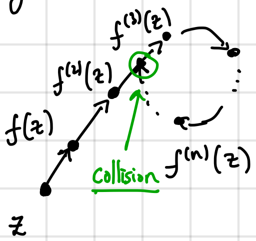
Collision finding with constant space (assuming function behaves like a random function)

↳ for concrete cryptographic hash functions (SHA-256, SHA-3), this is a common modeling heuristic (often called the random oracle model)

Suppose $f : \{0,1\}^\lambda \to \{0,1\}^\lambda$ is a random function (i.e., each output is distributed uniformly over $\{0,1\}^\lambda$)

To find $x \neq y$ such that $f(x) = f(y)$, we can do the following



sequence $z, f(z), f^{(2)}(z), \ldots$ are essentially random draws from $\{0,1\}^\lambda$ by birthday bound, after $O(\sqrt{2^\lambda})$ samples, we will have a collision and sequence will start to cycle

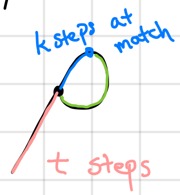
Note: after $\sqrt{2^\lambda}$ samples, probability of colliding with initial element is $\sqrt{2^\lambda}$
$= \text{negl}(\lambda)$

so there will be a collision with high probability

To find the collision: use Floyd's cycle finding algorithm:

1. Initialize $z_{slow}, z_{fast} \leftarrow z$
2. Update $z_{slow} \leftarrow f(z_{slow})$
   $z_{fast} \leftarrow f(f(z_{fast}))$

Count number of steps taken before they match (at input z^*)



Slow pointer: $t + k$ steps

fast pointer: $t + k + nc$ steps where c is cycle length and $n \in \mathbb{N}$

$$\Rightarrow t + k + nc = 2(t+k)$$
$$\Rightarrow nc = t + k$$

start at initial point z and intersection point z^*, apply f to each point iteratively until collision point is found:

Let $\tilde{z} = f^{(t)}(z)$
$$\Rightarrow z^* = f^{(t+k)}(z) = f^{(k)}(\tilde{z})$$
$$\Rightarrow f^{(t)}(z^*) = f^{(t+k)}(\tilde{z}) = f^{(nc)}(\tilde{z}) = \tilde{z}$$

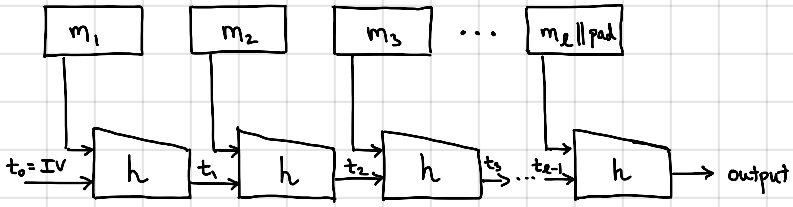
will succeed after t iterations

→ HMAC (most widely used MAC)

So how do we use hash functions to obtain a secure MAC? Will revisit *after* studying constructions of CRHFs.

Many cryptographic hash functions (e.g., MD5, SHA-1, SHA-256) follow the Merkle-Damgård paradigm: start from hash function on *short* messages and use it to build a collision-resistant hash function on a long message:

1. Split message into blocks
2. Iteratively apply *compression function* (hash function on short inputs) to message blocks



h : compression function

$t_0, \dots, t_\ell$: chaining variables

padding introduced so last block is multiple of block size

↳ *must* also include an encoding of the message length: typically of the form $100\cdots0 \| \langle s \rangle$ where $\langle s \rangle$ is a *fixed-length* binary representation of message length in blocks

Recall: $100\cdots0$ padding was used in the ANSI standard

if not enough space to include the length, then extra block is *added* (similar to CBC encryption)

Hash functions are *deterministic*, so IV is a fixed string (defined in the specification) — can be taken to be all-zeroes string, but usually set to a custom value in constructions

for SHA-256:

$\mathcal{X} = \{0,1\}^{256} = \mathcal{Y}$

Theorem. Suppose $h : \mathcal{X} \times \mathcal{Y} \to \mathcal{X}$ be a compression function. Let $H : \mathcal{Y}^{\leq \ell} \to \mathcal{X}$ be the Merkle-Damgård hash function constructed from h. Then, if h is collision-resistant, H is also collision-resistant.

Proof. Suppose we have a collision-finding algorithm A for H. We use A to build a collision-finding algorithm for h:

1. Run A to obtain a collision M and M' ($H(M) = H(M')$ and $M \neq M'$).
2. Let $M = m_1 m_2 \cdots m_u$ and $M' = m_1' m_2' \cdots m_v'$ be the blocks of M and M', respectively. Let $t_0, t_1, \dots, t_u$ and $t_1' t_2' \cdots t_v'$ be the corresponding chaining variables.
3. Since $H(M) = H(M')$, it must be the case that

$$H(M) = h(t_{u-1}, m_u) = h(t_{v-1}', m_v') = H(M')$$

If either $t_{u-1} \neq t_{v-1}'$ or $m_u \neq m_v'$, then we have a collision for h.

Otherwise, $m_u = m_v'$ and $t_{u-1} = t_{v-1}'$. Since m_u and m_v' include an encoding of the length of M and M', it must be the case that $u = v$. Now, consider the second-to-last block in the construction (with output $t_{u-1} = t_{u-1}'$):

$$t_{u-1} = h(t_{u-2}, m_{u-1}) = h(t_{u-2}', m_{u-1}') = t_{u-1}'$$

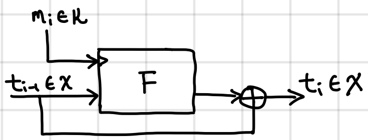
Either we have a collision or $t_{u-2} = t_{u-2}'$ and $m_{u-1} = m_{u-1}'$. Repeat down the chain until we have collision or we have concluded that $m_i = m_i'$ for all i, and so $M = M'$, which is a contradiction.

Note: Above constructing is *sequential*. Easy to adapt construction (using a *tree*) to obtain a parallelizable construction.

Sufficient now to construct a compression function.

Typical approach is to use a block cipher.

Davies-Meyer: Let $F: \mathcal{K} \times \mathcal{X} \to \mathcal{X}$ be a block cipher. The Davies-Meyer compression function $h: \mathcal{K} \times \mathcal{X} \to \mathcal{X}$ is then



$$h(k,x) := F(k,x) \oplus x$$

Many other variants also possible: $h(k,x) = F(k,x) \oplus k \oplus x$

[used in Whirlpool hash family]

Need to be careful with design!

- $h(k,x) = F(k,x)$ is not collision-resistant: $h(k,x) = h(k', F^{-1}(k', F(k,x)))$
- $h(k,x) = F(k,x) \oplus k$ is not collision-resistant: $h(k,x) = h(k', F^{-1}(k', F(k,x) \oplus k \oplus k'))$

Theorem. If we model F as an ideal block cipher (i.e., a truly random permutation for every choice of key), then Davies-Meyer is collision-resistant.

Conclusion: Block cipher + Davies-Meyer + Merkle-Damgård $\Rightarrow$ CRHFs

Examples: SHA-1: SHACAL-1 block cipher with Davies-Meyer + Merkle-Damgård ← no longer secure [first collision found in 2017!]

→ birthday attack run-time: $\sim 2^{80}$
attack ran in time $\sim 2^{64}$ (100,000× faster)

January, 2020: chosen-prefix collision in $\sim 2^{63.4}$ time!

SHA-256: SHACAL-2 block cipher with Davies-Meyer + Merkle-Damgård

- SHA-1 extensively used (e.g., git, svn, software updates, PGP/GPG signatures, certificates) → attacks show need to transition to SHA-2 or SHA-3

Why not use AES?

- Block size too small! AES outputs are 128-bits, not 256 bits (so birthday attack finds collision in 2^{64} time)
- Short keys means small number of message bits processed per iteration.
- Typically, block cipher designed to be fast when using same key to encrypt many messages
  - ↳ In Merkle-Damgård, different keys are used, so alternate design preferred (AES key schedule is expensive)

Recently: SHA-3 family of hash functions standardized (2015)

↳ Relies on different underlying structure ("sponge" function)

↳ Both SHA-2 and SHA-3 are believed to be secure (most systems use SHA-2 – typically much faster)

Back to building a secure MAC (or even better, a large-domain PRF) from a CRHF — can we do it more directly than using CRHF + small-domain MAC?

↳ Main difficulty seems to be that CRHFs are keyless but MACs are keyed

Idea: include the key as part of the hashed input

By itself, collision-resistance does not provide any "randomness" guarantees on the output

↳ For instance, if H is collision-resistant, then $H'(m) = m_0 \| \cdots \| m_{10} \| H(m)$ is also collision-resistant even though H' also leaks the first 10 bits/blocks of m

↳ Constructing a PRF/MAC from a hash function will require more than just collision resistance

- **Option 1:** Model hash function as an "ideal hash function" that behaves like a fixed truly random function (modeling heuristic called the random oracle model — will encounter later in this course)
- **Option 2:** Start with a concrete construction of a CRHF (e.g., Merkle-Damgård or the sponge construction) and reason about its properties
  - ↳ We will take this approach

Suppose H is a Merkle-Damgård hash function built from a <u>secure</u> compression function

Several ways to build a keyed function:

1. Prepend key: $F(k, m) := H(k \| m)$
   - ↳ Insecure due to structure of Merkle-Damgård: can mount an "extension attack:" given $H(k \| m)$, can compute $H(k \| m \| m')$ by extending Merkle-Damgård chain
2. Append key: $F(k,m) := H(m \| k)$
   - ↳ Similar to hash-then-MAC construction and vulnerable to same offline attack: adversary finds a collision in the Merkle-Damgård prefix and uses that to construct a forgery
     - ↳ Structure exploited in SHA-1 collision demonstration (can generate arbitrary collisions once prefix matches)

       → for SHA-1, they used PDF files
3. Envelope method: $F(k,m) := H(k \| m \| k)$
4. Two-key nest: $F((k_1, k_2), m) := H(k_2 \| H(k_1 \| m))$

} for reasonable pseudorandomness assumptions on h (e.g., both $F_1(k,m) := h(k,m)$ and $F_2(k,m) := h(m,k)$ is a PRF), both of these constructions are secure PRFs on a variable-size domain

hash-based MAC

HMAC is a PRF/MAC based on the two-key nest (though with correlated keys):

$$\text{HMAC}(k, m) := H(k_1 \| H(k_2, m))$$

where $k_1 \leftarrow k \oplus \text{ipad}$ and $k_2 \leftarrow k \oplus \text{opad}$

and ipad and opad are fixed strings (specified in the HMAC standard)

ipad: 0x36 repeated   opad: 0x5C repeated

<u>Security</u>: Since k_1 and k_2 are correlated, need to make stronger assumption on security (e.g., h remains pseudorandom under a <u>related-key</u> attack)

<u>Instantiations</u>: Typically, denoted HMAC-H where H is the hash function

e.g., HMAC-SHA1

HMAC-SHA256 — one of the most widely-used MAC on the web (used in SSL/TLS, IPsec, SSH, and more)

<u>HMAC for key-derivation</u>: Recall that under reasonable assumptions, HMAC is a secure PRF

In many protocols, we need to derive multiple keys from a single master key (e.g., derived from a password)

↳ To derive multiple independent cryptographic keys, a PRF is a natural primitive:

$$\underbrace{k_{\text{enc}}}_{\text{derived keys}} \leftarrow \text{HMAC}(\underbrace{k_{\text{master}}}_{\text{master key}}, \underbrace{\text{"enc"}}_{\text{tag (just has to be unique)}})$$

$$k_{\text{mac}} \leftarrow \text{HMAC}(k_{\text{master}}, \text{"mac"})$$

} PRF security says derived keys are computationally indistinguishable from uniform

This approach is used in TLS and IPsec to derive session keys durin session setup

↳ General paradigm is the "expand" step in hash-based key-derivation (HKDF — RFC 5869)

↳ Consists of two procedures:

- <u>Extract</u>: derive a master key from entropy source (e.g., a user password)
- <u>Expand</u>: derive sub-keys from the master key

Both steps rely on HMAC