CIS 5560

Cryptography
Lecture 18

Announcements

- HW 6 released; due on Friday
- HW 7 will be released tomorrow
- Project will be released early next week

Recap of last lecture

New primitive: Digital Signatures

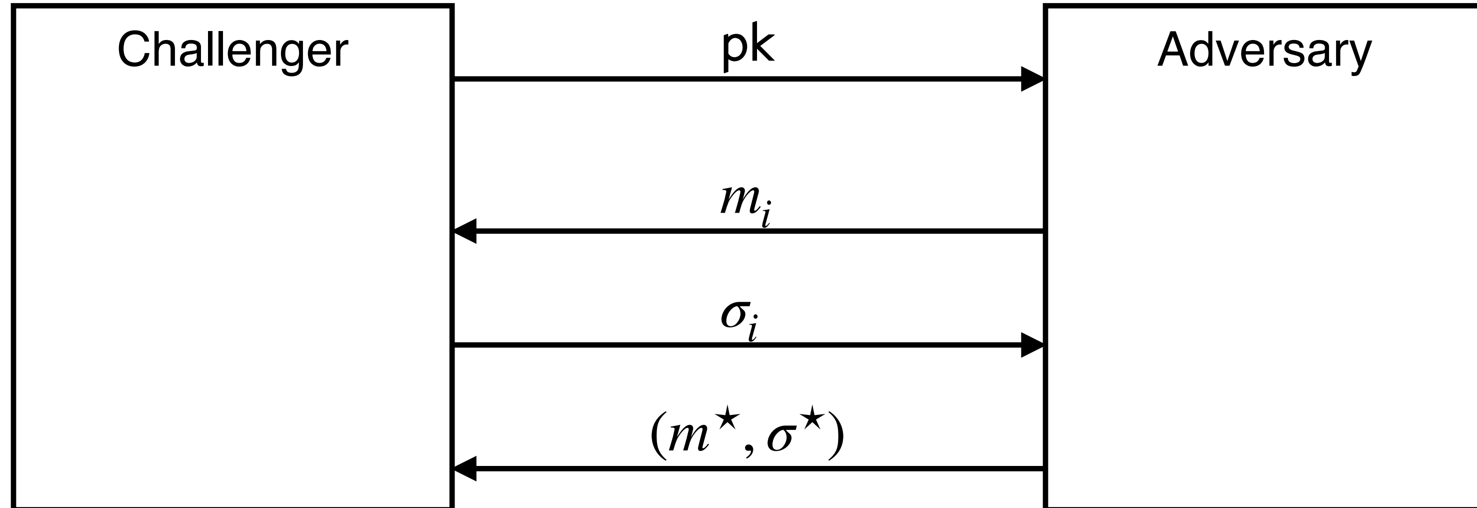
Digital Signatures: Definition

A triple of PPT algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ such that

- Key generation: $\mathsf{Gen}(1^n) \rightarrow (\mathsf{sk}, \mathsf{pk})$
- Message signing: $\mathsf{Sign}(\mathsf{sk}, m) \rightarrow \sigma$
- Signature verification: $\mathsf{Verify}(\mathsf{pk}, m, \sigma) \rightarrow b \in \{0,1\}$

Correctness: For all vk, sk, m: $\mathsf{Verify}(\mathsf{pk}, m, \mathsf{Sign}(\mathsf{sk}, m)) = 1$

EUF-CMA for Signatures



$$\Pr\left[\begin{array}{c} m^\star \notin \{m_i\} \\ \text{and} \\ \mathsf{Verify}(\mathsf{pk}, m^\star, \sigma^\star) = 1 \end{array}\right] = \mathsf{negl}(\lambda)$$

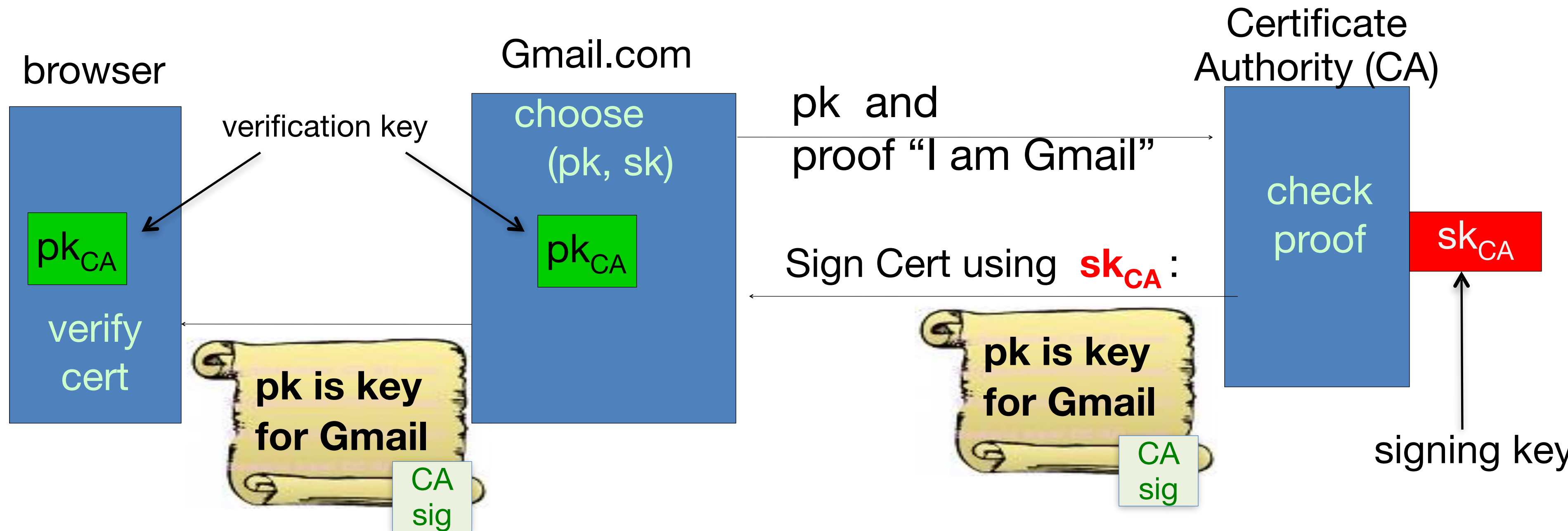
Today

- Signatures from CRH and OWF
- Signatures from RSA

Important application: Certificates

Problem: browser needs server's public-key to setup a session key

Solution: server asks trusted 3rd party (CA) to sign its public-key pk

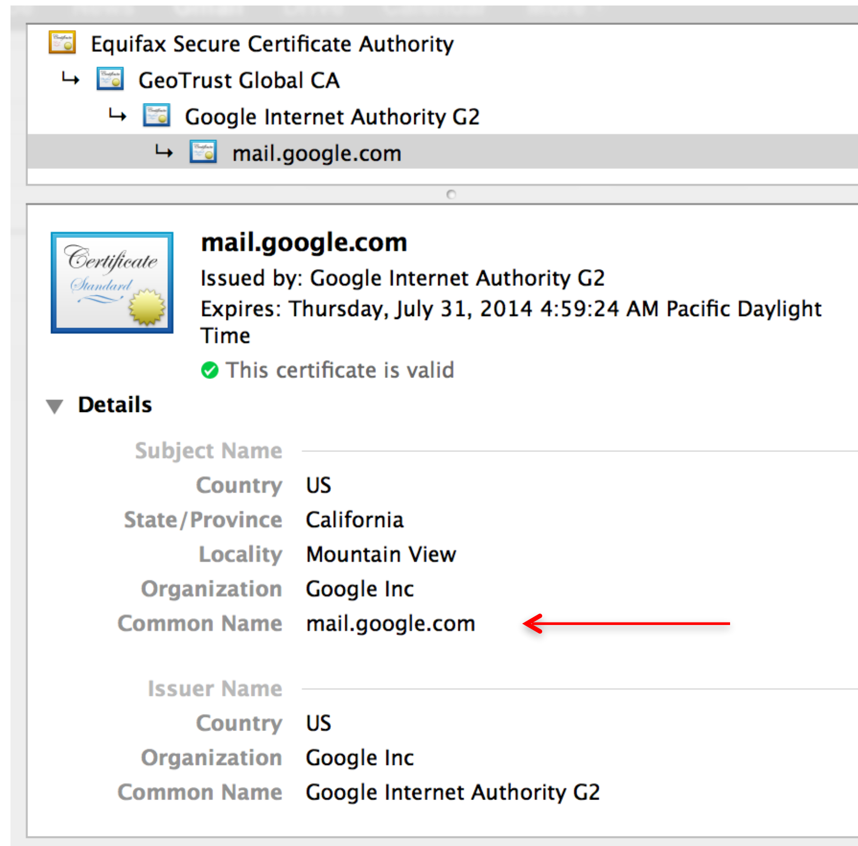


Server uses Cert for an extended period (e.g. one year)

Certificates: example

Important fields:

Serial Number	5814744488373890497
Version	3
Signature Algorithm	SHA-1 with RSA Encryption (1.2.840.113549.1.1.5)
Parameters	none
Not Valid Before	Wednesday, July 31, 2013 4:59:24 AM Pacific Daylight Time
Not Valid After	Thursday, July 31, 2014 4:59:24 AM Pacific Daylight Time
Public Key Info	
Algorithm	Elliptic Curve Public Key (1.2.840.10045.2.1)
Parameters	Elliptic Curve secp256r1 (1.2.840.10045.3.1.7)
Public Key	65 bytes : 04 71 6C DD E0 0A C9 76 ...
Key Size	256 bits
Key Usage	Encrypt, Verify, Derive
Signature	256 bytes : 8A 38 FE D6 F5 E7 F6 59 ...



What entity generates the CA's secret key sk_{CA} ?

- the browser
- Gmail
- the CA
- the NSA

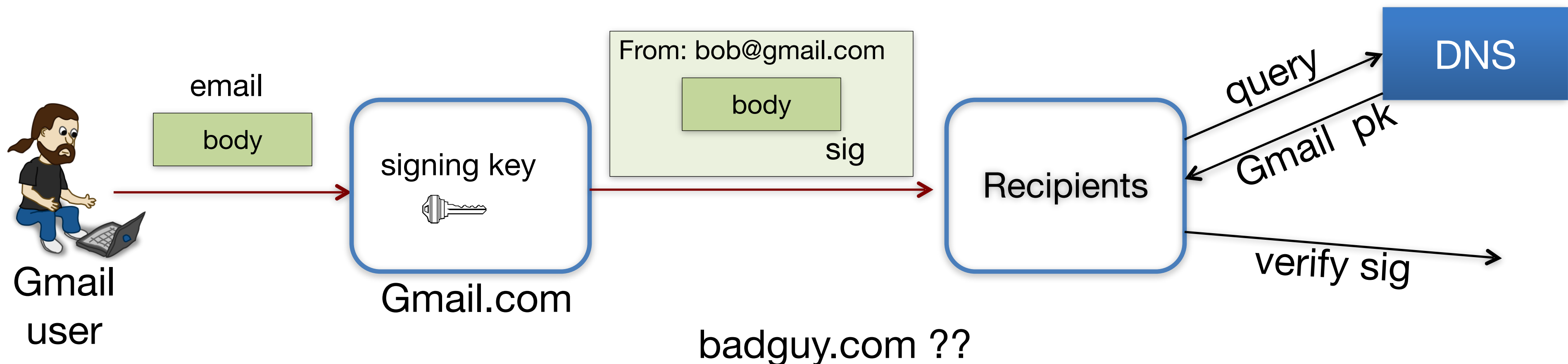
Signing email: DKIM (domain key identified mail)

Problem: bad email claiming to be from **someuser@gmail.com**

but in reality, mail is coming from domain **badguy.com**

⇒ Incorrectly makes gmail.com look like a bad source of email

Solution: **gmail.com** (and other sites) sign every outgoing mail



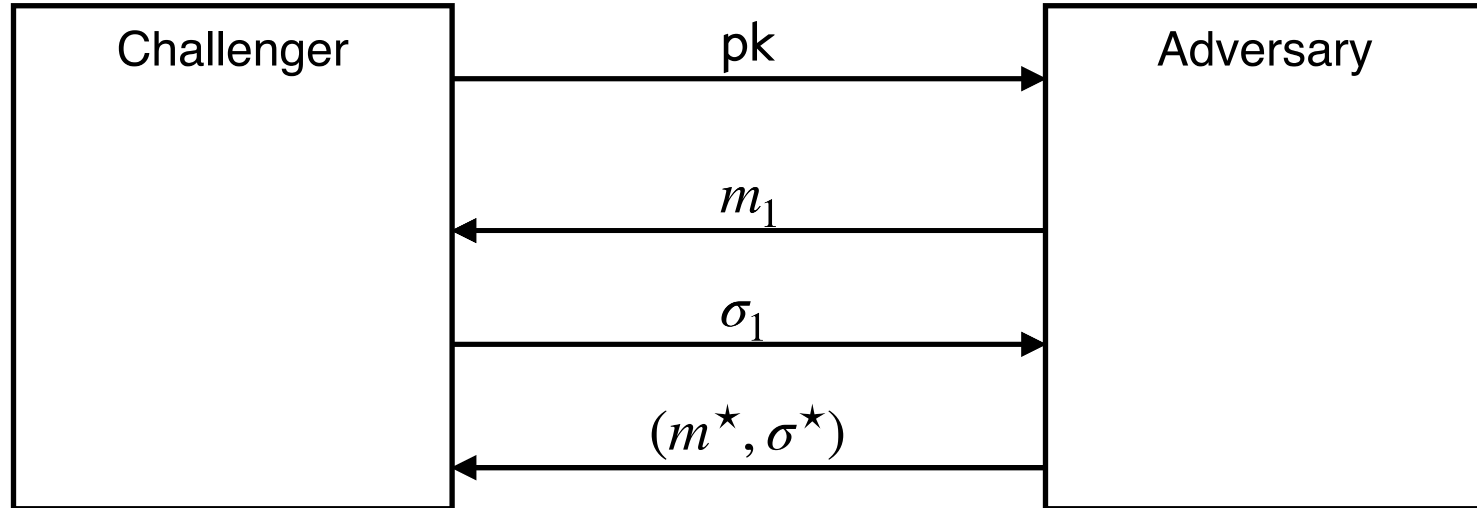
When to use signatures

Generally speaking:

- If one party signs and **one** party verifies: **use a MAC**
  - Often requires interaction to generate a shared key
  - Recipient can modify the data and re-sign it before passing the data to a 3rd party

- If one party signs and **many** parties verify: **use a signature**
  - Recipients **cannot** modify received data before passing data to a 3rd party (non-repudiation)

Simpler Goal: EUF-CMA for *1-time Signatures*



$$\Pr\left[\begin{array}{c} m^\star \neq m_1 \\ \text{and} \\ \mathsf{Verify}(\mathsf{pk}, m^\star, \sigma^\star) = 1 \end{array}\right] = \mathsf{negl}(\lambda)$$

Lamport (One-time) Signatures from OWFs

Signing Key sk: $\begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$

Public Key pk: $\begin{pmatrix} y_0 = f(x_0) \\ y_1 = f(x_1) \end{pmatrix}$

Signing a bit b: The signature is $\sigma = x_b$

Verifying (b, σ): Check if $f(\sigma) = y_b$

Claim: Assuming f is a OWF, no PPT adversary can produce a signature of $\bar{b}$ given a signature of b.

Lamport One-time Signatures for n-bit msgs

Secret Key sk: $\begin{pmatrix} x_{1,0} & x_{2,0} & \cdots & x_{n,0} \\ x_{1,1} & x_{1,1} & \cdots & x_{n,1} \end{pmatrix}$

Public Key pk: $\begin{pmatrix} y_{1,0} & y_{2,0} & \cdots & y_{n,0} \\ y_{1,1} & y_{2,1} & \cdots & y_{n,1} \end{pmatrix}$ where $y_{i,b} = f(x_{i,b})$.

Signing $m = (m_1, \ldots, m_n)$: $\sigma = (x_{1,m_1}, x_{2,m_2}, \ldots, x_{n,m_n})$

Claim: Assuming f is a OWF, no PPT adv can produce a signature of m given a signature of a single $m' \neq m$.

Claim: Can forge signature on any message given the signatures on (some) two messages.

Lamport (One-time) Signatures for arbitrary bits

Secret Key sk: $\begin{pmatrix} x_{1,0} & x_{2,0} & \cdots & x_{n,0} \\ x_{1,1} & x_{1,1} & \cdots & x_{n,1} \end{pmatrix}$

Public Key pk: $\begin{pmatrix} y_{1,0} & y_{2,0} & \cdots & y_{n,0} \\ y_{1,1} & y_{2,1} & \cdots & y_{n,1} \end{pmatrix}$ where $y_{i,b} = f(x_{i,b})$.

Signing m:

1. $z := H(m)$
2. $\sigma = (x_{1,z_1}, x_{2,z_2}, \ldots, x_{n,z_n})$

Claim: Assuming H is CRH and f is a OWF, no PPT adv can produce a signature of m given a signature of a single $m' \neq m$.

Claim: Can forge signature on any message given the signatures on (some) two messages.

So far, only one-time security…

Constructing a Signature Scheme

Step 0. Still one-time, but arbitrarily long messages.

Step 1. Many-time: Stateful, Growing Signatures.

Step 2. How to Shrink the signatures.

Step 3. How to Shrink Alice's storage.

Step 4. How to make Alice stateless.

Step 5 (*optional*). How to make Alice stateless and deterministic.

Constructing a Signature Scheme

Theorem [Naor-Yung'89, Rompel'90]
(EUF-CMA-secure) Signature schemes exist assuming that one-way functions exist.

TODAY:
(EUF-CMA-secure) Signature schemes exist assuming that collision-resistant hash functions exist.

(Many-time) Signature Scheme

In four+ steps

Step 1. Stateful, Growing Signatures. Idea: Signature ***Chains***

Step 2. How to Shrink the signatures. Idea: Signature ***Trees***

Step 3. How to Shrink Alice's storage.
Idea: ***Pseudorandom Trees***

Step 4. How to make Alice stateless.
Idea: ***Randomization***

Step 5 (*optional*). How to make Alice stateless and deterministic. Idea: ***PRFs.***

Step 1: Stateful Many-time Signatures

Idea: Signature Chains.

Alice starts with a secret signing key sk_0

When signing a message m_1:

1. Generate a new pair $(\mathsf{sk}_1, \mathsf{pk}_1)$
2. Produce signature $\sigma_1 \leftarrow \mathsf{Sign}(\mathsf{sk}_0, m_1 \| \mathsf{pk}_1)$
3. Output $\tau_1 = \mathsf{pk}_1 \| \sigma_1$ as the signature.
4. Remember $\mathsf{pk}_1 \| m_1 \| \sigma_1$ as well as sk_1.

To verify a signature $\tau_1 = \mathsf{pk}_1 \| \sigma_1$ for message m_1:

Run $\mathsf{Verify}(\mathsf{pk}_0, \mathsf{pk}_1 \| m_1, \sigma_1) = 1$

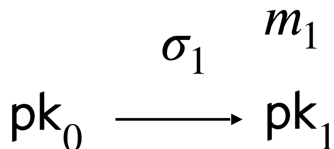
Step 1: Stateful Many-time Signatures

Idea: Signature Chains.

Alice starts with a secret signing key sk_0

When signing a message m_1:

1. Generate a new pair $(\mathsf{sk}_1, \mathsf{pk}_1)$
2. Produce signature $\sigma_1 \leftarrow \mathsf{Sign}(\mathsf{sk}_0, m_1 \| \mathsf{pk}_1)$
3. Output $\tau_1 = \mathsf{pk}_1 \| \sigma_1$ as the signature.
4. Remember $\mathsf{pk}_1 \| m_1 \| \sigma_1$ as well as sk_1.



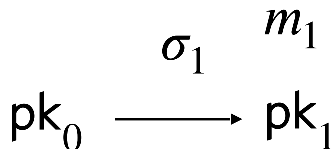
Step 1: Stateful Many-time Signatures

Idea: Signature Chains.

Alice starts with a secret signing key sk_0

When signing a message the next message m_2:

1. Generate a new pair $(\mathsf{sk}_2, \mathsf{pk}_2)$
2. Produce signature $\sigma_2 \leftarrow \mathsf{Sign}(\mathsf{sk}_1, m_2 \| \mathsf{pk}_2)$
3. Output ???
4.



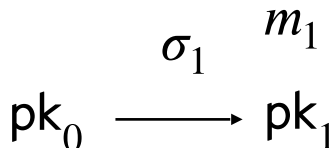
Step 1: Stateful Many-time Signatures

Idea: Signature Chains.

Alice starts with a secret signing key sk_0

When signing a message the next message m_2:

1. Generate a new pair (sk_2, pk_2)
2. Produce signature $\sigma_2 \leftarrow \mathrm{Sign}(sk_1, m_2 || pk_2)$
3. Output $pk_2 || \sigma_2$?
4.



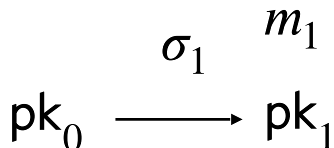
Step 1: Stateful Many-time Signatures

Idea: Signature Chains.

Alice starts with a secret signing key sk_0

When signing a message the next message m_2:

1. Generate a new pair $(\mathsf{sk}_2, \mathsf{pk}_2)$
2. Produce signature $\sigma_2 \leftarrow \mathsf{Sign}(\mathsf{sk}_1, m_2 \| \mathsf{pk}_2)$
3. Output $\mathsf{pk}_1 \| \mathsf{pk}_2 \| \sigma_2$??
4.



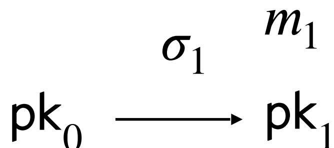
Step 1: Stateful Many-time Signatures

Idea: Signature Chains.

Alice starts with a secret signing key sk_0

When signing a message the next message m_2:

1. Generate a new pair $(\mathsf{sk}_2, \mathsf{pk}_2)$
2. Produce signature $\sigma_2 \leftarrow \mathsf{Sign}(\mathsf{sk}_1, m_2 \| \mathsf{pk}_2)$
3. Output $(\mathsf{pk}_1 \| m_1 \| \sigma_1) \| \mathsf{pk}_2 \| \sigma_2$
4. (additionally) remember $\mathsf{pk}_2 \| m_2 \| \sigma_2$ and sk_2.



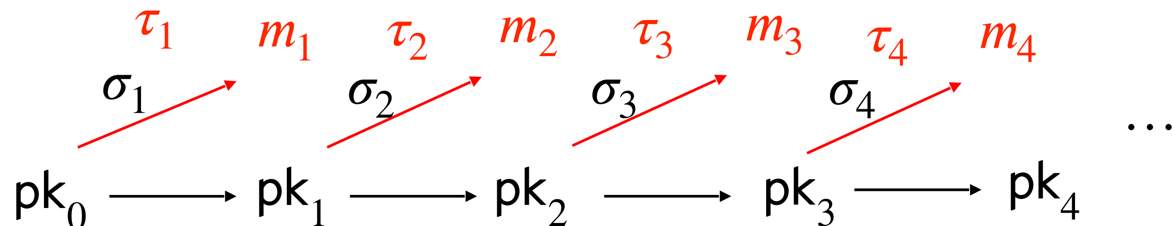
Step 1: Stateful Many-time Signatures

Idea: Signature Chains.

Two major problems:

1. Alice is *stateful*: Alice needs to remember a whole lot of things, $O(T)$ information after T steps.

2. The *signatures grow*: Length of the signature of the T-th message is $O(T)$.



(Many-time) Signature Scheme

In four+ steps

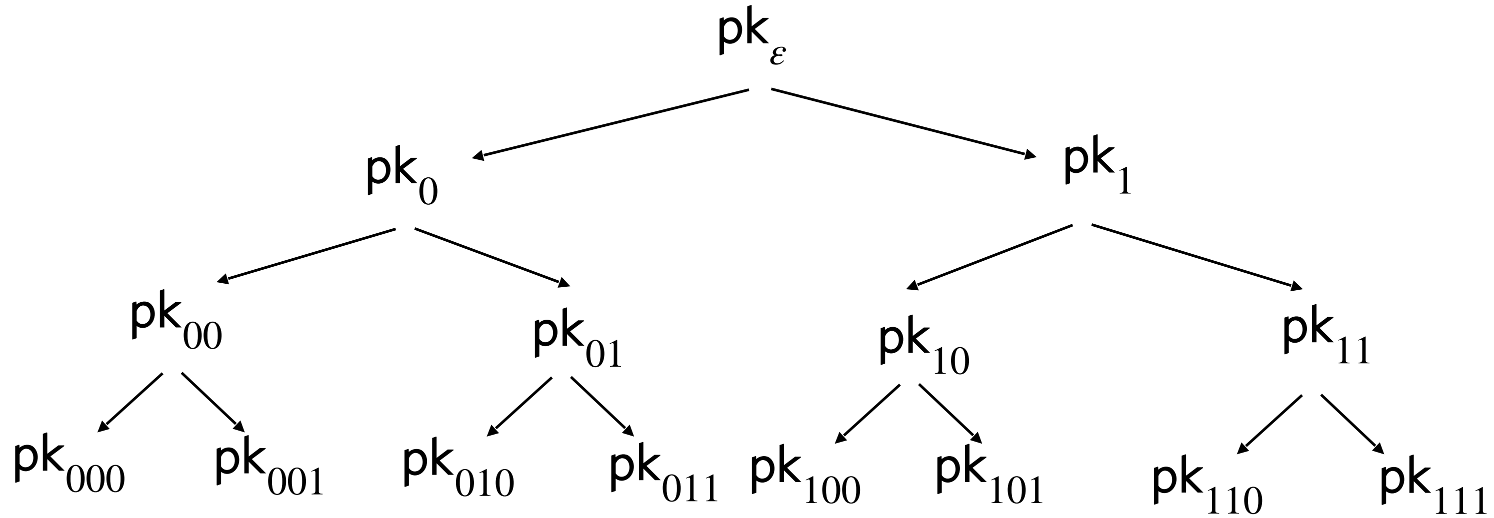
Step 1. Stateful, Growing Signatures. Idea: Signature ***Chains***

Step 2. How to Shrink the signatures. Idea: Signature ***Trees***

Step 2: Shrinking signatures

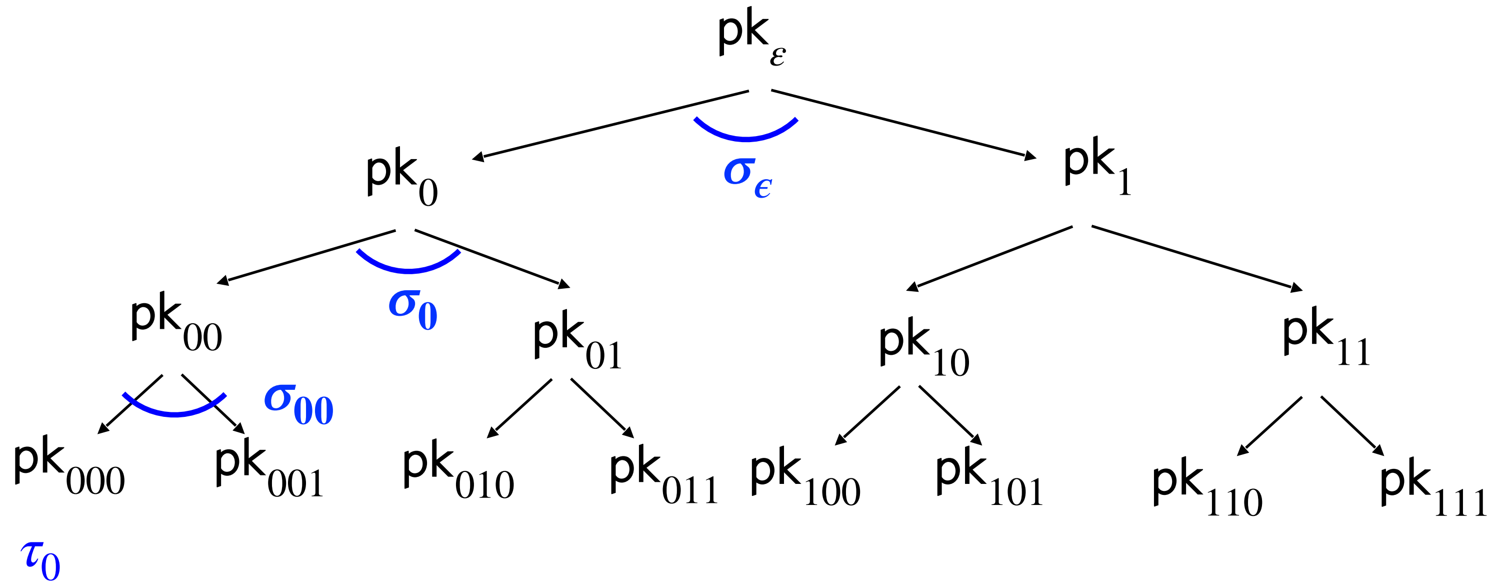
$\mathsf{pk}_{\mathcal{E}}$

Step 2: Shrinking signatures



Alice (the *stateful* signer) computes many $(\mathsf{pk}, \mathsf{sk})$ pairs and arranges them in a tree of depth = sec. param. λ

Step 2: Shrinking signatures

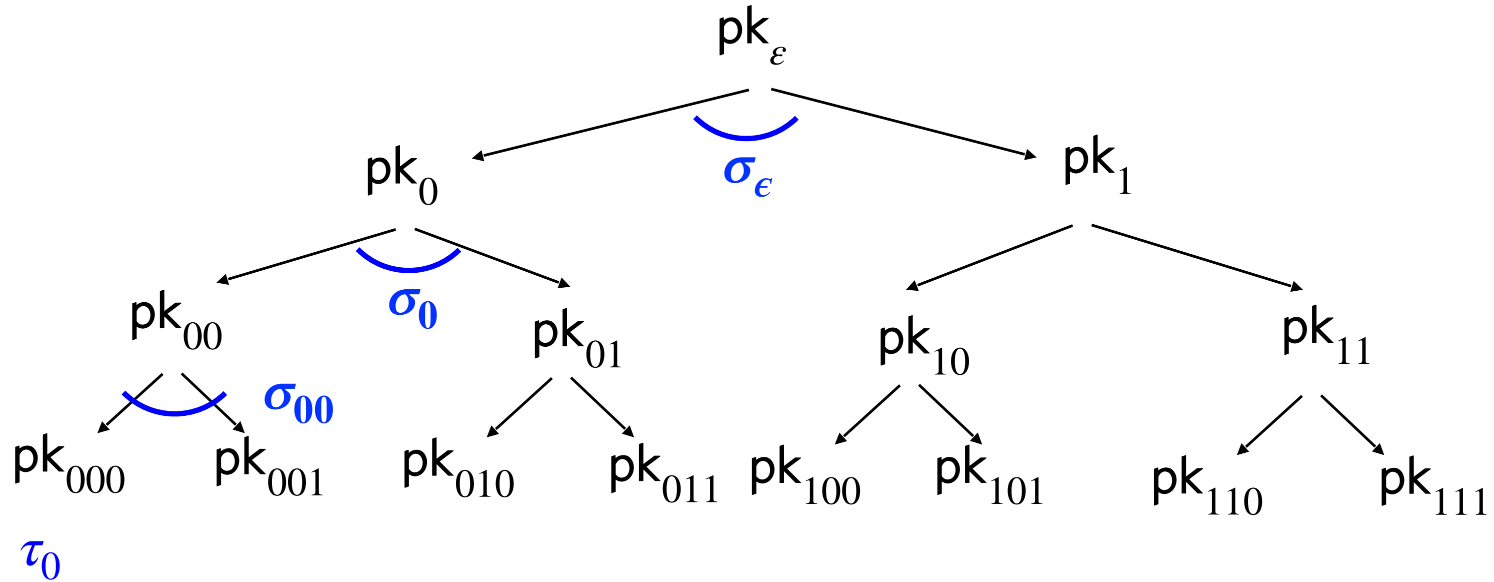


Signature of the zeroth message m_0:

Use sk_{000} to sign m_0.

"Authenticate" pk_{000} using the "signature path".

Step 2: Shrinking signatures

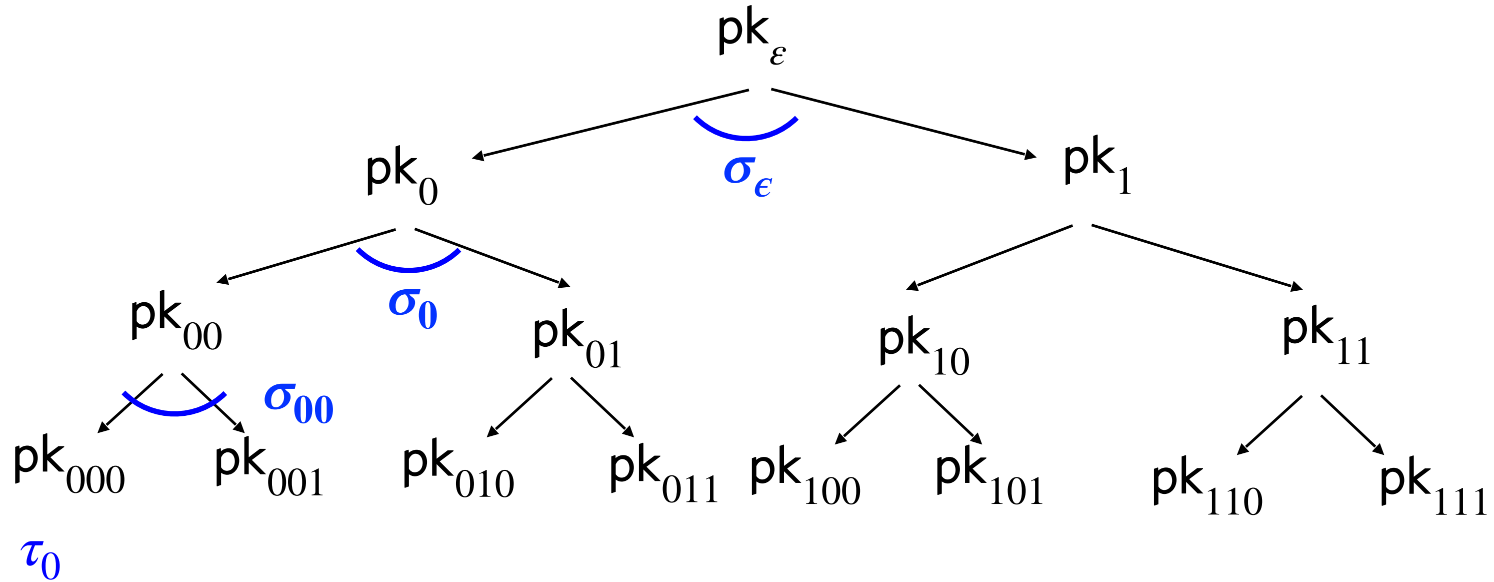


Signature of the zeroth message m_0:

$\big(\sigma_\varepsilon \leftarrow \mathsf{Sign}(\mathsf{sk}_\epsilon, \mathsf{pk}_0 \| \mathsf{pk}_1),\ \sigma_0 \leftarrow \mathsf{Sign}(\mathsf{sk}_0, \mathsf{pk}_{00} \| \mathsf{pk}_{01}),$

$\sigma_{00} \leftarrow \mathsf{Sign}(\mathsf{sk}_{00}, \mathsf{pk}_{000} \| \mathsf{pk}_{001}),\ \tau_0 \leftarrow \mathsf{Sign}(\mathsf{sk}_{000}, m_0)\big)$

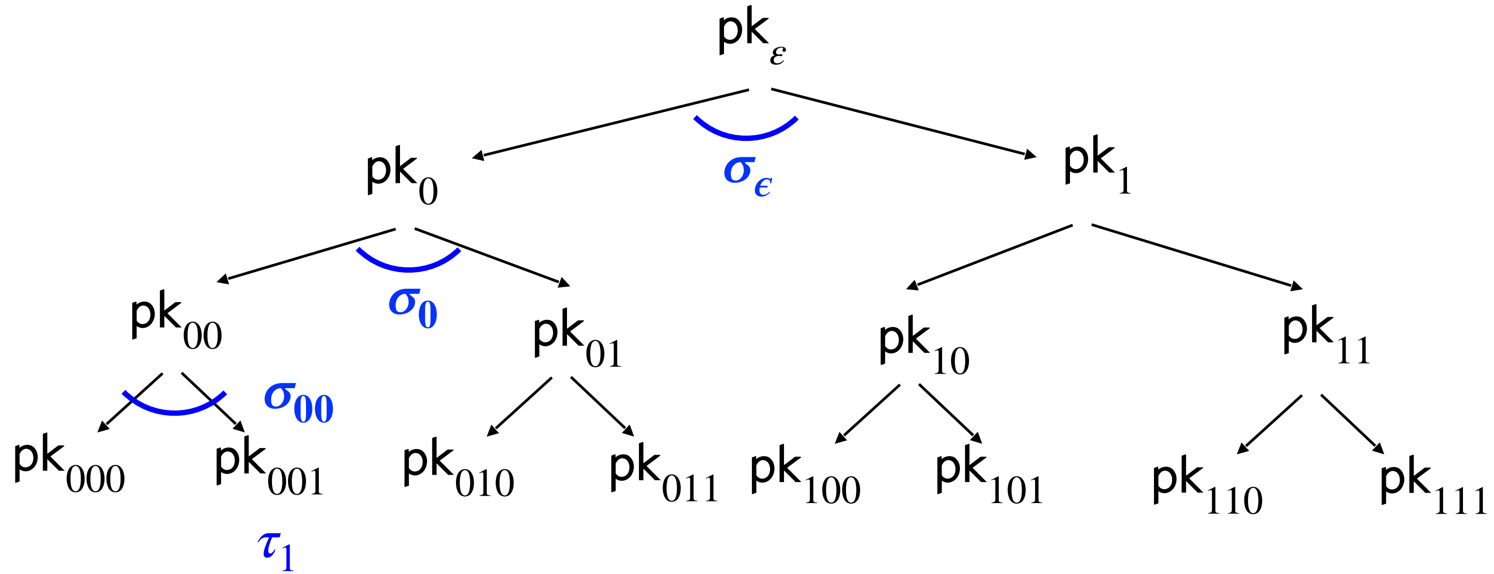
Step 2: Shrinking signatures



Signature of the zeroth message m_0:

(Authentication path for pk_{000}, $\tau_0 \leftarrow \mathsf{Sign}(\mathsf{sk}_{000}, m_0)$)

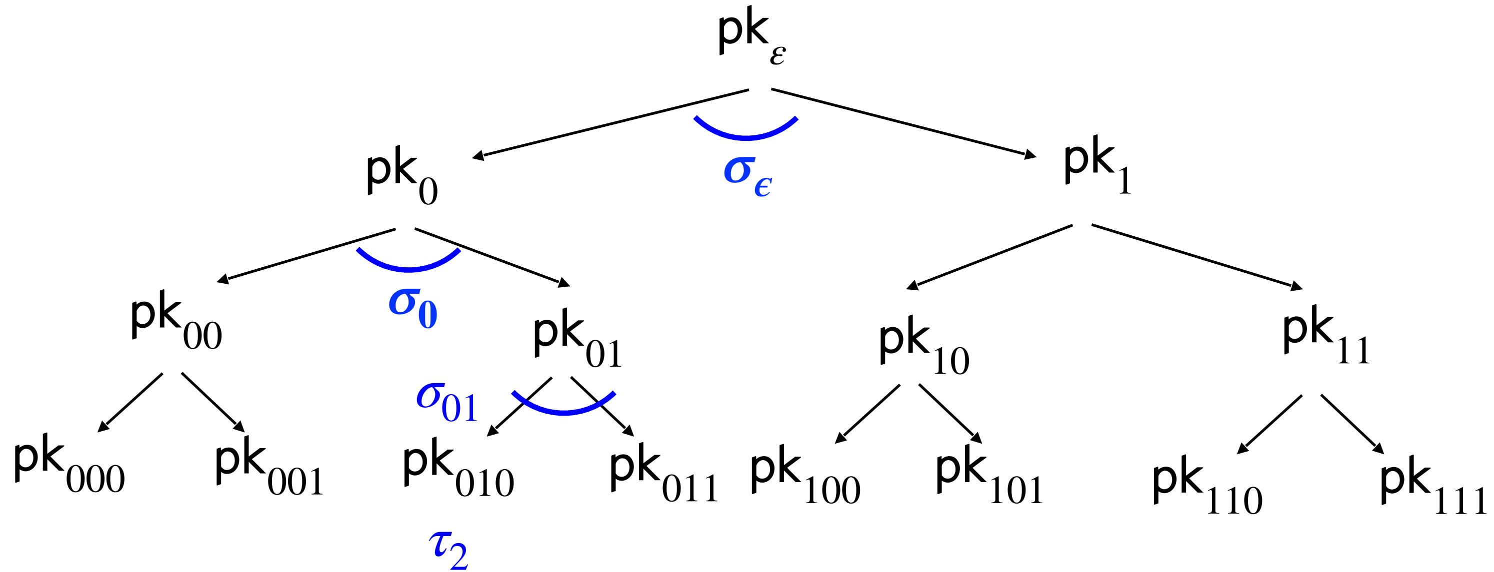
Step 2: Shrinking signatures



Signature of message m_1

(Authentication path for pk_{001}, $\tau_1 \leftarrow \mathsf{Sign}(sk_{001}, m_1)$)

Step 2: Shrinking signatures



Signature of message m_2

(Authentication path for pk_{010}, $\tau_2 \leftarrow \text{Sign}(sk_{010}, m_2)$)

Step 2: Shrinking signatures

pk_ε

pk_0 $\boldsymbol{\sigma_\epsilon}$ pk_1

pk_{00} $\boldsymbol{\sigma_0}$ pk_{01} pk_{10} pk_{11}

σ_{01}

pk_{000} pk_{001} pk_{010} pk_{011} pk_{100} pk_{101} pk_{110} pk_{111}

τ_2

GOOD NEWS:

Each verification key (incl. at the leaves) is used only once, so one-time security suffices!

Step 2: Shrinking signatures

pk_ε

pk_0 $\boldsymbol{\sigma_\epsilon}$ pk_1

pk_{00} $\boldsymbol{\sigma_0}$ pk_{01} pk_{10} pk_{11}

σ_{01}

pk_{000} pk_{001} pk_{010} pk_{011} pk_{100} pk_{101} pk_{110} pk_{111}

τ_2

GOOD NEWS:

Signatures consist of λ one-time signatures and do now grow with time!

Step 2: Shrinking signatures

pk_ε

σ_ϵ

pk_0 pk_1

σ_0

pk_{00} pk_{01} pk_{10} pk_{11}

σ_{01}

pk_{000} pk_{001} pk_{010} pk_{011} pk_{100} pk_{101} pk_{110} pk_{111}

τ_2

BAD NEWS:

Signer generates and keeps the entire ($\approx\ 2^\lambda$-size) signature tree in memory!

(Many-time) Signature Scheme

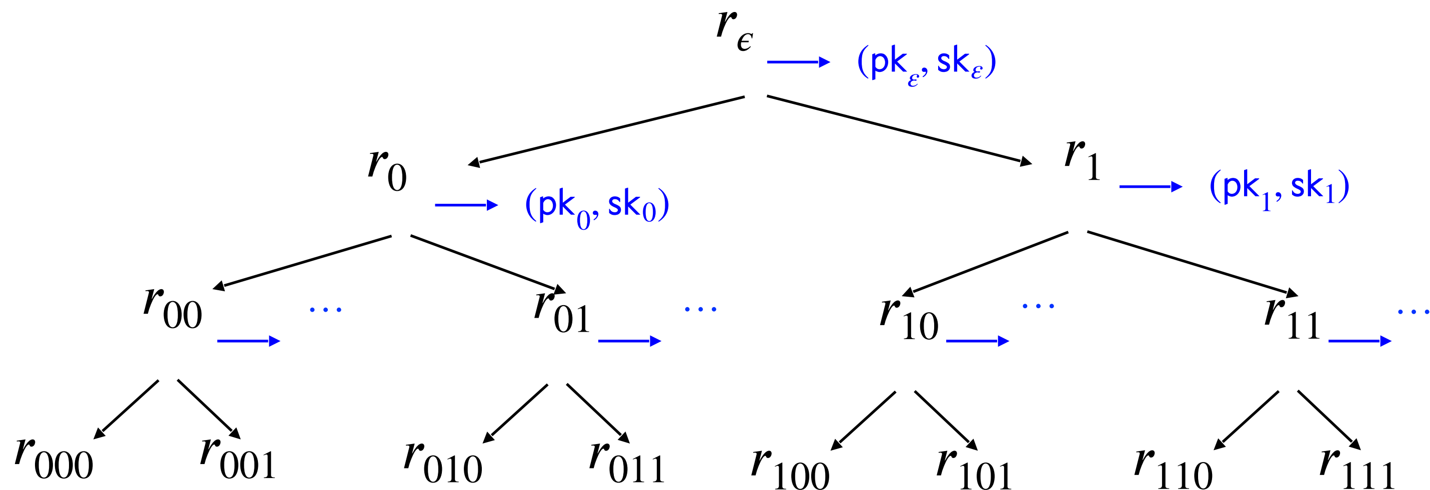
In four+ steps

Step 1. Stateful, Growing Signatures. Idea: Signature ***Chains***

Step 2. How to Shrink the signatures. Idea: Signature ***Trees***

Step 3. How to Shrink Alice's storage.
Idea: ***Pseudorandom Trees***

Step 3: Pseudorandom signature trees



Tree of pseudorandom values:

The signing key is a PRF key k.

"Lazily" populate the nodes with $r_x := \mathsf{PRF}(k, x)$.

Use r_x to derive the keys $(\mathsf{pk}_x, \mathsf{sk}_x) \leftarrow \mathsf{Gen}(1^\lambda; r_x)$.

Step 3: Pseudorandom signature trees

$$\mathsf{pk}_\varepsilon$$

pk_0 pk_1

pk_{00} pk_{01} pk_{10} pk_{11}

pk_{000} pk_{001} pk_{010} pk_{011} pk_{100} pk_{101} pk_{110} pk_{111}

GOOD NEWS:

Short signatures and small storage for the signer

Step 3: Pseudorandom signature trees

pk_ε

pk_0 pk_1

pk_{00} pk_{01} pk_{10} pk_{11}

pk_{000} pk_{001} pk_{010} pk_{011} pk_{100} pk_{101} pk_{110} pk_{111}

τ_2

m_2

BAD NEWS:

Signer needs to keep a counter indicating which ***leaf*** (which tells her which secret key) to use next.

(Many-time) Signature Scheme

In four+ steps

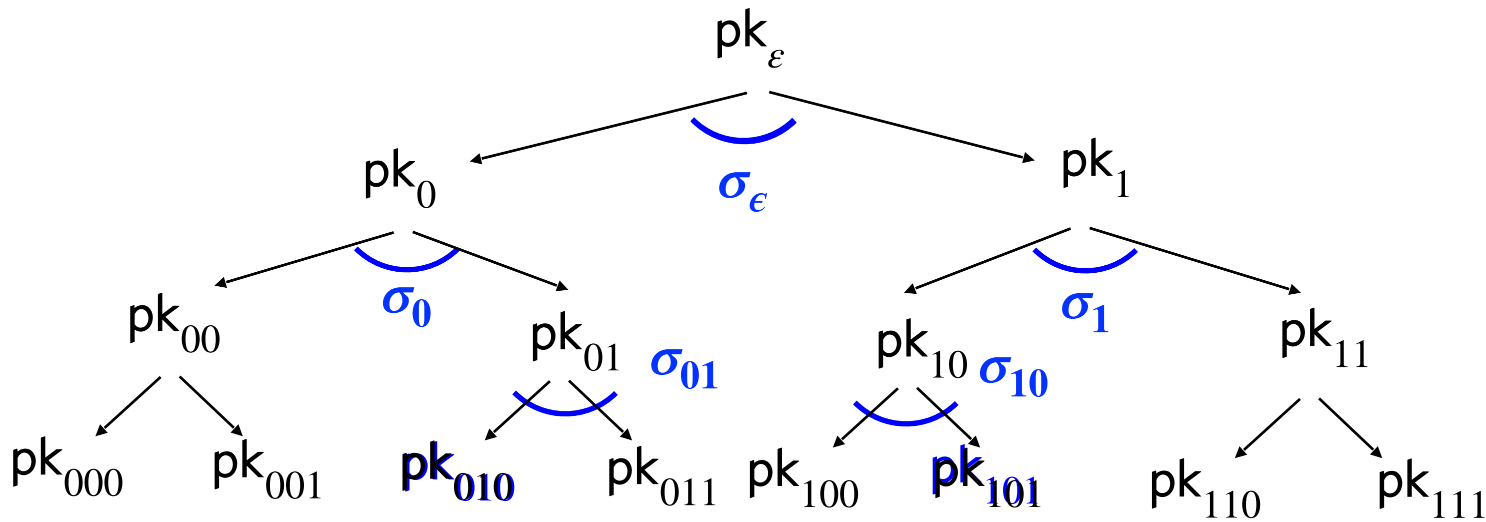
Step 1. Stateful, Growing Signatures. Idea: Signature ***Chains***

Step 2. How to Shrink the signatures. Idea: Signature ***Trees***

Step 3. How to Shrink Alice's storage.
Idea: ***Pseudorandom Trees***

Step 4. How to make Alice stateless.
Idea: ***Randomization***

Step 4: Statelessness via randomization



Signature of a message m:

Pick a **random** leaf r. Use pk_r to sign m.

$\sigma_r \leftarrow \mathsf{Sign}(\mathsf{sk}_r, m)$

Output (r, σ_r, authentication path for pk_r)

Step 4: Statelessness via randomization

pk_ε

σ_ϵ

pk_0 pk_1

σ_0 σ_1

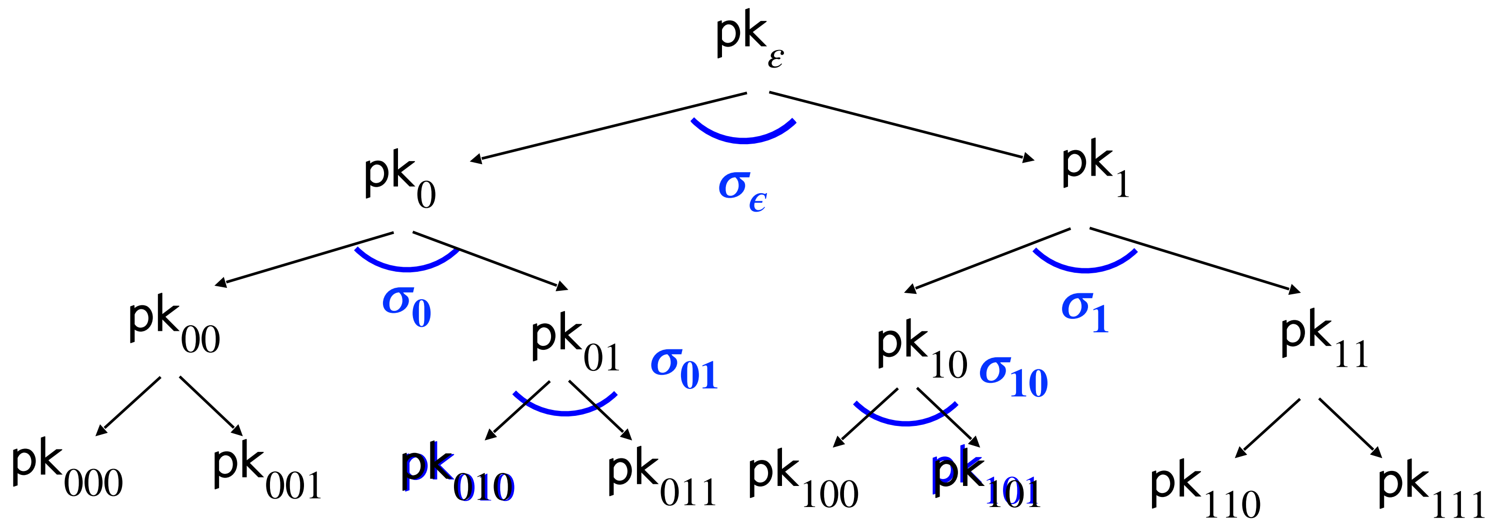
pk_{00} pk_{01} σ_{01} pk_{10} σ_{10} pk_{11}

pk_{000} pk_{001} pk_{010} pk_{011} pk_{100} pk_{101} pk_{110} pk_{111}

GOOD NEWS:

No need to keep state.

Step 4: Statelessness via randomization



Key Idea:

If the signer produces q signatures, the probability she picks the same leaf twice is $\leq q^2/2^\lambda$.

(Many-time) Signature Scheme

In four+ steps

Step 1. Stateful, Growing Signatures. Idea: Signature ***Chains***

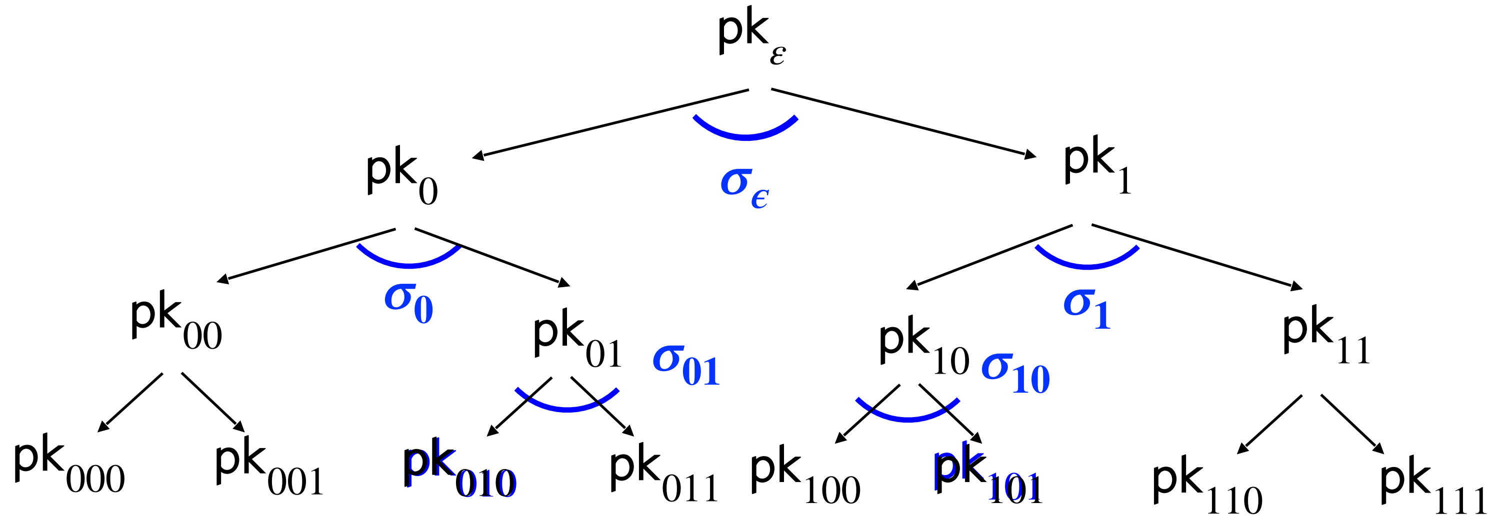
Step 2. How to Shrink the signatures. Idea: Signature ***Trees***

Step 3. How to Shrink Alice's storage.
Idea: ***Pseudorandom Trees***

Step 4. How to make Alice stateless.
Idea: ***Randomization***

Step 5 (*optional*). How to make Alice stateless and deterministic. Idea: ***PRFs.***

Step 4: Deterministic signer



Key Idea:

Generate r pseudo-randomly.

Have another PRF key k' and let $r = \text{PRF}(k', m)$

That’s it for the construction.

Digital Signature Construction

- Historically regarded as inefficient; therefore, never used in practice.

- However, this signature scheme (or variants thereof) are now called “hash-based signatures” and seeing a re-emergence as a candidate post-quantum secure signature scheme. E.g. https://sphincs.org/

“Vanilla” RSA Signatures

Start with any trapdoor permutation, e.g. RSA.

Gen(1^λ): Pick primes (p, q) and let $N = pq$. Pick e relatively prime to $\varphi(N)$ and let $d = e^{-1} \pmod{\varphi(N)}$.

sk $= (p, q, d)$ and pk $= (N, e)$

Sign(sk, m): Output signature $\sigma = m^d \mod N$

Verify(pk, m, σ): Check if $\sigma^e = m \mod N$

Problem: Existentially forgeable!

"Vanilla" RSA Signatures

Sign(sk, m): Output signature $\sigma = m^d \mod N$

Verify(pk, m, σ): Check if $\sigma^e = m \mod N$

Problem: Existentially forgeable!

Attack: Pick random σ and output ($m = \sigma^e$, σ) as forgery.

Problem: Malleable!

Attack: Given signature of m, you can produce signature for $2^e * m, 3^e * m, \ldots, m^2, m^3, \ldots$

"Vanilla" RSA Signatures

Sign(sk, m): Output signature $\sigma = m^d \mod N$

Verify(pk, m, σ): Check if $\sigma^e = m \mod N$

Fundamental Issues:

1. Can "reverse-engineer" the message starting from the signature (Attack 1)

2. Algebraic structure allows malleability (Attack 2)

How to Fix Vanilla RSA

Start with any trapdoor permutation, e.g. RSA.

Gen(1^λ): Pick primes (p, q) and let $N = pq$. Pick e relatively prime to $\varphi(N)$ and let $d = e^{-1} \pmod{\varphi(N)}$.

sk $= (p, q, d)$ and pk $= (N, e, H)$

Sign(sk, m): Output signature $\sigma = H(m)^d \mod N$

Verify(pk, m, σ): Check if $\sigma^e = H(m) \mod N$

So, what is H? Some very complicated "hash" function.

How to Fix Vanilla RSA

Start with any trapdoor permutation, e.g. RSA.

Gen(1^λ): Pick primes (p, q) and let $N = pq$. Pick e relatively prime to $\varphi(N)$ and let $d = e^{-1} \pmod{\varphi(N)}$.

$$\text{sk} = (p, q, d) \text{ and } \text{pk} = (N, e, H)$$

Sign(sk, m): Output signature $\sigma = H(m)^d \mod N$

Verify(pk, m, σ): Check if $\sigma^e = H(m) \mod N$

***H* should be at least one-way to prevent Attack #1.**

How to Fix Vanilla RSA

Start with any trapdoor permutation, e.g. RSA.

Gen(1^λ): Pick primes (p, q) and let $N = pq$. Pick e relatively prime to $\varphi(N)$ and let $d = e^{-1} \pmod{\varphi(N)}$.

sk $= (p, q, d)$ and pk $= (N, e, H)$

Sign(sk, m): Output signature $\sigma = H(m)^d \mod N$

Verify(pk, m, σ): Check if $\sigma^e = H(m) \mod N$

Hard to "algebraically manipulate" $H(m)$ into $H($related $m')$.

(to prevent Attack #2)

How to Fix Vanilla RSA

Start with any trapdoor permutation, e.g. RSA.

Gen(1^λ): Pick primes (p, q) and let $N = pq$. Pick e relatively prime to $\varphi(N)$ and let $d = e^{-1} \pmod{\varphi(N)}$.

$$\text{sk} = (p, q, d) \text{ and pk} = (N, e, H)$$

Sign(sk, m): Output signature $\sigma = H(m)^d \mod N$

Verify(pk, m, σ): Check if $\sigma^e = H(m) \mod N$

Collision-resistance does not seem to be enough.

(Given a CRHF $H(m)$, you may be able to produce $H(m')$ for related m'.)