1 Recap and Looking Ahead

Previously we demonstrated a signature scheme that was derived from any one-way function, plus a collision-resistant hash family; this latter assumption can be replaced with a slightly weaker primitive that also follows from one-way functions. This means that even though they are a "public-key" object, signature schemes exist in minicrypt.

Unfortunately our scheme was quite inefficient. Now we'll consider an alternative signature scheme that is efficient and simpler to analyse. To do this, we'll rely on stronger assumptions: we will rest our scheme on trapdoor permutations instead of just OWFs, and the security analysis will be in an *idealized* setting called the Random Oracle Model.

2 Random Oracle Model

A random oracle is a "public truly random function:"

- Anyone can query it as a "black box."
- It gives a *consistent*, *uniform*, and *independent* random output for each distinct input.

Is a random oracle substantially different from a PRF? On the surface, no: a PRF can be queried, and gives consistent outputs that appear uniformly random and independent. However, the point of having a random oracle is for it to be *public*. If the seed of a PRF is revealed, then it loses all its security guarantees.

2.1 Methodology

Bellare and Rogaway introduced a methodology for using random oracles in cryptography:

1. Design a scheme in the RO model: all algorithms (of the scheme, and the adversary) have "black-box" access to the oracle.
2. Analyze/prove security (in the RO model) by giving a security reduction.
3. *Instantiate* the oracle with a "sufficiently crazy" public hash function (such as SHA-1), and hope that this does not introduce any security vulnerabilities in the scheme.

2.2 Advantages and Caveats

Why does having a random oracle help? In the RO model, security is often easier to prove, even for very simple and efficient schemes. Generally speaking, this is because the simulator/security reduction has some extra flexibility: it gets to provide the view of the oracle to the adversary. This not only allows the simulator to know what queries the adversary makes, but also to 'control' (or 'program') the oracle outputs that the adversary sees. We shall see both of these properties used in the security proof below.

However, there is (sometimes heated) dispute over the value of a security proof in the random oracle model. The difficulty lies with the *instantiation* step. Because its code is fixed and public, a given hash function is *not* a black box, nor does it give uniformly random and independent outputs. In fact, Canetti, Goldreich, and Halevi showed that there exists (under standard assumptions) a signature scheme that is secure in the random oracle model, but which becomes *trivially breakable* with *any* public instantiation of the oracle! In this sense, the random oracle methodology does *not* yield secure schemes in general. However,

the problematic CGH signature scheme is very contrived, and would never be proposed for practice. Most signature schemes used in the real world are much more "natural," and don't seem to have obvious (or even non-obvious) flaws when instantiated with strong hash functions.

3 A Signature Scheme in the RO Model

Recall the definition of a trapdoor permutation family: it is a set $\{f_s \colon D_s \to D_s\}$ of permutations where f_s is easy to invert with knowledge of the trapdoor (which is generated along with the index s), and otherwise hard to invert on uniformly random values $y \in D_s$. The RSA and Rabin function families are plausible trapdoor permutation candidates.

Previously, we attempted to design a secure signature scheme based on trapdoor permutations. Our old scheme was simple: the public verification key was a function f_s, the secret signing key was its trapdoor (also produced by the function sampler S), and the signature for a message $m \in D_s$ was $f_s^{-1}(m)$. Unfortunately, this succumbs to a trivial attack: first pick some signature $\sigma \in D_s$, then set $m = f_s(\sigma)$. Then (m, σ) is clearly is a valid message/signature pair.

Fortunately, we can resurrect this scheme in the random oracle model, with a small change: we first *hash* the message using the oracle, then invert the hash. The following scheme SIG, often called *full-domain hash*, uses a trapdoor permutation family $\{f_s\}$ (where the domain of every f_s is $\{0,1\}^n$, for convenience) and a function $H : \{0,1\}^* \to \{0,1\}^n$, modelled as a random oracle.

- Gen^H: choose $(s,t) \leftarrow \mathsf{S}$ and output $vk = s$, $sk = t$.
- $\mathsf{Sign}_t^H(m \in \{0,1\}^*)$: output $\sigma = f_s^{-1}(H(m))$.
- $\mathsf{Ver}_s^H(m, \sigma \in \{0,1\}^n)$: accept if $f_s(\sigma) = H(m)$, otherwise reject.

Despite the similarly with our previous signature scheme, we cannot apply the same kind of attack: after choosing $\sigma \in \{0,1\}^n$, we would then have to find a message m such that $H(m) = f_s(\sigma)$. Because H is a random oracle, this could only be done by brute-force search, which would take about 2^n attempts.

Of course, just because *this* attack is thwarted does not mean that our scheme is secure. Happily, we can prove security in the idealized model.

Theorem 3.1. *The SIG scheme described above is strongly unforgeable under chosen-message attack (in the RO model), assuming that $\{f_s\}$ is a TDP family.*

Proof. First note that for any oracle H, every message m has exactly one signature, so to proving strong unforgeability it suffices to prove standard unforgeability.

We want to design a simulator algorithm $\mathcal{S}(s, y)$ that uses a hypothetical forger $\mathcal{F}$ to find $x = f_s^{-1}(y)$, given s generated by S and uniformly random $y \leftarrow \{0,1\}^n$. (Note that $y = f_s(x) \in \{0,1\}^n$ is uniformly random because $x \leftarrow \{0,1\}^n$ is, and because f_s is a permutation.) The basic idea is that $\mathcal{S}$ will answer the random oracle queries ("program the oracle") so that it knows the preimages (under f_s) of all but one of them, which it answers as its challenge value y. This allows the simulator to answer signing queries for all but one message, which it hopes is the message that $\mathcal{F}$ will forge, in which case the forged signature is exactly $f_s^{-1}(y)$, as desired.

We now proceed in more detail. Without loss of generality, we may make some simplifying assumptions about $\mathcal{F}$. In particular, we assume that:

- *$\mathcal{F}$ makes at most $q = \mathrm{poly}(n)$ queries to H.* Because $\mathcal{F}$ runs in polynomial time, it can't make more than $\mathrm{poly}(n)$ queries no matter what its execution path is.
- *$\mathcal{F}$ always outputs some (possibly invalid) forgery (m^*, σ^*).* We can always "wrap" $\mathcal{F}$ in a program that outputs an arbitrary "junk" forgery if $\mathcal{F}$ fails to.
- *$\mathcal{F}$ always queries $H(m)$ (or $H(m^*)$) before querying $\mathsf{Sign}(m)$ (or outputting its forgery (m^*, σ^*), respectively).* We can wrap $\mathcal{F}$ in another algorithm that inserts the needed queries, and ignores the results.
- *$\mathcal{F}$ never repeats any query to H.* We can always wrap $\mathcal{F}$ in another algorithm that watches all of $\mathcal{F}$'s queries, and store all queries/response pairs in a table. Before forwarding each query to the random oracle, it first checks its table and returns the answer if it is already known. (We note that we can enforce this condition on either the forger's end, or in the simulator's responses, with the same effect.)

The forger $\mathcal{F}$ expects to perform a chosen-message attack against the scheme, which means that $\mathcal{S}$ must provide it with access to:

1. a verification key vk;
2. a signing oracle $\mathsf{Sign}_{sk}(\cdot)$;
3. a random oracle H.

Our simulator $\mathcal{S}(s, y)$ works as follows: it first chooses $i^* \leftarrow [q]$ uniformly at random; this is a "guess" as to which query to H will correspond to the message in the eventual forgery. It then invokes $\mathcal{F}$ on $vk = s$, and simulates the oracles H and $\mathsf{Sign}(\cdot)$ as follows:

Algorithm 1 Simulation of $H(\cdot)$.

On $\mathcal{F}$'s ith query $m_i \in \{0,1\}^*$,
if $i = i^*$ **then**
  Store $(\perp, y, m_{i^*})$ in an internal table.
  return $y_{i^*} = y$ to $\mathcal{F}$
else
  Choose $x_i \leftarrow \{0,1\}^n$
  Let $y_i = f_s(x_i)$
  Store (x_i, y_i, m_i) in the internal table.
  return y_i to $\mathcal{F}$

Algorithm 2 Simulation of $\mathsf{Sign}(\cdot)$.

On a query $m \in \{0,1\}^*$, look up (x, y, m) in the internal table. {Such an entry exists because $\mathcal{F}$ always queries $H(m)$ before querying $\mathsf{Sign}(m)$.}
if $x \neq \perp$ **then**
  return x to $\mathcal{F}$
else
  Abort the simulation and fail.

Finally, when $\mathcal{F}$ outputs its purported forgery (m^*, σ^*), check that $m^* = m_{i^*}$. If so, $\mathcal{S}$ outputs σ^*; otherwise, it fails.

We shall analyze the simulation in the claims below. By Lemma 3.2 below, we conclude that the existence of a forger of non-negligible advantage implies that we have a non-negligible advantage in inverting the trapdoor permutation, as desired. □

Question 1. Justify that the $H(\cdot)$ constructed by $\mathcal{S}$ actually behaves as a random oracle from the point of view of $\mathcal{F}$.

3.1 Analysis

Lemma 3.2. *For the above simulation strategy,*

$$\mathbf{Adv}_{TDP}(\mathcal{S}) = \frac{1}{q} \cdot \mathbf{Adv}_{SIG}(\mathcal{F}).$$

This fact follows from two claims:

Claim 3.3. *Conditioned on not failing, $\mathcal{S}(s, y)$ simulates the chosen-message attack perfectly, and hence outputs $f_s^{-1}(y)$ with probability $\mathbf{Adv}_{SIG}(\mathcal{F})$.*

Proof. We first note that $vk = s$ is distributed precisely as in SIG, because $\mathcal{S}$ is playing the TDP inversion game. As you saw in Question 1, $\mathcal{S}$ simulates H perfectly. Lastly, because $\mathcal{S}$ did not fail, it answered every signing query with its unique valid signature.

Now $\mathcal{S}$ does not fail exactly when m^* is $\mathcal{F}$'s i^*th query to H, where (m^*, σ^*) is $\mathcal{F}$'s output forgery. Observe that in this case, σ^* is a valid signature if and only if $\sigma^* = f_{s^{-1}}(y)$. Because $\mathcal{S}$ simulates the chosen-message attack perfectly, $\mathcal{F}$ outputs such σ^* with probability $\mathbf{Adv}_{\mathrm{SIG}}(\mathcal{F})$, as desired. □

Claim 3.4. *The probability of $\mathcal{S}$ not failing equals $1/q$.*

Proof. By assumption, $\mathcal{F}$ must query H on m^* before outputting its forgery (m^*, σ^*). By construction of $\mathcal{S}$, it does not fail exactly when m^* is the i^*th query to H. Moreover, we claim that m^* is the i^*th query to H with probability exactly $1/q$. This is because we can imagine an alternative experiment in which *every* query to Sign is answered correctly, and i^* is chosen uniformly; in such an experiment, i^* is independent of $\mathcal{F}$'s view an hence m^* is the i^*th query with probability $1/q$. □

4 Variants

Notice the factor of $\frac{1}{q}$ in the advantage of the simulator. The reduction (from breaking the TDP to breaking the signature scheme) is "loose" in the sense that the simulator's probability of inverting is much less than the forger's probability of forging. For concrete security, this might mean that we need to use a large security parameter for our TDP in order to achieve a small enough forging probability for our signature scheme. Also notice that q is the number of *hash* queries, not signing queries. In practice, the number of times an attacker can compute the hash function is probably much larger than the number of signatures to which it has access.

People have tried to improve the "quality" of the reduction in a number of ways. One notable way, due to Katz and Wang, is to use "claw-free" pairs of TDPs; these are analogous to collision-resistant hash functions. In a claw-free family, the function sampler generates two functions f_{s_0} and f_{s_1}, and the security

property is that it should be hard to find a pair of (not necessarily distinct) inputs $x_0, x_1 \in \{0,1\}^n$ such that $f_{s_0}(x_0) = f_{s_1}(x_1)$. As usual, there are trapdoors that make it easy to invert the functions. (The connection to collision resistance is the following: define $f(b, x) = f_{s_b}(x)$. Then f maps $n+1$ bits to n bits, and a collision in f immediately gives us a claw in f_{s_0}, f_{s_1}; the values of b must be different because the functions f_{s_b} are permutations.)

We can tweak the signature scheme so that the public verification key consists of two functions f_{s_0}, f_{s_1}, and the signing algorithm on message m chooses one of the functions at random and inverts it on $H(m)$.[1] In the security proof, the simulator is attempting to find a claw in f_{s_0}, f_{s_1}. When simulating the random oracle on each new message m_i, it *always* chooses a fresh x_i and applies one of the two functions (chosen at random) to compute and return y_i. If the forger later queries the signing oracle on m_i, the simulator can always reply with a properly distributed signature. (But it only knows *one* valid signature, which is why the real scheme must also give away only one distinct signature for each message.) Finally, when the forger outputs a forgery, half of the time it will end up forging using the same function that the simulator used (so we fail), and the other half of the time it finds a preimage under the other function, yielding a claw. So the simulator's advantage is *half* that of the forger, which is a drastic improvement!

[1] If the forger queries the signing oracle on the same message more than once, the signer should invert the same function every time; this can be enforced by keeping state or using a PRF. Without this condition, it is easy to see that the signer will eventually give away a claw!

Answers

Question 1. Justify that the $H(\cdot)$ constructed by $\mathcal{S}$ actually behaves as a random oracle from the point of view of $\mathcal{F}$.

Answer. First, clearly H returns the same result when queried on the same input thanks to the internal table. When $i \neq i^*$, the string returned by H is uniformly random as expected. When $i = i^*$, y is returned. Recall that in the trapdoor permutation game, $y = f_s(x)$ for an x that is chosen uniformly at random. Since x is uniformly random and f_s is a permutation, y is uniformly random as well.