MTAT.07.003 Cryptology II
Spring 2012 / Exercise session ?? / Example Solution

Exercise (Security of hash functions). *A standard way to protect data against malicious corruption is hashing. Namely, there are many industry standard algorithms, like MD5, SHA-256 and WHIRLPOOL, that take in a long file and output a short digest. If the digest is securely stored, then the validity of the file can be tested by recomputing the digest and comparing it to the stored value.*

1. *Formalise the functional requirements and describe the attack scenario if the original data is generated by flipping a fair coin.*
2. *Describe the attack scenario if an attacker gets to know the randomly generated original data before spoofing the file.*
3. *Describe the attack scenario if attacker can influence the content of the original file. Show that no function can be secure against such attacks. What does it mean in real life applications?*

Solution. Let us consider the hash function $h : \mathcal{M} \to \mathcal{D}$ with the message space $\mathcal{M}$ and the digest space $\mathcal{D}$. In practice, the message space often consist of all possible binary strings $\{0,1\}^*$. The first attack scenario where the data is random can be modelled as follows:

$$\begin{array}{l} \mathcal{G}_1^{\mathcal{A}} \\ \left[\begin{array}{l} x \xleftarrow{u} \mathcal{M} \\ x_* \leftarrow \mathcal{A}(h(x)) \\ \textbf{return } [h(x_*) = h(x)] \end{array}\right. \end{array}$$

if we assume that the adversary succeed when it provides any file that gives the same hash value. In other words, adversary is successful when it either reconstructs the original file or comes up with completely bogus one. When the files are long then restoring the original form the short hash is extremely unlikely and thus the first option never occurs in practice. Nevertheless, one could explicitly rule out the first option and come up with more strict security game

$$\begin{array}{l} \mathcal{G}_*^{\mathcal{A}} \\ \left[\begin{array}{l} x \xleftarrow{u} \mathcal{M} \\ x_* \leftarrow \mathcal{A}(h(x)) \\ \textbf{return } [h(x_*) = h(x) \wedge x_* \neq x] \end{array}\right. . \end{array}$$

The second attack scenario can be modelled through the following game

$$\begin{array}{l} \mathcal{G}_2^{\mathcal{A}} \\ \left[\begin{array}{l} x \xleftarrow{u} \mathcal{M} \\ x_* \leftarrow \mathcal{A}(x) \\ \textbf{return } [x_* \neq x \wedge h(x_*) = h(x)] \end{array}\right. . \end{array}$$

Note that it is not necessary to give $h(x)$ to the adversary $\mathcal{A}$, since the function h is fixed and $\mathcal{A}$ can compute $h(x)$ from x directly. Here, the implicit assumption is that h is efficiently computable function, which is one of the design goals for hash functions. If h would be difficult to compute, then the fact whether we give $h(x)$ to $\mathcal{A}$ or not could make a difference. But in this case one often assumes that $\mathcal{A}$ can have oracle access to h so that it could evaluate $h(x)$ for free. To be precise, we would then count oracle calls to h separately and $\mathcal{A}$ should still pay for preparing the argument x and for reading the result $h(x)$. The corresponding security game is following

$$\begin{array}{l} \mathcal{G}_2^{\mathcal{A}} \\ \left[\begin{array}{l} x \xleftarrow{u} \mathcal{M} \\ x_* \leftarrow \mathcal{A}^{h(\cdot)}(x) \\ \textbf{return } [x_* \neq x \wedge h(x_*) = h(x)] \end{array}\right. \end{array}$$

where the superscript $\mathcal{A}^{h(\cdot)}$ emphasises that $\mathcal{A}$ can make oracle calls to the function h.

If the adversary $\mathcal{A}$ can influence the content of the original file, then it can gear the content generation to the direction where it makes finding bogus files easier. At the limit $\mathcal{A}$ has the power to specify the file completely and thus the third attack scenario can be modelled as the following game.

$$\mathcal{G}_3^{\mathcal{A}}$$
$$\left[\begin{array}{l}(x_1, x_2) \leftarrow \mathcal{A} \\ \textbf{return } [x_1 \neq x_2 \wedge h(x_1) = h(x_2)] \end{array}\right. .$$

Since the hash function h is fixed in the game G_3, there exists an *extremely efficient* adversary $\mathcal{A}$ which just prints two constants x_1 and x_2 such that $h(x_1) = h(x_2)$. Therefore, no function can be secure for such game. However, in reality, we know such adversary exists, but we have no idea how to find such adversary, so there formalisation is somehow incorrect. One possibility is to choose hash function form a large function family $\mathcal{H}$:

$$\mathcal{G}_*^{\mathcal{A}}$$
$$\left[\begin{array}{l}h \leftarrow \mathcal{H} \\ (x_1, x_2) \leftarrow \mathcal{A}(h) \\ \textbf{return } [x_1 \neq x_2 \wedge h(x_1) = h(x_2)] \end{array}\right. .$$

In this case, it is important that $\mathcal{A}$ gets a description of h, briefly denoted by $\mathcal{A}(h)$. As there are too many candidate functions, we cannot hardwire a collision table consisting of triples (h, x_1, x_2) into $\mathcal{A}$ and thus security against collisions is not ruled out by the design of the game. Also, note that the function family might assign different probabilities to different functions and thus h is chosen according to the distribution of $\mathcal{H}$, emphasised by $h \leftarrow \mathcal{H}$, and not uniformly, which is denoted by $h \xleftarrow{u} \mathcal{H}$ in our syntax.