Introduction to SHA-3
CubeHash vs. Keccak

Alexander Ros, 850408-4933, alr@kth.se
Carl Sigurjonsson, 830310-0054, carlsi@kth.se

Handledare Christian Bogdan

Abstract

Recently there's been discoveries in the current hash algorithm standard SHA-1 that make it vulnerable to attacks much faster than brute force. Although it is a serious weakness in theory, it still requires a tremendous amount of computing power to mount such an attack. However, with the rapid development of computer hardware the attacks are today more or less feasible even below government standards in terms of resources. Thus the need for a modern set of secure hash algorithms. In an effort to realize this NIST announced in 2007 a public competition to develop the best possible replacement to be named SHA-3 by 2012. This report will take a look at two of the fourteen candidates currently being evaluated - CubeHash and Keccak.

Table of Contents

1. Background

Like signing a formal document to ensure it's authentic, message digests or hashes are used to validate digital documents. The general idea is that when sending a message over an insecure channel, revealing the hash of the message would provide the receiver with the possibility to verify it's the correct message by comparing the message hash to the one supplied. However, these methods come with a security flaw, what if someone could generate a message that corresponds to the same hash? This is known as a hash collision and computer scientists have long tried to solve this problem by developing algorithms with the sole purpose of generating a unique hash for any given input.

This inevitably led to a demand for a standard to digitally sign data and in 1993 the National Institute of Standards and Technology (NIST) proposed an algorithm called Secure Hash Algorithm (SHA), also known as SHA-0. This algorithm was quickly withdrawn and replaced with a revised version known as SHA-1. And despite the minor change of adding a bitwise rotation in its compress function, NSA claimed the algorithm was secure. However, in 2005 it was suggested that the SHA-1 algorithm might have a flaw after Chinese scientists found a way to generate a collision a lot faster then the expected 2^{80} tries [1]. This lead scientist to the conclusion that SHA-1 was no longer to be considered secure and proposed a new algorithm to replace the standard. This resulted in a competition being announced in November 2007 to find a new hash algorithm. This paper will focus on two of the candidates in that competition, the Keccak and the CubeHash algorithms.

1.1 State of art

The Keccak algorithm is based on a hashing concept known as Sponges, which makes it part of the sponge family of cryptographic hash functions. The peculiar name derives from the fact that the Sponge compresses data in what is known as the absorption stage and then extract arbitrary length data at the later squeeze stage. What really makes this algorithm differ from the rest (MD5, SHA and RIPEMD) is the absence of an iterated application of a compress function, also known as Merkle-Damgård construction. While this function guarantees a collision resistance hash if the compression function is collision resistant recent, studies has shown that the Merkle-Damgård construction does not hold all the necessary properties that are expected from cryptographic hash functions [2]. Instead the Sponge relies on a permutation *f*. In Keccak the permutation basically consists of a function, Keccak-*f*[], which calculates a new state *n* times and then returns it. The value of *n* is determined using two of the algorithm parameters and mentioned later in the text.

```
KECCAK[r,c,d](M) {
  Initialization and padding
  S[x,y] = 0,                                   forall (x,y) in (0…4,0…4)
  P = M || 0x01 || byte(d) || byte(r/8) || 0x01 || 0x00 || … || 0x00

  Absorbing phase
  forall block Pi in P
    S[x,y] = S[x,y] xor Pi[x+5*y],              forall (x,y) such that x+5*y < r/w
    S = KECCAK-f[r+c](S)

  Squeezing phase
  Z = empty string
  while output is requested
    Z = Z || S[x,y],                            forall (x,y) such that x+5*y < r/w
    S = KECCAK-f[r+c](S)

  return Z
}
```

Image 1.1, Sponge construction pseudo code [1]

2. CubeHash

2.1 History

CubeHash is written by Daniel J. Bernstein (commonly known as djb), a professor at the University of Illinois in the US and well known for his work in cryptography and computer software industry. He is the author of popular programs such as djbdns and qmail as well as the stream cipher Salsa20 found in ECRYPT Stream cipher. Another interesting project he's currently involved in is DNSCurve with the goal of securing Internet name resolutions. In short this man is very determined to make a difference where IT security is concerned.

2.2 Description

CubeHash takes four parameters as input, they are as follows:

1. *r* - the number of rounds in {1,2,3,...}.
2. *b* - the number of bytes per message block in {1,2,3,...,128}.
3. *h* - the number of output bits in {2,4,8,...,512}
4. *m* - the actual message as a string of bits in size $\{0...2^{128-1}\}$

The short notation for the different parameter settings is CubeHash*r*/*b*-*h*. The author recommends CubHash16/32-512 as the default setting although when submitted to NIST in round 1 the suggested setting was CubeHash8/1. Djb later confessed it was 'much

[1] || indicates string concatenation and *w* is the *r*+*c* value, see Parameters for further information

more conservative than necessary' [2] and is about 16 times slower than the configuration recommended for round 2.

The algorithm consists of the following steps:

- Initialization the state *S* (1024 bits) based on (*h*, *b*, *r*)
- If required apply padding to *m* so that *m* is divisible by *b*.
- For every *b*-byte block in the padded *m* XOR with first *b*-bytes of *S* and perform *r* rounds of transformation on state *S.*
- Finalization of state *S.*
- Deliver the first *h* bits of state *S* as the final result.

The state is made up of 32 values each 32 bits. In the initialization step the first 3 values are set to *h*/8, *b*, *r* respectively and the rest are set to 0. Then *S* is transformed in 10*r* rounds described further down.

The padding is done by first appending a bit of value 1 to *m* then proceed with adding bits with value 0 until *m is* divisible by b.

The transformation consists of the following:

- Add x_{0jklm} into x_{1jklm} modulo 2^{32}, where *jklm* in {0000...1111} ie 16 iterations.
- Rotate x_{0jklm} upwards by 7 bits, for each (j; k; l;m).
- Swap x_{00klm} with x_{01klm}, for each (k; l;m).
- Xor x_{1jklm} into x_{0jklm}, for each (j; k; l;m).
- Swap x_{1jk0m} with x_{1jk1m}, for each (j; k;m).
- Add x_{0jklm} into x_{1jklm} modulo 2_{32}, for each (j; k; l;m).
- Rotate x_{0jklm} upwards by 11 bits, for each (j; k; l;m).
- Swap x_{0j0lm} with x_{0j1lm}, for each (j; l;m)
- Xor x_{1jklm} into x_{0jklm}, for each (j; k; l;m).
- Swap x_{1jkl0} with x_{1jkl1}, for each (j; k; l).

In the last step a integer of value 1 is xored to the last 32 bit value of the state as a measure to break any preserved symmetry through the transformations [3]. Finally the state is tranformed through 10*r* rounds once again.

2.3 Advantages

CubeHash is a comparatively simple hash algorithm. It is very small, in fact it is the smallest candidate only requiring 128 bytes of memory. This is required by the state as it is fixed at 128 bytes, all the operations basically rewrites the values in that same address space. This could be very important in special hardware implementations with limited resources.

The code size is also very small and the operations used like xor, add, rotate, swap, etc are all handled very well in most of todays processors.

The code itself is also designed to allow for parallizability as in most steps the operations are done independently in 16 iterations [4].

CubeHash also seems easy to configure given the input parameters. Trading speed for security or vice versa is simply a matter of adjusting those parameters. If security is desired, one can use CubeHash 8/1-512 but that would mean a cost of 200 cycles per byte . If instead energy usage weighs a major factor one can use CubeHash 16/32, for instance, that is far cheaper.

3. Keccak

3.1 History

Developed by Bertoni, Daemen, Peeters and Van Assche, Joan Daemen being the most recognized having co-designed the current encryption standard AES. Interestingly enough the Keccak algorithm shows no resemblance to AES rather builds upon a new concept in cryptography known as Sponges.

3.2 Description

The algorithm takes three parameters: *r*, *c* and *d*, where *r* is the bitrate parameter, *c* capacity and *d* the diversifier. *r* and *c* are performance parameters and controls how the algorithm behaves. Choosing a *c* that is two times the desired output size makes the sponge construction as strong as a random oracle and provides a preimage resistance of 2^n [4]. The *r* parameter affects the speed of the hash process and with the algorithm designed for 64bit processors the recommended sum of *r* and *c* is 1600 (*c* = 576 and *r* = 1024), making *r* a tradeoff between security and speed. The *r*+*c* sum value is also known as *b* or the permutation width *w* and used to determine how many rounds the Keccak-*f*[] should update the state before returning as $n = 12+2l$, where $2^l = b$. The diversifier parameter simply gets included in the padding process and works like a hash salt.

As seen in image 1.1 Keccak uses a 5x5 byte state *S* throughout the hashing process and the main methods of the algorithm consist the sponge inherited functions pad, absorb and squeeze. Like all other algorithms of the sponge family these functions rely on the permutation, Keccak-*f*[]. Choosing a good permutation is what tells the sponge based algorithms apart, in this case the Keccak-*f*[] is divided into five steps: θ, ρ, π, χ and ι.

```
KECCAK-f[b](A) {
  forall i in 0…n_r-1
    A = Round[b](A, RC[i])
  return A
}
```

```
Round[b](A,RC) {
  θ step
  C[x] = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4],   forall x in 0…4
  D[x] = C[x-1] xor rot(C[x+1],1),                            forall x in 0…4
  A[x,y] = A[x,y] xor D[x],                          forall (x,y) in (0…4,0…4)

  ρ and π steps
  B[y,2*x+3*y] = rot(A[x,y], r[x,y]),                forall (x,y) in (0…4,0…4)

  χ step
  A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y]), forall (x,y) in (0…4,0…4)

  ι step
  A[0,0] = A[0,0] xor RC

  return A
}
```

Image 1.2, Keccak-f[] pseudo code

3.2.1 Initiation process

The first step of the initiation is setting the elements of the state to zero. The second step is the padding [3]. This means extending the message by 1 and 0 to ensure the message can be fit into the fixed size state. It can be viewed as two separate methods, pad and enc. Having || symbolize string concatenation the input message *M* is extended according to

$$P = \text{pad}(M, 8)||\text{enc}(d, 8)||\text{enc}(r/8, 8)$$
$$P = \text{pad}(P, r)$$

Image 1.3, padding procedure

- pad(*M, n*) function adds a single 1 to the message and then adds as few 0 as possible to reach a length that is a multiple of *n*.
- enc(*x, n*) returns a string of *n* bits taken from *x* from LSB to MSB. I.e. works like a string truncate.

3.2.2 Absorb process

After the message has been padded it is split into 5x5 blocks that are xor'ed into the state one by one. Between every xor the state is updated using the Keccak-*f*[] function.

3.2.3 Squeeze process

This step builds the output string and works in a similar fashion as the absorb step. But instead of xor'ing blocks into the state it appends the output from the Keccak-*f*[] function to the output string until no further output is requested.

3.2.3 Keccak-*f*

The permutation of the Keccak is the set of methods θ, ρ, π, χ and ι, run consecutively *n* times where *n* is the number of rounds [4].

- θ is a linear mapping aimed at diffusion
- ρ used for translation between the lanes and aimed at providing inter-slice dispersion.
- π provides dispersion by applying a transpose of the lanes in the state. Can be viewed as multiplying every node pair (x,y) of the state with the matrix [[0,1],[2,3]].
- χ the only non-linear mapping of Keccak. Worth mentioning is that χ is a complement of the linear function called γ used in several other ciphers, e.g. PANAMA or RADIOGATUN.
- ι provides protection against symmetry attacks, e.g. slide attack. By adding round constants to the state and therefore adding symmetry dispersion.

3.3 Advantages

4. Method

5. Result

5.1 Discussion

6. References