Application: hash functions

A hash function H is a function, which takes an input x of arbitrary length, and produces an output $H(x)$ of fixed length, say, b bit.

Example 193. (error checking) When Alice sends a long message m to Bob over a potentially noisy channel, she also sends the hash $H(m)$. Bob, who receives m' (which, he hopes is m) and h, can check whether $H(m') = h$.

Comment. This only protects against accidental errors in m (much like the check digits in credit card numbers we discussed earlier). If Eve intercepts the message $(m, H(m))$, she can just replace it with $(m', H(m'))$ so that Bob receives the message m'.

Eve's job can be made much more difficult by sending m and $H(m)$ via two different channels. For instance, in software development, it is common to post hashes of files on websites (or announce them otherwise), separately from the actual downloads. For that use case, we should use a one-way hash function (see next example).

- The hash function $H(x)$ is called **one-way** if, given y, it is computationally infeasible to compute m such that $H(m) = y$. [Also called **preimage-resistant**.]

  Similarly, a hash function is called (weakly) **collision-resistant** if, given a message m, it is difficult to find a second message m' such that $H(m) = H(m')$. [Also called **second preimage-resistant**.]

- It is called **(strongly) collision-resistant** if it is computationally infeasible to find two messages m_1, m_2 such that $H(m_1) = H(m_2)$.

  Comment. Every hash function must have many collisions. On the other hand, the above requirement says that finding even one must be exceedingly difficult.

Example 194. (error checking, cont'd) Alice wants to send a message m to Bob. She wants to make sure that nobody can tamper with the message (maliciously or otherwise). How can she achieve that?

Solution. She can use a one-way hash function H, send m to Bob, and publish (or send via some second route) $y = H(m)$. Because H is one-way, Eve cannot find a value m' such that $H(m') = y$.

Some applications of hash functions include:

- **error-checking:** send m and $H(m)$ instead of just m
- **tamper-protection:** send m and $H(m)$ via different channels (H must be one-way!)

  If H is one-way, then Eve cannot find m' such that $H(m') = H(m)$, so she cannot tamper with m without it being detected.

- **password storage:** discussed later (there are some tricky bits)
- **digital signatures**: more later
- **blockchains**: used, for instance, for cryptocurrencies such as Bitcoin

Armin Straub
straub@southalabama.edu

Some popular hash functions:

	published	output bits	comment
CRC32	1975	32	not secure but common for checksums
MD5	1992	128	common; used to be secure (now broken)
SHA-1	1995	160	common; used to be secure (collision found in 2017)
SHA-2	2001	256/512	considered secure
SHA-3	2015	arbitrary	considered secure

- CRC is short for **Cyclic Redundancy Check**. It was designed for protection against common transmission errors, not as a cryptographic hash function (for instance, CRC is a linear function).
- SHA is short for **Secure Hash Algorithm** and (like DES and AES) is a federal standard selected by NIST. SHA-2 is a family of 6 functions, including SHA-256 and SHA-512 as well as truncations of these.

  SHA-3 is not meant to replace SHA-2 but to provide a different alternative (especially following successful attacks on MD5, SHA-1 and other hash functions, NIST initiated an open competition for SHA-3 in 2007). SHA-3 is based on Keccak (like AES is based on Rijndael; Joan Daemen involved in both). Although the ouput of SHA-3 can be of arbitrary length, the number of security bits is as for SHA-2.

  https://en.wikipedia.org/wiki/NIST_hash_function_competition
- MD is short for **Message Digest**. These hash functions are due to Ron Rivest (MIT), the "R" in RSA. Collision attacks on MD5 can now produce collisions within seconds. For a practical exploit, see: https://en.wikipedia.org/wiki/Flame_(malware)

  MD6 was submitted as a candidate for SHA-3, but later withdrawn.

Constructions of hash functions

Recall that a hash function H is a function, which takes an input x of arbitrary length, and produces an output $H(x)$ of fixed length, say, b bit.

Example 195. (Merkle–Damgård construction) Similarly, a **compression function** $\tilde{H}$ takes input x of length $b+c$ bits, and produces output $\tilde{H}(x)$ of length b bits. From such a function, we can easily create a hash function H. How?

Importantly, it can be proved that, if $\tilde{H}$ is collision-resistant, then so is the hash function H.

Solution. Let x be an arbitrary input of any length. Let's write $x = x_1x_2x_3...x_n$, where each x_i is c bits (if necessary, pad the last block of x so that it can be broken into c bit pieces).
Set $h_1 = 0$ (or any other initial value), and define $h_{i+1} = \tilde{H}(h_i, x_i)$ for $i \geqslant 1$. Then, $H(x) = h_{n+1}$ (b bits).
[In $\tilde{H}(h_i, x_i)$, we mean that the b bits for h_i are concatenated with the c bits for x_i, for a total of $b+c$ bits.]
Comment. This construction is known as a Merkle–Damgård construction and is used in the design of many hash functions, including MD5 and SHA-1/2.
Careful padding. Some care needs to be applied to the padding. Just padding with zeroes would result in easy collisions (why?), which we would like to avoid. For more details:
https://en.wikipedia.org/wiki/Merkle-Damgård_construction

Example 196. Consider the compression function $\tilde{H}\colon \{3\,\text{bits}\} \to \{2\,\text{bits}\}$ defined by

x	000	001	010	011	100	101	110	111
$\tilde{H}(x)$	00	10	11	01	10	00	01	11

[This was not chosen randomly: the first output bit is the sum of the digits, and the second output bit is just the second input bit.]

(a) Find a collision of $\tilde{H}$.

(b) Let H be the hash function obtained from $\tilde{H}$ using the Merkle–Damgård construction (using initial value $h_1 = 0$). Compute $H(1101)$.

(c) Find a collision with $H(1101)$.

Solution.

(a) For instance, $\tilde{H}(001) = \tilde{H}(100)$.

(b) Here, $b = 2$ and $c = 1$, so that each x_i is 1 bit: $x_1x_2x_3x_4 = 1101$.
$h_1 = 00$
$h_2 = \tilde{H}(h_1, x_1) = \tilde{H}(001) = 10$
$h_3 = \tilde{H}(h_2, x_2) = \tilde{H}(101) = 00$
$h_4 = \tilde{H}(h_3, x_3) = \tilde{H}(000) = 00$
$h_5 = \tilde{H}(h_4, x_4) = \tilde{H}(001) = 10$
Hence, $H(1101) = h_5 = 10$.

(c) Our computation above shows that, for instance, $H(1) = 10 = H(1101)$.

The construction of good hash functions is linked to the construction of good ciphers. Below, we indicate how to use a block cipher to construct a hash function.

Why linked? The ciphertext produced by a good cipher should be indistinguishable from random bits. Similarly, the output of a cryptographic hash function should look random, because the presence of patterns would likely allow us to compute preimages or collisions.

However. The design goals for a hash function are somewhat different than for a cipher. It is therefore usually advisable to not crossbreed these constructions and, instead, to use a specially designed hash function like SHA-2 when a hash function is needed for cryptographic purposes.

First, however, a cautionary example.

Example 197. (careful!) Let E_k be encryption using a block cipher (like AES). Is the compression function $\tilde{H}$ defined by

$$\tilde{H}(x, k) = E_k(x)$$

one-way?

Solution. No, it is not one-way.

Indeed, given y, we can produce many different (x, k) such that $\tilde{H}(x, k) = y$ or, equivalently, $E_k(x) = y$. Namely, pick any k, and then choose $x = D_k(y)$.

Example 198. Let E_k be encryption using a block cipher (like AES). Then the compression function $\tilde{H}$ defined by

$$\tilde{H}(x,k) = E_k(x) \oplus x$$

is usually expected to be collision-resistant.

Let us only briefly think about whether $\tilde{H}$ might have the weaker property of being one-way (as opposed to the previous example). For that, given y, we try to find (x,k) such that $\tilde{H}(x,k) = y$ or, equivalently, $E_k(x) \oplus x = y$. This seems difficult.

Just getting a feeling. We could try to find such (x,k) with $x = 0$. In that case, we need to arrange k such that $E_k(0) = y$. For a block cipher like AES, this seems difficult. In fact, we are trying a known-plaintext attack on the cipher here: assuming that $m = 0$ and $c = y$, we are trying to determine the key k. A good cipher is designed to resist such an attack, so that this approach is infeasible.

Comment. Combined with the Merkle–Damgård construction, you can therefore use AES to construct a hash function with 128 bits output size. However, as indicated before, it is advisable to use a hash function designed specifically for the purpose of hashing.

For several other (more careful) constructions of hash functions from block ciphers, you can check out Chapter 9.4.1 in the *Handbook of Applied Cryptography* (Menezes, van Oorschot and Vanstone, 2001), freely available at: http://cacr.uwaterloo.ca/hac/

We have seen how hash functions can be constructed from block ciphers (though this is usually not advisable). Similarly, hash functions can be used to build PRGs (and hence, stream ciphers).

Example 199. A hash function $H(x)$, producing b bits of output, can be used to build a PRG as follows. Let x_0 be our b bit seed. Set $x_n = H(x_{n-1})$, for $n \geqslant 1$, and $y_n = x_n \pmod 2$. Then, the output of the PRG are the bits $y_1 y_2 y_3 \ldots$

Comment. As for the B-B-S PRG, if b is large, it might be OK to extract more than one bit from each x_n.

Comment. Technically speaking, we should extract a "hardcore bit" y_n from x_n.

Comment. It might be a little bit better to replace the simple rule $x_n = H(x_{n-1})$ with $x_n = H(x_0, x_{n-1})$. Otherwise, collisions would decrease the range during each iteration. However, if b is large, this should not be a practical issue. (Also, think about how this alleviates the issue in the next example.)

Comment. Of course, one might then use this PRG as a stream cipher (though this is probably not a great idea, since the design goals for hash functions and secure PRGs are not quite the same). Our book lists a similar construction in Section 8.7: starting with a seed $x_0 = k$, bytes x_n are created as follows $x_1 = H(x_0)$ and $x_n = L_8(H(x_0, x_{n-1}))$, where L_8 extracts the leftmost 8 bits. The output of the PRG is $x_1 x_2 x_3 \ldots$ However, can you see the flaw in this construction? (Hint: it repeats very soon!)

Example 200. Suppose, with the same setup as in the previous example, we let our PRG output $x_1 x_2 x_3 \ldots$, where each x_n is b bits. What is your verdict?

Solution. This PRG is not unpredictable (at all). After b bits have been output, x_1 is known and $x_2 = H(x_1)$ can be predicted perfectly. Likewise, for all the following output.

Comment. While completely unacceptable for cryptographic purposes, this might be a fine PRG for other purposes that do not need unpredictability.

Here is a compression function, which is provably strongly collision-resistant.

However, it is rather slow and so is not practical for hashing larger data. On the other hand, its slowness could be beneficial for applications like password hashing.

Example 201. (the discrete log hash) Let p be a large safe prime (that is, $q=(p-1)/2$ is also prime). Let g_1, g_2 be two primitive roots modulo p. Define the compression function h as:

$$h\colon \{0,1,\ldots,q^2-1\}\to\{1,2,\ldots,p-1\},\quad h(m_1+m_2q)=g_1^{m_1}g_2^{m_2} \pmod p.$$

[Note that, although not working with inputs and outputs of certain size in bits, this is a compression function, because the input space is much larger than the output space.]

Show that finding a collision of h is as difficult as determining the discrete logarithm x in $g_1^x = g_2 \pmod p$.

Solution. Suppose we have a collision: $g_1^{m_1}g_2^{m_2}\equiv g_1^{m_1'}g_2^{m_2'} \pmod p$

Hence, $g_1^{(m_1-m_1')+(m_2-m_2')x}\equiv 1 \pmod p$ or, equivalently, $(m_1-m_1')+(m_2-m_2')x\equiv 0 \pmod{p-1}$ (because g_1 is a primitive root and so has order $p-1$).

This final congruence can now be solved for x.

More precisely, if $d=\gcd(m_2-m_2', p-1)$, there are actually d solutions for x. Since we chose p to be safe, the only factors of $p-1$ are $1, 2, (p-1)/2, p-1$.

Since $|m_2-m_2'|<q$, the only possibilities are $d=1,2$ (unless $m_2=m_2'$; however, this cannot be the case since then also $m_1=m_1'$, so that we wouldn't have a collision in the first place).

Passwords

Let's say you design a system that users access using personal passwords. Somehow, you need to store the password information.

- The worst thing you can do is to actually store the passwords m.

  This is an absolutely atrocious choice, even if you take severe measures to protect (e.g. encrypt) the collection of passwords.

  Comment. Sadly, there are still systems out there doing that. An indication that this might* be happening is systems that require you to update passwords and then complain that your new password is too close to the original one. Any reasonably designed system should never learn about your actual password in the first place!

  *: On the other hand, think about how you could check for (certain kinds of) closeness of passwords without having to store the actual password.

- Better, but still terrible, is to instead store hashes $H(m)$ of the passwords m.

  Good. An attacker getting hold of the password file, only learns about the hash of a user's password. Assuming the hash function is one-way, it is infeasible for the attacker to determine the corresponding password (if the password was random!!).

  Still bad. However, passwords are (usually) not random. Hence, an attacker can go through a list of common passwords (dictionary attack), compute the hashes and compare with the hashes of users (similarly, a brute-force attack can simply go through all possible passwords).

  Even worse, it is immediately obvious if two users are using the same password (or, if the same user is using the same password for different services using the same hash function).

  Comment. So, storing password hashes is not OK unless all passwords are completely random.

- Better, a random value s is generated for each user, and then s and $H(m, s)$ are stored. The value s is referred to as **salt**.

  In other words, instead of storing the hash of the password m, we are storing the hash of the salted password, as well as the salt.

  Why? Two users using the same password would have different salt and hence different hashes stored. As a consequence, an attacker can (of course) still mount a dictionary or brute-force attack but only against a single user, not all users at once.

  Comment. Note how the concept of salt is similar to a nonce.

  Comment. To be future-proof, the hash+salt is often stored in a single field in a format like (hash-algo, salt, salted hash).

  Comment. There's also the concept of **pepper** (usually, sort of a secret salt). This provides extra security if the pepper is stored separately. [Sometimes pepper is used as a sort of small random salt, which is discarded; this only slows a brute-force attack down and should instead be addressed using the item below.]

  https://en.wikipedia.org/wiki/Pepper_(cryptography)

- Finally, we should not use the usual (fast!) hash functions like SHA-2.

  Why? One of the things that makes SHA-2 a good hash function in practice is its speed. However, that actually makes SHA-2 a poor choice in this context of password hashing. An attacker can compute billions of hashes per second, which makes a dictionary or brute-force attack very efficient.

  To make a dictonary or brute-force attack impractical, the hashing needs to be slowed down. See Example 202 for some scary numbers.

Hashing functions like SHA-2 are not secure password hashing algorithms.

Instead, options that are considered secure include: PBKDF2, bcrypt, scrypt.

Comment. For instance, WPA2 uses PBKDF2 based on SHA-1 with 4096 (fairly small!) iterations.

Comment. Only increasing the number of iterations increases computation time but not memory usage. scrypt is designed to also consume an arbitrarily specified amount of memory.

For a nice discussion about password hashing:

https://security.stackexchange.com/questions/211/how-to-securely-hash-passwords

Example 202. (the power of brute-force) In April 2021, the Bitcoin hashrate is about $150\text{E}{=}1.5 \cdot 10^{20}$ hashes per second. How long would it take to brute-force a (completely random!) 8 character password, using all 94 printable ASCII characters (excluding the space)?

Solution. There are $94^8 \approx 6.1 \cdot 10^{15}$ possible passwords. Hence, it would take about 0.00004 seconds!

Comment. Even using 10 random characters (almost no human password has that kind of entropy), there are $94^{10} \approx 5.4 \cdot 10^{19}$ possible passwords. It would take less than 0.36 seconds to go through all of these!

Comment. https://bitinfocharts.com/comparison/bitcoin-hashrate.html

Example 203. Your king's webserver contains the following code to check whether the king is accessing the server. [As is far too common, his password derives from his girlfriend's name and year of birth.]

```
def check_is_king(password):
    return password=="Ludmilla1310"
```

Obviously, anyone who might be able to see the code (including its binary version) learns about your king's password. With minimal change, how can this be fixed?

Solution. The password should be hashed. For instance, in Python, using SHA-2 (why is that actually not a good choice here?) with 256 output bits:

```
from hashlib import sha256
def check_is_king(password):
    phash = sha256(password).hexdigest()
    return phash == "9e4b4fe180e22bc6cdf01fe9711cf2558507e5c3ae1c3c1f6607a25741941c66"
```

Comment. 256 bits are 64 digits in hexadecimal.

Python comment. Of course, a real implementation would use `digest()` instead of `hexdigest()`.

[For Python 3, if operating with strings (instead of bytes), `sha256(password)` needs to be replaced with something like `sha256(password.encode('utf-8'))`.]

Why is SHA not good here? Too fast to discourage brute-force attacks.

Example 204. Suppose you don't like the idea of creating random salt.

(a) How about using the same salt for all your users?

(b) Is it a good idea to use the username as salt?

Solution.

(a) This is a terrible idea and defeats the purpose of a salt. (For instance, again an attacker can immediately see if users have the same password.)

Comment. Essentially, this is a form of pepper (if the value is kept secret, i.e. stored elsewhere).

(b) That is a reasonable idea. One reason against it is that, ideally, the salt should be unique (globally). However, this could be easily achieved by using the username combined with something identifying your service (like your hostname).

Comment. A possible practical reason against choosing the username for salt is that the username might change.

Example 205. You need to hash (salted) passwords for storage. Unfortunately, you only have SHA-2 available. What can you do?

Solution. Iterate many times! (In order to slow down the computation of the hash.) The naive way would be to simply set $h_0 = H(m)$ and $h_{n+1} = H(h_n)$. Then use as hash the value h_N for large N.

In current applications, it is typical to choose N on the order of $100,000$ or higher (depending on how long is reasonable to have your user wait each time she logs in and needs her password hashed for verification).

Application: digital signatures

Goal: Using a private key, known only to her, Alice can attach her **digital signature** s to any message m. Anyone knowing her public key can then verify that it could only have been Alice, who signed the message.

- Consequently, in contrast to usual signatures, digital signatures must depend on the message to be signed so that they cannot be simply reproduced by an adversary.
- This should sound a lot like public-key cryptography!

Cryptographically speaking, a digitally signed message (m, s) from Alice to Bob achieves:

- **integrity**: the message has not been accidentally or intentionally modified
- **authenticity**: Bob can be confident the message came from Alice

  In fact, we gain even more: not only is Bob assured that the message is from Alice, but the evidence can be verified by anyone. We have "proof" that Alice signed the message. This is referred to as **non-repudiation**. We refer to a technical not a legal term here: if you are curious about legal aspects of digital signatures, see, e.g.:

  https://security.stackexchange.com/questions/1786/how-to-achieve-non-repudiation/6108

For comparison, sending a message with its hash $(m, H(m))$ only achieves integrity.

Example 206. (authentication using digital signatures) Last time, we saw that using human generated and memorized passwords is problematic even if done "right" (Example 202). Among other things, digital signatures provide an alternative approach to authentication.

Authentication. If Alice wants to authenticate herself with a server, the server sends her a (random) message. Using her private key, Alice signs this message and sends it back to the server. The server then verifies her (digital) signature using Alice's public key.

Obvious advantage. The server (like everyone else) doesn't know Alice's secret, so it cannot be stolen from the server (of course, Alice still needs to protect her secret from it being stolen).

(RSA signatures) Let H be a collision-resistant hash function.

- Alice creates a public RSA key (N, e). Her (secret) private key is d.
- Her signature of m is $s = H(m)^d \pmod N$.
- To verify the signed message (m, s), Bob checks that $H(m) = s^e \pmod N$.

This is secure if RSA is secure against known plaintext attacks.

Example 207. We use the silly hash function $H(x) = x \pmod{10}$.
Alice's public RSA key is $(N, e) = (33, 3)$, her private key is $d = 7$.

(a) How does Alice sign the message $m = 12345$?

(b) How does Bob verify her message?

(c) Was the message $(m, s) = (314, 2)$ signed by Alice?

Solution.

(a) $H(m) = 5$. The signature therefore is $s = 5^7 \pmod{33}$ (note how computing that signature requires Alice's private key). Computing that, we find $s = 14$.

(b) Bob receives the signed message $(m, s) = (12345, 14)$.
He computes $H(m) = 5$ and then checks whether $H(m) \equiv s^3 \pmod{33}$ (for which he only needs Alice's public key). Indeed, $14^3 \equiv 5 \pmod{33}$, so the signature checked out.

(c) We compute $H(m) = 4$ and then need to check whether $H(m) \equiv s^3 \pmod{33}$. Since $2^3 \equiv 8 \not\equiv 4 \pmod{33}$, the signature does not check out. Alice didn't sign the message.

Just to make sure. What's a collision of our hash function? Why is it totally not one-way?

Example 208. Why should Alice sign the hash $H(m)$ and not the message m?

Solution. A practical reason is that signing $H(m)$ is simpler/faster. The message m could be long, in which case we would have to do something like chop it into blocks and sign each block (but then Eve could rearrange these, so we would have to do something more clever, like for block ciphers). In any case, we shouldn't just sign $m \pmod{N}$ because then Eve can just replace m with any m' as long as $m \equiv m' \pmod{N}$.

There is another issue though. Namely, Eve can do the following **no message attack**: she starts with any signature s, then computes $m = s^e \pmod{N}$. Everyone will then believe that (m, s) is a message signed by Alice. This does not work if H is a one-way function: Eve now needs to find m such that $H(m) = s^e \pmod{N}$, but she fails to find such m if H is one-way.

Example 209. Is it enough if the hash function for signing is one-way but not collision-resistant?

Solution. No, that is not enough. If there is a collision $H(m) = H(m')$, then Eve can ask Alice to sign m to get (m, s) and later replace m with m', because (m', s) is another valid signed message. (See also the comments after the discussion of birthday attacks.)

Comment. This question is of considerable practical relevance, since hash functions like MD5 and SHA-1 have been shown to not be collision-resistant (but are still considered essentially preimage-resistant, that is, one-way). In the case of MD5, this has been exploited in practice:

https://en.wikipedia.org/wiki/MD5#Collision_vulnerabilities

Example 210. Alice uses an RSA signature scheme and the (silly) hash function $H(x) = x_1 + x_2$, where $x_1 = x^{-1} \pmod{11}$ and $x_2 = x^{-1} \pmod{7}$ [with 0^{-1} interpreted as 0] to produce the signed message $(100, 13)$. Forge a second signed message.

Solution. Since we have no other information, in order to forge a signed message, we need to find another message with the same hash value as $m = 100$. From our experience with the Chinese remainder theorem, we realize that changing x by $7 \cdot 11$ does not change $H(x)$. Hence, a second signed message is $(177, 13)$.

Comment. The hash $H(m)$ for $m = 100$ is $H(100) = (100^{-1})_{\mathrm{mod}\,11} + (100^{-1})_{\mathrm{mod}\,7} = 1 + 4 = 5$.

Similar to what we did with RSA signatures, one can use ElGamal as the basis for digital signatures. A variation of that is the **DSA** (digital signature algorithm), another federal standard.

https://en.wikipedia.org/wiki/ElGamal_signature_scheme

https://en.wikipedia.org/wiki/Digital_Signature_Algorithm

Not surprisingly, the hashes mandated for DSA are from the SHA family.