Alice just downloaded a large file x that was supposedly released by her friend Bob. How can she tell that Eve didn't tamper with it? Its integrity can be certified by a message authentication code like CBC-MAC. MACs require that Alice and Bob have set up a shared key in advance. This might not be a realistic assumption if, say, Bob is a big software company and Alice is one of its many users.

There is another type of cryptographic function that guarantees integrity without authenticity. It needs no secret key. After releasing x, Bob posts a short *hash* $y = h(x)$ on some public forum like his webpage. The hash y may be several hundred bits long. If Alice is in possession of the same file x, their hash values should match. What happens if her file x^* is corrupted, that is if x^* is different from x?

1 Collision-resistant hash functions

A *collision-resistant hash function* (CRH) is a function h that maps long inputs into short outputs for which finding a collision is intractable. A *collision* is any pair of distinct inputs that produce the same output:

$$x \neq x^*, \qquad \text{yet} \qquad h(x) = h(x^*).$$

Evaluating h requires no special keys. Its description is completely known to Alice, Bob, and Eve.

To make sense of this requirement it helps, as usual, to start with some non-examples.

Example 1: Is the following h that maps 3-bit inputs into 2-bit outputs

x	000	001	010	011	100	101	110	111
$h(x)$	10	00	10	11	01	01	00	10

collision-resistant? Clearly not: Inspecting the table we see that $h(000) = h(010) = 10$. This is merely one among many collisions. Collisions in h are in fact unavoidable. As h takes $2^3 = 8$ possible inputs but produces only $2^2 = 4$ possible outputs, some pair of inputs will have to collide into the same output. A collision must in fact occur somewhere among the first five inputs. This is the pigeonhole principle.

For this reason, collision resistance makes little sense for functions that produce two, or a handful of outputs. If the hash is k bits long, Eve can be certain to find a collision somewhere among the first $2^k + 1$ inputs. We will soon see that she is in fact quite likely to find a collision among some $2^{k/2}$ inputs. CRHs used in practice produce $k = 256$ or more bits of output. They are designed so that Eve is unlikely to spot a colliding pair of inputs without about $2^{k/2}$ evaluations.

Example 2: $h(x, y) = x + y$ (bitwise modulo two). For concreteness, assume x and y are 256-bits long, so that h maps 512 input bits into 256 output bits. A brute-force search for collisions is now intractable. Finding a collision is, however, not particularly challenging: As long as the sums $x + y$ and $x^* + y^*$ match, the inputs xy and x^*y^* collide. For instance, $h(00\cdots0, 00\cdots0) = h(11\cdots1, 11\cdots1) = 0$. Moderately long outputs are necessary but not sufficient for collision resistance.

Example 3: Randomness was necessary for secure encryption. Could randomization also prevent finding collisions? As an example, let's look at the randomized hash function $h(x, y) = ax + by$. The arithmetic is now modulo a large prime p, and the coefficients a and b are chosen at random. To be specific, let's take $p = 10\,000\,223$, $a = 4\,645\,725$, and $b = -4\,739\,702$.

In a CRH, Eve knows the random parameters a and b. Randomness cannot be used to hide information in a CRH construction. In this way CRHs are different from PRGs, PRFs, and PRPs. Since Eve knows a and b, she can easily find a collision. In this example,

$$h(0, 0) = 0 = a(-b) + ba = h(-b, a) = h(4\,739\,702, 4\,645\,725).$$

In general, randomizing a hash function is neither necessary nor sufficient for collision-resistance. Some candidate CRHs, like the Pedersen hash we will see shortly, do entail random choices. Others, like SHA-3, don't.

The Pedersen hash

The Pedersen hash is the function

$$h(X, Y) = g^X h^Y \bmod p. \tag{1}$$

Here p is a large safe prime number, and g and h are fixed quadratic residues modulo p.[1] The base-g discrete log of h should not be known to anyone—not to Eve, not to Alice, not to Bob. To keep a concrete example in mind, let $p = 10\,000\,223$, $g = 2$, and $h = 3\,863\,852$.

The exponents X and Y are taken modulo q with $q = (p-1)/2$. There are q^2 possible input pairs. The output is a quadratic residue modulo p, so there are q possible outputs. Some input pair must collide.

If Eve knew the discrete logarithm of h to the base g she would be able to find a collision. In our example, it happens that $h \equiv g^{2\,946\,219}$. Therefore $g^X h^Y \equiv g^{X+2\,946\,219Y}$ modulo p. Since $2\,053\,892 + 2\,946\,219 = q$, we have $h(0, 0) = h(2\,053\,892, 1) = 1$. The inputs $(0, 0)$ and $(2\,053\,892, 1)$ collide.

It turns out that this is the only strategy Eve can use to find a collision. Here is why. Suppose Eve has found two pairs of inputs (X, Y) and (X', Y') that collide, namely $g^X h^Y \equiv g^{X'} h^{Y'}$. Eve can manipulate this equation into $g^{X'-X} \equiv h^{Y-Y'}$. There are two possibilities. One is that $Y - Y'$ is zero (modulo q). Then $g^{X-X'} \equiv 1$, so $X - X'$ is also zero. The pairs (X, Y) and (X', Y') are equal, so Eve has not found a collision at all.

The other possibility is that $Y - Y'$ is nonzero (modulo q). Then it has a multiplicative inverse. Raising both sides to the $(Y - Y')^{-1}$ power, Eve gets

$$g^{(X'-X)\cdot(Y-Y')^{-1}} = h$$

Eve has thus found the discrete logarithm of h to the base g. This is supposed to be hard to find. You will work out an example in the homework.

The birthday attack

When q is on the order five million, as in this Pedersen hash example, brute-force search of a collision entails about q steps of computation. It turns out that Eve can do a lot better. She can find a collision in about 3000 steps.

The "birthday attack" applies to any hash function h. Eve evaluates h on *random* inputs x_1, x_2, and so on, until at some point she spots a collision.

Here is what happened when I ran the birthday attack on the Pedersen hash (1) with $g = 2$, $h = 3\,863\,852$, and $p = 10\,000\,223$ (using `random.seed(4108)`):

step	X	Y	$g^X h^Y$
1	3428888	164304	1189098
2	2866261	1903133	2512182
3	929068	3076717	6928499
⋮		⋮	
3312	1485857	2984950	6818020
⋮		⋮	
3830	2356472	781755	6818020

[1] In fact g and h need not be residues, but assuming they are avoids some technical difficulties.

The inputs picked at steps 3312 and 3830 collide: $g^{1485857}h^{2984950} \equiv 6818020 \equiv g^{2356472}h^{781755}$. A collision shows up after 3830 attempts. This is a big improvement over $q \approx 5$ million.

There is a reason behind the success of this attack. Take any function h that maps one of N possible inputs into one of K possible outputs. If a pair of inputs x, x' was picked at random, the probability that $h(x)$ equals $h(x')$ is at least $1/K$. (It is exactly $1/K$ if the outputs of h are equally likely, and can only be bigger if they are not.) The probability that x and x' were picked equal is only $1/N$. The probability that the inputs are distinct but the outputs collide is $1/K - 1/N$.

Now suppose Eve picked T random inputs. There are $\binom{T}{2} = T(T-1)/2$ unordered input pairs among them. The expected (average) number of input pairs that are distinct but their outputs collide is therefore

$$\binom{T}{2} \cdot \left(\frac{1}{K} - \frac{1}{N}\right).$$

As Eve tries more and more inputs, this number becomes larger. As soon as it becomes one, Eve is expected to see a colliding pair. (A more involved probabilistic analysis shows that she is in fact quite likely to run into a colliding pair at this point.)

The Pedersen hash has $N = q^2$ possible inputs and $K = q$ possible outputs. When q is around 5 million, $1/q$ dominates $1/q^2$, so $1/K - 1/N \approx 1/q$. The number $\binom{T}{2}$ is very close to $T^2/2$, give or take one. Eve is expected to see a collision as soon as $(T^2/2) \cdot (1/q)$ reaches one, or after $T = \sqrt{2q}$ trials. When q is 5 million, T is about 3162.

The birthday attack is universal. It works on any candidate CRH. If the CRH produces K possible outputs, a collision arises after Eve has tried about $\sqrt{2K}$ inputs.[2] When the output is a sequence of k bits, K equals 2^k and the birthday attack has complexity about $2^{k/2}$. If Alice and Bob want to force Eve to expend 2^{128} steps in her attack, they should work with a CRH with at least 256 output bits. This is a common parameter setting in CRH applications.

2 Domain extension

A collision-resistant hash can serve as a certificate of data integrity. For Alice to be sure that her downloaded file x^* matches Bob's original x, she calculates $h(x^*)$ and compares it to the authentic value $y = h(x)$. Eve could not have mauled x into a different x^* without finding a collision.

Files, however, can be gigabytes long. Hashes like Pedersen's only shrink the length of their input by a factor of two. A domain-extension transform turns a short-input hash into a hash for long messages. The simplest and best-known is the *Merkle–Damgård transform*.

To be concrete, suppose h hashes 768 bits of input into 256 bits of output. To hash a long message m, divide it into chunks $m_1 m_2 m_3 \cdots m_n$ of 256 bits each and output

$$h(\cdots h(h(h(m_1 m_2 m_3) m_4 m_5) m_6 m_7) \cdots m_{n-1} m_n).$$

See Figure 1. To understand why this is collision-resistant, suppose Eve found a collision, say

$$h(h(m_1 m_2 m_3) m_4 m_5) = h(h(m'_1 m'_2 m'_3) m'_4 m'_5).$$

As the outer h is collision-resistant, Eve could not have accomplished this with reasonable effort unless its inputs are equal: $h(m_1 m_2 m_3) m_4 m_5 = h(m'_1 m'_2 m'_3) m'_4 m'_5$. But that means the fourth and fifth chunks of m and m' are equal, and so are the hashes of the first three chunks. For the collision to be legitimate, m and m' must however differ somewhere in these first three chunks. Eve cannot accomplish this owing to the collision-resistance of the inner copy of h.

The same argument extends to any number of blocks. Eve cannot find a collision between two n-block messages m and m' in the Merkle-Damgård hash without finding a collision for the fixed-length hash h. If

[2] The attack is named after the birthday paradox: If $h(x)$ is the birthday of person x, then among $\sqrt{2 \cdot 365} \approx 27$ random people, two are likely to share a birthday.

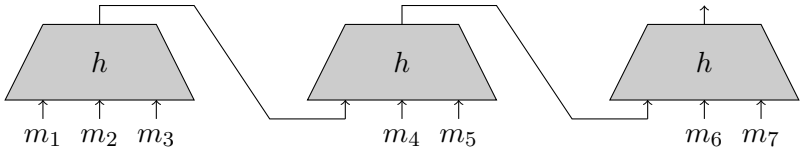


FIGURE 1: Merkle-Damgård domain extension.

m and m' are of different lengths, however, collisions are not so hard to find. Message $m = m_1m_2m_3m_4m_5$ collides with $m' = \hat{m}_3m_4m_5$, when $\hat{m}_3$ is set to $h(m_1m_2m_3)$.

This attack can be thwarted by writing down the length of m into the last block of m itself:

$$\text{encode message } m_1m_2 \dots m_n \text{ by } m_1, m_2 \dots, m_n, n.$$

For example, if the block length is 4, the message `001101110110` has length 12, or `1100` in binary. It would be encoded by the block sequence `0011`, `0111`, `0110`, `1100`.

If m and m' now have different lengths, their encodings will differ in the last block. If Eve managed to make them collide, she would have had to pick differing inputs to the last (outermost) h in Figure 1 that map to the same output. Assuming h is collision-resistant, she cannot do so with reasonable effort.

Hash-then-MAC

In Lecture 7 we had to work somewhat hard to extend a MAC so it can handle messages of arbitrary length. CRHs are an elegant solution to this problem: If h hashes arbitrarily long messages into k-bit outputs, and F is a PRF whose inputs are k-bits long, then the (keyed) function

$$F'(x) = F(h(x))$$

is a PRF that can take arbitrarily long inputs.

To understand why, suppose Eve wants to distinguish F' from a truly random function. She gets to observe the outputs $F'(x)$ for different inputs x of her choice. One strategy for her is to look for distinct inputs $x \neq x'$ that collide under h. Their outputs under F' will also collide: $F'(x') = F(h(x')) = F(h(x)) = F'(x)$. As h is a CRH, however, she cannot find them with reasonable effort. If this strategy fails, then all the evaluations of F she gets to observe must be performed on pairwise distinct inputs $h(x)$. As F is pseudorandom, all outputs $F'(x) = F(h(x))$ will be indistinguishable from independent uniformly random values to her, as required for a PRF.

If PRF F is used as a PRF MAC for fixed-length messages, its extension F' can now handle arbitrary length inputs:

Hash-then-PRF MAC:

Shared key: The key k used to set up PRF F.

Tagging: Alice attaches the tag $t = F(h(m))$ to message m of arbitrary length.

Verification: Upon receiving the pair (m, t), Bob accepts if t equals $F(h(m))$, and rejects if not.

One technical obstacle in instantiating this construction is matching the output length of h to the input length of F. For example, AES has a 128-bit block, whereas a collision-resistant hash commonly produces 256 bits. Practical MACs such as HMAC fuse F and h together into a single component that resolves these mismatches.

3 SHA-3

Evaluating the Pedersen hash over a reasonably secure modulus like `ffdhe2048` entails computationally taxing operations like modular exponentiations. There are substantially more efficient tailored alternatives like SHA-3. This is a family of hash functions that can be instantiated with different parameters. We describe SHA3-256.

SHA3-256 is based on the "sponge" construction. Its internal state contains 1600 bits. The message is divided into 1088-bit blocks. In each round, a message block is added to the top 1088 bits of state. A fixed permutation p (called Keccak-f[1600]) is then applied to all 1600 state bits. After all message blocks have been consumed, the first 256 bits of the state are output as the hash. See Figure 2.

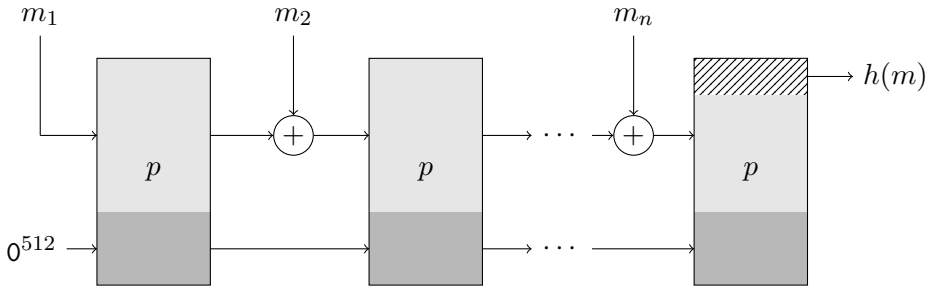


FIGURE 2: SHA3-256. The 1088-bit message blocks m_i are added to the first 1088 bits of state. The output consists of the first 256 bits of the rate from the last round.

The permutation p is designed so that no useful structure should survive the repeated iterations. Its implementation is based on the confusion-diffusion paradigm. The state is divided into a $5 \times 5 \times 64$ array of bits, to which 24 repeated rounds of diffusion, shuffling, and shifting are applied. The permutation leaves no substantial parts of the state invariant. It avoids symmetries and short cycles. Such weaknesses could help Eve find collisions.

Keccak-f[1600] applies 24 rounds of several different operations. Some spread local changes across the state, one introduces nonlinearity, and round constants break symmetries.

4 Database integrity

Alice wants to entrust her database A to Bob's cloud storage. To ensure integrity Alice remembers its hash value $y = h(A)$. At some point Alice wants to retrieve a specific item $A(x)$. How can she ensure that Bob didn't tamper with it? One possibility is for Bob to send her back the whole database so that she can verify that the hashes are consistent. This requires costly communication between them.

The *Merkle tree* is a substantially more efficient alternative. Its building block is a length-halving CRH h. The database items are associated to the leaves of a complete binary tree. The hash h is then applied recursively to every consecutive pair of inputs. The database hash is the root value y. See Figure 3.

When Alice asks to retrieve $A(x)$, Bob returns this value together with the *sibling* value at every level on the path to the root. For instance, if Alice queries $A(\texttt{100})$, Bob returns $A(\texttt{100})$ and the authentication path $A(\texttt{101}), y_{11}, y_0$. These values are marked in blue in Figure 3. Alice can then compute the intermediate values y_{10}, y_1, and y (dashed in Figure 3) herself. She verifies that the value obtained for root equals the actual hash y.

In general, to verify the hash, Alice checks that each pair of siblings hashes to the value of their parent, and the hash of the two top siblings is consistent with the database hash y at the root. If Bob tries to claim an incorrect value for $A(x)$ he must either create a collision with the true hashes somewhere along the root-to-leaf path or end up with an inconsistent hash at the root.

If the database A has N items, an authentication path contains $\log_2 N$ sibling values. In addition to the requested item, Bob therefore communicates only $\log_2 N$ hashes. When $N = 2^{30}$, for instance, this is

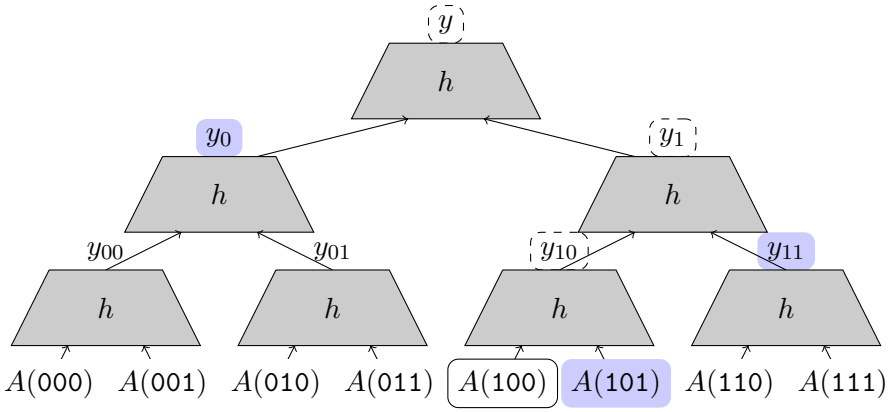


FIGURE 3: The Merkle tree. The certificate for $A(\texttt{000})$ consists of the values marked in blue.

only 30 hashes, an enormous improvement in communication.

5 Password hashing and random oracles

Passwords are a problematic yet popular method for identification. The reason is that a password is insufficiently random to prevent a brute-force attack. A 4-digit PIN can be guessed in 10,000 attempts. There are about $2^{47.6}$ possible eight-symbol alphanumeric passwords, but users do not choose them uniformly. The effective search space of real passwords is often much smaller.

Interactive identification protocols may limit the success of password-guessing attacks by imposing restrictions on the bandwidth. If several attempts in a row are unsuccessful, further ones are denied.

To enable password identification in the first place, the server must store a record of all its users' passwords. Storing passwords in plaintext is particularly risky. Compromising the record would enable the attacker immediate unauthorized access to the underlying service, and to other services as passwords are routinely reused.

One defense strategy is to encrypt the passwords under a secret key. The server must then routinely apply this secret key for decryption. If the key is ever compromised the encryption becomes useless.

Password-hashing functions can be used to make stolen password records harder to exploit. Instead of storing the actual password x, the server stores a hash $y = h(x)$ of it. To check an identification attempt, it repeats the computation. If Eve gets hold of the password record, all she gets to see are hashes of the passwords. She can still attempt a brute-force attack: Hash all possible passwords and check which ones are consistent with the record. While this attack is not completely infeasible, it requires considerable computational resources. Hash functions tailored to password hashing, like Argon2id, are somewhat costly to evaluate, adding further friction to the attack.

Can Eve improve on the brute-force attack? The collision-resistance requirement is insufficient for this purpose. (For example, the CRH definition does not forbid a short password from hashing to itself.) Cryptographic hashes such as SHA-3 and password-hashing functions such as Argon2id are designed so that their outputs appear unstructured. They are specifically designed so that $h(x)$ does not reveal information about the input x.

This random-looking behavior of common hashes is not easy to pin down mathematically. Functions like SHA-3 are, after all, deterministic. No random parameters or keys are involved. When reasoning about security it is, however, sometimes convenient to think of them as random functions called *random oracles*. There is an important distinction between random oracles and PRFs. Evaluating a random oracle does not require possession of a secret key. Its operation is known to Alice, Bob, and Eve alike.

An additional measure is often employed to further lessen the likelihood of successful password cracking. When password x is created, the server associates it to a random value called a *salt*. The password record stores the salt in plaintext together with the salt-password hash. For example, if the password is

$x =$ `luv4ever`, the record associated to x might look like

$$s = \texttt{01101001 10011101},$$
$$h(s, x) = h(\texttt{01101001 10011101 luv4ever}) = \texttt{10100100}.$$

The salt serves two purposes. First, it prevents work performed for one salt from being reused directly against accounts with different salts. Eve has to rerun the guessing attack for each salt. Second, it defeats universal precomputed lookup tables. Without the salt, Eve could have computed the hashes of all possible passwords well before attempting any exploit. She would have instant access to all the passwords as soon as the attack succeeds. A fresh random salt thwarts the effectiveness of large-scale precomputation.

6 RSA-based key exchange

In Lecture 6 we talked about the insecurities of RSA as an encryption scheme. Recall that the RSA encryption of message m under private key e is $m^e \bmod n$. Plain RSA encryption is deterministic and therefore insecure for standalone use. Nevertheless, RSA is a plausible basis for a secret key exchange mechanism.[3]

To agree on a shared key, Bob starts by sampling a modulus $n = pq$ and a key pair e, d as in regular RSA. Alice then chooses a *random* m and encrypts it as $c = m^e \bmod n$. Bob decrypts $m = c^d \bmod n$. Alice's and Bob's shared key is chosen as $k = h(m)$ for a suitable hash function h like SHA-3. The scheme is functional: Alice and Bob end up with the same key.

Is it secure? What can Eve learn about the key $k = h(m)$ by observing the value $m^e \bmod n$? If h was a truly random function, there are two possbilities. Either Eve never evaluates h on m, in which case she learns nothing about k. Otherwise, she must have learned the entirety of m from observing $m^e \bmod n$. Unless she can reliably decrypt, this is impossible. As long as RSA does not reveal the message in its entirety and h lacks any apparent structure, Eve learns nothing about the key.

7 Summary

A collision-resistant hash (CRH) is a public function (no keys involved) that compresses an arbitrarily long input x into a short digest $h(x)$. Even though colliding pairs $x' \neq x$ with $h(x') = h(x)$ abound, it takes unreasonable effort to produce one. The birthday attack finds one in about $2^{k/2}$ random attempts for a hash with k bits of output. In well-designed CRHs like SHA-3, no better attack is known.

Domain-extension methods like the Merkle-Damgård transform turn a fixed input-length hash into one for arbitrarily long inputs. The hash-and-MAC construction turns a MAC for fixed-length messages into one for arbitrary-length messages.

The cryptographic utility of hashing extends well beyond MACs. Merkle trees are used to succinctly verify the integrity of a single item in a large database. Dedicated password-hashing functions such as Argon2id are commonly used to store password verifiers on a server. Provided that hash outputs lack apparent structure, a compromised password record requires substantial effort to crack, especially if the passwords are salted with random identifiers. This lack of structure does not follow from collision-resistance alone, but is a reasonable modelling assumption for hashes like SHA-3 and Argon2. Such structureless hashes can be combined with RSA into a secure key exchange mechanism.

[3] The technical term is *key encapsulation*: The secret key is encapsulated into the ciphertext.