Modern Cryptography

600.442

Lecture #18

Dr. Christopher Pappacena

Fall 2013

The Digital Signature Standard (DSS)

- Published by NIST in 1991.
- Security is based on the difficulty of the discrete logarithm problem.
- No known security proof, even in the random oracle model.

DSS

Let $\mathcal{G}$ be a group generation algorithm that, on input n, returns (p, q, g) where p and q are primes with $\|q\| = n$, q exactly divides $p - 1$, and g is a generator for the subgroup of $\mathbb{Z}_p^*$ of order q.

- Gen(n) calls $\mathcal{G}(n)$ to obtain (p, q, g), chooses $x \leftarrow \mathbb{Z}_q$ at random, and sets $y = g^x \pmod p$. The public key is (H, p, q, g, y) and the private key is (H, p, q, g, x), where $H : \{0,1\}^* \rightarrow \mathbb{Z}_q$ is a hash function.

- To sign m, choose $k \leftarrow \mathbb{Z}_q$, set $r = [g^k \pmod p] \pmod q$, compute $s = (H(m) + xr)k^{-1} \pmod q$, and output (r, s).

- To verify (m, r, s), compute $u_1 = H(m)s^{-1} \pmod q$ and $u_2 = rs^{-1} \pmod q$ and verify that $r = [g^{u_1} y^{u_2} \pmod p] \pmod q$.

So far we haven't given any actual signature schemes that we can prove secure.

In fact, it is hard to produce such schemes. To get there, we will begin with a weaker type of scheme called a *one-time signature scheme*.

In this setup, the adversary $\mathcal{A}$ is allowed to ask for the signature of a *single* message m' before producing his putative forgery (m, σ).

We write $\mathsf{SigForge}^{\mathsf{1time}}_{\mathcal{A},\Pi}(n) = 1$ if $\mathcal{A}$ succeeds.

Lamport's One-Time Signature Scheme

Let f be a one-way function.

- Gen(n): Choose random $x_{i,0}, x_{i,1} \in \{0,1\}^n$ for $1 \leq i \leq \ell$ and compute $y_{i,b} = f(x_{i,b})$. Set $\mathtt{pk} = \vec{y}$ and $\mathtt{sk} = \vec{x}$.

- To sign the message $m = (m_1, \ldots, m_\ell)$, output

$$\sigma = (x_{1,m_1}, x_{2,m_2}, \ldots, x_{\ell,m_\ell}).$$

- To verity (m, σ), check that $f(\sigma_i) = y_{i,m_i}$ for all i.

Theorem: If f is a one-way function then Π is a one-time signature scheme for messages of length ℓ.

Proof: Let $\mathcal{A}$ be an adversary which succeeds with probability $\epsilon(n)$. We define an algorithm $\mathcal{I}$ for inverting f which calls $\mathcal{A}$ as a subroutine.

- Given y, $\mathcal{I}$ chooses i, b at random and set $y_{i,b} = y$. Otherwise, $\mathcal{I}$ chooses $x_{i,b}, y_{i,b}$ as above.

- $\mathcal{I}$ calls $\mathcal{A}$ as a subroutine with input $\vec{y}$. If $\mathcal{A}$ asks for the signature of m' with $m'_i = b$, stop. Otherwise, return correct σ.

- If $\mathcal{A}$ outputs (m, σ) with $m_i = b$, then $\mathcal{I}$ returns $\sigma_{i,b}$. Otherwise $\mathcal{I}$ returns a random value.

Success Probability for $\mathcal{I}$

Note that $\mathcal{I}$ succeeds whenever $\mathcal{A}$ outputs a successful forgery with $m_i = b$.

The probability that $m_i = b$, *conditioned on the forgery being successful*, is at least $1/2\ell$. So,

$$\Pr[\text{Invert}_{\mathcal{I},f}(n) = 1] \geq \frac{\epsilon(n)}{2\ell(n)}.$$

Since $\ell(n)$ is a polynomial in n and f is one-way, $\epsilon(n)$ is negligible.

Stateful Signature Schemes

A *stateful signature scheme* outputs an initial state s_0 in addition to $\mathtt{pk}, \mathtt{sk}$.

Given a message m and state s_{i-1}, $\mathsf{Sign}_{\mathtt{sk},s_{i-1}}(m)$ updates the state to s_i and outputs signature σ_i.

We require $\mathsf{Verify}_{\mathtt{pk}}(\mathsf{Sign}_{\mathtt{sk},s_{i-1}}(m)) = 1$, except possibly with negligible probability.

Note that s_i is not needed to verify the signature and in some cases it must be kept secret.

Example: ℓ-time Signatures

- Generate one-time signature schemes $(\mathtt{pk}_i, \mathtt{sk}_i)$ for $i \leq i \leq \ell$. The initial state is $s_0 = 1$.

- To sign message m with state i, use $\mathtt{sk}_i$ and update the state to $i + 1$.

- To verify (m, σ), check if σ is a valid signature for m with respect to *some* $\mathtt{pk}_i$.

Note that we need to choose ℓ ahead of time.

Example: Signature Chains

Initially, generate $(\mathtt{pk}_1, \mathtt{sk}_1)$ for a one-time signature scheme. The initial state is empty.

When a message m_1 needs to be signed, generate new one-time signature keys $(\mathtt{pk}_2, \mathtt{sk}_2)$, sign the message $m_1||\mathtt{pk}_2$, and publish $(m_1, \sigma_1, \mathtt{pk}_2)$. The internal state is updated to $(m_1, \sigma_1, \mathtt{pk}_2, \mathtt{sk}_2)$.

To sign message m_i, generate new keys $(\mathtt{pk}_{i+1}, \mathtt{sk}_{i+1})$, set

$$\sigma_i = \mathsf{Sign}_{\mathtt{pk}_i}(m_i||\mathtt{pk}_{i+1})$$

and output $(m_i, \sigma_i, \mathtt{pk}_{i+1}, \{m_j, \sigma_j, \mathtt{pk}_{j+1}\}_{j \leq i-1})$. The internal state is $(\{m_j, \sigma_j, \mathtt{pk}_{j+1}, \mathtt{sk}_{j+1}\}_{j \leq i})$.

Validating the Chain

To validate a signature chain, use the fact that $\sigma_j = \mathsf{Sign}_{\mathtt{sk}_j}(m_j || \mathtt{pk}_{j+1})$. All the information needed to validate these steps is included in the i^{th} signature.

Intuitively, the signature chain is existentially unforgeable: Forging a signature requires forging links in the chain, and each link is signed using a different secure one-time signature scheme.

We won't prove this formally.

Issues

- We must be able to sign messages that are strictly longer than the key. So Lamport's one-time signature scheme doesn't work unless we use "hash and sign".

- The signature σ_i for message m_i contains all messages m_j for $j < i$.

- The size of the signature is linear in the number of previous signatures issued.

Tree-Based Signatures

Fix a binary tree of depth n, with leaf nodes corresponding to all possible n-bit messages. Fix a one-time signature scheme Π and a pair of keys $\mathtt{pk}_\epsilon, \mathtt{sk}_\epsilon$ corresponding to the root.

To sign a message m:

- Generate keys $\mathtt{pk}_i, \mathtt{sk}_i$ for each node on the path from the root to the leaf labeled m.
- Certify the path to m by signing $\mathtt{pk}_{w0}||pk_{w1}$ with key $\mathtt{sk}_w$ for each prefix w of m.
- Finally, sign m with private key $\mathtt{sk}_m$.

Details

For an n-bit message $m = (m_1, \ldots, m_n)$, let $m_{\leq j}$ denote the prefix $(m_1, \ldots, m_j)$.

At level j of the tree, the signature is $\sigma_{m_{\leq j}} = \mathsf{Sign}_{\mathtt{sk}_{m_{\leq j}}}(\mathtt{pk}_{m_{\leq j}0} || \mathtt{pk}_{m_{\leq j}1})$.

The final signature is $(\{\sigma_{m_{\leq j}}, \mathtt{pk}_{m_{\leq j}0} || \mathtt{pk}_{m_{\leq j}1}\}_{j \leq n-1}, \sigma_m)$.

Remarks

- The state of the signature is the full list of keys generated for all messages that have been signed so far.

- If a new message passes through a node which already has keys associated to it, the keys are reused.

- The total number of messages signed is polynomial in n, so even though the size of the tree is exponential in n only polynomially-many nodes are ever visited.

Finally, a Digital Signature!

Theorem: If Π is a secure one-time signature scheme, then the tree-based signature is existentially unforgeable under an adaptive chosen-message attack.

Proof: Denote the tree-based signature by Π^*, let $\mathcal{A}^*$ be an adversary who can break Π^* with advantage $\epsilon(n)$, and let $\ell^*(n)$ be an upper bound on the number of oracle calls made by $\mathcal{A}^*$.

Set $\ell(n) = n\ell^*(n) + 1$, an upper bound on the number of public keys needed to answer $\mathcal{A}^*$'s oracle queries.

Proof, II

Let $\mathcal{A}$ be an adversary attacking Π which uses $\mathcal{A}^*$ as a subroutine:

- $\mathcal{A}$ chooses a random index $i \leftarrow \{1, \ldots, \ell\}$, sets $\mathtt{pk}^i = \mathtt{pk}$, and computes $(\mathtt{pk}^j, \mathtt{sk}^j)$ via Gen(n) for all $j \neq i$.

- $\mathcal{A}$ calls $\mathcal{A}^*$ as a subroutine with input $\mathtt{pk}_\epsilon = \mathtt{pk}^1$. When $\mathcal{A}^*$'s query passes through a prefix w, $\mathcal{A}$ uses the next two unused values $\mathtt{pk}^j, \mathtt{pk}^{j+1}$ for $\mathtt{pk}_{w0}, \mathtt{pk}_{w1}$.

- If $\mathcal{A}$ needs to use the secret key $\mathtt{sk}^i$ corresponding to $\mathtt{pk}^i$, it requests the appropriate signature from its oracle.

Proof, III

$\mathcal{A}^*$ outputs the signature $\sigma = (\{\sigma'_{m_{\leq j}}, \mathtt{pk}'_{m_{\leq j}0} || \mathtt{pk}'_{m_{\leq j}1}\}_{j \leq n-1}, \sigma'_m)$ for a message m.

If σ is valid, let j be the smallest index for which $\mathtt{pk}'_{m_{\leq j},0} \neq \mathtt{pk}_{m_{\leq j},0}$ or $\mathtt{pk}'_{m_{\leq j},1} \neq \mathtt{pk}_{m_{\leq j},1}$. Let i' be the index such that $\mathtt{pk}^{i'} = \mathtt{pk}_{m_{\leq j}} = \mathtt{pk}'_{m_{\leq j}}$.

If no such j exists, then $\mathtt{pk}_m = \mathtt{pk}'_m$. In this case let i' be the index with $\mathtt{pk}^{i'} = \mathtt{pk}_m$.

Proof, IV

In the first case, if $i = i'$, then $\mathcal{A}$ outputs $(\mathtt{pk}'_{m_{\leq j},0}||\mathtt{pk}'_{m_{\leq j},1}, \sigma'_{m_{\leq j}})$. Otherwise $\mathcal{A}$ outputs a random message and signature.

In the second case, if $i = i'$ then $\mathcal{A}^*$ never requested a signature with respect to $\mathtt{pk}^i = \mathtt{pk}_m$, yet σ'_m is a valid signature. So $\mathcal{A}$ outputs (m, σ'_m).

Conditioned on $\mathcal{A}^*$ outputting a forgery, $\mathcal{A}$ outputs a forgery with probability $1/\ell(n)$. So the probability that $\mathcal{A}$ succeeds is $\epsilon(n)/\ell(n)$. Since Π is secure, $\epsilon(n)$ is negligible.

Making it Stateless

The tree-based scheme requires maintaining state. If we fix a pseudo-random function F with keys k, k', we can make it stateless.

If we need to generate a pair of keys at prefix w, we use the pseudo-random bits $F_k(w)$ as the source of random for generating $\mathtt{pk}_w, \mathtt{sk}_w$..

Similarly, we use the pseudorandom bits $F_{k'}(w)$ for the random bits needed to construct the signature $\sigma_w = \mathsf{Sign}_{\mathtt{sk}_w}(\mathtt{pk}_{w0}||\mathtt{pk}_{w1})$.

This way, the intermediate states can be constructed on the fly and do not need to be stored. The deterministic nature of the pseudorandom function ensures we always construct the same values at a given node.

The 400 Pound Gorilla

Although we have not made it explicit, we have assumed some amount of *authentication* on the communication channels used for public-key encryption and key exchange.

- For key exchange, how does Alice know that the person she is communicating with is Bob?

- For public-key encryption, how does Bob know that Alice's public key actually belongs to Alice?

Man in the Middle

Suppose that Alice and Bob want to agree on a shared secret key for a private key algorithm Π using a key exchange protocol.

An eavesropper Eve can potentially insert herself *between* Alice and Bob during the exchange:

$$\texttt{Alice} \xrightarrow{A} \texttt{Eve} \xrightarrow{A'} \texttt{Bob}$$

$$\texttt{Alice} \xleftarrow{B'} \texttt{Eve} \xleftarrow{B} \texttt{Bob}$$

- Alice shares key $k = f(a, B')$ with Eve, thinking she is Bob.
- Bob shares key $k' = f(A', b)$ with Eve, thinking she is Alice.

Man in the Middle

What happens when Alice sends message m to Bob?

- She sends $c \leftarrow \mathsf{Enc}_k(m)$ to Eve, thinking its Bob.
- Eve computes $m = \mathsf{Dec}_k(c)$ and sends $c' \leftarrow \mathsf{Enc}_{k'}(m)$ to Bob.
- Bob receives c' and computes $m = \mathsf{Dec}_{k'}(c')$.

Alice and Bob are able to communicate successfully and, they believe, securely, but Eve gets a copy of every message.

Even worse, Eve can *modify* the message if she wants!

Remarks

- While we have described man-in-the-middle attacks against key exchange protocols, they also apply to public-key encryption schemes: Eve simply swaps her public key for Alice's.

- To defeat these attacks, Alice and Bob need a way to authenticate themselves.

- Digital signatures provide authentication but lead to a chicken and egg problem: How do you trust the identity of the signer?

Public Key Infrastructure

The basic idea is simple. Suppose that Bob trusts Charlie, and Charlie attests to the validity of Alice's public key. Then Bob can trust that Alice's public key really belongs to Alice.

How does Charlie do this? He signs Alice's public key with his own signing key!

If Charlie's signing keys are $\mathtt{pk}_C, \mathtt{sk}_C$, then Charlie publishes the signed message

$$\mathsf{Cert}_C(A) = \mathsf{Sign}_{\mathtt{sk}_C}(\texttt{"Alice's public key is pk}_A\texttt{"}).$$

Alice then gives $(\mathtt{pk}_A, \mathsf{Cert}_C(A))$ to Bob. Bob verifies $\mathsf{Cert}_C(A)$ with $\mathtt{pk}_C$ and then knows he can use $\mathtt{pk}_A$ to communicate with Alice.

Certificates and Certificate Authorities

Charlie's digital signature is called a *certificate*. Certificates serve to bind electronic data such as public keys to entities (people, companies, IP addresses, etc.).

Note that in reality, the certificate needs to do a much better job of uniquely identifying Alice.

Issuers of certificates are sometimes called *certificate authorities*, or CAs.

We're back to the chicken and the egg: How does Bob come to trust Charlie, and why does Charlie trust Alice?

Building Trust

Charlie trusts Alice because she authenticates herself to him using a trusted communication channel. For example, Alice can *physically* present herself to Charlie (with 2 forms of ID) along with her public key.

Building Trust

Bob trusts Charlie for one of several reasons:

- Bob has no choice.
- Everybody else trusts Charlie.
- Bob knows Charlie to be trustworthy.

Single CA

In some settings, e.g. a company's internal network, there is a *single* CA who validates everyone's keys.

Employees are authenticated in person when issued their keys and the CA maintains a list of signed keys.

Similar to a KDC in that the CA is a single point of failure – if the CA's secret key is compromised then it is possible to subvert the system.

One big advantage over the KDC: The CA does not participate in each communication – once it signs a key, it is not needed.

Multiple CAs

In other applications, there is no single CA. Instead there are multiple CAs and Alice may get her key signed by one, or several, of these authorities.

Alice sends $(\mathtt{pk}_A, \mathsf{Cert}_{C_1}(A), \ldots, \mathsf{Cert}_{C_t}(A))$ to Bob. If Bob trusts some C_i, he verifies the signature and trusts that $\mathtt{pk}_A$ belongs to Alice.

In this model, Charlie can *delegate* signing privileges to another CA, Derek:

$$\mathsf{Del}_C(D) = \mathsf{Sign}_{\mathtt{sk}_C}(\texttt{"Derek is trustworthy and has key pk}_D\texttt{"}).$$

If Alice obtains a certificate $\mathsf{Cert}_D(A)$ from Derek, she sends the triple $(\mathtt{pk}_A, \mathsf{Cert}_D(A), \mathsf{Del}_C(D))$ to Bob.

Multiple CAs in Practice

The multiple CA model is used for SSL/TLS. Your web browser comes pre-loaded with public signing keys for CAs that it trusts.

Some CAs are root certificate authorities, others have delegated signing privileges.

(Demo)

Web of Trust

In this *peer to peer model*, users of a system attest to the validity of *each other's* keys. There are no "authorities", and individuals decide whether to trust certificates issued by other users.

The more "independent" signatures Alice obtains, the more likely Bob is to trust her public key.

Maybe Alice can get a handful of people to collude with her, but the more signatures she gets, the less likely it is that all signers are malicious.

Web of Trust in Practice

The web of trust model is used by the encryption program PGP.

People hold "key signing parties" where people sign each other's keys.

Keys, along with certificates, can be uploaded and downloaded from sites, e.g. `pgp.mit.edu`.

Webs of trust work well for non-commercial uses.

Revocation

At times it is necessary to *revoke* a public key associated to an individual.

- The individual leaves the company.
- The private key is comprimised.
- The individual is found to be untrustworthy.

Revocation is also handled by certificate authorities.

Expiration Dates and Revocation Lists

For starters, public keys often come with *expiration dates*. When the expiration date passes, a new key needs to be issued and certified.

For example, a company may reissue keys every year.

To revoke keys, a CA signs a *revocation* which is publicly posted:

$$\mathsf{Rev}_C(A) = \mathsf{Sign}_{\mathtt{sk}_C}(\texttt{"Key pk}_A \texttt{ is revoked"}).$$

Bob must check if Alice's key has been revoked before using it.

Public Key Infrastructure

The details of how to issue, retrieve, and verify public keys, including the role of certificate authorities, is known as *public key infrastructure* or PKI.

Establishing a secure, robust, and efficient PKI is no easy task. For details, consult references on protocols and/or network security.