Cryptography and Computer Security

SHA-3

Why and how improving the SHA-2 standard even though there are no known attacks against it?

Niklas Schelten

Technische Universität Berlin - Electrical Engineering and Computer Science

26th January 2018

Abstract

Cryptographic Hash functions are widely needed in the cryptography for proof-of-work systems (such as digital signatures), verifying passwords and a lot more. For that it is, however, very important that the cryptographic hash functions are secure and fast to compute. In 2012 a new hash function was standardized by the NIST in a competition, the so called SHA-3. In this paper we explain why a new hash function is needed and how it works. We are doing that by having a look at the history of cryptographic hash functions and examining the competition for the SHA-3 function. Furthermore, we describe and analyze the new hash function and compare its security to other established hash functions. It reveals that there is no immediate need for a new hash function, nor was there when starting the competition. However, NIST decided after publications of exploits against the approach of the predecessor SHA-2 that it would be good to have a new and better hash function. Until now there was no real need for the new hash algorithm but the cryptographic publicity is calmed by the knowledge that in the case of an exploit that would deem SHA-2 insecure, there is a new hash function with a substantially different approach.

Contents

1 Introduction

Cryptographic hash function are a complicated but really important problem for today's mathematicians. There is a lot of research going on and, therefore, also a lot of ideas how to improve cryptographic hash algorithms. With this paper I want to clarify the need of a new standard of hash algorithms because it is far from apparent why a working standard should be improved spending more than five years of heavy research. Furthermore, I want to compare different kind of hash functions and point to their differences and what consequences are following from that. My third motivation for writing this paper is to clarify how exactly the SHA-3 standard can be implemented because there are a lot of different approaches on how to calculate the hash but all of them share that they are difficult to understand. Therefore, I try to make the complicated calculations as easy to follow as possible.
As mentioned, there is a whole lot of work going into cryptographic hash algorithm from which probably NIST is the organization with the most publications as they are standardizing the SHA-Family. However, there are also a lot of publication about possible attacks against different kind of hash algorithm which I will mention throughout my paper. Also for the general definitions of cryptographic hash algorithms there are a lot of good books and I chose to base my paper partly on the Second Edition of Stinson's *Cryptography: Theory and Practice.*
My paper is organized in roughly four sections: First we will introduce the general definition of cryptographic hash algorithms in Section 3. Then we will have a look at the history of hash algorithms and specifically the SHA-Family in Section 4. Furthermore, we will examine the detailed structure of SHA-3 in Section 5 and, finally, come to the security concerns of cryptographic hash algorithms in Section 6. In that section we will define the security concerns of hash algorithms and also compare the security of SHA-3 to other hash functions.

2 Glossary

2.1 Parameters and Variables

b	The length of the state vector in bits.
c	The capacity of a Sponge construction. $c = b - r$.
l	A predefined constant of the KECCAK algorithm. For SHA-3 it is defined as $l = 6$.
n	The hash length in bits.
m	The message length in bit.
q	The amount of padding-bytes.
q_b	The amount of padding-bits.
r	The rate/block length in bits of the sponge function.
w	length of a lane in the state vector. $w = \frac{b}{25}$.

2.2 Basic Operations

$0x00^n$	A string consisting of n consecutive bytes (8 bits) of 0s.
$X \oplus Y$	Binary exclusive-OR, also written as XOR. For example $0011 \oplus 0101 = 0110$.
$X\&Y$	Binary AND. For example $0011\&0101 = 0001$.
$\neg X$	Binary NOT. For example $\neg 01 = 10$.
$X\|\|Y$	Concatenating of two strings. For example $1010\|\|1100 = 10101100$.
$a[i,j,k]$	a is the state vector which is aligned as a three-dimensional block (Figure 1) with $0 \leq i < 5$, $0 \leq j < 5$ and $0 \leq k < w$. For a one-dimensional vector the index is $l = (5i + j) \cdot w + k$.
$a[i,j]$	concatenating $a[i,j,0]\|\|a[i,j,1]\|\|\ldots\|\|a[i,j,w-1]$. This is also called a *line*.
$x \lll n$	rotates a string x bitwise by n bits to the left. For example $101000 \lll 2 = 100010$.

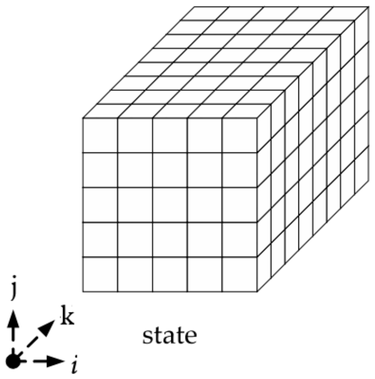


Figure 1: The layout of the state vector

3 Cryptographic Hash Algorithms

Douglas R. Stinson defines a hash family as follows: [2]
Definition. *A hash family is a four-tuple $(\mathcal{X}, Y, \mathcal{K}, \mathcal{H})$, where the following conditions are satisfied:*

1. *$\mathcal{X}$ is a set of possible messages*
2. *$\mathcal{Y}$ is a set of possible message digests or authentication tags*
3. *$\mathcal{K}$, the keyspace, is a finite set of possible keys*
4. *For each $K \in \mathcal{K}$, there is a hash function $h_K \in \mathcal{H}$. Each $h_K : \mathcal{X} \to \mathcal{Y}$.*

He also mentions *unkeyed hash functions* $h : \mathcal{X} \to \mathcal{Y}$ where $\mathcal{X}$ and $\mathcal{Y}$ are the same as in the above definition. The SHA-Family is a family of such unkeyed hash functions, which is why we will focus on those.

In most cases a hash function is a function that maps an arbitrary sized input into a fixed sized output. Stinson called the output *message digest* or *authentification tag*, however, we will continue to refer to the output plainly as the *hash*. A hash function becomes a *cryptographic* hash functions if it is infeasible to invert and infeasible to generate a message x_1 that generates the same hash as $h(x_0)$. We will go into much more detail in Section 6. Whenever we say hash function we referring to a cryptographic hash function unless otherwise specified.
Additionally, there are some "nice-to-have" features for hash functions: Firstly, that it is "pseudo-random", meaning that only a small change of the message results in a totally different hash and, secondly, that it is somewhat fast to calculate. Again, we will go into more detail about what *fast* means in Section 6.

Hash functions have a variety of use-cases:

- A **Proof-Of-Work** is for example used in Bitcoin-Mining and also a lot of other systems where it is important to proof who did a specific set of work. For example digital signatures also rely on hashes to proof that the message is from the owner you expect it to be and that the message integrity holds.
- **Password Verification** is a really important topic in the 21st century as everyone is using passwords to log-in to any kind of systems on a daily basis. Passwords need to be very secure and, on the other side, often need to be validated on the other end of the world. To make sure that this is working without an interfering attacker being able to "steal" the password, the passwords are salted and then hashed. The hash of the password is then sent to the receiving end making it impossible (really hard, see Section 6 again) for an attacker to get the plain text password.
- **File Identifier**s are another important use-case for hash functions. A lot of file systems, for example source code management systems like Git or peer-to-peer systems rely on checksums to safely identify a file. As a checksum is basically a hash function without some of its security factors they are not as much researched as hash functions. Therefore, most systems use hash functions for identifying files.

4 History of the Secure Hash Algorithms

The *Secure Hash Algorithms* are a family of cryptographic hash function that are standardized and published by the *National Institute of Standards and Security (*NIST*)*.

4.1 Until SHA-2

The first standard is today known as SHA-0 which was published as a component of *DSA (Digital Signature Algorithm)* for the *DSS (Digital Signature Standard)* in 1993. It was developed in cooperation with the *NSA (National Security Agency)* and has similarities in its design to the *MD4 (Message Digest Algorithm 4)* by Ronald L. Rivest. However, it was withdrawn shortly after its publications and replaced with the standard today know as SHA-1 which altered the algorithm just a bit but in the same time improved its security by a lot, see F. Chabaud and

A. Joux [5].
SHA-2 is a set of four cryptographic hash algorithm published in 2001 by the NIST as a successor for SHA-1. The four different variants feature different hash lengths of 224, 256, 384 and 512 bits. As of publications for attacks against SHA-1, NIST advised the transition to the successor SHA-2 in 2006, see the official NIST policies [6].

4.2 SHA-3 Competition

As of these breakthroughs in attacks against hash functions of the SHA-1 and also MD4 and MD5, NIST decided to held two public workshops to assess the current situation of their approved hash algorithms in 2004/2005. In November 2007 they announced that there will be a competition to standardize a new hash algorithm (family) that will be referred to as SHA-3. They announced this competition while knowing that at that moment no attacks against their current hash algorithm family SHA-2 existed, see [7].
64 teams applied for the first round of the competition with only 51 being accepted to participate in autumn of 2008. In late 2009 14 teams advanced to the second round of the competition where five finalists were announced after a full year of public review. Those five finalists were BLAKE, GRØSTL, JH, KECCAK and SKEIN.
Finally on the second October of 2012 NIST announced the final winner of the competition after eight month of public review of the finalists as KECCAK. We will compare the finalists algorithms to some extend in Section 6.3, see the NIST competition reports of the three rounds [8, 9, 10].

5 Design of SHA-3

I have implemented SHA-3 in C to make sure I get all the details correct. My implementation can be found in Listing 1.

The SHA-3 function is a sponge construction, which is based on a fixed-length permutation and will be explained in more detail in Section 6.2.1. That means that the message is hashed in two basic steps: First it needs to be padded so that the total length is a multiple of the *rate* r, which is the length of a block that will be `XOR`ed with the state each round. The rate is calculated by

$$r = b - 2 \cdot n \tag{1}$$

where b is the size of the state vector and n the length of the hash. The size of the state vector is derived from

$$b = 25 \cdot 2^l \tag{2}$$

$l = 6$ is just a parameter which still exists because there may be variants of KECCAK where l differs from 6.

amount of bytes	padded message
$q = 1$	$M\|\|0x86$
$q = 2$	$M\|\|0x0680$
$q > 2$	$M\|\|0x06\|\|0x00^{q-2}\|\|0x80$

Table 1: Padded message depending on the amount of padded bytes for byte-aligned messages.

5.1 Padding

The total amount of bits that the message needs to be padded with, such that the resulting size is a multiple of r is calculated from the message length m as follows:

$$q_b = r - (m \bmod r). \tag{3}$$

Assuming that the message is byte-aligned (meaning that $m \bmod 8 = 0$ holds), the padded message looks as shown in Table 1, where $q = {}^{q_b}/_8$ is the amount of bytes to be padded, see the NIST standard [1]. The assumption that the message is byte-aligned is valid most of the time because the use-cases imply that usually files are hashed and they are always byte-aligned.

5.2 Block-wise Permutation

Each block gets consecutively "absorbed" by the "sponge" and with that changes the internal state vector. When all blocks got absorbed the first n bits of the internal state are "squeezed out" as the resulting hash.

Each absorbing iteration starts with bitwise `XOR`ing the current state vector with the given message chunk. After that the round function is applied to the state vector. It applies five permutations in each of its 24 rounds.[1] The permutations are defined as follows where accessing the state vector is done by three indices. The index of a one-dimensional state vector is calculated by $l = (5i + j) \cdot w + k$. Additionally, access can also be done to a whole "lane", meaning a string of concatenated bits for varying k, see the Glossary 2.2 for more details.

1. θ-Permutation: A linear mixing operation which computes parity bits and `XOR`s them with the state vector:
    a) $p_j = a[0, j] \oplus a[1, j] \oplus a[2, j] \oplus a[3, j] \oplus a[4, j]$
    b) $a'[i, j] = a[i, j] \oplus p_{j-1} \oplus (p_{j+1} \lll 1)$
2. ρ-Permutation: A rotation of each lane by a different offset for each line. The offsets are listed in Table 2. For performance reasons they should be taken $\pmod w$. It becomes obvious that all offsets are triangular numbers.
3. π-Permutation: A simple permutation of bits of the following scheme:
    a) $a'[i, j, k] = a[(i + 3j) \bmod 5, i, k]$

[1]The amount of rounds can be reduced to compromise better performance for lower security. See Section 6.3.

4. χ-Permutation: A nonlinear operation:
    a) $a'[i,j] = a[i,j] \oplus (\neg a[(i+1) \bmod 5, j] \,\&\, a[(i+2) \bmod 5, j])$
5. ι-Permutation: `XOR`ing with a round constant $RC[r]$ that can be examined in Table 3.

5.3 Variants

There are six predefined variants of SHA-3 which differ in the hash size and with that also the rate. With changes to the hash size, also the security and the computation time changes, apparently (See Section 6). The hash lengths are 224, 256, 384 and 512.
Additionally, there are two so called SHAKE variants which have an arbitrary hash length. However, we will not go into more detail here besides that the padding scheme for SHAKE differs from the scheme presented in Section 5.1 to make sure that their results differ.

6 Security

It is important that a cryptographic hash function meets some security criteria, which we briefly covered in Section 3. Here we will explain what those criteria mean in detail and how SHA-3 manages to meet those and also compare SHA-3 with other important hash algorithms.

6.1 Security Concerns of Hash Algorithms

A cryptographic hash function needs to hold **all** three of the following resistances: [2]

- **Pre-Image Resistance:** For a given $h \in \mathcal{H}$ and $y \in \mathcal{Y}$, finding $x \in \mathcal{X}$ such that $h(x) = y$ is infeasible.
- **Second Pre-Image Resistance:** For a given $h \in \mathcal{H}$ and $x \in \mathcal{X}$, finding $x' \in \mathcal{X}, x' \neq x$ such that $h(x) = h(x')$ is infeasible.
- **Collision Resistance:** For a given $h \in \mathcal{H}$, finding $x, x' \in \mathcal{X}, x \neq x'$ such that $h(x) = h(x')$ is infeasible.

It is not defined what infeasible in that specific context means. However, one can assume it to mean that it is most certainly beyond current computers to be able to break any of the above resistances within a timespan where the systems needs its security. Because that does not really make it specific, we will give a more specific answer to that question here. It should be enjoyed with care as it won't hold for ever. The resistances above hold if it takes at least 2^{80} computations of $h \in \mathcal{H}$ to break it.
There is also another, theoretical definition which origins from the computational complexity theory. It states that a problem is infeasible to solve if there exists no algorithm which solves it in polynomial time. However, that definition is probably even more difficult because for small hash length even a exponential algorithm for breaking any of the stated resistances could

terminate in a reasonable time.[2] See the key management paper from NIST [11].

Additionally, it can be shown that collision resistance implies second image resistance but not preimage resistance. Furthermore, in reality a hash function is needed to behave as random as possible (often called *random oracle*) which means especially that it should not be able to retrieve any valuable information about the message when only having the hash. On the other hand it needs to be deterministic and efficiently to compute.

6.2 Comparison with Previous Cryptographic Hash Algorithms

Besides the hash algorithms we briefly introduced in Section 4 (SHA-1 and SHA-2) there are also MD4 and MD5 where for MD4 the first full collision attack has been published in 1995 and MD5's security has also been compromised by different exploits. As of 2009 it is widely considered broken states CERT [13]. MD5 has been one of the most used hash algorithms and, therefore, is still in use in a lot of software even though its discouragement.
As MD4 and SHA-1 are both succeeded by another standard we will compare the security of SHA-3 only to the successors MD5 and SHA-2.

6.2.1 Different Construction Approaches

Both algorithms from above, MD5 and SHA-2, are based on the *Merkle-Damgård construction* and it is, therefore, assumed that both algorithm face similar weaknesses. On the other side there is the sponge construction used by SHA-3. Here we will compare those two and with that also compare the hash functions.

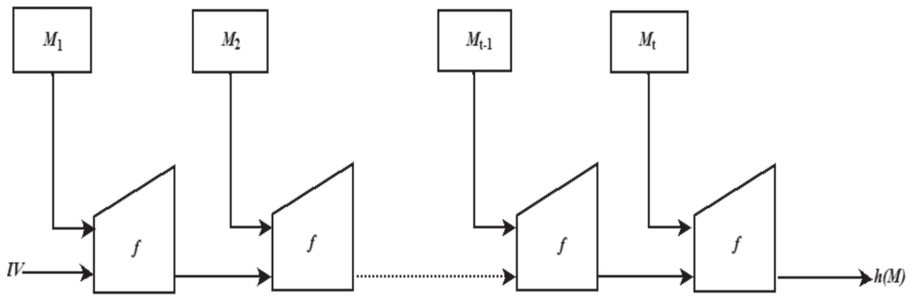


Figure 2: A Merkle-Damgård construction

Merkle-Damgård Construction This construction was discovered in 1989 by Merkle and Damgård independently and relies on a *cryptographic compression function*

$$c : \{0,1\}^m \times \{0,1\}^n \to \{0,1\}^n \tag{4}$$

[2]Similarities to RSA can be found here: RSA relies on integer factorization being infeasible. At the moment no algorithm with polynomial run-time is known to solve an integer factorization. However, RSA is only deemed secure with a key length of at least 1024 bits because with smaller keys it could be possible to solve the integer factorization in reasonable time, see [12].

It takes a m bit message and a n bit chaining value and is to be distinguished from a *data compression function* because the purpose of a cryptographic compression function is to be difficult to be inverted. That is, why it is often also referred to as a *one-way compression function.* A data compression function's use, on the other hand, is to be able to invert (lossless) or approximately invert (lossy) the function which would be useless for hashing.
A Merkle-Damgård hash function, h, is created by first padding the message to a multiple of m bits and then chaining compression functions as seen in Figure 2. Formally spoken it is defined as follows:

$$H_0 = \text{IV} \tag{5}$$

$$H_i = c(M_i, H_{i-1}) \qquad i \in \{1, 2, \ldots, t\} \tag{6}$$

$$h(M) = h_t \tag{7}$$

Here `IV` is an initiation value with a n-bit length and the whole message M (including the padding) is broken up into t m-bit sized message blocks M_i.
The padding is done by adding a 1 at the end of the message and then filling up with the necessary amount of 0's while the binary encoding of the message length is added at the end of the message. This kind of padding is named after the construction itself as it was unique at that time, see [14].
Merkle and Damgård claim that the security, both, collision resistance and preimage resistance, of the compression function is being promoted to the hash function itself. Though, it is sufficient to use a secure compression function in order to have a secure hash function. However, with recent publications it became apparent that the Merkle-Damgård approach is limited by its vulnerability to those three attacks:

- **Long Second Pre-Image:** It is shown by J. Kelsey and B. Schneider [15] that Second Pre-Image resistance is greatly reduced in Merkle-Damgård constructions. Their findings are based on a pattern expanding a message while still resulting in the same hash.
- **Multicollision Attack:** With a similar approach, they also show that multiple collisions can be found and, thus, break the collision resistance.
- **Herding Attack:** The so called herding attack is able to first provide the hash and *afterwards* "herd" any appropriate prefix to the message and the hash will still be correct. This is, though, only possible when being able to compute multiple collisions on the hash function (multicollision attack), see [16].

Sponge Construction On first sight a sponge construction may look similar to a Merkle-Damgård even though it is substantially different from it and, thus, not vulnerable to the above mentioned attacks. The key is that it builds upon a fixed length permutation

$$p : \{0,1\}^n \to \{0,1\}^n \tag{8}$$

The process of computing a hash with a sponge construction is a two step process. The first is called the "absorbing" where it iteratively absorbs the padded message as blocks. The second phase is called the "squeezing" where it outputs the hash of the message.
The sponge construction operates on a state vector which is modified each iteration and has a size of $r + c$ bits. In each iteration the first r bits are `XOR`ed with the message and after

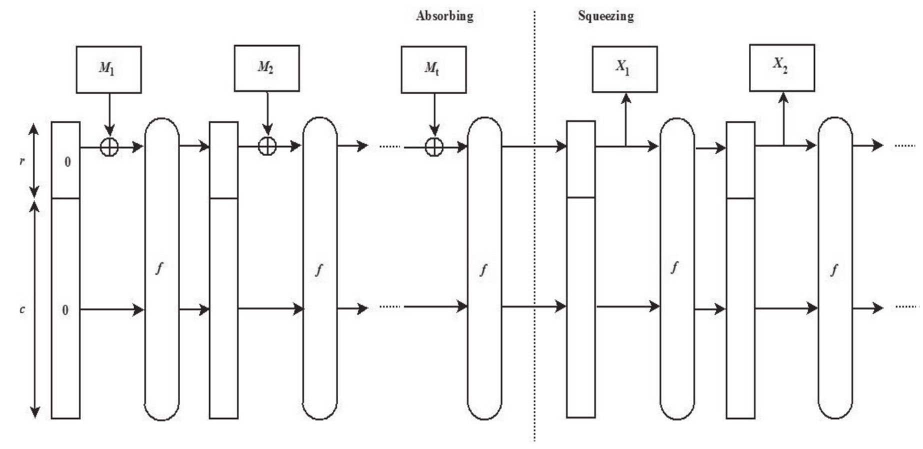


Figure 3: A Sponge construction

that the permutation p is applied to the state vector. This behavior is visualized in Figure 3. Additionally, between two iteration also blank rounds can be applied where no message block is XORed with the state vector but only p is applied. That is, however, not done in SHA-3. The complexity of a collision attack is $\min(2^{c/2}, 2^{n/2})$ while the complexity for a preimage and second preimage attack is $\min(2^{c/2}, 2^{n})$, see [14]. That holds until today because there are no attacks known that exposes any vulnerability to the sponge construction.

6.3 Comparison with Competitors

The competition for SHA-3 took about five years where the publicity inspected the submitted hash functions and Nist announced the winner Keccak in October 2012. But how did they decided which hash function is the best? They splitted the competition in three rounds, each eliminating possible competitors. First we will summarize the competitors of the third and final round and afterwards explain why Nist chose Keccak as a winner:

- Blake is based on a stream cipher (a ChaCha variant of Salsa20). However, before each round a permuted message block (XORed with round constants) is added.
- GRØSTL is very similar to, for example, MD5 and the previous SHAs because it also uses the Merkle-Damgård construction. One main difference is that it uses a state twice the hash size and truncates it in the end.
- JH has a constant state and input size of 1024- and 512-bits and also works with S-Boxes. It can, therefore, also be compared to previous SHA and MD5 hash algorithms.
- Keccak (the winner) has been explained in Section 5. It is based on a sponge construction.
- Skein is, similar as Blake, based on a cipher, the *Threefish*. Similar to the Salsa20 cipher it does not use any S-Boxes but alternating additions and XORs.

As seen in Section 6.1, there are a different factors that need to be taken into account. Therefore, we will give a brief overview to what criteria were taken into account when evaluating the hash functions. We are listing them in an ordered fashion, meaning that the first criteria was the most important one for NIST and so on, see [10].

Security The first and most important criteria was, of course, the security of a hash function. It takes onto account the proven resistance to the various attacks described in Section 6.1, cryptoanalysis of the hash function and also its parts (e.g. the Sponge construction, the permutation), tweaks the finalists made after the second round to their hash function and some other, minor, security concerns.

None of the five finalists had any real exploits that would open a vulnerability against them. All of them were deemed very secure until further than 2030. In Table 5 the proven security of the finalists and SHA-2, which is added as a reference, can be seen.
NIST also introduces the *Security Margin*, which can be calculated for round functions, which all of the finalists are. They define it as "the fraction of the hash or compression function that has not been successfully attacked. (For example, an attack on six rounds of a ten-round hash function would give a 40 % security margin.)" [10]. The security margins of the finalists can be viewed in Table 4. Here a hint is given that KECCAK may be the most secure hash function out of those 5. However, the security margin is only one indication and, additionally, all of the other algorithms were also considered really secure. Therefore, other criteria should also influence the decision to which hash function should succeed SHA-2.

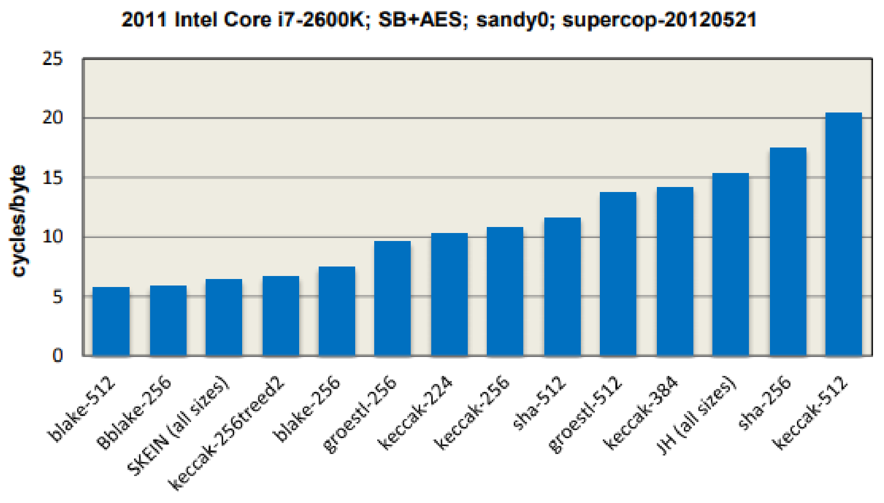


Figure 4: Performance of software solution on current CPU with Current Vector Unit [10]

Cost and Performance The cost and performance is the easiest defined criteria as it is straight forward. Here the computational complexity of the hash function and its memory usage is taken into account. That is due to the importance for a hash function to be fast computable on devices with low computational power and memory, e.g. Smart Cards.[3]

[3]It may be good to know that in some rare cases (e.g. hashing salted passwords) it is not needed for a hash function to be fast. When checking passwords, the user does not gain a lot by being able to log in in a

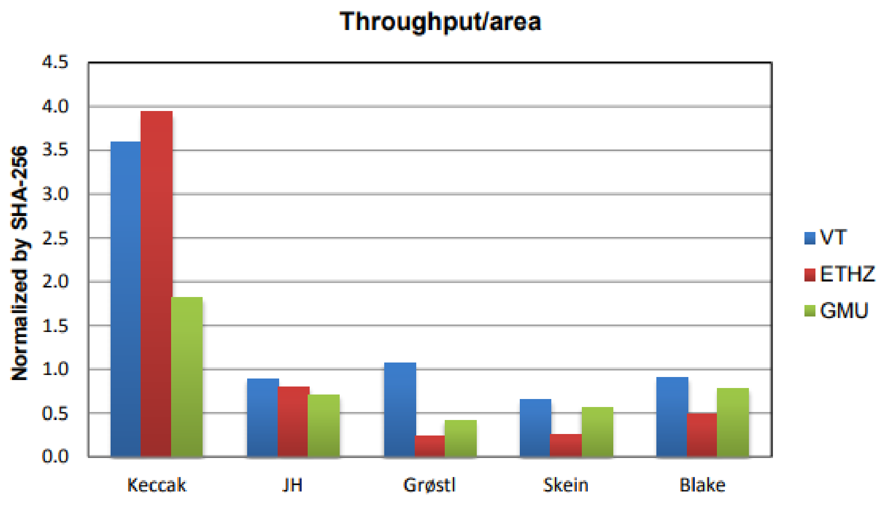


Figure 5: Normalized throughput for three ASIC implementations of 256-bit variants.[4] [10]

It is important to notice that the cost and performance may vary a lot for software and hardware implementations, so both should be taken into account.

The performance of an algorithm may depend on the hash and message length. As all algorithms (including the SHA-2 for reference) implement a 224-, 256-, 384- and 512-bit variant they can be compared with each other. However, different approaches result in different performances for a different hash size with a constant message size: KECCAK runs the same permutation for all hash lengths, however, the higher the hash length, the smaller each message block is. Therefore, it takes more iterations for the same message length. SKEIN and JH, both use the same compression function for all hash lengths and run in about the same time. GRØSTL, BLAKE and SHA-2 use one compression functions for 224- and 256-bit, and another for 384- and 512-bit hash length. This usually results in two different run times.
As previously mentioned the software performance can be substantially different from the hardware performance, which is why NIST looked at both. In Figure 4 the performance of the different hashing algorithms can be seen on a current general-purpose processor and in Figure 5 one can see the performances of three different ASICs (Application-specific integrated circuit) which are hardware implementations of the 256-bit variants.
The software performance of KECCAK is comparable bad, for example its 512-bit implementation is about four-times slower than BLAKE-512. However, when looking at the hardware performances KECCAK is the only algorithm that is faster than SHA-2 at all. In some implementations it is even faster by a factor of 4.

Algorithm and Implementation Characteristics The hash functions are also judged on their *flexibility* and *simplicity* as a hash function that can easily be adjusted to all kind of platforms is preferred.

fraction of seconds. On the other side, however, a slower hash function limits a possible attacker to less attacks per second.

Some of the finalists suggested ways to use their hash algorithm in different approaches that current hash functions do not cover. Skein was probably the one suggesting the most other uses as for example their tweakable blockcipher, Threefish, and different applications like tree-mode hashing etc.. However, Keccak provided with the sponge construction an enormous flexibility as it can easily be modified to output different sized hashes or trade off security for performance in a controlled manner (See the security margin in Table 4).

All together, Nist decided for Keccak as the final winner, however, mentioning that the other finalists would, also, have been a good choice as a successor for SHA-2. Contradictory to the expectation in the beginning of the competition, SHA-2 is a non-broken hash function until now. However, Nist still partly based their choice on the fact that Keccak takes a completely different approach than SHA-2 and also has its strength in rather different fields (performance on ASICs, flexibility, etc.).

Critical voices can never be avoided and so it comes that they also face Nist with SHA-3. After the announcement of Keccak as the winner Nist put a lot of effort into creating a standard based on the Keccak paper. For that there were tweaks necessary and also some minor changes to the capacity and padding scheme which we will not describe in more detail here. However, as in 2013 the so called *Snowden leaks*[5] inflicted a very skeptical view on all US-national institutes, especially the Nsa, concerns were rising that Nist made tweaks to Keccak in order to provide back-doors. The fear of back-doors is not unknown to hash algorithms as it also exists for back-doors in SHA-2 but in general it is assumed that there is no back-door. That is due to the fact that no one was ever able to find one until now and there are no suspicious, e.g., constants in the algorithm.
As a result there are also publications trying to convince the public that SHA-3 is influenced by the Nsa and, therefore, insecure. However, most of the publications I have found, provide misleading arguments or even plainly wrong statements. Therefore, I decided not to dedicate a whole section to this topic. If, however, you are further interested, I am including two references: The first is a post by *Joseph Lorenzo Hall* which contains most arguments for SHA-3 being supposedly insecure. Additionally, he included an answer of the official Keccak team (not Nist but the independent developer team that applied for the competition in the first place) which obliterates each argument in a detailed way: [18]. The second post is from the official Keccak website to summarize that SHA-3 still is the same as Keccak: [19].

7 Conclusion

There is a lot of research going into the development of cryptographic hash functions as they become an increasingly important part in the todays world. The first hash algorithm dates back to the 1970s, however, the first standard the Nist published was in the mid 1990s. From

[4]VT: Virginia Tech; ETHZ: Eidgenössische Technische Hochschule Zürich; GMU: George Mason University The implementations had different optimizing approaches and, therefore, vary from each other.

[5]Edward Snowden leaked about 9,000 to 10,000 private documents of the Nsa providing in depth detail on the methods Nsa uses to spy on everyday people with the help of back-doors and other unknown tricks, see the (German) news report by "Spiegel" [17].

that point on a lot of exploits were published against different kind of approaches and concrete algorithms. One of the most used approach to design a cryptographic hash function is the Merkle-Damgård approach, however, due to its massive usage, also a lot of researching energy went into developing exploits. It is based on a compression function that is iteratively used to alter the state which then results in the hash. Originally the authors proved that the resistances of the compression function will be inherited by the whole hash function. However, it became apparent that some exploits could surpass this limitation making the approach vulnerable. Combined with exploits against the compression function of SHA-1 the hash algorithm was broken. Nevertheless, SHA-2 is still considered secure because its hash function is yet to be broken.
NIST decided for a standardization of a new hash algorithm anyways and chose KECCAK. That decision was to some extent influenced by the reason that it has its strength in other fields than its predecessor: The sponge approach used by KECCAKs SHA-3 was never used in a commercial cryptographic hash algorithm before but is still deemed to be very secure. It is, in contrast to the Merkle Damgård approach, based on a fixed-length permutation which only permutes the state each time a message block gets iteratively `XOR`ed with it. Secondly, the hardware implementations of SHA-3 are up to four times faster than the SHA-2 hardware implementations which becomes more important in today's world as ASICs become more popular. Additionally it is a very flexible approach, meaning that a trade-off between performance and security can be done in a very secure manner by reducing its numbers of rounds (See security margin in Table 4).

With these aspects in mind we can conclude that there was no immediate need for a new hash algorithm standardization, however, it took the cryptographic community about five years to come up with a hash function that mostly everyone considers to be safe. Additionally, the fact that the today's world desperately needs a cryptographic hash function makes it apparent that it is good to have SHA-3 even though SHA-2 is still deemed to be very secure.

8 Appendix

	$i = 0$	$i = 1$	$i = 2$	$i = 3$	$i = 4$
$j = 0$	0	1	190	28	91
$j = 1$	36	300	6	55	276
$j = 2$	3	10	171	153	231
$j = 3$	105	45	15	21	136
$j = 4$	210	66	253	120	78

Table 2: Offsets for the ρ-Permutation.[3]

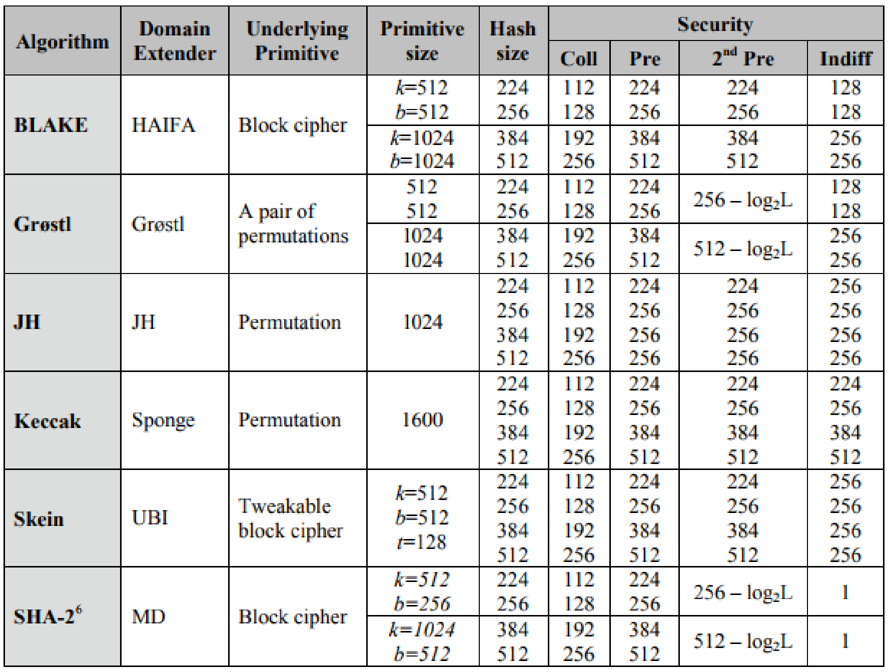
RC[0]	0x0000000000000001	RC[12]	0x000000008000808B
RC[1]	0x0000000000008082	RC[13]	0x800000000000008B
RC[2]	0x800000000000808A	RC[14]	0x8000000000008089
RC[3]	0x8000000080008000	RC[15]	0x8000000000008003
RC[4]	0x000000000000808B	RC[16]	0x8000000000008002
RC[5]	0x0000000080000001	RC[17]	0x8000000000000080
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 3: The round constants for the ι-Permutation.[3]

Algorithm	Security Margin	Depth of Analysis
BLAKE	71%	High
GRØSTL	40%	Very High[6]
JH	38%	Low
KECCAK	79%	Medium
SKEIN	56%	High[6]
SHA-2	62%	Medium

Table 4: Security Margin of the finalists [10]

[6]These hash function have been substantially tweaked for the final round



Algorithm	Domain Extender	Underlying Primitive	Primitive size	Hash size	Security			
					Coll	Pre	2nd Pre	Indiff
BLAKE	HAIFA	Block cipher	*k*=512 *b*=512	224	112	224	224	128
				256	128	256	256	128
			k=1024 *b*=1024	384	192	384	384	256
				512	256	512	512	256
Grøstl	Grøstl	A pair of permutations	512	224	112	224	$256 - \log_2 L$	128
			512	256	128	256		128
			1024	384	192	384	$512 - \log_2 L$	256
			1024	512	256	512		256
JH	JH	Permutation	1024	224	112	224	224	256
				256	128	256	256	256
				384	192	256	256	256
				512	256	256	256	256
Keccak	Sponge	Permutation	1600	224	112	224	224	224
				256	128	256	256	256
				384	192	384	384	384
				512	256	512	512	512
Skein	UBI	Tweakable block cipher	*k*=512 *b*=512 *t*=128	224	112	224	224	256
				256	128	256	256	256
				384	192	384	384	256
				512	256	512	512	256
SHA-2[6]	MD	Block cipher	*k=512 b=256*	224	112	224	$256 - \log_2 L$	1
				256	128	256		
			k=1024 b=512	384	192	384	$512 - \log_2 L$	1
				512	256	512		

Table 5: Proven Security of the SHA-3 finalists [10]

```c
#include<stdlib.h>
#include<string.h>
#include<stdint.h>
#include<stdio.h>
#include"util.h"
#include"sha3.h"

#define NUMBER_OF_ROUNDS 24

// rotates qword by n to the left
#define ROTL64(qword, n) \
  ((qword) << (n) ^ ((qword) >> (64 - (n))))

uint64_t keccak_round_constants[NUMBER_OF_ROUNDS] = {
  0x0000000000000001ULL, 0x0000000000008082ULL,
  0x800000000000808AULL, 0x8000000080008000ULL,
  0x000000000000808BULL, 0x0000000080000001ULL,
  0x8000000080008081ULL, 0x8000000000008009ULL,
  0x000000000000008AULL, 0x0000000000000088ULL,
  0x0000000080008009ULL, 0x000000008000000AULL,
  0x000000008000808BULL, 0x800000000000008BULL,
  0x8000000000008089ULL, 0x8000000000008003ULL,
  0x8000000000008002ULL, 0x8000000000000080ULL,
  0x000000000000800AULL, 0x800000008000000AULL,
  0x8000000080008081ULL, 0x8000000000008080ULL,
  0x0000000080000001ULL, 0x8000000080008008ULL
};

void keccak_theta(uint64_t *a) {
  unsigned int j;
  uint64_t p[5], q[5];
  for (j = 0; j < 5; j++)
    p[j] = a[j] ^ a[j+5] ^ a[j+10] ^ a[j+15] ^ a[j+20];

  q[0] = ROTL64(p[1], 1) ^ p[4];
  q[1] = ROTL64(p[2], 1) ^ p[0];
  q[2] = ROTL64(p[3], 1) ^ p[1];
  q[3] = ROTL64(p[4], 1) ^ p[2];
  q[4] = ROTL64(p[0], 1) ^ p[3];

  for (j = 0; j < 5; j++) {
    a[j]      ^= q[j];
    a[j +  5] ^= q[j];
    a[j + 10] ^= q[j];
    a[j + 15] ^= q[j];
    a[j + 20] ^= q[j];
  }
}

void keccak_rho(uint64_t *state) {
  state[ 1] = ROTL64(state[ 1],  1);
  state[ 2] = ROTL64(state[ 2], 62);
  state[ 3] = ROTL64(state[ 3], 28);
  state[ 4] = ROTL64(state[ 4], 27);
  state[ 5] = ROTL64(state[ 5], 36);
  state[ 6] = ROTL64(state[ 6], 44);
  state[ 7] = ROTL64(state[ 7],  6);
  state[ 8] = ROTL64(state[ 8], 55);
  state[ 9] = ROTL64(state[ 9], 20);
  state[10] = ROTL64(state[10],  3);
  state[11] = ROTL64(state[11], 10);
  state[12] = ROTL64(state[12], 43);
  state[13] = ROTL64(state[13], 25);
  state[14] = ROTL64(state[14], 39);
  state[15] = ROTL64(state[15], 41);
  state[16] = ROTL64(state[16], 45);
  state[17] = ROTL64(state[17], 15);
  state[18] = ROTL64(state[18], 21);
  state[19] = ROTL64(state[19],  8);
  state[20] = ROTL64(state[20], 18);
  state[21] = ROTL64(state[21],  2);
  state[22] = ROTL64(state[22], 61);
  state[23] = ROTL64(state[23], 56);
  state[24] = ROTL64(state[24], 14);
}

void keccak_pi(uint64_t *a) {
  uint64_t a1;
  a1 = a[1];
  a[ 1] = a[ 6];
  a[ 6] = a[ 9];
  a[ 9] = a[22];
  a[22] = a[14];
  a[14] = a[20];
  a[20] = a[ 2];
  a[ 2] = a[12];
  a[12] = a[13];
  a[13] = a[19];
  a[19] = a[23];
  a[23] = a[15];
  a[15] = a[ 4];
  a[ 4] = a[24];
  a[24] = a[21];
  a[21] = a[ 8];
  a[ 8] = a[16];
  a[16] = a[ 5];
  a[ 5] = a[ 3];
  a[ 3] = a[18];
  a[18] = a[17];
  a[17] = a[11];
  a[11] = a[ 7];
  a[ 7] = a[10];
  a[10] = a1;
  // a[0] is left as is
}

void keccak_chi(uint64_t *a) {
  unsigned int i;
  for (i = 0; i < 25; i += 5) {
    uint64_t a0 = a[0 + i];
    uint64_t a1 = a[1 + i];
    a[0 + i] ^= ~a1       & a[2 + i];
    a[1 + i] ^= ~a[2 + i] & a[3 + i];
    a[2 + i] ^= ~a[3 + i] & a[4 + i];
    a[3 + i] ^= ~a[4 + i] & a0;
    a[4 + i] ^= ~a0       & a1;
  }
}

void permutation(uint64_t *state) {
  unsigned int round;
  for (round = 0; round < NUMBER_OF_ROUNDS; round++) {
    keccak_theta(state);
    keccak_rho(state);
    keccak_pi(state);
    keccak_chi(state);
    // iota
    *state ^= keccak_round_constants[round];
  }
}

void process_block(uint64_t *hash, const uint64_t *block, unsigned rate) {
  // one hash array item represents one lane (64bit)

  // XOR the block with the state
  unsigned int i;
  for (i = 0; i < rate/64; i++) {
    hash[i] ^= block[i];
  }
  permutation(hash);
}

void sha3(int n, char* message, size_t length, char* hash, int debug) {
  unsigned rate = 1600-2*n;
  unsigned block_size = rate / 8; // in byte

  if (debug) {
    printf("message:\n");
    print_hex_memory(message, length);
  }

  // add the padding
  unsigned q = block_size-(length % block_size);
  char *msg;
  if (q == 1) {
    msg = malloc(length+1);
    memcpy(msg, message, length);
    msg[length++] = 0x86;
  } else if (q == 2) {
    msg = malloc(length+2);
    memcpy(msg, message, length);
    msg[length++] = 0x06;
    msg[length++] = 0x80;
  } else {
    msg = malloc(length+q);
    memcpy(msg, message, length);
    msg[length] = 0x06;
    memset(&msg[length+1], 0, q-2);
    length += q;
    msg[length-1] = 0x80;
  }

  if (debug) {
    printf("padded message:\n");
    print_hex_memory(msg, length);
  }

  // loop through all blocks
  unsigned int i;
  uint64_t h[25];
  memset(h, 0, sizeof(h[0])*25);
  for (i = 0; i < length; i += block_size) {
    uint64_t *block = malloc(block_size);
    memcpy(block, &msg[i], block_size);
    process_block(h, block, rate);
    free(block);
  }
  free(msg);
  memcpy(hash, h, n/8);
}
```

Listing 1: My implementation of SHA-3

References

[1] Morris J. Dworkin, *NIST*: **SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions** (published Aug 2015),
`https://www.nist.gov/publications/sha-3-standard-permutation-based-hash-and-extendable-output-functions` (last accessed 2018-01-09)

[2] Douglas R. Stinson: **Cryptograhpy: Theory and Practice 2nd Edition**, *CRC Press*, 2002

[3] TeamKeccak: **Keccak specifications summary**,
`https://keccak.team/keccak_specs_summary.html` (last accessed 2018-01-10)

[4] National Institute of Standards and Technology: **Secure Hash Standard (SHS)** (published Aug 2015)
`https://csrc.nist.gov/publications/detail/fips/180/4/final` (last accessed 2018-01-12)

[5] Florent Chabaud and Antoine Joux: **Differential Collisions in SHA-0** (published at CRYPTO '98)
`http://fchabaud.free.fr/English/Publications/sha.pdf` (last accessed 2018-01-13)

[6] National Institute of Standards and Technology: **NIST Policy on Hash Functions** (published Mar 2006, Sep 2012 and Aug 2015)
`https://csrc.nist.gov/Projects/Hash-Functions/NIST-Policy-on-Hash-Functions` (last accessed 2018-01-13)

[7] National Institute of Standards and Technology: **SHA-3 Project**
`https://csrc.nist.gov/projects/hash-functions/sha-3-project` (last accessed 2018-01-13)

[8] Andrew Regenscheid (NIST), Ray Perlner (NIST), Shu-jen Chang (NIST), John Kelsey (NIST), Mridul Nandi (NIST), Souradyuti Paul (NIST): **Status Report on the First Round of the SHA-3 Cryptographic Hash Algorithm Competition** (published Sep 2009)
`https://csrc.nist.gov/publications/detail/nistir/7620/final` (last accessed 2018-01-13)

[9] Meltem Sönmez Turan (NIST), Ray Perlner (NIST), Lawrence Bassham (NIST), William Burr (NIST), Donghoon Chang (NIST), Shu-jen Chang (NIST), Morris Dworkin (NIST), John Kelsey (NIST), Souradyuti Paul (NIST), Rene Peralta (NIST): **Status Report on the Second Round of the SHA-3 Cryptographic Hash Algorithm Competition** (published Feb 2011)
`https://csrc.nist.gov/publications/detail/nistir/7764/final`
(last accessed 2018-01-13)

[10] Shu-jen Chang (NIST), Ray Perlner (NIST), William Burr (NIST), Meltem Sönmez Turan (NIST), John Kelsey (NIST), Souradyuti Paul (NIST), Lawrence Bassham (NIST): **Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition** (published Nov 2012)

`https://csrc.nist.gov/publications/detail/nistir/7896/final`
(last accessed 2018-01-13)

[11] Elaine Barker (NIST): **Recommendation for Key Management, Part 1: General** (published Jan 2016)
`https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-4/final`
(last accessed 2018-01-14)

[12] Thorsten Kleinjung and Kazumaro Aoki and Jens Franke and Arjen Lenstra and Emmanuel Thomé and Joppe Bos and Pierrick Gaudry and Alexander Kruppa and Peter Montgomery and Dag Arne Osvik and Herman te Riele and Andrey Timofeev and Paul Zimmermann: **Factorization of a 768-bit RSA modulus** (published Feb 2010)
`https://eprint.iacr.org/2010/006` (last accessed 2018-01-14)

[13] CERT: **MD5 vulnerable to collision attacks** (published Jan 2009)
`https://www.kb.cert.org/vuls/id/836068` (last accessed 2018-01-14)

[14] Harshvardhan Tiwari; Centre for Incubation, Innovation, Research and Consultancy (CIIRC) Jyothy Institute of Technology, Bangalore, Karnataka, India; India: **Merkle-Damgård Construction Method and Alternatives: A Review** (published 12 2017)
`https://jios.foi.hr/index.php/jios/article/view/1070/787` (last accessed 2018-01-14)

[15] John Kelsey and Bruce Schneier: **Second Preimages on n-bit Hash Functions for Much Less than 2^n Work** (published Nov 2004)
`https://eprint.iacr.org/2004/304.pdf` (last accessed 2018-01-14)

[16] John Kelsey and Tadayoshi Kohno: **Herding Hash Functions and the Nostradamus Attack** (published Feb 2006)
`https://eprint.iacr.org/2005/281` (last accessed 2018-01-14)

[17] Der Spiegel: **NSA-Enthüllungen Chronologie der Snowden-Affäre** [**German news report**] (published Jul 2013) `http://www.spiegel.de/politik/ausland/nsa-spaehaktion-eine-chronologie-der-enthuellungen-a-910838.html` (last accessed 2018-01-20)

[18] Joseph Lorenzo Hall: **What the heck is going on with NIST's cryptographic standard, SHA-3?** (published Sep 2013)
`https://cdt.org/blog/what-the-heck-is-going-on-with-nist%E2%80%99s-cryptographic-standard-sha-3/` (last accessed 2018-01-20)

[19] KECCAK: **Yes, this is still Keccak!** (published Oct 2013)
`https://keccak.team/2013/yes_this_is_keccak.html` (last accessed 2018-01-20)