COMP 3704 Computer Security

Christian Grothoff
christian@grothoff.org

http://grothoff.org/christian/
Design Criteria for Hash Functions

• $H : \{0, 1\}^n \rightarrow \{0, 1\}^m$ for fixed $m \in \mathbb{N}$ and any $n \in \mathbb{N}$

• Given $M$, it is easy to compute $h = H(M)$

• Given $h$, it is hard to find an $M$ such that $H(M) = h$

• Given $M$, it is hard to find an $M' \neq M$ such that $H(M) = H(M')$

• It is hard to find random messages $M$ and $M' \neq M$ such that $H(M) = H(M')$
Birthday Attack!

Probability of not finding a $n$-bit collision after generating $2^{n/2}$ messages is less than 50%:

$$p(k) = \prod_{i=0}^{k} \left( 1 - \frac{i}{2^n} \right)$$  \hspace{1cm} (1)

$$\approx \prod_{i=0}^{k} e^{-\frac{i}{2^n}}$$  \hspace{1cm} (2)

$$= e^{-\frac{(k(k-1))}{2^{n+1}}}$$  \hspace{1cm} (3)
General Construction

Difficult to define function $H : \{0, 1\}^n \rightarrow \{0, 1\}^m$. Instead use:

$$h_i = f(M_i, h_{i-1})$$

(4)

$f : \{0, 1\}^b \times \{0, 1\}^m \rightarrow \{0, 1\}^m$ for a fixed $b$ is called a compression function.
General Implementation

```c
struct hash_context;
void hash_init_context(struct hash_context * ctx);
void hash_process_bytes(const void * buf,
                        size_t len,
                        struct hash_context * ctx);
void hash_finish(struct hash_context * ctx,
                  void * result);
```
Example: MD5

Figure 1: MD5 consists of four rounds of 16 operations.
MD5 Functions

\[ F(X, Y, Z) = (X \land Y) \lor (\neg X \land Z) \] \hspace{1cm} (5)

\[ G(X, Y, Z) = (X \land Y) \lor (Y \land \neg Z) \] \hspace{1cm} (6)

\[ H(X, Y, Z) = X \oplus Y \oplus Z \] \hspace{1cm} (7)

\[ I(X, Y, Z) = Y \oplus (X \lor \neg Z) \] \hspace{1cm} (8)
Common Hash Functions

- MD5 – 128 bits
- RIPE160MD – 160 bits
- SHA1 – 160 bits
- SHA-2 – 256-512 bits
- WHIRLPOOL – 512 bits
Miyaguchi-Preneel Constructions

Example: WHIRLPOOL = Miyaguchi-Preneel + AES
Successful Attacks

• SHA-1: collisions found in 2005
• MD4, MD5 and RIPEMD-128: collisions found in 2004
⇒ Use 256 or more bits
Password Crackers

• Passwords do not usually have 128-bits of entropy

• We could actually compute hash codes for all $2^{64}$ “realistic” passwords (8 ASCII characters)

• However, we could not store all $2^{64}$ values

⇒ Precompute and use space-computation trade-off when cracking!
Precomputed Hash Chains

- Have set $P$ of realistic passwords and domain $D$ of $H$
- Define reduction function $F : D \rightarrow P$
- Pre-compute chains $X(I) = H(F(H(F(H(F(H(I)))))))$ for many $I$
- When cracking $C$, check if $C = X(I)$ or $H(F(C)) = X(I)$ or $H(F(...(H(F(C)))))) = X(I)$.  

$\Rightarrow$ reduce storage space by chain length $L$ at the expense of $O(L)$ more computation during cracking.
Problems with Hash Chains

- $F$ can cause collisions in two chains, merging the chains
- Collisions reduce effectiveness of table construction (to often less than 70%) and bound chain length

$\Rightarrow$ Tables are much too big!

$\Rightarrow$ Some chains are discarded as ineffective

$\Rightarrow$ Wasted time during construction!

$\Rightarrow$ Possibility of “false alarms”
Rainbow Tables

- Key idea: use different functions $F_i$ in chain
- Pre-compute chains $X(I) = H(F_3(H(F_2(H(F_1(H(I))))))))$

⇒ Collisions only merge chains if they also happen at same position

⇒ Can achieve 99% effectiveness

⇒ Cracking overhead increases from $O(L)$ to $O(L^2)$ for chain traversal

⇒ Cracking overhead decreases from $O(L)$ to $O(1)$ due to fewer chains
Defense: Salt!

- $hash = H(password + salt)$
- Extends length of the password
- Rainbow tables commonly only support 8 characters

$\Rightarrow$ Add 16 characters (or more) of salt
Reality

- UNIX NIS/YP/shadow: salted for a long time
- Windows NT/2000 LAN Manager: unsalted, easily cracked
Questions
Exercise

Generate a rainbow table (and password cracker) for SHA1 that can invert passwords of up to 5 characters (A-Za-z). You may link against libgcrypt or OpenSSL for hashing.