Decentralised Hash Function

Petr Ležák†

Faculty of Electrical Engineering and Communication Brno University of Technology

Summary
In this article, a new hash function is introduced. The hash function is calculated by several parties – the client and one or more servers. If a user or attacker wants to calculate the hash of any message, he has to query all the servers. This means that if the attacker wants to invert the hash function by brute-force attack, he has to query the servers frequently. Servers can detect a heavy load and deny queries submitted by the attacker, or limit them. The calculation is done in a way that no server can detect which message is hashed nor trick the client to calculate a wrong value. This new decentralised hash function is useful in the case of wanting to hide some information by hashing, but the information has relatively low entropy, so brute-force attack is possible. One example is hashing of passwords to store them.

Key words:
Hash function, security, confidentiality, decentralisation

1. Introduction

The classical hash function h=H(m) assigns the fixed length hash h to a variable length message m. There are three requirements the that hash function must fulfil [1]:

Requirement 1 *Pre-image resistance - it is not possible to find message* m *for given hash h so* h=H(m).

Requirement 2 *Second pre-image resistance - it is not possible to find message* m_2 *for given message* m_1 *so* $H(m_1)=H(m_2)$.

Requirement 3 *Collision resistance - it is not possible to find two messages* m_1 *and* m_2 *so* $H(m_1)=H(m_2)$.

One possible use of the hash function is to safely store passwords. Instead of storing password p we can store h=H(p) or better h=H(s||p), where s a is randomly chosen string called salt. The salt s is stored with the hash h and is a protection against the pre-creation of hash dictionaries, called rainbow tables. To verify the password, we simply perform the calculation again and compare the calculated hash with the stored one. The attacker who gets h and s cannot inverse the hash function so, cannot calculate the correct password p. If the password p has enough entropy, then it is stored safely. But if it has not, the attacker can try to hash different passwords and see if the calculated hash is the same as the stored one. The only way of preventing this brute-force or dictionary attack is to limit the number of hash calculations by the attacker. A classical method of doing this is to make the hash calculation slow by hashing the input many times. In this article, another method is described – the use of one or more trusted servers.

2. General definition of decentralized hash function

Let us define the decentralised hash H' by Equation (1):

$$H'(m)=H_2(f(H_1(m))||m) \qquad (1)$$

H_1 and H_2 are hash functions and f is a function that can be calculated just by the server and returns a fixed length byte sequence. We have to prove that this definition preserves the properties of the hash function H_2.

We make a substitution $g(m)=f(H_1(m))||m$. Please note that g(m) is defined in such a way that it maps different messages m to different values g(m) regardless of the definition of functions f and H_1 provided that the result of function f has a constant length. This is ensured by concatenation of message m. Now we can prove that function H' has the properties of the hash function

1. Pre-image resistance: Suppose that we can find message m for given hash H'(m). But then we can calculate g(m) which means that we can invert hash function H_2. If hash function H_2 is pre-image resistant then hash function H' is also pre-image resistant.
2. Second pre-image resistance: Suppose that we can find message m_2 for a given message m_1 so that $H'(m_1)=H'(m_2)$. But then $g(m_1)$ and $h(m_2)$ are different messages with the same hash H_2. If hash function H_2 is second pre-image resistant for all messages (especially for messages in the form of g(m)), then hash function H' is also second pre-image resistant for all messages.
3. Collision resistance: Suppose that we can find two different messages m_1 and m_2 with the same hash $H'(m_1)=H'(m_2)$. This means that $H_2(g(m_1)) = H_2(g(m_2))$ so we have two different messages $g(m_1)$ and $g(m_2)$ with the same value of hash function H_2. If hash function H_2 is collision resistant, then hash function H' is collision resistant too.

If we model hash function H_2 as a random oracle, then the call of H(m)' for different messages m means the call of H_2 with different inputs. This means that H(m)' can be modelled as a random oracle too.

Manuscript received August 5, 2015
Manuscript revised August 20, 2015

If the function f is some kind of message authentication code (MAC), then we need to call this MAC function for every message m to calculate H(m)'. If the MAC can be calculated just by any server, then the server can control the ability of H(m)' calculation. However, further problems arise. The server receives $H_1(m)$ so it can try a brute-force attack on $H_1(m)$ to get message m. We have to prevent this somehow, to ensure the confidentiality of message m. The server can also return a wrong value instead of $f(H_1(m))$ so we need a method of auditing the server.

3. Preconditions

Let us have a group G with generator g and prime order q. Requirement 4 must hold on group G:

Requirement 4 We suppose that the Decision Diffie-Hellman Theorem (DDH) holds on group G. This theorem states that for a given randomly chosen a and b the attacker who knows g^a, g^b and g^c cannot decide if c=a*b mod q. More information about the DDH theorem can be found in [2].

Then we need a function Rd that converts a fixed length sequence of bytes (hash value) to a group element. We also need function Wr that converts the group element to sequence of bytes. We suppose that Requirement 5 holds on function Rd and hash function H_1:

Requirement 5 *For every message m* $Rd(H_1(m))$ *is equivalent to* g^a *for unknown randomly chosen value a.*

We require H^2 to be a hash function, so it fulfils Requirements 1, 2 and 3. In addition, we have a requirement about hash value distribution:

Requirement 6 *For two different randomly chosen messages* m_1 *and* m_2 *probability* $P(H(m_1)=H(m_2))\leq Y$ *for some fixed negligible Y.*

4. Definition of decentralised hash

We define the function f by Equation (2) so the decentralised hash is defined by Equation (3).

$$F(x)=Wr((Rd(x))_k) \quad (2)$$

$$H'(m)=H_2(Wr((Rd(H_1(m)))^k||m) \quad (3)$$

Here, k is a private key known by the server or shared by several servers.

5. Calculation of hash

Let us have n servers with indices from 1 to n. Each server has a private key k_i. The compound private key k is defined as $k\equiv\prod_{i=1}^{n} k_i (\bmod q)$. The situation is illustrated in Figure 1. The algorithm for the calculation of a decentralised hash of message m is described below:

1. Client calculates $S_0=Rd(H_1(m))$.
2. For $1\leq i\leq n$ client calculates $S_i=S^{ki}_{i-1}$ using *i*-th server.
3. Client calculates $H'(m)=H_2(Wr(S_n||m)$.

Calculation of $S_i=S^{ki}_{i-1}$ using i-th server is described below:

1. Client obtains server's certificate with public key L_i for verification of response signatures.
2. Client obtains server's public hashing key K_i signed by the server and verifies it's signature using key L_i.
3. Client randomly selects non-empty set S of distinct numbers in a range from 1 to Z. The order is not important. z is the security parameter described later.
4. For $1\leq j\leq z$ do:
    1. Client randomly selects number $1\leq v_{i,j}\leq q$.
    2. If $j\in S$ then client asks server for calculation of S^{Ki}_{i-1}:
        1. Client sends $c_{i,j}=S^{vi,j}_{i-1}$ to *i*-th server.
        2. *i*-th server sends $d_{i,j}=c^{ki}_{i,j}$ to the client.
        3. Client verifies message signature using public key L_i.
        4. Client calculates $e_{i,j}=d^{(v_{i,j})-1}_{i,j}$.
    3. If $j\neq s$ then client asks server for calculation of check value:
        1. Client sends $C_{i,j}=g^{Vi,j}$ to *i*-th server.
        2. *i*-th server sends $d_{i,j}=c^{k}_{i,j}$ to the client.
        3. Client verifies message signature using public key L_i.
        4. Client verifies that $d_{i,j}=K^{v_{i,j}}_{i}$. If not, then he knows that the server is behaving incorrectly and stops the protocol.
5. The client verifies that all calculated values $e_{i,j}$ are the same. If not, then he knows that the server is behaving incorrectly and stops the protocol. Result S_i is one of the calculated equal values $e_{i,j}$.

Please note that value $e_{i,j}$ does not depend on $v_{i,j}$ because $e_{i,j}=d^{(vi,j)-1}_{i,j}= c^{v_{i,j}\bullet k\bullet v-1}_{i,j}= c^{k}_{i,j}$. The server's responses are digitally signed. The signature must be constructed in such a way that it also contains a corresponding request. If a server cheats, then the client will have proof of that cheating. In the case of a wrong response to check value, the proof are values $v_{i,j}$, $v_{d,j}$ and the signature. In the case of two different values $e_{i,j}$, the proof are the value S_{i-1}, corresponding values $v_{i,j}$ and $v_{d,j}$ and the signatures.

6. Properties of the decentralised hash function

Now we consider some properties of the system. Firstly, we should consider the confidentiality of messages. The client sends to the first server a set of messages $S^{v_{i,j}}_0$ for at most z different randomly chosen values $v_{i,j}$. Then the client sends a set of values $S_1^{v_{2,j}}=S_0^{k_1 v_{2,j}}$ to the second server and, so on. Generally, the client sends the set of values $c_{i,j}=s_0^{v_{i,j}\prod_{l=1}^{i-1} k_l}$. We can pessimistically assume that servers know the private keys K_l of the preceding servers, so they know values (cryptograms) $C_{i,j}'=S_0^{V_{i,j}}$ for $1\leq i\leq n$ and $1\leq j\leq z$ in the worst case scenario. Please note that for every group generator S_0 there is exactly one vector of random values $V_{i,j}$ that leads to the vector of cryptograms $C'_{i,j}$. This means that the attacker cannot distinguish between different generators S_0. So the attacker cannot test if a given message is hashed - value S_0 is encrypted by a one-time pad. Also, the generator g is one of the possible values S_0. This means that the server cannot distinguish between a real query and a check value.

6.1 Confidentiality of message

The message m is hashed by the hash function H_1. This hash is then transformed by the function Wr to a group generator and powered by several random values $v_{i,j}$. As previously discussed, this powering is equal to one-time padding, so the server(s) or attacker listening on wire cannot calculate the message or even test if a given message is hashed.

6.2 Correctness of the calculated hash

One or more servers can calculate wrong values. Suppose that i-th server want to cheat and trick the client to calculate the wrong hash. The client sends z different requests to the server, some of them are real queries used for the calculation of the hash, some ask just for the calculation of the check value. To be able to trick the client to use the wrong value $d_{i,j}$ he has to correctly calculate $d_{i,j}$ for those indices j where the client has used check values and use the same incorrect value for those indices j where the client has queried a real message. So the server has to guess which queries are real and which are just checks - it cannot distinguish between them. The probability that this

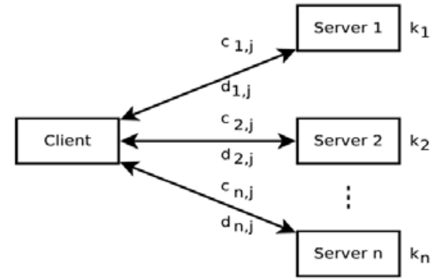

Figure 1: System schema

guess is correct is $\frac{1}{2^z}-1$ because we have 2^z possible sets s without one illegal (all queries are checks).
The client can control how confident he should be about the hash value by security parameter z. For instance, if the system is used to hash authentication passwords, then when the password is established, we need to be confident of the correctness of the hash so we can use $z\approx 20$ to have the probability of a successful attack approximately one in a million. On the other hand, when the user just logs in, we can use z=2 because, in the worst case scenario, he can try to log in again.

6.3 Necessity of communication with servers

We now prove that without interaction with all the servers it is not possible to calculate the hash H'(m).
Suppose that the attacker can calculate the hash without interaction with all of the servers. Without losing generality, suppose that the attacker can interact with all the servers except j-th. This means that he can calculate S_{j-1}. If S_0 is by Requirement 5 equal to g^a for an unknown randomly chosen value a then S_{j-1} is equal to $g^{\alpha\cdot\prod_{i=1}^{j-1} k_i}=g^\beta$ where $\beta=\alpha\cdot\prod_{i=1}^{j-1} k_i \bmod q$. Because q is prime, then for any fixed set of values $1\leq ki\leq q$ the relation $\beta\rightarrow\alpha$ is bijective so S_{j-1} is equal to g^β with the unknown randomly chosen value β.
Now suppose that the attacker wants to break the DDH theorem and test if value S'_j is equal to correct value S_j. He can then calculate value $s'_n=s'^{\prod_{i=j+1}^{n} k_i}_j$. Because mapping $s_j\rightarrow s_n$ is bijective then S'_n is correct if and only if S'_j is correct. Then he can calculate H''(m). If S'_j is correct then H''(m) is of course also correct. If S'_j is incorrect then because of requirement 6 there is very high probability that H''(m) is also incorrect. So if the attacker knows the correct hash H'(m) he can use it to break DDH theorem. If we suppose that DDH theorem is valid then the attacker must query all servers to calculate correct hash.

7. Implementation

As a group G we can use the subgroup of quadratic residues of the multiplicative group p where p=2q and both P and q are primes. We can use the function $Rd(x) = X^2 \bmod q$ with x treated as a number in big endian. Squaring by two makes the number a quadratic residue. More can be found in [3]. We can use function Wr which simply converts a number into a fixed length sequence in big endian notation.

We can define $H_1(m)$ = SHAKE256('A'||m, 128+8 • log256(p)) and $H_2(m)$=SHAKE256('BB||m,512). The shake is a hash function with configurable output length. The length of is at least 128 bit greater than the bit width of . This ensures that after all remainders have nearly the same probability of occurrence. Please note that the choice of hash functions is on the client; the same server will work with any hash function. For the server only group G is mandatory. If the client uses more servers, they must use the same group G.

The client communicates with the server using any transport protocol. For simplicity we have used Java RMI, but any protocol, for instance TCP or TLS over TCP, can be used. There are several methods that the client can call on the server:

- int getProtocolVersion() - returns maximal supported version of the protocol, 1 for now.
- Certificate getCertificate() - returns certificate of the server with public key for verification of signatures.
- int getKeyCount() - returns *keyCount*, the number of hashing keys.
- SignedResponse getKeyAt (final int keyIndex) - returns hashing key with given index. This key contains public key K_i and parameters of group G.
- SignedResponse powerByKey (final int keyIndex, final byte[] base) - powers group element encoded in field *base* by public key with index *keyIndex*.

SignedResponse contains a response from the server with a digital signature. The signature is computed from the request and response, so if the server sends a wrong response, the client will have proof of this cheating. The signing key with certificate can be obtained by method getCertificate. Hashing public keys k_i can be obtained by method getKeyAt. Keys are numbered by index $0 \leq$ keyIndex < keyCount. The server can add new keys or make old ones deprecated but it should not delete or change them. This ensures backward compatibility. The structure of response with request can be seen in Figure 2. The signature is calculated from both request and response (the structure of request is specified just for the purpose of the signature; of course that request is not sent back).

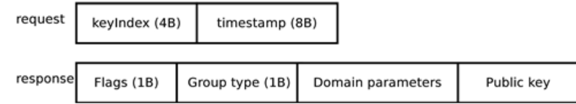

Figure 2: Key request and response

Another type of request is the query for calculation of the decentralised hash. The client sends $c_{i,j}$ to the server and the server replies $d_{i,j}=c^k_{i,j}$. It is realised by the method powerByKey. Hash request and response can be seen in Figure 3.

Figure 3: Hash response

For the present, only one group type is defined: the subgroup of quadratic residues of group p where p=2q and both p and q are primes. This group has groupType = 0. Domain parameters contains just a number q and the public key is just is just a number K_i. Hash query is a number $c_{i,j}$ and hash response is a number $d_{i,j}$. All numbers are encoded as shown in Figure 4. In the beginning, there is a number of bytes required for encoding of the number. This size is written as 4 bytes in big endian. There after a number is encoded as big endian using size bytes.

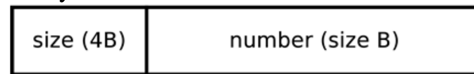

Figure 4: Number encoding

Conclusion

This decentralised hash function can be used instead of the classic hash function anywhere we have access to the server. It is beneficial to use it in situations where we are hashing some secret information with possibly low entropy, e.g. passwords or security questions. The testing server and the client are implemented in the AALG library, see [4].

We have proved that the decentralised hash function is at least as secure as the hash function H_2. We also know that it is necessary to query the server(s) to calculate the hash value; this depends on the DDH theorem. We have also proved that the server cannot obtain a hashed message nor even test if any concrete message is hashed. The server also cannot trick the client to calculate a wrong hash value. These properties are unconditional as long as random values $v_{i,j}$ and set S are true random and the security parameter z is large enough.

References

[1] SMART, N. Cryptography: An Introduction. [online]. 3rd Edition. URL (http://www.cs.bris.ac.uk/~nigel/Crypto_Book/)

[2] BONEH, D. The decision Diffie-Hellman problem. In: Third Algorithmic Number Theory Symposium: Lecture Notes in Computer Science. Springer-Verlag, 1998 Vol. 1423. [online]. URL (http://crypto.stanford.edu/~dabo/pubs/abstracts/DDH.html)

[3] Wolfram Research, Inc. Quadratic Residue [online]. URL (http://mathworld.wolfram.com/QuadraticResidue.html)

[4] LEŽÁK, P. The abstract algebra library [online]. URL (http://sourceforge.net/projects/aalg/)