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 190. 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 the failure in the next example does not occur like that.)

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 hashes 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 191. 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.

Finally, here is a compression function, which is provably strongly collision-free.

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 192. (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:

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

[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 $h(x)$ 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).

Armin Straub
straub@southalabama.edu

9.2 Passwords

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

- 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 is still systems out there doing that. An indication of this 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!

- 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. Since 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 also 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 things that makes SHA-2 such a good hash function for practical purposes is its speed. However, that good property actually makes SHA-2 a poor choice in this context of password hashing. An attacker can compute billions of SHA-2 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 193 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 iterations (actually, much larger numbers of iteration are common).

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 193. (the power of brute-force) In early 2017, the Bitcoin hashrate is about $3.5\text{P}{=}3.5 \cdot 10^{15}$ 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 less than 2 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 only take 15389 seconds, or less than 4.5 hours to go through all of these!

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

Example 194. 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 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 is_king(password):
    phash = sha256(password).hexdigest()
    return phash == "9e4b4fe180e22bc6cdf01fe9711cf2558507e5c3ae1c3c1f6607a25741941c66"
```

Comment. 256 bit is 64 digits in hexadecimal.

Comment. Of course, a real implementation should use `digest()` instead of `hexdigest()`.

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

Example 195. (homework) 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 196. (homework) 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).