# Cryptography: More Primitives

## 1 Digital Signatures

In the lecture, we briefly mentioned digital signatures as an application of hashing. Now we introduce digital signatures as a stand-alone primitive.

A digital signature scheme has a pair of functions Sign and Verify

$$\sigma = \mathsf{Sign}(sk_A, m) \tag{1}$$
$$b = \mathsf{Verify}(pk_A, m, \sigma) \tag{2}$$

In the first step, Alice signs a message $m$ she wants to send using her secret key $sk_A$, producing a digital signature $\sigma$. Then we want anyone who receives the pair $m, \sigma$ from Alice to be able to verify if $m$ indeed originates from Alice, using Alice's public key $pk_A$.

### 1.1 What properties do we want from digital signatures?

- Correctness: if $\sigma$ is a signature of $m$, then $b$ should be true; otherwise, $b$ should be false.
- Unforgeability: an adversary who has seen some valid message-signature pairs $(m_1, \sigma_1)$, $(m_2, \sigma_2), \cdots, (m_t, \sigma_t)$ should not be able to come up with a forgery for a new message, i.e., $(m^*, \sigma^*)$ where $m^* \neq m_i$ for $i \in [1..t]$ such that $\mathsf{Verify}(pk_A, m^*, \sigma^*)$ outputs true.

Notice that there is no (known) way to prevent an adversary from passing along a valid message-signature pair it has seen before. That's why we define the unforgeability requirement in the above way.

### 1.2 First attempt

In the early years, researchers proposed making digital signatures the inverse of public-key encryption. Sign is decryption, and Verify is encryption and compare. Use RSA as an example, the basic RSA signature scheme works as follows:

$$\mathsf{Sign} : \sigma = m^d \mod n$$
$$\mathsf{Verify} : b = \sigma^e \stackrel{?}{\equiv} m \mod n$$

where $d$ is the RSA secret key and $(n, e)$ is the RSA public key.

The hope behind this design is that: (1) think of $m$ as a ciphertext, if we decrypt it and then encrypt it again, we will get back $m$, (2) to forge a signature for $m^*$, an adversary needs to decrypt $m^*$, which he cannot do without the secret key.

(Can challenge the class to break it.) One problem lies in the malleability (multiplicative homomorphism) of RSA. If an adversary has seen two valid message-signature pairs $(m_1, \sigma_1)$, $(m_2, \sigma_2)$, it can easily come up with a forgery $(m_1 m_2, \sigma_1 \sigma_2)$: $\sigma_1 \sigma_2 = m_1^d m_2^d \equiv (m_1 m_2)^d \mod n$ is a valid signature of message $m_1 m_2$. Another attack: choose a $\sigma^*$, compute $m^* = \sigma^{*e} \mod n$, then $\sigma^*$ is a valid signature of $m^*$.

### 1.3 Second attempt

How about just use the hash-then-sign scheme mentioned in the lecture, i.e., replace $m$ with $h(m)$ in (1) and (2)? A 'good' hash function does seem to fix the above problems. It should not be multiplicative, preventing the first attack; it should be one way, preventing the second attack (if $h(m^*) = \sigma^{*e}$, one cannot figure out $m^*$). Of course, we have also seen that $h$ needs to be collision-resistant; otherwise the composed signature scheme is clearly forgeable.

Indeed, this is much better. In fact, there are digital signature standards similar to this approach, signing the hash of the message with some padding. For example, ANSI X9.31 standard signs 6bbb $\cdots$ bbba || $h(m)$ || 33cc. PKCS #1 v1.5 standard signs 0001ff $\cdots$ ff00 || length_of($h$) || $h(m)$.

But how do we know there are no other attacks? Well, we do not. This is a flaw of these designs: they build on ad-hoc security. We do not know how to break them, but we do not know how to prove their security, either. This is where modern cryptography departs from this approach. It tries to reduce the security of a scheme to one or several simple and easy-to-describe assumptions. Digital signatures in modern cryptography, however, are out of the scope of 6.046.

## 2 Message Authentication Codes

So far, we have seen three of the most common cryptographic primitives: public-key encryption, private-key encryption and digital signatures. If we categorize them as 'confidentiality' vs. 'integrity', 'asymmetric' vs. 'symmetric', we have

| | symmetric | asymmetric |
|---|---|---|
| for confidentiality | private-key encryption | public-key encryption |
| for integrity | *message authentication codes* | digital signatures |

The remaining one, the "symmetric integrity scheme" is Message Authentication Codes (MAC). Its definition and requirements are similar to digital signatures', except that there is only one key $k$.

$$\sigma = \mathsf{MAC}(k, m) \tag{3}$$

Verification simply checks if $\sigma \stackrel{?}{=} \mathsf{MAC}(k, m)$. Correctness and unforgeability are defined similarly as for digital signatures.

Q: Is a hash not a MAC (What's the relation/difference between a hash and a MAC)?

A: No, because a hash is a public function that everyone can compute. So, it's trivial to forge. But it's close. A popular way to construct a MAC is to use a keyed hash. The simplest one (and the standard today) is to use SHA-3 hash on the message with the key prepended, $\sigma = \mathsf{SHA3}(k \mid\mid m)$, though not every secure hash function can be turned into a MAC by simply prepending the key.

# 3 Merkle Tree

Now consider another scenario where we want integrity. Alice stores a bunch of files on a server (e.g., Google Drive). How does Alice verify her files are not modified? In this scenario, we want freshness: when Alice accesses a file, it should be the latest version of that file (what Alice wrote there last time).

In this case, MAC and digital signatures do not help. An attacker (e.g., a malicious server) can always return Alice an older version of that file with its corresponding MAC/signature (recall their unforgeability definitions). For this application, we need a new primitive.

A naive method (but maybe reasonable in practice) is to let Alice store a hash of every file (the most recent version) locally, on her own computer. Assuming data on Alice's own computer cannot be modified by an adversary, Alice can detect any modification to the files. Here we say Alice's own computer is *trusted storage* or *local storage*. If Alice has $n$ files, she needs to store $n$ hashes locally. This requires $O(n)$ trusted storage, which is arguably not ideal. Alternatively, Alice could concatenate all her files together, and produce a single hash. Then the trusted storage requirement becomes $O(1)$, but time complexity is $O(n)$: verifying one file requires downloading all the files, and updating a single file requires recomputing the hash, which involves all files.

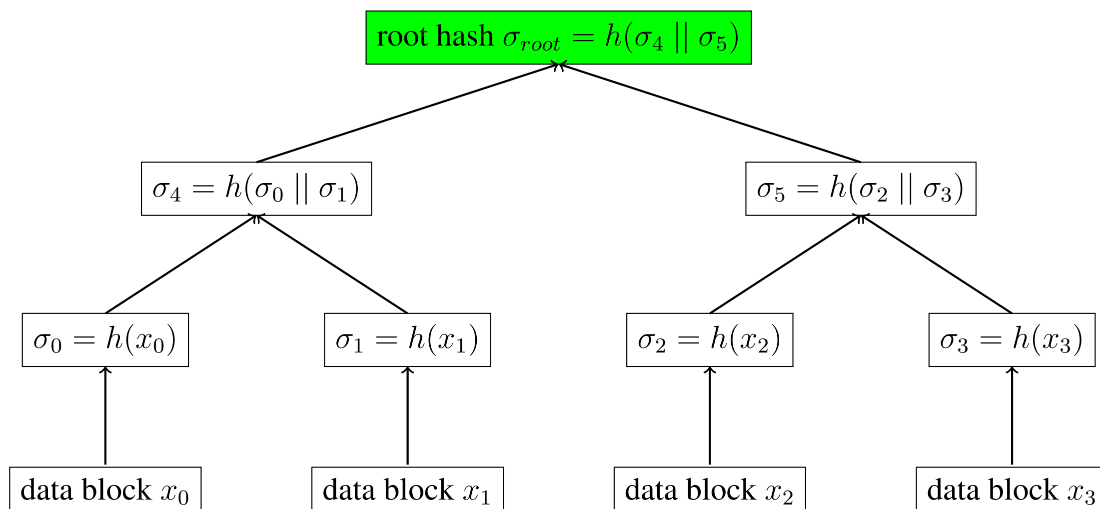


**Figure 1**: A Merkle tree with 4 leaf nodes.

Merkle tree or hash tree [Merkle, 1980] is a solution to this problem. In a Merkle tree, all data blocks are first hashed at the leaf nodes. Each intermediate node stores a hash of its child hashes

concatenated, until we get a root hash, which is stored in trusted storage. All the intermediate hashes and all data blocks are stored in untrusted storage and can be modified by an adversary. Therefore, a Merkle tree needs $O(1)$ trusted storage. It has $O(\lg n)$ time complexity for verification and updating: verifying/updating a block checks/updates a path in the hash tree.

The hash tree is collision resistant if the underlying hash function is collision resistant. Intuition: if $x_i$ is modified, $\sigma_i$ will change; otherwise a collision is found. If $\sigma_i$ changes, its parent will change; otherwise a collision is found ... Repeat the same argument all the way to the root: either the root hash changes or a collision is found.

# 4 Review Knapsack Cryptosystems

For this part, review the Merkle-Hellman cryptosystem (super-increasing knapsack as the private key, general knapsack as the public key), and go over the example on page 9 of the lecture note.

## 4.1 Why is it broken?

This subsection gives intuition, not a rigorous proof. Let the trapdoor knapsack problem be $sk = \{u_1, u_2, \cdots, u_n\}$, and the transformed knapsack problem be $pk = \{w_1, w_2, \cdots, w_n\}$, where $w_i = Nu_i \mod M$.

First we show that we need $M > \sum u_i$. Let $m_1 m_2 \cdots m_n$ be the bit decomposition of an input message $m$. Encryption of $m$ gives $S = \sum m_i w_i$. We will then transform the sum $S$ back to the super-increasing knapsack problem for decryption:

$$\begin{aligned} T &= N^{-1} S \mod M \\ &= N^{-1} \sum m_i w_i \mod M \\ &= N^{-1} \sum m_i N u_i \mod M \\ &= \sum m_i u_i \mod M \end{aligned}$$

If and only if $M > \sum u_i$, solving the trapdoor knapsack problem always gives the same result as solving the public-key knapsack.

Next, we also need each $u_i$ to have a reasonably large range for us to choose from. If the range is too small, an attacker can brute force all the possible choices of each $u_i$. A reasonable strategy is to select $u_1$ from $[1, t]$, $u_2$ from $[t+1, 2t]$, $u_3$ from $[3t+1, 4t]$, $u_i$ from $[(2^{i-1}-1)t+1, 2^{i-1}t]$ ... Then, $M$ should be larger than $2^{n-1}t$. Note that the largest element among $w_i$'s will be close to $M$. The density is then,

$$d = \frac{n}{\max(\log_2 w_i)} \approx \frac{n}{\log_2 M} \approx \frac{n}{n - 1 + \log_2 t}$$

Now we have a dilemma. If $t$ is small, the range for each $u_i$ is small and may be brute-forced. For reference, there is another class of attack [Shamir, 1984] that can find a trapdoor knapsack

that is not necessarily the same one as $sk$ in polynomial time with high probability when $t$ is small. If $t$ is large, the density is low and the attack on low-density knapsack problems [Lagarias and Odlyzko, 1985] will succeed. How low a density is considered low? Lagarias and Odlyzko conjectured their attack had a high success rate if $d < 0.645$, and this threshold had been improved. The original MH scheme [Merkle and Hellman, 1978] proposed setting $t = 2^n$, making the density roughly 0.5, and is therefore vulnerable to low-density attacks.

While most of the knapsack-based cryptosystems have been broken, there are a few that have so far resisted all attacks. They are still of interest due to the high encryption/decryption speed, as well as our desire of a variaty of cryptosystems (if one is broken, we have another one). But the original motivation of knapsack-based cryptosystems turned out to be unsuccessful: it is unlikely to base cryptography on NP-completeness. NP-complete problems may be hard only in the worst-case, while cryptography needs average-case hardness.[1]

# A Clarification for the Lecture

Deciding if a integer $N$ is prime or composite is in P, thanks to the AKS primality algorithm [Agrawal, Kayal and Saxena, 2004] that checks if a number is a prime in polynomial time. What's unknown if in NPC or not is, deciding if $N$ is composite **with a factor within a range**. (The lecture note has that extra condition in it but Prof. Devadas did not write it on the board.)

[1]The best summary on this topic may be Impagliazzo's "five worlds" in his "A personal view of average-case complexity", 1995.

MIT OpenCourseWare
http://ocw.mit.edu

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.