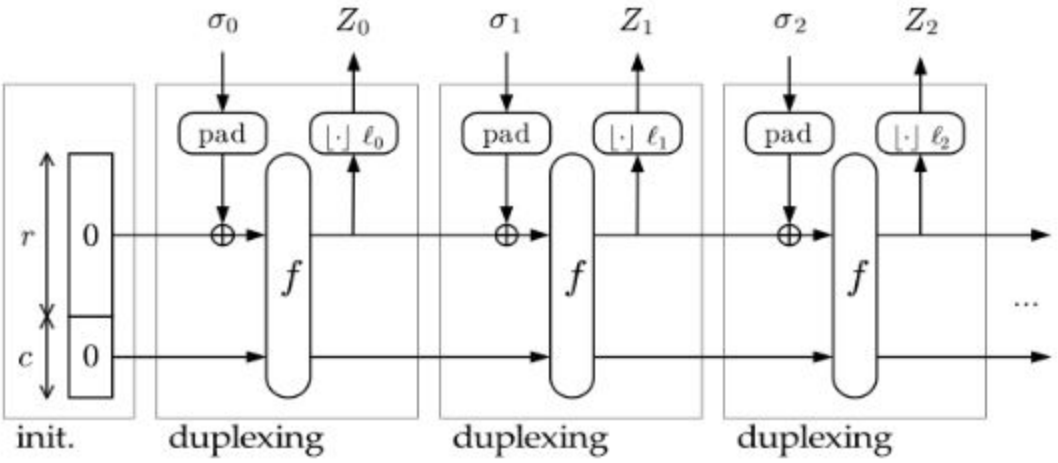

Fig. 14 Duplex sponge construction [5]

We finally arrive at a Full-State Keyed Duplex Sponge construction (FKDS) by combining the structures together. The result produces a block cipher with very similar security to the Keccak hash algorithm.

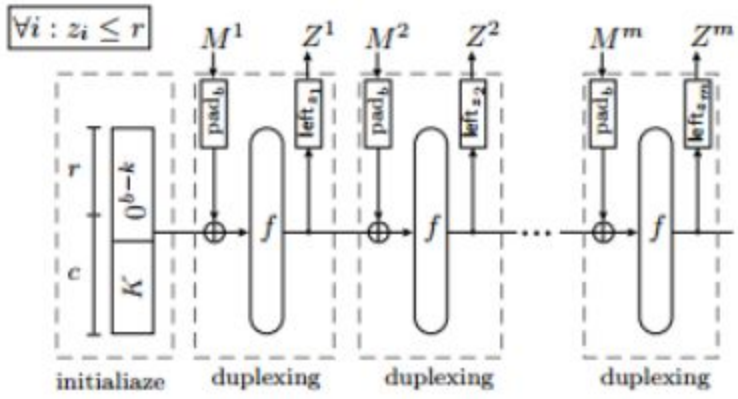

Fig. 15 Full-State Keyed Duplex Sponge construction (FKDS) [14]

Keyak managed to survive until the third-round of Caesar competition, but ultimately has not made it to the list of finalists announced earlier this year. Time will tell if Keyak will still find use in other areas of cryptography even if it is not to become a government standard. It is

clear, however, that the sponge construction has and will continue to leave its mark on the cryptography community as a smart, flexible, and robust method of building encryption schemes.

References

1. FIPS, P. (1995). 180-1. secure hash standard. National Institute of Standards and Technology, 17, 45.

2. Stevens, M., Bursztein, E., Karpman, P., Albertini, A., & Markov, Y. (2017, August). The first collision for full SHA-1. In *Annual International Cryptology Conference* (pp. 570-596). Springer, Cham. from https://shattered.io/

3. Wang, X., Yin, Y. L., & Yu, H. (2005, August). Finding collisions in the full SHA-1. In *Annual international cryptology conference* (pp. 17-36). Springer, Berlin, Heidelberg.

4. Schneier, B. (n.d.). Schneier on Security. Retrieved December 11, 2018, from https://www.schneier.com/blog/archives/2005/02/sha1_broken.html

5. Schneier, B. (n.d.). Schneier on Security. Retrieved December 11, 2018, from https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

6. Regenscheid, A., Perlner, R., Chang, S. J., Kelsey, J., Nandi, M., & Paul, S. (2009). Status report on the first round of the SHA-3 cryptographic hash algorithm competition. *Information Technology Laboratory National Institute of Standards and Technology Gaithersburg, MD*, 20899-8930.

7. Description of Known Answer Test (KAT) and Monte Carlo Test (MCT) for SHA-3 Candidate Algorithm Submissions. (February 20, 2008). https://csrc.nist.gov/CSRC/media/Projects/Hash-Functions/documents/SHA3-KATMCT1.pdf

8. Nandi, Mridul. (February 26, 2009). NIST's views on security requirements and Evaluation of Attacks.https://csrc.nist.gov/CSRC/media/Events/First-SHA-3-Candidate-Conference/documents/Nandi.pdf

9. Turan, M. S., Perlner, R., Bassham, L. E., Burr, W., Chang, D., jen Chang, S. & Peralta, R. (2011). Status report on the second round of the SHA-3 cryptographic hash algorithm competition. *NIST Interagency Report*, *7764*.

10. The Skein Hash Function Family. (n.d.). Retrieved December 11, 2018, from http://www.skein-hash.info/node/44

11. Chang, S. J., Perlner, R., Burr, W. E., Turan, M. S., Kelsey, J. M., Paul, S., & Bassham, L. E. (2012). Third-round report of the SHA-3 cryptographic hash algorithm competition. *NIST Interagency Report*, *7896*. https://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7896.pdf

12. Dworkin, M. J. (2015). *SHA-3 standard: Permutation-based hash and extendable-output functions* (No. Federal Inf. Process. Stds.(NIST FIPS)-202). https://doi.org/10.6028/NIST.FIPS.202

13. Bertoni, G. et. al, Team Keccak. Retrieved December 3, 2018, from https://keccak.team/keccak_specs_summary.html

14. Paar, C., Pelzl, J., & Preneel, B. (2014). *Understanding Cryptography A Textbook for Students and Practitioners*. SHA-3 and The Hash Function Keccak. Berlin: Springer Berlin.

15. Wetzels, J., & Bokslag, W. (2015). Sponges and Engines: An introduction to Keccak and Keyak. *arXiv preprint arXiv:1510.02856*.

Supplemental Links and Image Credits

s1. http://computer-trickster.blogspot.com/2015/11/encryption.html

s2. https://www.talwork.net/has-your-password-been-leaked

s3. https://en.wikipedia.org/wiki/Hash_function

s4. https://slideplayer.com/slide/9400105/

s5. https://en.wikipedia.org/wiki/Merkle%E2%80%93Damg%C3%A5rd_construction

s6. https://crypto.stackexchange.com/questions/44131/what-is-the-new-attack-on-sha-1-shattered-and-how-does-it-work

s7. https://github.com/XKCP/XKCP

s8. https://github.com/mgoffin/keccak-python/blob/master/Keccak.py