Signatures

Noah Stephens-Davidowitz

September 30, 2025

1 Public keys!

We have now seen how Alice and Bob can communicate both authentically and securely. However, we have mostly glossed over one gigantic assumption that we made: that Alice and Bob somehow can manage to share a secret key. For both SKE and MACs to be useful, Alice and Bob need to somehow first exchange some bit string k, which they must keep secret from Eve. Of course, this is still quite useful in many situations because once they have figured out how to agree on a shared secret key k (e.g., by meeting in person), Alice and Bob can use this secret key to communicate securely as much as they like on any channel (e.g., over the internet while Alice is traveling).

At first, it might seem like a shared secret key is necessary. For example, if Alice and Bob want to send messages in a way that prevents Eve from reading them, how can they possibly do this unless they have some information that Eve does not know? (Since we're going to see that this is in fact possible, it's a little difficult for me to try to convey the intuition that one might have suggesting that it's not possible, but I hope you recognize that it does seem kind of ridiculous that one should be able to do anything interesting without a shared secret key. Or, perhaps you're simply smarter than I am and it's immediately obvious to you that this is possible...)

Of course, if shared secret keys really were necessary for secure communication, then the internet would be a very different place. We might, for example, need to get a secret key sent to us in the mail before we can set up an account with a website for the first time! Or, maybe we would all rely on some central authority (the government? a large corporation?) that shares secret keys with everyone and facilitates communication.

1.1 A little history

The first people to *publicly* consider that something better might be possible were Whitfield ("Whit") Diffie and Martin Hellman, in their extraordinarily influential and prescient paper "New directions in cryptography" from 1976 [DH76]. (That is an extremely bold title for a paper, and it is actually justified.) They considered the possibility of both private and authenticated communication *without* a shared secret and suggested some approaches (and, just for good measure, they mentioned other possible cryptographic primitives, such as obfuscation). They very deservedly won the Turing award for this achievement.

However, it turns out that James Ellis, a cryptographer working for the British spy agency GCHQ, came up with this idea all the way back in 1970. Ellis and his colleagues discovered many of the ideas now used in public-key cryptography long before they were discovered publicly—all of which remained classified until 1997. In particular, Clifford Cocks discovered RSA encryption

in 1973. (They called it "non-secret encryption," which is a horrible name in my opinion.) And, Malcolm Williamson discovered Diffie-Hellman key agreement in 1974. We will see both of these ideas soon. (We of course do not know whether they discovered other things as well that are still classified.)

1.2 Splitting the key in two

One of the ideas that Diffie and Hellman had was to observe that Alice and Bob play different roles in our stories. E.g., when Alice sends a message privately to Bob, Alice encrypts the message, while Bob decrypts the message. For authentication, Alice tags a message, and Bob just checks the tag.

So, since Alice and Bob do different things, maybe there's a way to make this work in which they have different keys as well? In particular, instead of requiring Alice and Bob to share some k, we require one of them to have a secret key sk and the other to have a public key pk. The public key is, well, public—in the sense that even the adversary Eve gets to know the public key.

2 Signatures

The public-key counterpart of a MAC is a *digital signature*. (I think that they are formally called *digital* signatures—rather than just signatures—to distinguish them from the little squiggly things that we put on contracts and checks. When there is no risk of confusion, we typically just call them signatures, as the term "digital" just sounds rather archaic to me. When do we talk about anything in this class that is *not* digital?) In keeping with the intuition described above, the key observation here is that Alice and Bob do different things when Alice sends an authenticated method to Bob. Specifically, Alice *produces* a signature, and Bob *verifies* it. Notice that we definitely do not want the adversary to be able to produce signatures on her own. However, it might be okay if the adversary can *verify* signatures. So, maybe we can come up with a definition in which Alice and Bob have different pieces of information, so that *only* Alice is able to produce signatures, but anyone at all can verify them.

So, yeah, let's do that :). As we often do, we first define correctness and then worry about security.

Definition 2.1. *A* digital signature scheme *consists of three PPT algorithms* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ *with the following properties.*

1. Gen *takes as input* 1^n *and outputs* two *keys* sk, vk*. sk is called the* secret key *(or sometimes the signing key), and vk is called the* verification key *(or sometimes the public key, in which case it is written pk).*
2. Sign *takes as input a secret key sk and a plaintext m and outputs a* signature *σ.*
3. Ver *takes as input a verification key vk, a plaintext m, and a signature σ' and outputs either* 1 *or* 0 *(i.e., either "that's signature looks valid" or "that signature looks invalid").*
4. **(Correctness.)** *For any n and any plaintext m,*

$$\Pr_{(sk,vk)\leftarrow\mathsf{Gen}(1^n)}[\mathsf{Ver}(vk, m, \mathsf{Sign}(sk, m)) = 1] = 1 \ .$$

Notice that the above definition is identical to the definition of a MAC except that we have "divided the MAC key k into two keys sk, vk"—one used by the signing algorithm and the other used by the verification algorithm. Again, we will think of vk as *public*, e.g., Alice might publish it on her website, while sk is of course secret. In other words, signatures are "publicly verifiable," in the sense that we think of the information needed to verify the signature (the verification key) as public.

2.1 Security

Our security notion for a digital signature will be the same basic notion that we used for a MAC, i.e., existential unforgeability against adaptive chosen message attacks (EUACMA), as defined in the game below. The one *crucial* difference between this game and the MAC game is that in this game the adversary is given access to vk. This formally captures the notion that vk is public. In particular, what does it mean from a security perspective that "it is okay to release vk publicly"? Well, it means we're even willing to give vk to our worst enemy—the adversary!

Adversary		**Challenger**
INPUT: 1^n		INPUT: 1^n
		$(sk, vk) \leftarrow \mathsf{Gen}(1^n)$
	$\xleftarrow{\quad vk \quad}$	
		i = 1
		DO UNTIL the adversary aborts
	$\xrightarrow{\quad m_i \in \{0,1\}^n \quad}$	
	$\xleftarrow{\quad \mathsf{Sign}(sk, m_i) \quad}$	
		i++
		END DO
	$\xrightarrow{\quad m', \sigma' \quad}$	
		IF $m' \notin \{m_1, \ldots, m_\ell\}$ AND $\mathsf{Ver}(vk, m', \sigma') = 1$, OUTPUT "WIN"

Definition 2.2. *A digital signature is* secure *(EUACMA) if for any PPT adversary $\mathcal{A}$ there exists a negligible $\varepsilon(n)$ such that the probability that $\mathcal{A}$ wins the above game is at most $\varepsilon(n)$.*

As in the case of MACs, we can write this definition without explicitly describing the above protocol by using oracles. (To be clear, these two definitions are exactly the same. We are just showing different notation for the same thing.)

Definition 2.3. *A digital signature is* secure *if for any PPT adversary $\mathcal{A}$ there exists a negligible $\varepsilon(n)$ such that*

$$\Pr_{(sk,vk)\leftarrow\mathsf{Gen}(1^n)}[(m', \sigma') \leftarrow \mathcal{A}^{\mathsf{Sign}(sk,\cdot)}(1^n, vk),\ m' \notin Q \text{ and } \mathsf{Ver}(vk, m', \sigma') = 1] \leq \varepsilon(n) \ ,$$

where $\mathsf{Sign}(sk, \cdot)$ is an oracle that takes as input a plaintext m and outputs $\mathsf{Sign}(sk, m)$, and Q is the list of queries that $\mathcal{A}$ makes to this oracle.

Notice that we do not need to give our adversary $\mathcal{A}$ a verification oracle in these games because the adversary $\mathcal{A}$ has access to the verification key. It can thus verify signatures itself, without any need for an oracle! So, a verification oracle would be redundant :).

3 Lamport's one-time signature

One of the magical things about signatures is that, even though they seem like public-key objects (since, well, there's a public key and a secret key), they exist in Minicrypt, i.e., they can be built from one-way functions. (We will also use a collision-resistant hash function, but this is not actually necessary.) Our first hint of this possibility comes from Leslie Lamport's construction of a weaker object called a *one-time signature* [Lam79]. (This idea was actually first published by Diffie and Hellman in their gigantic breakthrough paper in which they proposed public-key cryptography [DH76]. However, Diffie and Hellman credit the idea to Lamport, and Lamport eventually did write the idea down himself in [Lam79], though it took a few years.)

As you might guess, a one-time signature is a signature that is secure if you use it only once. To formalize this, we only allow the adversary to make a single query to the signature oracle before she attempts to forge a fresh signature. Here's the definition. (We saw that the "one-time" versions of MACs and secret-key encryption were achievable information theoretically, i.e., we could achieve security against computationally unbounded adversaries using the one-time pad and pairwise-independent hash functions respectively. This is not possible for signatures (or, more generally, for public-key primitives).)

Definition 3.1. *A signature scheme is* one-time secure *if for every PPT $\mathcal{A}$* making at most one query to the signature oracle *in the (EUACMA) signature security game, there exists a negligible $\varepsilon(n)$ such that the probability that $\mathcal{A}$ wins is at most $\varepsilon(n)$. For technical reasons, we also assume that the* Sign *algorithm is deterministic.*[1]

Given this, Lamport's one-time signature is quite simple and elegant. Here's the construction, which can be used to sign n-bit messages. Let f be any one-way function.

- $\mathsf{Gen}(1^n)$: Sample $x_{1,0}, \ldots, x_{n,0}, x_{1,1}, \ldots, x_{n,1} \sim \{0,1\}^n$. Output $sk = (x_{1,0}, \ldots, x_{n,0}, x_{1,1}, \ldots, x_{n,1})$ and $vk = (y_{1,0} = f(x_{1,0}), \ldots, y_{n,0} = f(x_{n,0}), y_{1,1} = f(x_{1,1}), \ldots, y_{n,1} = f(x_{n,1}))$.
- $\mathsf{Sign}(sk, m)$: Output $x_{1,m_1}, x_{2,m_2}, \ldots, x_{n,m_n}$, where $m = (m_1, \ldots, m_n) \in \{0,1\}^n$ represents the bit decomposition of m.
- $\mathsf{Ver}(vk, m, (x'_1, \ldots, x'_n))$: Output 1 if and only if $f(x_i) = y_{i,m_i}$ for all i.

In words, the signer knows preimages $x_{i,b}$ for the $y_{i,b}$. To sign the plaintext $m \in \{0,1\}^n$, the signer simply provides the preimages x_{i,m_i} corresponding to the bits of m. Intuitively, this is secure because any plaintext $m' \neq m$ must differ from m in at least one bit, e.g., $m_i = 0$ and $m'_i = 1$. So, in order for an adversary to produce a valid signature for m', it must somehow produce a preimage of $y_{i,1}$. The below proof makes this formal. (It *does* require one little trick to make it actually work, which is that the reduction must guess ahead of time an index i and a bit b such that $m'_i = b$

[1] This is without loss of generality for one-time secure signatures. Since we only need to use it once, we can always include as part of the secret key the random coins r that we will use to generate the one-and-only signature that we will produce.

and $m_i \neq b$. This is necessary because the reduction "needs to know where to plant the challenge y^*.")

Theorem 3.2. *Lamport's signature is one-time secure.*

Proof. Suppose that there exists a PPT adversary $\mathcal{A}$ that wins the one-time signature game against Lamport's signature with non-negligible probability $\varepsilon(n)$.

Then, we construct an adversary $\mathcal{A}'$ in the one-way function game as follows. $\mathcal{A}'$ takes as input $y^* = f(x^*)$ for $x^* \sim \{0,1\}^n$ and samples a uniformly random index $i^* \sim \{1,\ldots,n\}$ and a uniformly random bit $b^* \sim \{0,1\}$. It sets $y_{i^*,b^*} := y^*$ and for all other pairs (i,b) (including $(i = i^*, b = 1-b^*)$), it samples $x_{i,b} \sim \{0,1\}^n$ and sets $y_{i,b} := f(x_{i,b})$. It then sends $vk = (y_{1,0},\ldots,y_{n,0},y_{1,1},\ldots,y_{n,1})$ to $\mathcal{A}$, receiving in response a single plaintext m to sign.

If $m_{i^*} = b^*$, $\mathcal{A}'$ simply fails (since $\mathcal{A}'$ does not know the inverse to $y_{i^*,b^*} = y^*$ and therefore cannot provide a signature of m). Otherwise, $\mathcal{A}'$ sends $(x_{1,m_1},\ldots,x_{1,m_n})$ to $\mathcal{A}$, receiving in response $m', \sigma = (x'_1,\ldots,x'_n)$. Finally, $\mathcal{A}'$ outputs x_{i^*}.

Clearly $\mathcal{A}'$ is efficient. First, notice that $\Pr[m_{i^*} = b^*] = 1/2$, since b^* was sampled uniformly at random, and the distribution of vk is independent of b^*. (This independence relies on the fact that $y^* = f(x^*)$ for uniformly random $x^* \sim \{0,1\}^n$.)

Now, let us assume without loss of generality that $\mathcal{A}$ always outputs $m' \neq m$ (since anyway $\mathcal{A}$ always loses when $m = m'$). Then, we claim that

$$\Pr[m'_{i^*} = b^* \text{ and } m_{i^*} \neq b^*] \geq 1/(2n) \ .$$

To see this, notice that i^* is independent of vk and b^*, so i^* is uniformly random and independent of m', even conditioned on $m_{i^*} \neq b^*$. Since there exists at least one index i such that $m'_i \neq m_i$, the claim follows.

Finally, we claim that

$$\Pr[f(x'_{i^*}) = y^* \mid m_{i^*} \neq b^*,\ m'_{i^*} = b^*] \geq \varepsilon(n) \ .$$

Indeed, conditioned on $m_{i^*} \neq b^*$ and $m'_{i^*} = b^*$, (1) the view of $\mathcal{A}$ is identical to its view in the signature game; and (2) whenever $\mathcal{A}$ wins the signature game, we must have $f(x'_{i^*}) = y^*$. The result follows. □

3.1 Don't use this twice!

Of course, there is a reason that we stressed that this is really a *one-time* signature. If an adversary knows a signature for *two* distinct plaintexts $m = (m_1,\ldots,m_n)$ and $m' = (m'_1,\ldots,m'_n)$, then she can trivially find a signature of any "mixed message," e.g., $(m_1, m'_2, m'_3, m_4, \ldots, m_{n-1}, m'_n)$.

4 Remember hash-then-MAC? Now it's time for hash-then-sign!

In some sense, there are two distinct issues with Lamport's signature scheme. The most serious issue is of course that it can only be used once! But, a minor issue is that the key and signature sizes are pretty silly. If we want to sign an n-bit plaintexts, then our secret key must be $2n^2$ bits long, and a signature is n^2 bits long

In this course, we often ignore such things—$2n^2$ is a polynomial in n, which is usually good enough for us. But, it turns out that focusing on this will help us make progress on solving the bigger

problem. In particular, below, we will want to build a sequence of keys $(sk_0, vk_0, \ldots, sk_m, vk_m)$ and use sk_{i-1} to sign vk_i. This clearly is not possible to do efficiently for large m using the above scheme, because we will need the number of bits in sk_{i-1} to be roughly the square of the number of bits in vk_i, which will quickly get out of hand. So, we will need a way to be able to sign plaintexts that are as long as the verification key.

Luckily, we've already thought about issues like this before in the context of MACs. Recall that in the last lecture, we used the hash-then-MAC construction to combine a collision-resistant hash function H_{hk} (where hk stands for *hash key*) with a MAC $(\mathsf{Gen}, \mathsf{MAC}, \mathsf{Ver})$ for n-bit plaintexts in order to build a MAC that works for arbitrarily large plaintexts. Essentially the exact same construction works in this context. For completeness, we show the full scheme below.

Let $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ be a one-time secure signature for n-bit plaintexts $m \in \{0,1\}^n$, and let $H_{hk} : \{0,1\}^* \to \{0,1\}^{|hk|}$ be a collision-resistant hash function. We build a new signature scheme $(\mathsf{Gen}^*, \mathsf{Sign}^*, \mathsf{Ver}^*)$ that can sign arbitrarily long plaintexts $M \in \{0,1\}^*$ as follows.

- $\mathsf{Gen}^*(1^n)$: Sample $hk \sim \{0,1\}^n$ and $(sk, vk) \leftarrow \mathsf{Gen}(1^n)$. Output $sk^* = (sk, hk)$ and $vk^* = (vk, hk)$.
- $\mathsf{Sign}^*((sk, hk), M)$: Output $\mathsf{Sign}(sk, H_{hk}(M))$.
- $\mathsf{Ver}^*((vk, hk), M, \sigma')$: Output $\mathsf{Ver}(vk, H_{hk}(M), \sigma')$.

The proof of this is essentially identical to the proof for MACs from the previous lecture, so we do not include it. (In fact, this construction does not only convert a one-time secure signature on n-bit plaintexts to a one-time signature on arbitrarily long plaintexts. It also converts a many-time secure signature on n-bit plaintexts to a many-time secure signature on arbitrarily long plaintexts. I.e., it inherits whatever security the original signature scheme has.)

Theorem 4.1. *The above hash-and-sign scheme is a one-time secure signature for arbtirarily long plaintexts $M \in \{0,1\}^*$.*

5 From one-time to many-time—a chain of keys, and a stateful scheme (informal)

Now, it is relatively straightforward to convert a one-time signature into a many-time signature if we are willing to change the definition slightly. Specifically, we will consider a scheme that is *stateful* in the sense that it "updates the keys at every step." This is analogous to the stream cipher construction that we saw for secret-key encryption. (In fact, this whole lecture might give you some serious déjà vu, as we're repeating in this new context a lot of the ideas that we already used on our long journey to building secret-key encryption.)

The scheme works as follows. Suppose we have a signature scheme that is one-time secure and can sign arbitrarily long plaintexts (which we know how to construct, since we did it above). We generate an initial key pair (sk_1, vk_1) using the Gen algorithm. Then, the first time that we sign some message $M_1 \in \{0,1\}^*$, we do not just output $\mathsf{Sign}(sk_1, M_1)$. Instead, we sample $(sk_2, vk_2) \leftarrow \mathsf{Gen}(1^n)$ and output $\sigma_1 := \mathsf{Sign}(sk_1, (M_1, vk_2))$. (Remember, we can sign arbitrarily long messages, so there's no issue with signing the longer message (M_1, vk_2).) You can interpret this as a signature of a message saying "M_1, and also the next time you wish to verify a signature

from me, please use my new verification key vk_2." (Indeed, we could sign this whole English sentence if we wanted to.) Then, when it comes time to sign the next plaintext M_2, we sample $(sk_3, vk_3) \leftarrow \mathsf{Gen}(1^n)$ and output $\sigma_2 := \mathsf{Sign}(sk_2, (M_2, vk_3))$. This second signature is verified with the key vk_2, and so on.

We will not bother to write a formal definition that captures this idea (since we will improve upon it soon), but this scheme does in fact satisfy a nice version of this formal definition. Intuitively, it works because (1) each signing key sk_i is only used to sign one thing (which is good since this is only a one-time secure signature!); and (2) each verification key is authenticated by the previous one, so "the adversary cannot replace a legitimate verification key vk_i with some illegitimate key vk_i^*."

The very frustrating thing about this scheme is that it is stateful. It is slightly annoying that the *signing* algorithm is stateful. But, it is a much bigger problem that the verification algorithm must be stateful. In particular, the verification algorithm needs access to the first $i-1$ signatures $\sigma_1, \ldots, \sigma_{i-1}$ in order to verify the ith signature σ_i. This means that the scheme is only useful if every party who ever wants to verify your signature on *any* plaintext M_i is willing to verify *all* of your signatures on all plaintexts $M_1, \ldots, M_{i-1}$ that you sent before that. That's rather inefficient! It also means that you need to be willing to share all of the previous plaintexts $M_1, \ldots, M_{i-1}$ that you have signed with whoever wants to verify M_i.

6 From one-time to many-time the right way—a tree of keys!

In the previous section, we used each key sk_i to sign one new verification key vk_{i+1}. Now, we use each key to sign *two* new verification keys. This gives a tree of key instead of a list of keys.

Specifically, imagine that we sample a key pair $(sk_x, vk_x) \leftarrow \mathsf{Gen}(1^n)$ for *every* bit string x with length at most n. (Of course, there are $2^{n+1}-1$ such strings, so we cannot do this efficiently. But we will fix this below.) E.g., we have $(sk_\emptyset, vk_\emptyset) \leftarrow \mathsf{Gen}(1^n)$ (where $\emptyset$ represents the empty string—the string with length zero), and then $(sk_0, vk_0) \leftarrow \mathsf{Gen}(1^n)$ and $(sk_1, vk_1) \leftarrow \mathsf{Gen}(1^n)$, and then (sk_{00}, vk_{00}), and so on. We think of these as arranged in a binary tree in the natural way. (I.e., (sk_x, vk_x) has as its children (sk_{x0}, vk_{x0}) and (sk_{x1}, vk_{x1}).)

We only release $vk_\emptyset$ as our verification key. So, the verification key is nice and small, but right now we will take our signing key to be $(sk_x, vk_x)_{|x| \le n}$—one pair of keys for every string of length at most n. That's a list with $2^{n+1}-1$ elements! That is of course way too long for the secret key, since we can't write it down in polynomial time. But, we will fix this later.

So, ignoring this gigantic problem for the moment, notice that we can then use $sk_\emptyset$ to compute the signature $\sigma_\emptyset := \mathsf{Sign}(sk_\emptyset, (vk_0, vk_1))$ of the verification keys vk_0 and vk_1 corresponding to its two children. And, more generally, we can use sk_x to compute $\sigma_x := \mathsf{Sign}(sk_x, (vk_{x0}, vk_{x1}))$ for all bit strings x with $|x| < n$. For convenience, we also define $\tau_x := (\sigma_x, vk_{x0}, vk_{x1})$. Then, to sign a plaintext $m \in \{0,1\}^n$, we release the signature $\sigma_m^* := \mathsf{Sign}(sk_m, m)$ of m under the key sk_m corresponding to m, together with the list $\tau_\emptyset, \tau_{m_1}, \tau_{m_1 m_2}, \ldots, \tau_{m_1 m_2 \cdots m_{n-1}}$. You can think of this lists as corresponding to all signatures and verification keys τ_x along the path to m in our tree of keys.

To verify this signature, the verification procedure first iterates through the list of the τ_x. For each $\tau_{m_1 m_2 \cdots m_i} = (\sigma_{m_1 m_2 \cdots m_i}, vk_{m_1 m_2 \cdots m_i 0}, vk_{m_1 m_2 \cdots m_i 1})$, it checks that

$$\mathsf{Ver}(vk_{m_1 m_2 \cdots m_i}, (vk_{m_1 m_2 \cdots m_i 0}, vk_{m_1 m_2 \cdots m_i 1}), \sigma_{m_1 m_2 \cdots m_i}) = 1 \ .$$

Finally, it checks that $\mathsf{Ver}(vk_m, m, \sigma^*_m) = 1$. It outputs 1 if and only if all of these checks pass.

6.1 Sampling the keys lazily (and pseudorandomly)

One can check that the above scheme is secure. Intuitively, it is secure because every secret key sk_x is only used to sign a single fixed plaintext—either (vk_{x0}, vk_{x1}) if x is not a leaf of the tree, or x itself if x is a leaf. (And, since we assumed that the signing algorithm Sign is deterministic, each time that we produce a signature under sk_x, it will be the same.) However, the key-generation procedure is of course horribly inefficient, since it requires us to build a list with $2^{n+1} - 1$ elements.

Of course, in order to sign any specific message m, we only need to know the keys along the path $\emptyset, m_1, m_1m_2, \ldots, m$. So, a natural idea is to try to sample these keys *lazily*. I.e., the key-generation algorithm might only sample $sk_\emptyset, vk_\emptyset$. And, when we are asked to sign message m, we could simply sample fresh keys for all of the keys

$$
\begin{aligned}
&(sk_0, vk_0), (sk_1, vk_1),\\
&(sk_{m_10}, vk_{m_10}), (sk_{m_11}, vk_{m_11}),\\
&(sk_{m_1m_20}, vk_{m_1m_20}), (sk_{m_1m_21}, vk_{m_1m_21}),\\
&\ldots,\\
&(sk_{m1m2\cdots m_{n-1}0}, vk_{m1m2\cdots m_{n-1}0}), (sk_{m1m2\cdots m_{n-1}1}, vk_{m1m2\cdots m_{n-1}1})
\end{aligned}
$$

necessary to create a signature for m. (These are all the keys along the path to m on the tree, *together with* all of their siblings, since every time we sign a verification key vk_{x0}, we also must sign vk_{x1}. If we didn't do this, then we would not have a tree!)

The problem with using *truly* fresh keys here is that it is insecure. In particular, suppose that our adversary asks us for two signatures. Then, with this approach, we would produce two different key pairs (sk_0, vk_0) and (sk'_0, vk'_0) and then produce signatures under $sk_\emptyset$ of two *different* plaintexts vk_0 and vk'_0 (well, technically, of (vk_0, vk_1) and (vk'_0, vk'_1))! This is not a good thing to do with a signature that is only one-time secure!

So, we cannot *actually* sample fresh keys each time that we sign. And, we cannot store $\approx 2^n$ randomly sampled strings efficiently. We could just remember every key that we've sampled, but only if we're willing to have a stateful signing algorithm, which is not ideal. What we *can* do instead is implicitly store $\approx 2^n$ *pseudorandom strings*, using a PRF. Specifically, given a PRF F_k, we can define $(sk_x, vk_x) \leftarrow \mathsf{Gen}(1^n; F_k(x))$, where here we have used $F_k(x)$ to replace the random coins of the key-generation algorithm.[2] Since F_k is a PRF, we intuitively expect using $F_k(x)$ to be just as good as using some uniformly random string r_x.

Here is the full construction. Let F_k be a PRF, and let $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ be a signature scheme that works for plaintexts of arbitrary length and is one-time secure. Then, we define $(\mathsf{Gen}^*, \mathsf{Sign}^*, \mathsf{Ver}^*)$ to sign n-bit plaintext, as follows. (The formal description is a bit of a mouthful, but what's going on is just what was described above.)

- $\mathsf{Gen}^*(1^n)$: Sample $k \sim \{0,1\}^n$ and $(sk_\emptyset, vk_\emptyset) \leftarrow \mathsf{Gen}(1^n)$. Output $sk^* := (k, sk_\emptyset, vk_\emptyset)$ and $vk^* := vk_\emptyset$.

[2] We originally defined PRFs to take input strings of length $n = |k|$ and to output strings of length n. Now, I am assuming that our PRFs take as input strings of length *at most* n and output strings whose length is n^C—whatever the maximal number of random coins needed by the Gen algorithm is. Of course, it is easy to build such a PRF from the kind that we defined earlier, so we will ignore this minor annoyance.

- $\mathsf{Sign}^*(sk^* = (k, sk_\emptyset, vk_\emptyset), m)$: For $i = 0, \ldots, n-1$, compute

$$\tau_i := (vk_{m_1 \cdots m_i 0}, vk_{m_1 \cdots m_i 1}, \mathsf{Sign}(sk_{m_1 \cdots m_i}, (vk_{m_1 \cdots m_i 0}, vk_{m_1 \cdots m_i 1})))$$

  and $\tau^* := \mathsf{Sign}(sk_m, m)$, where $(sk_x, vk_x) := \mathsf{Gen}(1^n; F_k(x))$ (and we interpret $m_1 \cdots m_i$ as the empty string when $i = 0$). Output $\sigma^* := (\tau_0, \ldots, \tau_{n-1}, \tau^*)$.

- $\mathsf{Ver}^*(vk^* = vk_\emptyset, m, \sigma' := (\tau'_0, \ldots, \tau'_{n-1}, \tau^*))$. Check that $\tau'_i = (vk_{m_1 \cdots m_i 0}, vk_{m_1 \cdots m_i 1}, \sigma_i)$ satisfies

$$\mathsf{Ver}(vk_{m_1 \cdots m_i}, (vk_{m_1 \cdots m_i 0}, vk_{m_1 \cdots m_i 1}), \sigma_i) = 1$$

  for all i. Finally, check that $\mathsf{Ver}(vk_m, m, \tau^*) = 1$, and output 1 if and only if all checks pass.

Before we prove our main theorem, we will need a lemma about the following game. Intuitively, the game allows the adversary to see one signature *each* from *many* different key pairs (sk_i, vk_i), and asks the adversary to forge a signature for any of these key pairs. (The adversary is allowed to request the signatures in any order. The annoying pseudocode on the right with the set Q is just to make sure that the adversary does not request *two* signatures from the same key pair (sk_i, vk_i).)

Adversary		**Challenger**
INPUT: 1^n		INPUT: 1^n
		$(sk_1, vk_1) \leftarrow \mathsf{Gen}(1^n), \ldots, (sk_q, vk_q) \leftarrow \mathsf{Gen}(1^n)$
	$\xleftarrow{vk_1, \ldots, vk_q}$	
		$Q = \emptyset$
		DO UNTIL the adversary aborts
	$\xrightarrow{i \in \{1, \ldots, q\}, M_i \in \{0,1\}^*}$	
		IF $i \in Q$, OUTPUT "LOSE"
		Add i to Q.
	$\xleftarrow{\mathsf{Sign}(sk_i, M_i)}$	
		END DO
	$\xrightarrow{i, M', \sigma'}$	
		IF $M' \neq M_i$ AND $\mathsf{Ver}(vk_i, M', \sigma') = 1$, OUTPUT "WIN"

Lemma 6.1. *If* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ *is a secure one-time signature, then no adversary has non-negligible probability of winning the above game for any* $q = q(n) \leq \mathrm{poly}(n)$.

Proof. We suppose that there exists an adversary $\mathcal{A}$ with non-negligible advantage $\varepsilon(n)$ in the above game and construct an adversary $\mathcal{A}'$ in the one-time signature game against $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$. $\mathcal{A}'$ receives as input a verification key vk^*, where $(sk^*, vk^*) \leftarrow \mathsf{Gen}(1^n)$. It then samples $i^* \sim \{1, \ldots, q\}$ uniformly at random and sets $vk_{i^*} := vk^*$. For all $i \neq i^*$, $\mathcal{A}'$ samples $(sk_i, vk_i) \leftarrow \mathsf{Gen}(1^n)$. It then sends $(vk_1, \ldots, vk_q)$ to $\mathcal{A}$.

$\mathcal{A}'$ then responds to the queries (i, M_i) made by $\mathcal{A}$ as follows. If $i \neq i^*$, it computes the signature $\mathsf{Sign}(sk_i, M_i)$ itself and sends the result to $\mathcal{A}$. If $i = i^*$, it passes M_i to its signing oracle, receiving in response some signature σ^*. It then simply passes σ^* to $\mathcal{A}$.

When $\mathcal{A}$ eventually sends (i, M', σ') to $\mathcal{A}'$, $\mathcal{A}'$ first checks if $i = i^*$. If not, then $\mathcal{A}'$ simply fails. Otherwise, $\mathcal{A}'$ passes (M', σ') to its challenger.

Clearly $\mathcal{A}'$ is efficient. We claim that $\mathcal{A}'$ wins the one-time signature game with probability equal to $\varepsilon(n)/q$, which is non-negligible because $q \leq \text{poly}(n)$. To see this, first notice that the view of $\mathcal{A}$ in interaction with $\mathcal{A}'$ is identical to the view of $\mathcal{A}$ in the game described above, and is independent of i^*. It follows that $\mathcal{A}'$ outputs a valid forgery (i, M', σ') forgery with probability $\varepsilon(n)$, and that this event is independent of i^*. Therefore,

$$\Pr[\mathcal{A} \text{ outputs a valid forgery and } i = i^*] = \varepsilon(n)/q \ ,$$

as needed. □

Theorem 6.2. *The above scheme is a secure (EUACMA) signature. In particular, secure signatures are implied by the existence of one-way functions (and CRHFs, though this is not necessary).*

Proof. As you might guess, the proof is via a sequence of (just two) games. (A good rule of thumb is that a sequence of games is the right proof strategy whenever you are proving the security of a construction that uses more than one different primitive. E.g., here we are using both a PRF and the underlying one-time signature.)

Game 1 is simply the signature security game (the EUACMA game) against Sig^*.

Game 2 is the same game, except we replace the keys $(vk_x, sk_x) = \mathsf{Gen}(1^n; F_k(x))$ by $(vk_x, sk_x) = \mathsf{Gen}(1^n; r_x)$ for uniformly random and independent r_x. (I.e., Game 2 is the signature security game against the inefficient signature scheme that we described in the previous section.)

The intuition for the rest of the proof is simply that (1) Game 1 and Game 2 are indistinguishable by the security of the PRF; and (2) no PPT adversary can have non-negligible advantage in Game 2 by the security of the one-time signature (because each key pair is only used to sign a single plaintext). The following two claims make this formal.

Claim 6.3. *Any for any PPT adversary $\mathcal{A}$, there exists negligible $\varepsilon(n)$ such that*

$$\Pr[\mathcal{A} \textit{ wins Game } 1] - \Pr[\mathcal{A} \textit{ wins Game } 2] \leq \varepsilon(n) \ .$$

Proof. Given an adversary $\mathcal{A}$ in Game 1 (or Game 2), we construct an adversary $\mathcal{A}'$ in the PRF security game as follows. I.e., $\mathcal{A}'$ is given oracle access to an oracle H_b, where $b \sim \{0,1\}$, H_0 is a random oracle, and H_1 is F_k for $k \sim \{0,1\}^n$, and $\mathcal{A}'$ wants to guess the bit b.

$\mathcal{A}'$ samples $(vk_\emptyset, sk_\emptyset) \leftarrow \mathsf{Gen}(1^n)$ and sends $vk^* := vk_\emptyset$ to $\mathcal{A}$. Then, when $\mathcal{A}$ makes a query m to its signature oracle, $\mathcal{A}'$ runs the signing algorithm defined above, except that it sets the keys to be $(sk_x, vk_x) := \mathsf{Gen}(1^n; H_b(x))$.

Finally, $\mathcal{A}'$ sends to $\mathcal{A}$ a purported forged signature m', σ'. $\mathcal{A}'$ then simply runs $\mathsf{Ver}^*(vk_\emptyset, m', \sigma')$ algorithm itself. If the verification algorithm outputs 1 AND m' was not one of the queries made by $\mathcal{A}$, then $\mathcal{A}'$ outputs 1. Otherwise, it outputs 0.

$\mathcal{A}'$ is clearly efficient. Furthermore, notice that when $b = 0$, the view of $\mathcal{A}$ is exactly the same as its view in Game 2. When $b = 1$, its view is exactly the same as its view in Game 1. It follows that the advantage of $\mathcal{A}'$ in guessing b is exactly (half of) the difference that we are trying to prove is negligible. Since F_k is a secure PRF, this difference must be negligible, as needed. □

Claim 6.4. *No PPT adversary can have non-negligible advantage in Game 2.*

Proof. We prove this using Lemma 6.1. So, we suppose that some adversary $\mathcal{A}$ that has non-negligible advantage in Game 2, and we build an adversary $\mathcal{A}'$ in the game from Lemma 6.1.

$\mathcal{A}'$ first receives verification keys $vk_1, \ldots, vk_{2nq}$ from its challenger, where $q \leq \text{poly}(n)$ is a bound on the number of queries made by $\mathcal{A}$. Then, $\mathcal{A}'$ simulates Game 2 with $\mathcal{A}$. Each time that the game requires $\mathcal{A}'$ to produce a verification key vk_x that it has not already produced, it uses the next unused key in list $vk_1, \ldots, vk_{2nq}$. Each time that it needs to produce a signature, it makes the corresponding signature query. (Notice that here we are relying on the fact that we only need to use each key pair (sk_x, vk_x) *once*.) Finally, $\mathcal{A}'$ outputs $m' \in \{0,1\}^n$ and a purported signature σ'. If m' matches one of the queries made by $\mathcal{A}$, then $\mathcal{A}'$ simply gives up. Otherwise, σ' must contain a purported signature σ^* of some string $M^* \in \{0,1\}^*$ with corresponding verification key vk_{x^*} where vk_{x^*} is one of the keys produced by $\mathcal{A}'$ earlier, and M^* was not signed with vk_{x^*}. $\mathcal{A}'$ then uses the index corresponding to vk_{x^*}, the message M^*, and the signature σ^* as its forgery.

Clearly, $\mathcal{A}'$ is efficient, and the success probability of $\mathcal{A}'$ is at least the success probability of $\mathcal{A}$. The result follows. □

□

References

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 1976. 1, 4

[Lam79] Leslie Lamport. Constructing digital signatures from a one way function. Technical Report CSL-98, October 1979. 4