Signatures and Hash Functions

Ahto Buldas   Aleksandr Lenin

Nov 18, 2019

Digital Signature Scheme

Involves three algorithms:

G (key-generation): generates a public key pk, and a corresponding private key sk

S (signing): on input the private key sk and a message m generates the signature $s \leftarrow S(\mathsf{sk}, m)$

V (verifying): on input the public key pk, a message m, and a signature s, either accepts or rejects

Correctness:

$$\mathsf{P}[(\mathsf{pk}, \mathsf{sk}) \leftarrow G, V(\mathsf{pk}, m, S(\mathsf{sk}, m)) = 1] = 1 \ .$$

Notions of Security: Attack Types

In 1988, Goldwasser, Micali, Rivest described attack types for digital signatures:

Key-only attack (KOA): adversary only has pk

Known message attack (KMA): adversary has valid signatures for a list of messages not chosen by adversary.

Adaptive chosen message attack (CMA): adversary learns signatures for arbitrary messages chosen by adversary.

Notions of Security: Attack Results

Goldwasser, Micali, Rivest (1988):

Total break (TB): recovery of the signing key.

Universal forgery (UF): ability to forge signatures for any message.

Selective forgery (SF): a signature on a message of the adversary's choice.

Existential forgery (EF): a valid message/signature pair not already known to the adversary.

EF-CMA Security

The strongest notion: security against existential forgery under an adaptive chosen message attack.

1. Sample key $(\mathsf{sk}, \mathsf{pk}) \leftarrow G$
2. Run the adversary $(m, s) \leftarrow A^{S(\mathsf{sk},\cdot)}(\mathsf{pk})$

The adversary A is successful if $V(\mathsf{pk}, m, s) = 1$ and A never queried $S(\mathsf{sk}, m)$.

Plain RSA Signatures

sk – RSA secret key (n, d), where n is the modulus and d is private exponent

pk – RSA public key (n, e), where e is the public exponent

$S(\mathsf{sk}, m) = m^d \bmod n$

$V(\mathsf{pk}, m, s) = 1$ if and only if $s^e \bmod n = m$

Not EF-CMA secure: Adversary, without making any S-queries:

1. Chooses s
2. Computes $m \leftarrow s^e \bmod n$
3. Outputs (m, s)

Secure RSA signatures use *paddings*: $s = P(m)^d \bmod m$

Signing Long Messages with Hash Functions

$H\colon \{0,1\}^* \to \{0,1\}^k$ converts any message m to a k-bit hash $H(m)$

Hash and Sign paradigm: Messages are hashed before applying the signature function S

$$s \leftarrow S(\mathsf{sk}, H(m))$$

Hash Functions

A hash function converts a large, possibly variable-sized amount of data into a small datum (hash) that may serve as an index to an array.

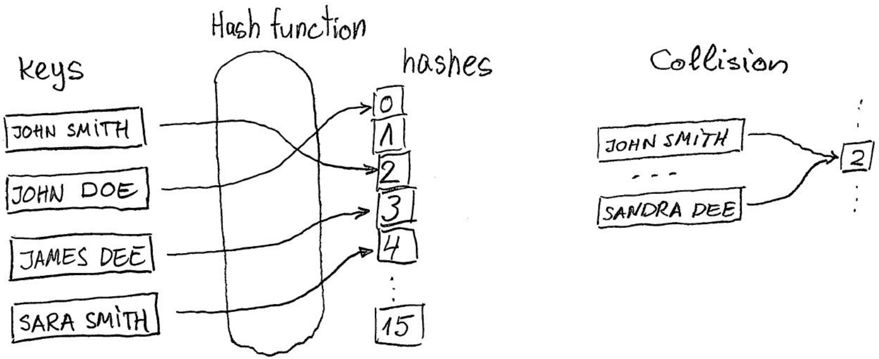
Hash functions are mostly used to accelerate table lookup or data comparison tasks—such as finding items in a database, detecting duplicated or similar records in a large file, finding similar stretches in DNA sequences, etc.

With a hash function in use, we can check the presence of an item in a database with just one single lookup!

The idea was proposed in the 1950s, but the design of good hash functions is still a topic of active research.

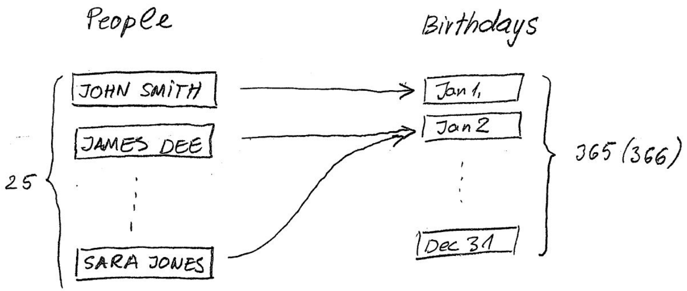
Collisions

A hash function may map two or more keys to the same hash value—this is what we call a *collision*. In most applications, it is desirable to *minimize the occurrence of collisions*, which means that the hash function must map the keys to the hash values as evenly as possible.



Birthday Paradox

Birthday paradox: For randomly chosen 25 people, very likely we have two with the same birthday.



NB! This does NOT mean that if you are among those people, you will very likely find someone with the same birthday!

How many different hash values do we need?

How large hash values should we use for uniquely identifying N items?

Answer: for a *good* hash function, we need about N^2 different hash values.

If $N = 2^{256}$ documents will be created all over the world during the existence of mankind, we need about $2 \cdot 256 = 512$ output bits to identify them uniquely.

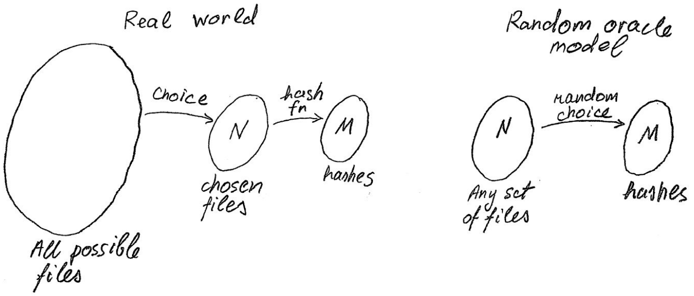
Eddington number: number of protons in the observable universe is about

$$1.57 \cdot 10^{79} \approx 136 \cdot 2^{256} \quad .$$

This is still just a small fraction of all possible 40 byte sequences!

Modeling Hash Functions With Random Oracles

The values of a *good hash function* are distributed as evenly as possible. A good hash function is modeled as a random function (a *random oracle*)—the computation of hash value is replaced with a uniformly random choice (all hash values occur with the same probability $1/M$).



Bounds for Collision Probability

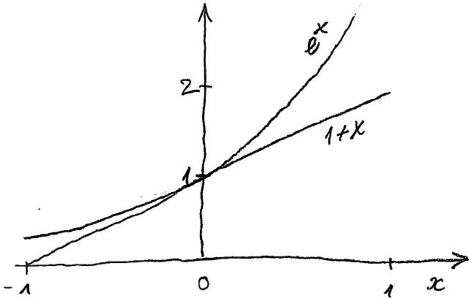
Say we have N files and the hash function has k output bits, i.e. there are $M = 2^k$ possible hash values. If the hash function is modeled as a random oracle, the probability $P(N, M)$ that all N files have different hash values has the following bounds:

$$1 - \frac{N(N-1)}{2M} \leq P(N, M) \leq e^{-\frac{N(N-1)}{2M}} \quad .$$

Homework exercise: Show that $P(N, M)$ can only decrease if the hash function deviates from random oracle, i.e. the output probabilities differ from $\frac{1}{M}$.

A Useful Inequality from Calculus

We used the inequality $1 + x \leq e^x$ that holds for all real x. This also means that $1 - x \leq e^{-x}$.



The inequality holds because e^x is *convex* and $1 + x$ is the *tangent line* of e^x at $x = 0$.

Upper Bound

For the upper bound, we use the observation that

$$
\begin{aligned}
P(N,M) &= \left(1-\frac{1}{M}\right)\cdot\left(1-\frac{2}{M}\right)\cdot\ldots\cdot\left(1-\frac{N-1}{M}\right)\\
&\le e^{-1/M}\cdot e^{-2/M}\cdot\ldots\cdot e^{-(N-1)/M} = e^{-(1+2+\ldots+N-1)/M}\\
&= e^{-N(N-1)/2M}\quad .
\end{aligned}
$$

Explanation: We compute the hashes of the N files one by one, considering each hash computation as a uniformly random selection of a hash value. Before the i-th selection, we already have $i-1$ selected hash values and the probability that the new hash is one of those is $\frac{i-1}{M}$.

Lower Bound

To get a lower bound for $P(N,M)$, we first observe that for any pair of files, the probability that they have the same hash is $1/M$. As there are $N(N-1)/2$ pairs, the probability of having a collision is upper bounded by $\frac{N(N-1)}{2M}$ and hence,

$$P(N,M) \geq 1 - \frac{N(N-1)}{2M} \quad .$$

Examples

For large $N \approx \sqrt{M}$, we have

$$\frac{1}{2} \leq \lim_{M \to \infty} P(\sqrt{M}, M) \leq e^{-1/2} \approx 0.6065306597 \ .$$

For the Birthday paradox, we set $N = 25$ and $M = 365$ and get

$$P(N, M) \leq 0.4395878005 \ .$$

Cryptographic Hash Functions

We study how to protect the uniqueness of identifiers against malicious behavior of users.

Cryptographic hash functions are designed for this purpose.

We describe the main properties that are required from cryptographic hash functions.

We go through some examples to show where these security properties are needed in practice.

Three Main Security Properties

Cryptographic hash functions are designed for having the following security properties:

◦ *One-wayness*—hardness of finding a message given its hash.

◦ *Second pre-image resistance*—hardness of modifying a message without

changing its hash.

◦ *Collision-freeness*—hardness of finding two different messages with the

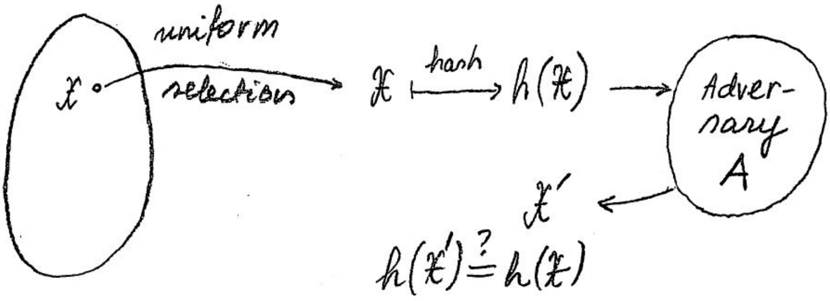
same hash.

Collision freeness is the strongest requirement of the three, because the adversary has the largest degree of freedom.

Note that `CRC32` does NOT meet any of the three requirements! It is intended to fight against occasional errors, not malicious attacks.

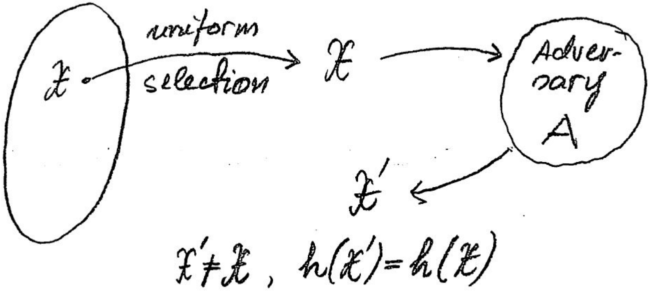
One-Wayness (or Pre-Image Resistance)

Given a hash $h(X)$ of a randomly chosen input X, it is hard to find an input X' with the same output $h(X') = h(X)$. Here, X is not known to the adversary!



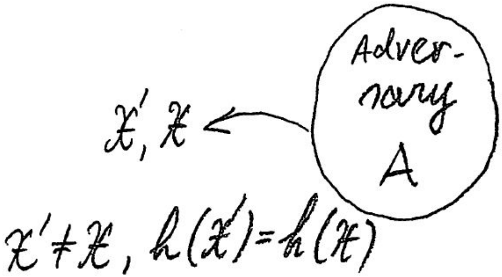
Second Pre-Image Resistance

Given a randomly chosen input X, it is hard to find a different input $X' \neq X$ with the same output $h(X') = h(X)$. Here, X is known to the adversary!



Collision Resistance (or Collision-Freeness)

It is hard to find two distinct inputs $X \neq X'$ with the same output $h(X') = h(X)$.



Examples for One-Wayness and Second Pre-Image Resistance

We need one-wayness in a situation, where X contains secret data. We have to uniquely identify X but do not want that the identifier reveals the contents of X.

We need second pre-image resistance if we want to protect a public document X from malicious modifications.

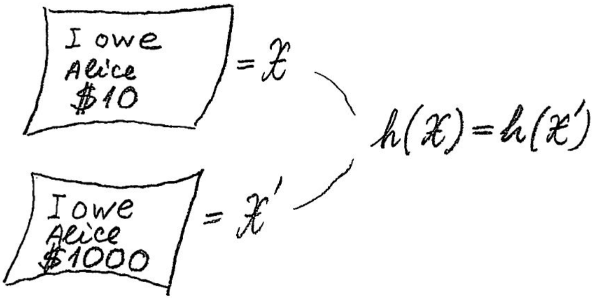
For example, if we keep the hash $h(X)$ of X in a safe place, it should be impossible to create a modified document X' with the same hash, because then the modifications were undetectable.

Digital Signatures and Collision Resistance

We create two contracts X and X' with the same hash, where X' is clearly favorable to us.

We show X to our partner who then signs the hash $h(X)$.

Later, we show X' in court with the partner's signature on $h(X') = h(X)$ and claim that he/she accepted the conditions of X'.



Additional Remark about Digital Signatures

It may seem that in the last example, we can overcome such a situation by stating (in law) that a signature that uses h is invalid whenever one comes up with a collision for h.

However, this would lead to another way of deception.

We prepare two contracts X and X', where X is the contract we actually sign and X' is another contract with the same hash $h(X') = h(X)$ but which is much more profitable to us than X.

We later deny having signed X by showing the other version X' and claiming that we wanted to sign X'

Relations Between the Security Properties

It turns out that some of the security properties for hash functions can be derived from others.

We prove that the collision-resistance is the strongest of the three

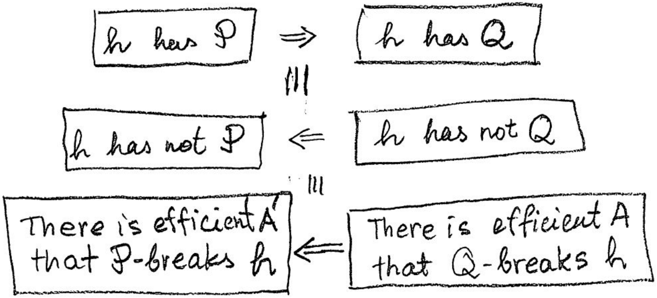
It implies both the one-wayness and the second pre-image resistance.

Relative Security Proofs by Reductions

To prove relations between security properties, we use *reductions*.

To prove that property $\mathcal{P}$ implies property $\mathcal{Q}$, we use the contraposition:

If there is an efficient A that breaks $\mathcal{Q}$, we construct an efficient A' that breaks $\mathcal{P}$.

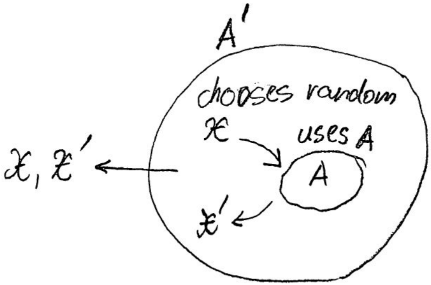

h has P
⇒
h has Q
≡
h has not P
⇐
h has not Q
≡
There is efficient A' that P-breaks h
⟸
There is efficient A that Q-breaks h

Collision Resistance Implies Second Pre-Image Resistance

Every adversary A that finds second pre-images for h can be modified to a collision finding adversary A':

◦ A' first generates a random input X and then

◦ uses A to find an $X' \neq X$ such that $h(X') = h(X)$.

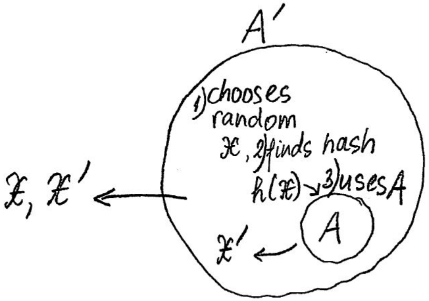


Collision Resistance Implies One-Wayness ...

... if the hash function is *sufficiently compressing*. Every adversary A that is able to invert h can be modified to a collision finding adversary A':

- ◦ A' generates a random X, computes $y = h(X)$ and
- ◦ uses A to find X' such that $h(X') = h(X)$.

If h is sufficiently compressing then $\Pr[X' = X]$ is small. Hence, A' very likely finds a collision when A is successful.



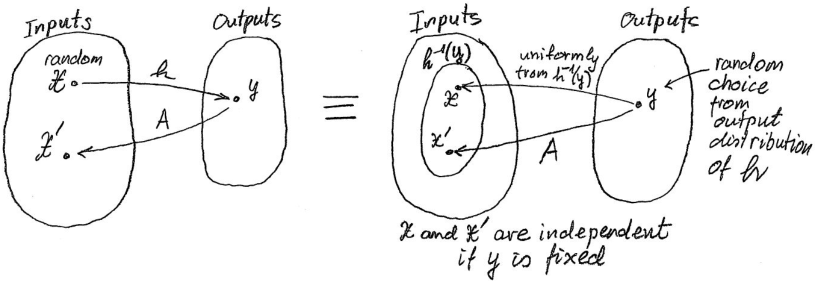
Mathematical Details

X' depends on X, and hence, X and X' are not independent variables.

However, if the output value y is fixed then X' is independent of X.

Hence, the work of A' is equivalent to the next (inefficient!) procedure:

1. Choose a uniformly random input Z and compute $y = h(Z)$
2. Choose a uniformly random X from the h-preimage of y
3. Apply A to obtain $X' = A(y)$.


Inputs
random x
h
y
Outputs
A
x'
$\equiv$
Inputs
$h^{-1}(y)$
uniformly from $h^{-1}(y)$
x
x'
A
y
Outputs
random choice from output distribution of h
x and x' are independent if y is fixed

The steps 2 and 3 are independent random choices and hance the probability $C(y)$ that a collision is produced for y is:

$$\begin{aligned} C(y) &= \mathsf{Pr}[A \text{ inverts } y] \cdot \mathsf{Pr}[X \neq X'] = \mathsf{Pr}[A \text{ inverts } y] \cdot \frac{|\,h^{-1}(y)\,| - 1}{|\,h^{-1}(y)\,|} \\ &= \mathsf{Pr}[A \text{ inverts } y] \cdot \left(1 - \frac{1}{|\,h^{-1}(y)\,|}\right) \ . \end{aligned}$$

The overall success $\delta_{A'}$ of A' is the average of this probability:

$$\begin{aligned} \delta_{A'} &= \sum_y \mathsf{Pr}[y] \cdot C(y) = \sum_y \mathsf{Pr}[y] \cdot \mathsf{Pr}[A \text{ inverts } y] \cdot \left(1 - \frac{1}{|\,h^{-1}(y)\,|}\right) \\ &= \underbrace{\sum_y \mathsf{Pr}[y] \cdot \mathsf{Pr}[A \text{ inverts } y]}_{\delta_A} - \sum_y \mathsf{Pr}[A \text{ inverts } y] \cdot \underbrace{\mathsf{Pr}[y] \cdot \frac{1}{|\,h^{-1}(y)\,|}}_{1/N} \\ &= \delta_A - \frac{1}{N} \cdot \underbrace{\sum_y \mathsf{Pr}[A \text{ inverts } y]}_{\leq M} \geq \delta_A - \frac{M}{N} \ , \end{aligned}$$

Insufficiently Compressing Hash Functions

Insufficiently compressing function can be collision-free but not one-way.

If there exists a collision free hash function h [with $(k-2)$-bit output] then there exists a function h' that is collision-free but not one-way.

We define $h'\colon \{0,1\}^k \to \{0,1\}^{k-1}$ as follows:

$$h'(x) = \begin{cases} 1\|z, & \text{if } x = 11z \\ 0\|h(x), & \text{otherwise} \end{cases}$$

◦ Finding collisions for h' is as hard as finding collisions for h.

◦ Inverting h' is relatively easy, because with probability $1/4$, a random x is of the form $11z$ and the output $1z$ of h' completely reveals the input.

Provably Secure vs Practical Hash Functions

We study two main goals when constructing a hash function.

Provably secure hash functions are more reliable but inefficient.

In practice, we use more efficient ad hoc designs, which have no guarantees against discovering more and more efficient attacks.

We present some basic ideas how to construct provably secure hash functions.

We characterize practical hash functions and summarize how secure some of them are, considering the known attacks against them.

Provably Secure Hash Functions: the Idea

The security of a hash function h can be based on a hard mathematical problem if we can prove that finding collisions for h is as hard as solving the problem.

This gives us much stronger security than just relying on complex mixing of bits.

A cryptographic hash function h has provable security against collision attacks if finding collisions is reducible to a problem P which is supposed not to be efficiently solvable. The function h is then called provably secure.

Proof by Reduction: If finding collisions if feasible by algorithm A, we could find and use an efficient algorithm S (that uses A) to solve P, which is supposed to be unsolvable. That is a contradiction. Hence, finding collisions cannot be easier than solving P.

Some Hard Mathematical Problems for Provably Secure Hashing

Provably secure hash functions are based on hard mathematical problems that are mostly related to well-known mathematical functions.
Some problems, that are assumed not to be efficiently solvable:
◦ *Discrete Logarithm Problem*: Given $a^x \mod p$, find x, where p is a large prime number.
◦ *Finding Modular Square Roots*: Given $a^2 \mod n$, find a, where n is a hard to factorize composite number.
◦ *Integer Factorization Problem*: Given $n = p \cdot q$, find p and q, where p, q are two large primes.

Example of a Provably Secure Hash Function

Modular exponent function:

$$h(x) = a^x \mod n,$$

where n is a hard to factor composite number, and a is a pre-specified base value.

A collision $a^x \equiv a^{x'} \pmod{n}$ reveals a multiple $x - x'$ of the order of a (in the multiplicative group of residues modulo n). Such information can be used to factor n assuming certain properties of a.

Such an h is inefficient because it requires 1.5 multiplications per input-bit.

Design Principles for Practical Hash Functions

Practical hash functions are combinations of bit operations which are hard to study, partly because of their "ugly" mathematical properties.

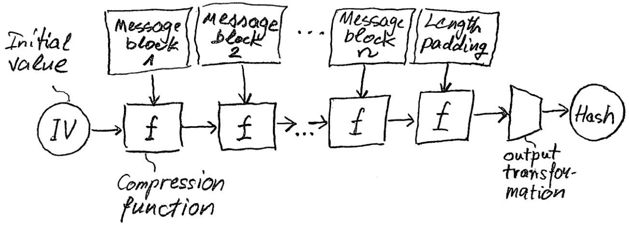
The *avalanche criterion* requires that if an input is changed slightly (for example, flipping a single bit) the output must change significantly (e.g., many output bits flip). First used by Horst Feistel (1915-1990).

The *Strict avalanche criterion (SAC)* is a generalization of the avalanche criterion. It is satisfied if, whenever a single input bit is complemented, each of the output bits changes with probability $\frac{1}{2}$. Introduced by Webster and Tavares in 1985.

The *bit independence criterion (BIC)*—two output bits should change independently when any single bit (not among these two) is inverted.

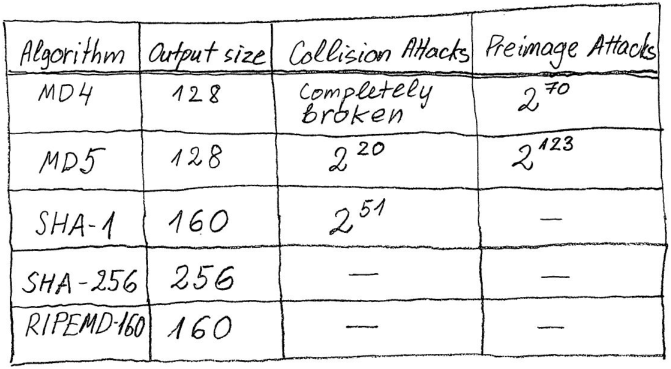
Merkle-Damgård Design

A hash function must be able to process an arbitrary-length message into a fixed-length output. This is achieved by breaking the input up into blocks.



The last block processed should also be unambiguously length padded. This construction is called the *Merkle-Damgård construction*.

Practical Hash Functions and their Known Strength



Algorithm	Output size	Collision Attacks	Preimage Attacks
MD4	128	Completely broken	2^{70}
MD5	128	2^{20}	2^{123}
SHA-1	160	2^{51}	—
SHA-256	256	—	—
RIPEMD-160	160	—	—