Lecture 11

Douglas Wikström
KTH Stockholm
dog@csc.kth.se

April 24, 2015

Signature Schemes

Signature Scheme

- Gen **generates a key pair** $(\mathsf{pk}, \mathsf{sk})$.
- Sig takes a secret key sk and a message m and **computes a signature** σ.
- Vf takes a public key pk, a message m, and a candidate signature σ, **verifies the candidate signature**, and outputs a single-bit verdict.

Existential Unforgeability

Definition. A signature scheme $(\mathsf{Gen}, \mathsf{Sig}, \mathsf{Vf})$ is **secure against existential forgeries** if for every polynomial time algorithm and a random key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^n)$,

$$\Pr\left[A^{\mathsf{Sig}_{\mathsf{sk}}(\cdot)}(\mathsf{pk}) = (m, \sigma) \wedge \mathsf{Vf}_{\mathsf{pk}}(m, \sigma) = 1 \wedge \forall i : m \neq m_i\right]$$

is negligible where m_i is the ith query to $\mathsf{Sig}_{\mathsf{sk}}(\cdot)$.

Proof of Knowledge of Exponent

In an **identification scheme** one party convinces another that it holds some special token.

Proof of Knowledge of Exponent

In an **identification scheme** one party convinces another that it holds some special token.

- Let G_q be a group of prime order q with generator g.

Proof of Knowledge of Exponent

In an **identification scheme** one party convinces another that it holds some special token.

- Let G_q be a group of prime order q with generator g.
- Let $x \in \mathbb{Z}_q$ and define $y = g^x$.

Proof of Knowledge of Exponent

In an **identification scheme** one party convinces another that it holds some special token.

- Let G_q be a group of prime order q with generator g.
- Let $x \in \mathbb{Z}_q$ and define $y = g^x$.
- Can we prove knowledge of x without disclosing anything about x?

Schnorr's Signature Scheme (1/3)

1. The prover chooses $r \in \mathbb{Z}_q$ randomly and hands $\alpha = g^r$ to the verifier.

Schnorr's Signature Scheme (1/3)

1. The prover chooses $r \in \mathbb{Z}_q$ randomly and hands $\alpha = g^r$ to the verifier.
2. The verifier chooses $c \in \mathbb{Z}_q$ randomly and hands it to the prover.

Schnorr's Signature Scheme (1/3)

1. The prover chooses $r \in \mathbb{Z}_q$ randomly and hands $\alpha = g^r$ to the verifier.
2. The verifier chooses $c \in \mathbb{Z}_q$ randomly and hands it to the prover.
3. The prover computes $d = cx + r \bmod q$ and hands d to the verifier.

Schnorr's Signature Scheme (1/3)

1. The prover chooses $r \in \mathbb{Z}_q$ randomly and hands $\alpha = g^r$ to the verifier.
2. The verifier chooses $c \in \mathbb{Z}_q$ randomly and hands it to the prover.
3. The prover computes $d = cx + r \mod q$ and hands d to the verifier.
4. The verifier accepts if $y^c \alpha = g^d$.

Schnorr's Signature Scheme (1/3)

1. The prover chooses $r \in \mathbb{Z}_q$ randomly and hands $\alpha = g^r$ to the verifier.
2. The verifier chooses $c \in \mathbb{Z}_q$ randomly and hands it to the prover.
3. The prover computes $d = cx + r \bmod q$ and hands d to the verifier.
4. The verifier accepts if $y^c\alpha = g^d$.

Suppose that a machine convinces us in the protocol with probability δ. Does it mean that it knows x such that $y = g^x$?

Schnorr's Signature Scheme (2/3)

Schnorr's Signature Scheme (2/3)

1. Run the machine to get α.

Schnorr's Signature Scheme (2/3)

1. Run the machine to get α.

2. Complete the interaction twice using **the same** α, once for a challenge c and once for a challenge c', where $c, c' \in \mathbb{Z}_q$ are chosen randomly.

Schnorr's Signature Scheme (2/3)

1. Run the machine to get α.
2. Complete the interaction twice using **the same** α, once for a challenge c and once for a challenge c', where $c, c' \in \mathbb{Z}_q$ are chosen randomly.
3. Repeat from (1) until the resulting interactions (α, c, d) and (α, c', d') are accepting and $c \neq c'$.

Schnorr's Signature Scheme (2/3)

1. Run the machine to get α.
2. Complete the interaction twice using **the same** α, once for a challenge c and once for a challenge c', where $c, c' \in \mathbb{Z}_q$ are chosen randomly.
3. Repeat from (1) until the resulting interactions (α, c, d) and (α, c', d') are accepting and $c \neq c'$.
4. Note that:

$$y^{c-c'} = \frac{y^c}{y^{c'}} = \frac{y^c\alpha}{y^{c'}\alpha} = \frac{g^d}{g^{d'}} = g^{d-d'}$$

which gives the logarithm $x = (d - d')(c - c')^{-1} \bmod q$ such that $y = g^x$.

Schnorr's Signature Scheme (3/3)

- Anybody can sample $c, d \in \mathbb{Z}_q$ randomly and compute $\alpha = g^d / y^c$.
- The resulting tuple (α, c, d) has **exactly** the same distribution as the transcript of an interaction!

Such protocols are called (honest verifier) **zero-knowledge proofs of knowledge**.

Schnorr's Signature Scheme In ROM

Let $H : \{0,1\}^* \to \mathbb{Z}_q$ be a random oracle.

- Gen chooses $x \in \mathbb{Z}_q$ randomly, computes $y = g^x$ and outputs $(\mathsf{pk}, \mathsf{sk}) = (y, x)$.
- Sig does the following on input x and m:
  1. it chooses $r \in \mathbb{Z}_q$ randomly and computes $\alpha = g^r$,
  2. it computes $c = H(y, \alpha, m)$,
  3. it computes $d = cx + r \bmod q$ and outputs (α, d).
- Vf takes the public key y, a message m, and a candidate signature (α, d), and accepts iff $y^{H(y,\alpha,m)}\alpha = g^d$.

Provably Secure Signature Schemes

Provably secure signature schemes exists if one-way functions exist (in plain model without ROM), but the construction is more involved and typically less efficient.

Provably secure signature schemes are rarely used in practice!

Standards used in practice: RSA Full Domain Hash, DSA, EC-DSA. The latter two may be viewed as variants of Schnorr signatures.

Problem

- We have constructed public-key cryptosystems and signature schemes.
- Only the holder of the secret key can decrypt ciphertexts and sign messages.
- How do we **know** who holds the secret key corresponding to a public key?

Signing Public Keys of Others

- Suppose that Alice computes a signature $\sigma_{A,B} = \mathsf{Sig}_{\mathsf{sk}_A}(\mathsf{pk}_B, Bob)$ of Bob's public key pk_B and his identity and hands it to Bob.
- Suppose that Eve holds Alice's public key pk_A.
- Then **anybody** can hand $(\mathsf{pk}_B, \sigma_{A,B})$ **directly** to Eve, and Eve will be convinced that pk_B is Bob's key (assuming she trusts Alice).

Certificate

- A **certificate** is a signature of a public key along with some information on how the key may be used, e.g., it may allow the holder to issue certificates.
- A certificate is valid for a given setting if the signature is valid and the usage information in the certificate matches that of the setting.
- Some parties must be trusted to issue certificates. These parties are called Certificate Authorities (CA).

Certificate Chains

A CA may be "distributed" using in certificate chains.

- Suppose that Bob holds valid certificates

$$\sigma_{0,1}, \sigma_{1,2}, \ldots, \sigma_{n-1,n}$$

  where $\sigma_{i-1,i}$ is a certificate of pk_{P_i} by P_{i-1}.

- Who does Bob trust?

Randomness

- Everything we have done so far requires randomness!

Randomness

- Everything we have done so far requires randomness!
- Can we “generate” random strings?

Physical Randomness and Deterministic Algorithms

- We could flip actual coins. This would be extremely impractical and slow (and booring unless you are Rain man).

Physical Randomness and Deterministic Algorithms

- We could flip actual coins. This would be extremely impractical and slow (and booring unless you are Rain man).
- We could generate "physical" randomness using hardware, e.g., measuring radioactive decay
  - Slow or expensive.
  - Hard to verify and trust.
  - Biased output.

Physical Randomness and Deterministic Algorithms

- We could flip actual coins. This would be extremely impractical and slow (and booring unless you are Rain man).
- We could generate "physical" randomness using hardware, e.g., measuring radioactive decay
  - Slow or expensive.
  - Hard to verify and trust.
  - Biased output.
- We could use a deterministic algorithm that outputs a "random looking string", but would that be secure?

Pseudo-Random Generator

A pseudo-random generator requires a **short random string** and deterministically expands this to a **longer "random looking" string**.

Pseudo-Random Generator

A pseudo-random generator requires a **short random string** and deterministically expands this to a **longer "random looking" string**.

This looks promising:

- Fast and cheap?
- Practical since it can be implemented in software or hardware?
- What is "random looking"?

Pseudo-Random Generator

Definition. An efficient algorithm PRG is a **pseudo-random generator (PRG)** if there exists a polynomial $p(n) > n$ such that for every polynomial time adversary A, if a seed $s \in \{0,1\}^n$ and a random string $u \in \{0,1\}^{p(n)}$ are chosen randomly, then

$$|\Pr[A(\mathrm{PRG}(s)) = 1] - \Pr[A(u) = 1]|$$

is negligible.

Informally, A can not distinguish $\mathrm{PRG}(s)$ from a truly random string in $\{0,1\}^{p(n)}$.

Increasing Extension (1/2)

Before we consider how to construct a PRG we consider what the definition gives us:

Increasing Extension (1/2)

Before we consider how to construct a PRG we consider what the definition gives us:

- Suppose that there exists a PRG that extends its output by a **single bit**.

Increasing Extension (1/2)

Before we consider how to construct a PRG we consider what the definition gives us:

- Suppose that there exists a PRG that extends its output by a **single bit**.
- This would **not be very useful** to us, e.g., to generate a random prime we need many random bits.

Increasing Extension (1/2)

Before we consider how to construct a PRG we consider what the definition gives us:

- Suppose that there exists a PRG that extends its output by a **single bit**.
- This would **not be very useful** to us, e.g., to generate a random prime we need many random bits.
- Can we use the given PRG to construct another PRG which extends its output more?

Increasing Extension (2/2)

Construction. Let PRG be a pseudo-random generator. We let PRG_t be the algorithm that takes $s_{-1} \in \{0,1\}^n$ as input, computes $s_0, s_2, \ldots, s_{t-1}$ and $b_0, \ldots, b_{t-1}$ as

$$(s_i, b_i) = \mathrm{PRG}(s_{i-1})$$

and outputs $(b_0, \ldots, b_{t-1})$.

Increasing Extension (2/2)

Construction. Let PRG be a pseudo-random generator. We let PRG_t be the algorithm that takes $s_{-1} \in \{0,1\}^n$ as input, computes $s_0, s_2, \ldots, s_{t-1}$ and $b_0, \ldots, b_{t-1}$ as

$$(s_i, b_i) = \mathrm{PRG}(s_{i-1})$$

and outputs $(b_0, \ldots, b_{t-1})$.

Theorem. Let $p(n)$ be a polynomial and PRG a pseudo-random generator. Then $\mathrm{PRG}_{p(n)}$ is a pseudo-random generator that on input $s \in \{0,1\}^n$ outputs a string in $\{0,1\}^{p(n)}$.

Increasing Extension (2/2)

Construction. Let PRG be a pseudo-random generator. We let PRG_t be the algorithm that takes $s_{-1} \in \{0,1\}^n$ as input, computes $s_0, s_2, \ldots, s_{t-1}$ and $b_0, \ldots, b_{t-1}$ as

$$(s_i, b_i) = \mathrm{PRG}(s_{i-1})$$

and outputs $(b_0, \ldots, b_{t-1})$.

Theorem. Let $p(n)$ be a polynomial and PRG a pseudo-random generator. Then $\mathrm{PRG}_{p(n)}$ is a pseudo-random generator that on input $s \in \{0,1\}^n$ outputs a string in $\{0,1\}^{p(n)}$.

We can go on "forever"!

Random String From Random Oracle

Theorem. If $F : \{0,1\}^n \rightarrow \{0,1\}^m$ is a random function, then $(F(0), F(1), F(2), \ldots, F(t-1))$ is a *tm*-bit string.

Random String From Random Oracle

Theorem. If $F : \{0,1\}^n \rightarrow \{0,1\}^m$ is a random function, then $(F(0), F(1), F(2), \ldots, F(t-1))$ is a tm-bit string.

Can we do this using a pseudo-random function?

Can we replace the random function by SHA-2?

Pseudo-Random Function

Recall the definition of a pseudo-random function.

Definition. A family of functions $F : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$ is pseudo-random if for all polynomial time oracle adversaries A

$$\left| \Pr_K \left[A^{F_K(\cdot)} = 1 \right] - \Pr_{R:\{0,1\}^n \rightarrow \{0,1\}^n} \left[A^{R(\cdot)} = 1 \right] \right|$$

is negligible.

Pseudo-Random Generator From Pseudo-Random Function

Theorem. Let $\{F_K\}_{K\in\{0,1\}^k}$ be a pseudo-random function for a random choice of K. Then the PRG defined by:

$$\mathrm{PRG}(s) = (F_s(0), F_s(1), F_s(2), \ldots, F_s(t))$$

is a pseudo-random generator.