Cryptographic Hash Functions

Qiutai (Tom) Pan

**A. Abstract**

Cryptographic hash functions are an important aspect of computer security, and have many established and potential applications. Among the constructs that depend on cryptographic hash functions is the newly developed blockchain data structure. A cryptographic hash function is a specific type of function with several requirements such as preimage resistance and second preimage resistance. These properties will be discussed in detail, as well as the venerable Secure Hash Algorithm-2 (SHA-2) family of cryptographic hash functions. Finally, the recently developed SHA-3 functions, which are due to replace SHA-2 as the National Institute of Standards and Technology’s Secure Hash Algorithm standard, will be examined.

**B. Basics of Cryptographic Hash Functions**

1. Definitions

A hash function H is a function whose input is a variable-length block of data (i.e. string) M and whose output is a fixed-length value h where h = H(M). Then, M is referred to as the preimage of h (M is a data block whose hash value, using the hash function H, is h). Typically, the inputs of a hash function are first padded to an integer multiple of some fixed length, with the length of the original data block included in this padding. Since the outputs of a hash function are fixed in size while the inputs are not, for any particular hash value h and hash function H, there will usually be many preimages of h using the function H. In fact, if a hash function H outputs hash values of length n, takes as input data blocks of length b, and uniformly distributes hash values, then each hash value corresponds to 2^(b/n) preimages. A collision occurs when 2 distinct preimages of a particular hash value are known for a particular hash function H (x ≠ y, but H(x) = H(y)), and these are generally considered undesirable.

Generally, a hash function is considered to be “good” if it satisfies the following requirements. First, applying the function to a large set of possible inputs must produce outputs that are both evenly distributed and apparently random. Furthermore, a change to any bit or set of bits within a particular input must result in a very high probability of a change to the hash function’s output. Cryptographic hash functions are hash functions with some further restrictions. For a hash function to be considered a cryptographic hash function, it must be computationally infeasible (there is no algorithm to do so that is significantly more efficient than a brute force method) to find both an input data block that maps to a specified hash output (this is known as the one-way property) and a pair of input data blocks that map to the same hash output (collision resistant), among other properties. These requirements are described in depth below.

2. Requirements of cryptographic hash functions

As stated previously, cryptographic hash functions are a class of hash functions, and thereby must have variable input sizes but fixed output sizes. Furthermore, cryptographic hash functions should be reasonably efficient – for any such function H, H(x) should be relatively easy to compute for any possible input x. This allows one to develop practical implementations of H in either software or hardware. A cryptographic hash function must also be preimage resistant (one-way), as mentioned previously. A formal definition of this property is that for a cryptographic hash function H and a hash value h, it must be computationally infeasible to find a data block y such that H(y) = h. This essentially means that although it is easy to generate a hash value given an input, it is virtually impossible to go the other way, to generate an input “message” given a hash value. A diagram of what is prevented by preimage resistance is shown below in figure 1.

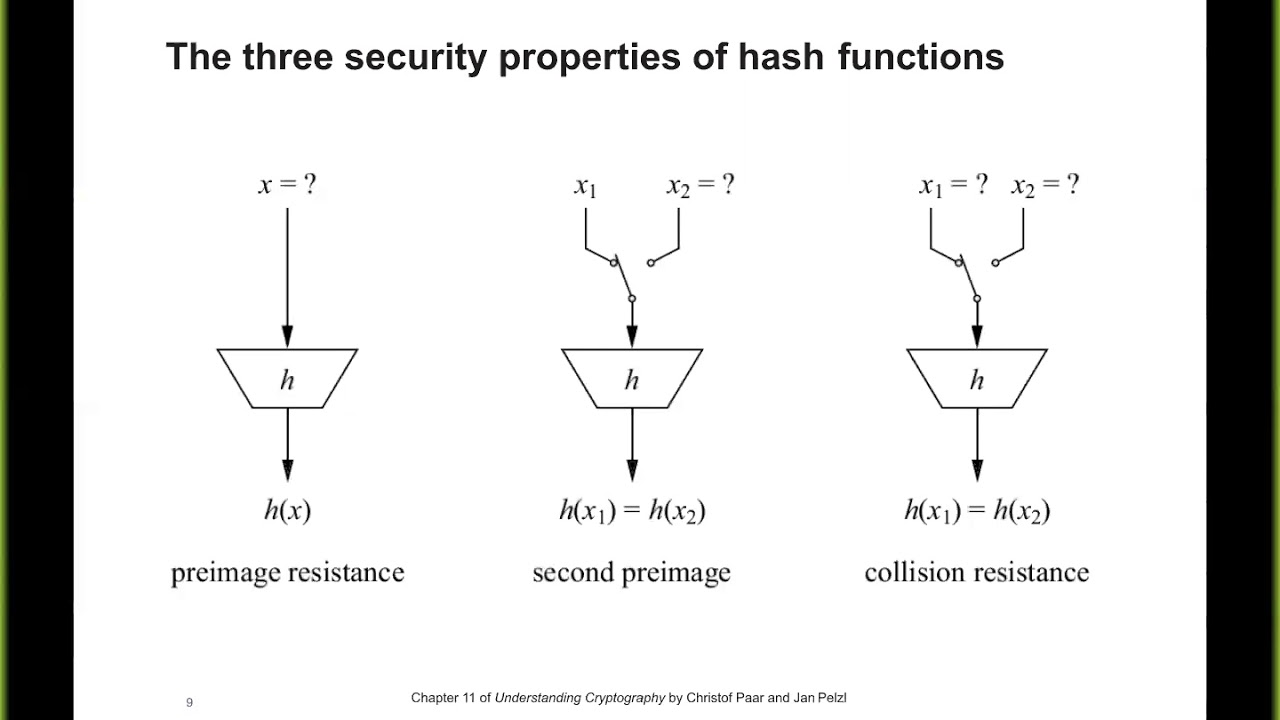


Figure 1 - Preimage Resistance

In addition, a cryptographic hash function must have the property of being second preimage resistant. The formal definition of this property is that for a hash function H and an input data block x, it must be computationally infeasible to find another data block y such that y ≠ x but H(y) = H(x). This essentially means that after one obtains a hash value for a particular data block, it must be virtually impossible to find a different input that gives the same hash value. A diagram of what is prevented by second preimage resistance is shown below in figure 2.

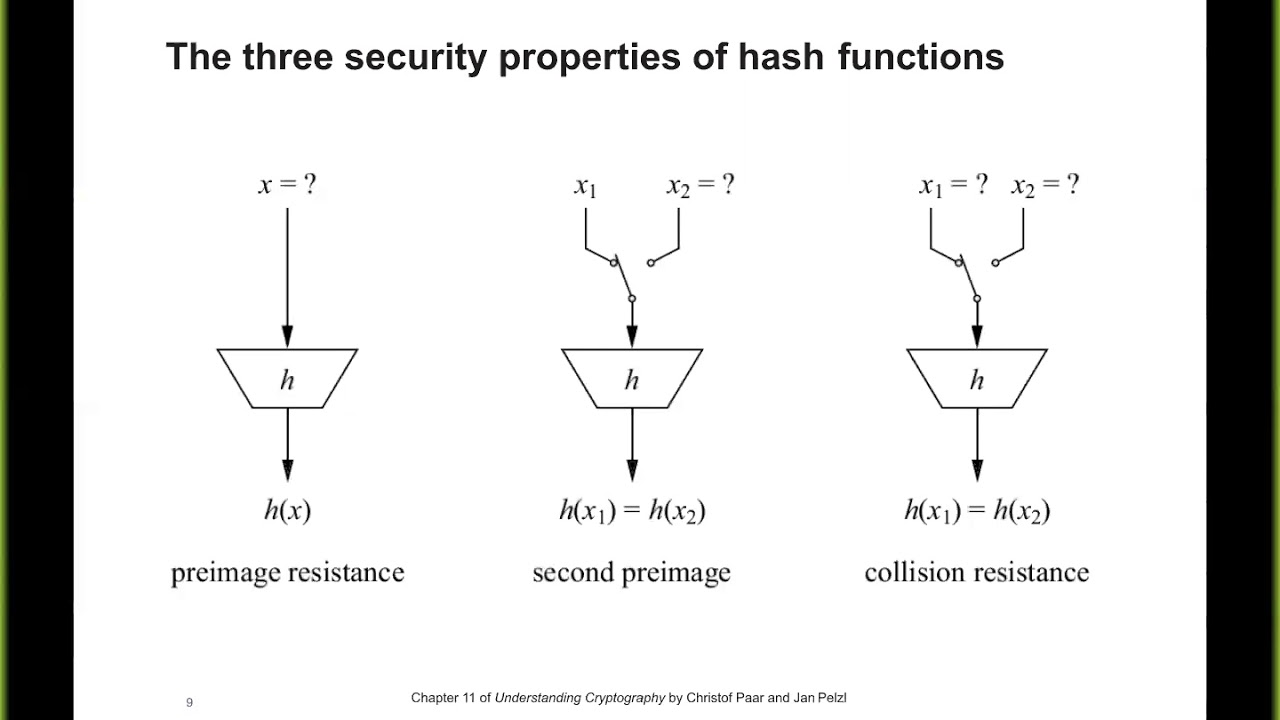


Figure 2 - Second Preimage Resistance

If a hash function satisfies the five previously discussed properties, then it is considered a weak cryptographic hash function. There is one more property that a hash function must satisfy in order to be considered a true (strong) cryptographic hash function – collision resistance. Formally, a hash function H is collision resistant if it is computationally infeasible to find any pair of input data blocks (x,y) such that H(x) = H(y). This essentially means that one cannot find two inputs that hash to the same value. One obvious benefit of this property is that it helps prevent attacks in which one party generates a message for another to sign. The first party would not be able to procure an alternate message that is more favorable to them in lieu of the original message. A function that is collision resistant is also necessarily second preimage resistant (collision resistance implies second preimage resistance), but the reverse is not necessarily true. Furthermore, there is unfortunately no relationship between collision resistance and preimage resistance, or between second preimage resistance and preimage resistance. A diagram of what is prevented by collision resistance is shown below in figure 3.

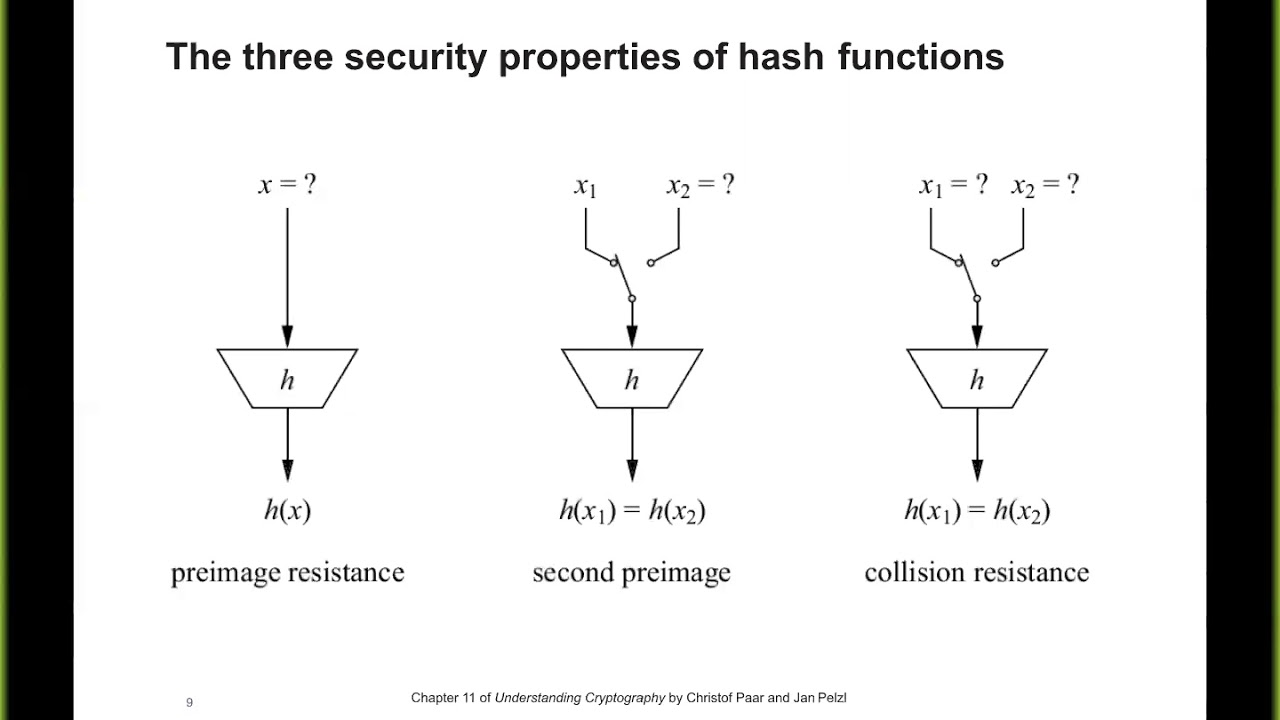


Figure 3 - Collision Resistance

Finally, there is another property that has not traditionally been considered a requirement for cryptographic hash functions, but is practically implied by the three resistance requirements – that the output of the function meets the standard tests for pseudorandomness. This is because the three resistance properties depend on the outputs of the hash function appearing to be random. A chart displaying the properties of some classes of hash functions is shown below in Table 1.



Table 1 - Properties Satisfied by Hash Functions

3. Attacks against cryptographic hash functions

The two main types of attacks are brute-force attacks and cryptanalysis attacks. For a brute-force attack, an adversary simply systematically tries all possible values until a “successful” value or set of values is found; no algorithm is used to increase the chances of success. For these attacks, the level of effort depends only on the length of the output of the hash function in question. When performing a brute-force preimage or second preimage attack against a cryptographic hash function H, an adversary attempts to find a value y such that H(y) is equal to some particular hash value. Values are picked at random (or in order) until a satisfactory one is found. Thus, if H outputs m-bit hash values, the level of effort is proportional to 2^m. When performing a brute-force collision attack, an adversary attempts to find two values x and y such that they yield the same hash value when used as input into H (H(x) = H(y)). Surprisingly, a brute-force collision attack actually requires considerably less effort than a brute-force preimage or second preimage attack. If H outputs m-bit hash values, then the level of effort required is proportional to only 2^(m/2). This can be explained by the birthday paradox – comparisons are made between pairs of outputs rather than comparing individual outputs to a fixed result.

Meanwhile, cryptanalysis attacks, rather than simply trying out possible values without rhyme or reason, seek to exploit some property of the cryptographic hash function. For example, most cryptographic hash functions involve the use of a compression function – a function that takes as input both an n-bit chaining variable (the output of the previous iteration of the function) and a b-bit data block (usually b is far greater than n), and produces an n-bit output. Generally, the chaining variable is initialized to some specified value at the start of the hash function’s internal algorithm while the final value of the chaining variable after all iterations of the compression function is the output of the hash function itself. Thus, if the internal compression function is collision resistant, then so is the resultant iterative hash function, and the problem of designing a collision resistant hash function reduces to that of designing a collision resistant compression function whose chaining variable length is equal to the desired hash value length. If a particular cryptographic hash function H has an internal compression function f, then a cryptanalytic collision attack on H would focus on the structure of f, and attempt to find an efficient way to produce a collision for a single execution of f (of course, the adversary must also account for the fixed initial value of the chaining variable). To measure the resistance of a cryptographic hash function to cryptanalysis, the level of effort required for such attacks is compared to the effort required for brute-force attacks on that function. Ideally, a cryptographic hash function would require a cryptanalytic effort equal to or greater than brute-force efforts for all potential attacks.

4. Applications of cryptographic hash functions

There are many applications for cryptographic hash functions for computer security. For example, they can be used to authenticate messages passed between entities. If used in this case, cryptographic hash functions are often referred to as message digests. Cryptographic hash functions are also used to create digital signatures and one-way password files (in which the hash of a password is stored rather than the password itself). Finally, cryptographic hash functions are necessary for the newly developed blockchain data structure, in which each individual entity (block) in the data structure has a field that contains the cryptographic hash of its other fields, as well as a field that contains a copy of the hash field of the previous block in the blockchain. This links each block in the blockchain with its immediate neighbors, and makes it very difficult for any block within the chain to be maliciously changed (as then its hash field would no longer match the previous hash field of the subsequent block, making it obvious that it has been modified).

**C. SHA2**

1. Origin of SHA-2

The SHA-2 cryptographic hash functions are the second latest generation of the Secure Hash Algorithm (SHA) family of cryptographic hash functions. The SHA-2 functions were developed from SHA-1, a cryptographic hash function with 160 bit outputs that was developed by the National Security Agengy in 1995. SHA-1 itself was based on earlier cryptographic hash functions such as MD5. Eventually, SHA-1 was found to be insecure (in fact, in 2010 an attack in which 2 distinct messages could be found within 2^69 operations that hash to the same SHA-1 value was described), and so improved versions of the function with longer hash values were developed. These were SHA-256, SHA-384, and SHA-512, whose outputs were 256, 384, and 512 bits long, respectively. The three functions were collectively referred to as SHA-2.

2. Overview of SHA-512

The SHA-512 cryptographic hash function, part of the SHA-2 standard, produces 512-bit hash value outputs, as its name suggests. It allows a maximum input length input of 2^128 bits. An input is modified and then processed in 1024-bit blocks, and these blocks themselves are divided into 64-bit words. The algorithm requires 80 rounds of an internal “round” function (i.e. compression function) to process a single input block.

The operations performed by the SHA-512 algorithm are as follows. First, the input is padded so that its length is congruent to 896 mod 1024. This essentially means that the resultant length will be 128 bits shy of being an exact multiple of 1024 bits. The padding bits are always added, even if the input message is already of the correct length (in this case, the padding will be 1024 bits long). This padding consists of a single 1 bit followed by the necessary amount (0-1023) of 0 bits. Next, a block of 128 bits is appended to the padded input that contains the length of the original input message (before padding). The block is interpreted as an unsigned 128 bit integer with the most significant bit first. The result of these first 2 steps is a data block that is an integer multiple of 1024 bits in length. Next, a 512-bit buffer that is used to hold the intermediate and final results of the internal round function is initialized. The buffer is represented as eight 64-bit registers (words), and each register is initialized to a specified 64-bit hexadecimal integer, with the most significant bytes stored in the lowest positions (big-endian format). These integers are actually obtained by taking the first 64 bits of the fractional parts of the square roots of the first 8 prime numbers. The modified input message is then processed in 1024-bit blocks.

The algorithm to process a block takes 80 rounds, where each round takes as input the current 512-bit buffer and updates the contents of the buffer. At the first round of processing the ith block Mi, the buffer contains an intermediate value Hi-1. Each round t makes use of a 64-bit value Wt derived from Mi, as well as a 64-bit constant Kt (where the Kt’s are the first 64 bits of the fractional parts of the cube roots of the first 80 prime numbers). Once obtained, the output of the 80th round (t = 79) is added to Hi-1 (which was the input to the first round) to produce Hi. This addition is performed independently for each of the 8 registers that form the buffer, and is done modulo 2^64. After all 1024-bit blocks of the modified input have been processed, the final value of the 512-bit buffer is the output hash value. The algorithm can be succinctly described as:

where IV is the initial value of the 512-bit buffer, abcdefghi is the output of the 80th round of processing for the ith input block, N is the number of blocks in the modified input message (including the padding and length field), SUM64 is addition modulo 64, and MD is the final output hash value.

3. SHA-512 round function

Each round in the processing of a 1024-bit block is defined by obtaining T1 and T2 as defined below:

where t is the current round (0 ≤ t ≤ 79)

h(e,f,g) = If e then f else g

Maj(a,b,c) = (a AND b) ⊕ (a AND c) ⊕ (b AND c) (true iff majority of a,b,c are true)

= ROTR14(e) ⊕ ROTR18(e) ⊕ ROTR41(e)

= ROTR28(a) ⊕ ROTR34(a) ⊕ ROTR39(a)

(ROTRn(x) is the circular right shift of the 64-bit input x by n bits)

Then, the 512-bit, 8-word buffer is updated as follows:

a = T1 + T2

b = a

c = b

d = c

e = d + T1

f = e

g = f

h = g

Furthermore, + is addition modulo 2^64, while Wt and Kt were defined previously.

Of the eight words of the buffer, six are simply assigned the previous value of another word. Only a and e are updated based on the data block being processed – a is modified based on all words except d as well as Wt and Kt, while e is modified based on d, e, f, g, h, Wt, and Kt.

**D. SHA-3**

1. Origin of SHA-3

Although a significant attack against one of the SHA-2 cryptographic hash functions has yet to be demonstrated, the fact that successful attacks have been performed on the functions that SHA-2 was derived from (such as MD5 and SHA-1) caused the National Institute of Standards and Technology (NIST) to look for an alternative, dissimilar cryptographic hash function standard. Thus, the NIST opened a competition for a new SHA standard in 2007, which was eventually won in 2015 by Keccak. Thus SHA-3 is a subset of the Keccak family of cryptographic hash functions. The primary mechanism of the Keccak functions is known as sponge construction. At its core, sponge construction consists of two phases – an absorbing phase and a squeezing phase. The absorbing phase consists of several iterations of XORing an input block into a subset of the current state, and then transforming the entire state based on a specified function. The squeezing phase consists of repeatedly reading output blocks from the same subset of the state, alternated with further applications of the specified function on the entire state.

In the Keccak family, the size of the portion of the state that is written to during the absorbing phase and read from during the squeezing phase is called the rate (r), while the size of the remaining portion of the state is called the capacity (c). For the subset of Keccak that forms SHA-3, the entire state is always 1600 bits, while the capacity is always twice the length of the output hash. Thus, the capacity determines the level of effort needed for brute-force attacks against a SHA-3 function: For brute-force preimage and second preimage attacks, the level of effort needed is 2^(c/2), while for collision preimage attacks the level of effort needed is 2^(c/4). Furthermore, the presence of the extra c bits in addition to the portion of the state actually used for input and output helps protect against the length extension cryptanalysis attacks than MD5 and its derivatives such as SHA-1 are susceptible to. The four primary members of the SHA-3 family are SHA3-224, SHA3-256, SHA3-384, and SHA3-512. These cryptographic hash functions output hash values of 224, 256, 384, and 512 bits, respectively. All aspects of the SHA-3 functions are provided in a 2015 NIST publication.

2. Overview of Keccak/SHA-3

A Keccak cryptographic hash function takes as input an input data block (string) N, a padding function *pad*, a function f that operates on data of size b, a rate r, and an output length d. The rate r must be strictly less than b, and the capacity c = b – r. Also, for the SHA-3 functions f must be a permutation function (its inverse must also be a function). The algorithm begins by padding the input N using the padding function *pad*. The padding function must be specified such that the result is a data block P with a length divisible by r (P = N || pad(r, len(N)). Letting n = len(P)/r, P is then divided into n consecutive r-bit strings Po…Pn-1. Next, the state S is initialized to be b 0 bits (S = 0b).

The absorbing phase then begins. For each data block Pi, the following operations are performed. First, Pi is extended by a block of c 0 bits, resulting in a block of length b. The resulting block is then XORed with the current state S. Finally, f is applied to the result of that operation, yielding a new state S. After all Pi’s have been processed, the output hash Z is initialized to be the empty string, and then the squeezing phase beings.

For the squeezing phase, the following operations are performed while the length of Z is less than d. First, the first r bits of S are appended to Z. If the length of Z is still less than d bits, then f is applied to S, yielding a new state S. However, for all SHA-3 functions the rate r is larger than d, and so this will never be necessary. Finally, Z is truncated to d bits if its length is currently greater than d. A high-level overview of the entire process is shown below in Figure 4.



Figure 4 - Diagram of Keccak/SHA-3

3. SHA-3 Padding Function

For each of the SHA-3 functions, the padding function used is 10\*1. This means that the padding consists of 1 bit followed by 0 or more 0 bits (the maximum amount is (r-1)), and a final 1 bit. Formally, the amount of 0 bits is (-len(N) – 2) mod r. The maximum amount of 0 bits occurs when the original input is of length congruent to (r-1) modulo r (1 bit shy of being an exact multiple of r bits). The initial 1 bit of the padding is required to ensure that inputs differing only in the amount of 0 bits at their ends do not hash to the same value. Furthermore, if the length of the input N is already divisible by r, then two 1 bits are appended, followed by another block consisting of a 1 bit followed by (r-2) 0 bits and finally another 1 bit. This specification is necessary so that an input with length divisible by r that ends with a block of bits resembling padding does not produce the same hash output as an input that is exactly the same except that the bits that look like padding are removed.

4. SHA-3 Permutation Function

For the SHA-3 functions, the 1600 bit state is divided into 64-bit words, and is usually represented as a 3-dimensional 5x5x(w = 64) array of single bits. For a state array A, several different concepts are defined. Among these are lanes and planes, where Lane(i,j) = A[i][j][0] || … || A[i][j][w-1] and Plane(j) = Lane(0,j) || … || Lane(4,j). Then the state S is defined as Plane(0) || … || Plane(4). A diagram of all divisions of the state array is shown below in Figure 5.

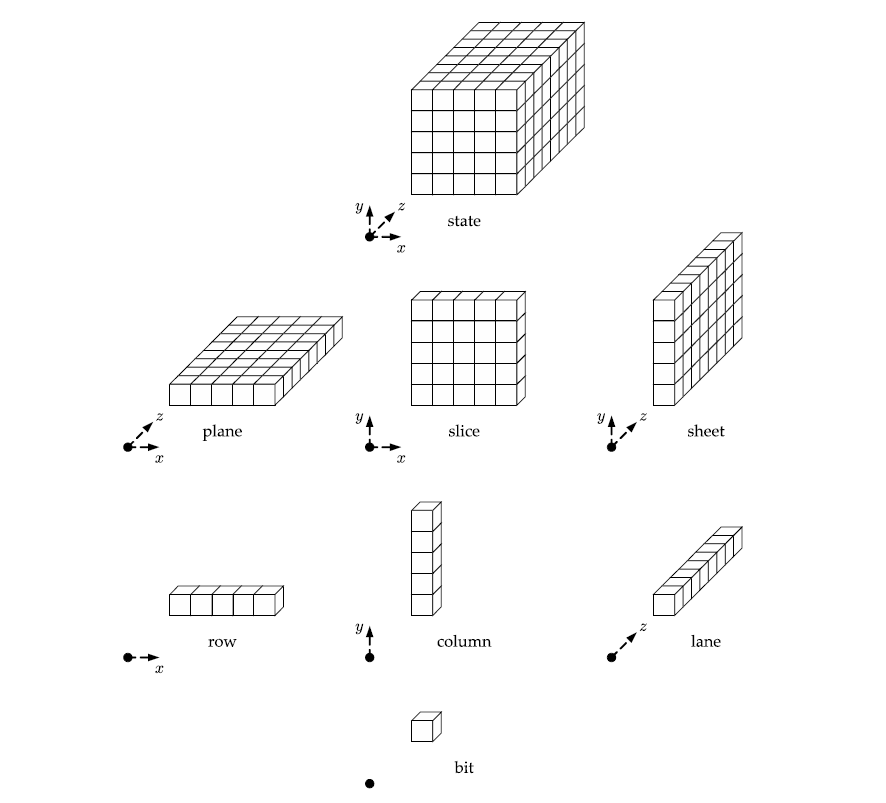


Figure 5 - Divisions of State Array

Furthermore, the state array is defined such that for any 0 ≤ x ≤ 4, 0 ≤ y ≤ 4, and 0 ≤ z < w, A[x, y, z] is the (w(5y + x) + z)th bit of the state S. The actual permutation function used in all SHA-3 functions consists of 5 subfunctions on the state array: θ, ρ, π, χ, and ι. There are 24 total rounds of the permutation function, and in each round θ, ρ, π, χ, and ι are applied to the current state array A in that exact order.

The function θ consists of the following operations:

a. For all pairs (x,z) such that 0 ≤ x ≤ 4 and 0 ≤ z < w, let C[x,z] = A[x,0,z] ⊕ A[x,1,z] ⊕ A[x,2,z] ⊕ A[x,3,z] ⊕ A[x,4,z]

b. For all pairs (x,z) as previously defined, let D[x,z] = C[(x-1) mod 5, z] ⊕ C[(x+1) mod 5, (z-1) mod w]

c. The result A’[x,y,z] = A[x,y,z] ⊕ D[x,z]

What this function essentially does is XOR each bit in the state array with the parity of two different columns in the array (neither of which it belongs to). This effect is shown below in Figure 6.

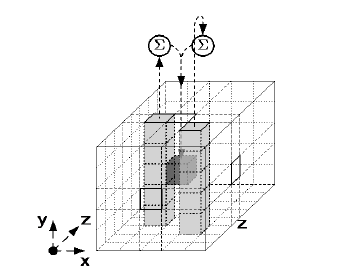


Figure 6 - The effect of θ on a single bit

The function ρ consists of the following operations:

a. For all z such that 0 ≤ z < w, A’[0,0,z] = A[0,0,z]

b. Let (x,y) = (1,0)

c. For t from 0 to 23, . Then (x,y) = (y, (2x + 3y) mod 5).

What this function essentially does is bitwise rotate each lane of the state array by an offset that is a triangular number and depends on the (x,y) coordinates of the lane (the (0,0) lane is unchanged). These offsets are given below in Table 2.

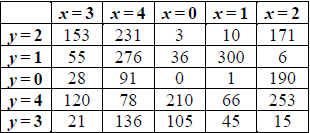


Table 2 - Offsets for ρ

The function π consists of a single operation: A’[x,y,z] = A[(x+3y) mod 5, x, z]. This serves to simply rearrange the lanes within the state array, and its effect on various slices of the state array is shown below in Figure 7.

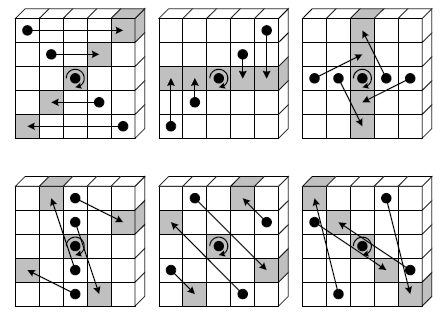


Figure 7 - Effect of π on various slices

The function χ also consists of a single operation: A’[x,y,z] = A[x,y,z] ⊕ ((A[(x+1) mod 5, y , z] ⊕ 1) & A[(x+2) mod 5, y , z]). This serves to XOR each bit with a non-linear function of 2 other bits in its row. Its application to a single row in the state array is shown below in Figure 8.

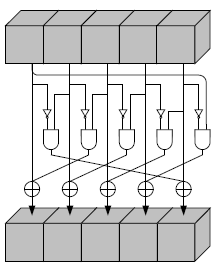


Figure 8 - Application of χ to a row of the State Array

Finally, the function ι has a relatively complex definition that is too long to be recounted here. However, its effect is that some bits of the (0,0) lane are modified in a manner that depends on the current round index. This is the only constituent of the permutation function that doesn’t necessarily preserve the symmetry of the state array.

5. Analysis of a JavaScript Implementation of SHA-3

An implementation of the Keccak family, and thereby all of the four main SHA-3 functions, is found at https://github.com/chrisveness/crypto/blob/master/sha3.js.

This implementation is written in the imperative language JavaScript. Unsually, the implementation expresses state as a 2-dimensional array of a user-defined data type that simulates 64-bit unsigned integers. Thus, the implementation of the 5 constituent functions of the permutation function differs slightly from their formal definitions given in the NIST publication. Furthermore, the implementation combined the operations of the ρ and π functions in order to lessen the amount of copying that needed to be performed on the entire state. Finally, the ι function was implemented very simply – instead of performing the formally defined operations, a round counter array was hard coded and the (0,0)th element of the state array (corresponding to the (0,0) lane of a 3-dimensional state array) was XORed with the relevant element of the round counter at each iteration. Testing the implementation is quite easy – simply go to https://www.movable-type.co.uk/scripts/sha3.html, choose one of the 4 SHA-3 functions, and type an input message. The hash output of the message should then display automatically.

**E. References**

Stallings, William. Cryptography and Network Security Principles and Practice. 5th ed. Boston: Pearson, 2017.

Dworkin, Morris J. "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions." 2015. doi:10.6028/nist.fips.202.

Veness, Chris. "Chrisveness/crypto." GitHub. January 09, 2018. https://github.com/chrisveness/crypto/blob/master/sha3.js.