Problem Set 5
Cryptographic Hash Functions
(Summer Term 2019)

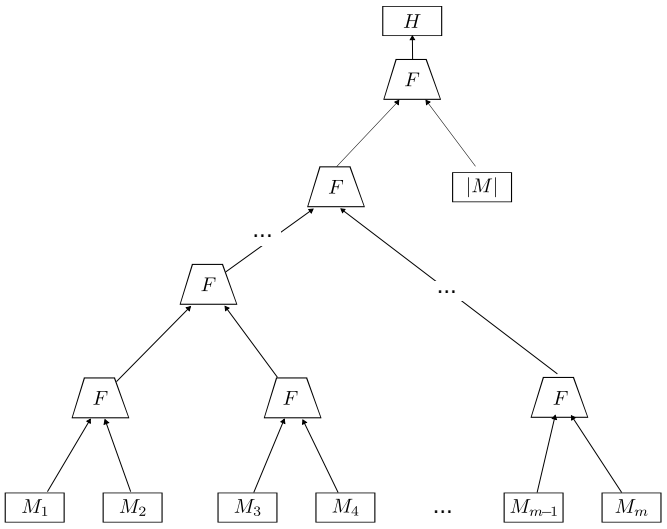
Bauhaus-Universität Weimar, Chair of Media Security
Prof. Dr. Stefan Lucks, Jannis Bossert, Nathalie Dittrich, Eik List
URL: `http://www.uni-weimar.de/de/medien/professuren/mediensicherheit/teaching/`

Due Date: 20 June 2019, 11:00 AM, via email to `jannis.bossert(at)uni-weimar.de`.

Goal of this Problem Set: Deepen the understanding of tree hashing and explore the primitives for password hashing.

Task 1 – Merkle Hash Tree (6 Credits)
Let m be an integer power of 2. Let $F : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ be a compression function that takes two n-bit values as input. To generate the hash value for a message $M = (M_1, M_2, \ldots, M_m)$ with $M_i \in \{0,1\}^n$, for $i = 1, 2, \ldots, m$, the message is processed as shown in the figure below; $|M|$ denotes the message length in bits.

Show that this construction is preimage-resistant if F is preimage-resistant. Thus, you have to show that, if you can find a preimage for the construction efficiently, then you can find a preimage for F efficiently.



Task 2 – Pebbling Game (6 Credits)

Recall Slides 159 pp. from Chapter 7 of the lecture. Let $H : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ be a PRF. Define $G \stackrel{\text{def}}{=} 2^g$ for $g \in \mathbb{N}$, and let $\pi : \mathbb{Z}_G \to \mathbb{Z}_G$ be a permutation. A graph-based password-hashing function $F(X_0)$ computes the output Y_{G-1} as following:

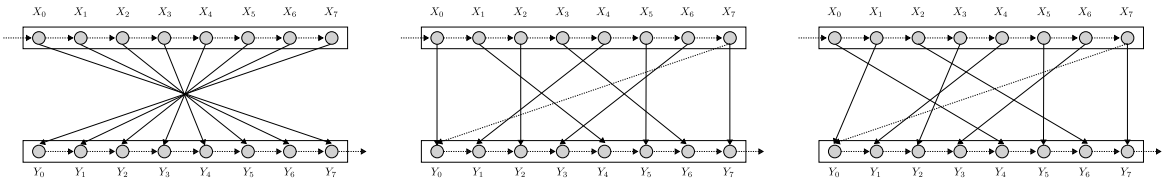
$$
\begin{aligned}
X_i &\leftarrow H\left(X_{i-1}, i\right), \qquad \text{for } 1 \le i \le G-1 \\
Y_0 &\leftarrow H\left(X_{G-1}, X_{\pi(0)}\right) \\
Y_i &\leftarrow H\left(Y_{i-1}, X_{\pi(i)}\right), \qquad \text{for } 1 \le i \le G-1 \\
F(X_0) &\text{ outputs } Y_{G-1}.
\end{aligned}
$$

The reverse permutation π_{REV} and the Gray-Reverse permutation π_{GRG2} for constant g are:

$$
\begin{aligned}
\pi_{\text{REV}}(i) &\stackrel{\text{def}}{=} (G-1-i) \\
\pi_{\text{BRG}}(i) &\stackrel{\text{def}}{=} \text{reverse}(i) \\
\pi_{\text{GRG2}}(i) &\stackrel{\text{def}}{=} \pi_{\text{BRG}}(i) \oplus \left(\pi_{\text{BRG}}\left(\bar{i}\right) \gg \lceil g/2 \rceil\right),
\end{aligned}
$$

where reverse inverts the bit order of the input, and $\bar{i}$ returns the bitwise negation of i, e.g. reverse$(1000) = 0001$, and $(\overline{1000}) = 0111$. The Bit-Reversal permutation π_{BRG} is given only for illustration.

Assume you have memory for storing S state words at a time. Let T be the number of calls to F. Choose *either* REV *or* GRG2, and provide an upper bound for the product of $S \cdot T$. You can do this theoretically, or heuristically (for some small values g of your choice) with the help of a Python script and an explanation of your results. The implementation is not graded, but must be efficiently executable to verify your results.



From left to right: Reverse, bit-reverse, and GRG2 graphs for $g = 3$.

Task 3 – `scrypt` (6 Credits)

Recall ROMix from Chapter 7 of the lecture. It consists of two loops, where the first loop fills an array $V = (V_0, \ldots, V_{n-1})$, before the second loop accesses a state word V_i in random fashion in every iteration.

Slide 154 in Chapter 7 of the lecture provides a memory-efficient sieve for ROMix. Assume, you are an adversary that managed to install a spy process on a legitimate user's computer. After a legitimate computation of ROMix with the secret password, you know the indices of the $k = (1 - 1/e) \cdot n$ cells of the n-word array that was used by the second loop – though, you do not know the values. Slide 153 then describes how you can run ROMix on your machine and reduce the time necessary for testing password candidates even if you have only a single state word of storage.

Assume you have not only a single, but a constant amount of $s < k$ state words that you can store while testing every password candidates.

a) Propose an algorithm which state words you would store, and maybe how and when you would move.

b) Analyze the average time needed to reject a wrong password candidate. You can do this theoretically, or heuristically (for some small values n, s of your choice) with the help of a Python script and explaining your results. The implementation is not graded, but must be efficiently executable to verify your results.

Task 4 – (Bonus) Fast Hashing (3 Credits)

Alice has designed an iterated password-hashing function F that executes a secure cryptographic hash function H once at the begin and once at the end, and a performant non-cryptographic function $H' : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ c times in between:

$$F(X) \stackrel{\text{def}}{=} H\left(H'\left(H'\left(\cdots H'\left(H\left(X\right), 1\right), c-1\right), c\right)\right).$$

H' takes two inputs: the previous hash value and a counter i that starts at 1 and is incremented for every call: $X_i \stackrel{\text{def}}{=} H'(X_{i-1}, i)$, for $1 \leq i \leq c$. The input value X_0 and the number of iterations c are parameters. Since Alice heard that multiplications are faster than full hashing, she proposed the following functions H':

a) $H'(X, i) \stackrel{\text{def}}{=} (X \cdot i) \bmod 2^n$,

b) $H'(X, i) \stackrel{\text{def}}{=} ((X \mid \texttt{0x3}) \cdot i) \bmod 2^n$, where $\mid$ denotes the bitwise OR.

For the functions above, *either* provide reasonable arguments why the resulting F is still a secure PRF *or* describe briefly an efficient attack.