CSE 127: Introduction to Security

Lecture 14: Public-Key Cryptography

Nadia Heninger and Deian Stefan

UCSD

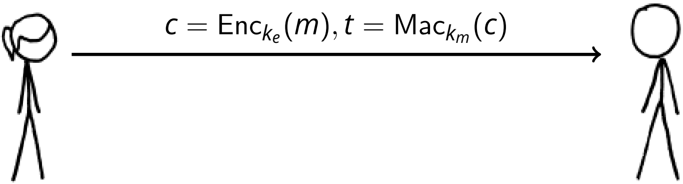
Fall 2019

Lecture Outline

- MAC Usage and Length Extension Attacks
- Key Exchange
- Public Key Encryption
- Digital Signatures

Recall: MAC Usage

MAC Security: $\mathsf{Mac}_k(c)$ should be unforgeable by an adversary.



Verify $t = \mathsf{Mac}_{k_m}(c)$

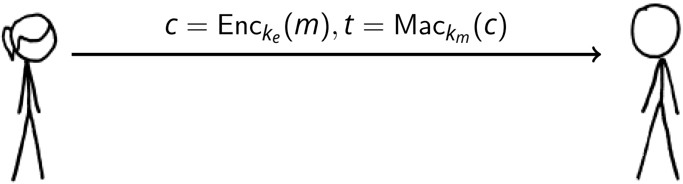
Compute $m = \mathsf{Dec}_{k_e}(c)$

Question: Is $\mathsf{Mac}(c) = H(c)$ for H a collision-resistant hash function a good MAC function?

Recall: MAC Usage

MAC Security: $\mathsf{Mac}_k(c)$ should be unforgeable by an adversary.

$c = \mathsf{Enc}_{k_e}(m), t = \mathsf{Mac}_{k_m}(c)$



Verify $t = \mathsf{Mac}_{k_m}(c)$

Compute $m = \mathsf{Dec}_{k_e}(c)$

Question: Is $\mathsf{Mac}(c) = H(c)$ for H a collision-resistant hash function a good MAC function?

No: H is public, so adversary can compute $H(m)$ for any m they desire.

Length extension attacks

Question: Is $\mathsf{Mac}_k(m) = H(k||m)$ a secure MAC?

Length extension attacks

Question: Is $\mathsf{Mac}_k(m) = H(k||m)$ a secure MAC?

A: Not if H is MD5, SHA-1, or SHA-2.

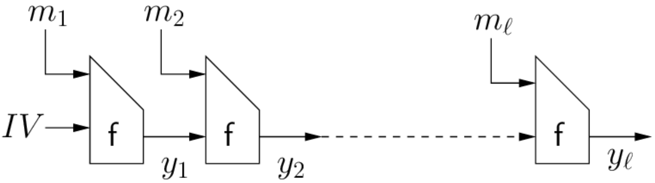
These all use the Merkle-Damgård construction, which is vulnerable to length extension attacks.

Merkle-Damgård Hash Function Construction

The Merkle-Damgård construction constructs a hash function that takes arbitrary length inputs from a fixed-length compression function.

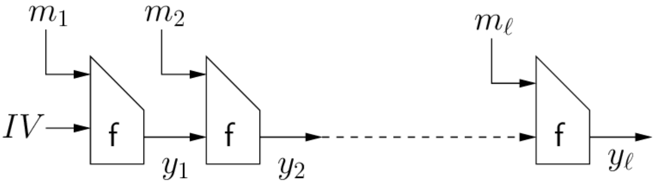
For MD5, it works like this:

1. Input $m = m_1 || m_2 || \dots || m_\ell$ where m_i are 512-bit blocks.
2. Append $1 || 000 \dots 000 || len(m)$ to the last block, where as many bits as necessary to make m_ℓ a multiple of 512.
3. Iterate



Length Extension Attack Against MD5

- Adversary observes $\mathsf{BadMac}_k(m) = H(k||m)$ for unknown k and possibly unknown m.
- Adversary would like to forge $\mathsf{BadMac}_k(m||r)$ for r of the adversary's choice.
- A length extension attack allows the adversary to construct $\mathsf{BadMac}_k(m||\text{padding}||r)$ for r of their choice.



If adversary knows or can guess the length of $k||m$, they can reconstruct the padding and append additional blocks corresponding to r to Merkle-Damgård construction.

Application: Flickr API length extension vulnerability

In 2009, Flickr required API calls to use an authentication token that looked like:

```
MD5(secret || arg1=val1&arg2=val2&...)
```

This was included in the argument list.

This construction was vulnerable to exactly the length extension attack we just described.

Secure Solution: Use a good MAC Construction

This is why HMAC is a good choice.

“We stand today on the brink of a revolution in cryptography.”

— Diffie and Hellman, 1976

Lecture Outline

- MAC Usage and Length Extension Attacks
- Key Exchange
- Public Key Encryption
- Digital Signatures

Asymmetric cryptography/public-key cryptography

Main insight: Separate keys for different operations.

Keys come in pairs, and are related to each other by the specific algorithm:

- Public key: known to everyone, used to encrypt or verify signatures
- Private key: used to decrypt and sign

Public-key encryption

- Encryption: (public key, plaintext) $\rightarrow$ ciphertext

$$\mathrm{Enc}_{pk}(m) = c$$

- Decryption: (secret key, ciphertext) $\rightarrow$ plaintext

$$\mathrm{Dec}_{sk}(c) = m$$

Properties:

- Encryption and decryption are inverse operations:

$$\mathrm{Dec}_{sk}(\mathrm{Enc}_{pk}(m)) = m$$

- Secrecy: ciphertext reveals nothing about plaintext
  - Computationally hard to decrypt without secret key
- What's the point:
  - Anybody with your public key can send you a secret message! Solves key distribution problem.

Modular Arithmetic Review

Division: Let n, d, q, r be integers.

$$\begin{aligned} \lfloor n/d \rfloor &= q \\ n &= qd + r \qquad 0 \leq r < d \\ n &\equiv r \bmod d \end{aligned}$$

Facts about remainders/modular arithmetic:

$$\begin{aligned} \text{Add:} \quad & (a \bmod d) + (b \bmod d) \equiv (a + b) \bmod d \\ \text{Subtract:} \quad & (a \bmod d) - (b \bmod d) \equiv (a - b) \bmod d \\ \text{Multiply:} \quad & (a \bmod d) \cdot (b \bmod d) \equiv (a \cdot b) \bmod d \end{aligned}$$

Modular Inverse: “Division” for modular arithmetic

If $a \cdot b \mod d = c \bmod d$ we would like $c/b \bmod d = a \bmod d$.

But if $3 \cdot 2 \bmod 4 = 2 \bmod 4$ this says $3 = 1 \bmod 4$. Problem!

Modular Inverse: "Division" for modular arithmetic

If $a \cdot b \mod d = c \bmod d$ we would like $c/b \bmod d = a \bmod d$.

But if $3 \cdot 2 \bmod 4 = 2 \bmod 4$ this says $3 = 1 \bmod 4$. Problem!

Fix: For rationals, $\frac{a}{b} = a \cdot \frac{1}{b}$ $\qquad b \cdot \frac{1}{b} = 1$.

Define modular inverse: $\frac{1}{b}$ means $b^{-1} \bmod d$.

- $b^{-1} \bmod d$ is a value such that $b \cdot b^{-1} \equiv 1 \bmod d$.
- Example: $3 \cdot (3^{-1} \bmod 5) \equiv 3 \cdot 2 \equiv 1 \bmod 5$.
- If $\gcd(a, d) = 1$ then a^{-1} is well defined.
- Efficient to compute.

Modular exponentiation and discrete log

Modular exponentiation

- Over the integers, $g^a = g \cdot g \cdot g \dots g \quad a$ times.
- $\mod d$ it's the same:
  $g^a \mod d = (((g \mod d) \cdot g \mod d) \dots g \mod d) \mod d$
  a times.
- This is efficient to compute using the binary representation of a.

Modular exponentiation and discrete log

Modular exponentiation

- Over the integers, $g^a = g \cdot g \cdot g \ldots g \quad a$ times.
- $\mod d$ it's the same:
  $g^a \bmod d = (((g \bmod d) \cdot g \bmod d) \ldots g \bmod d) \bmod d$
  a times.
- This is efficient to compute using the binary representation of a.

"Inverse" of modular exponentiation: Discrete log

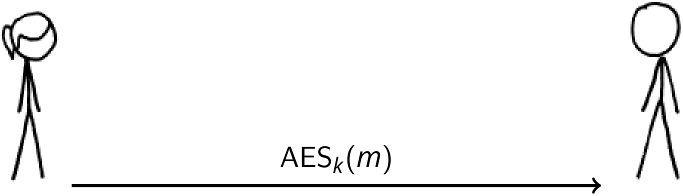
- Over the reals, if $b^a = y$ then $\log_b y = a$.
- Define discrete log similarly:
  Input b, d, y, discrete log is a such that $b^a \equiv y \bmod d$.
- No known polynomial-time algorithm to compute this.

New Directions in Cryptography

Invited Paper

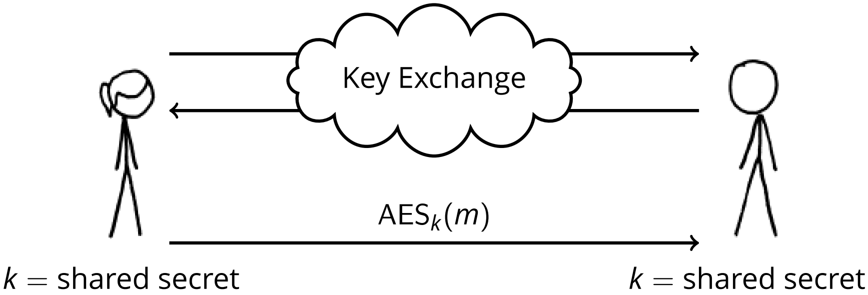
WHITFIELD DIFFIE AND MARTIN E. HELLMAN, MEMBER, IEEE

Symmetric cryptography



Public key crypto idea # 1: Key exchange

Solving key distribution without trusted third parties



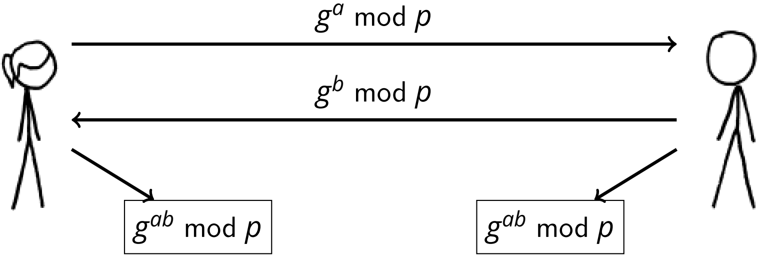
Textbook Diffie-Hellman Key Exchange

Public Parameters

p a prime

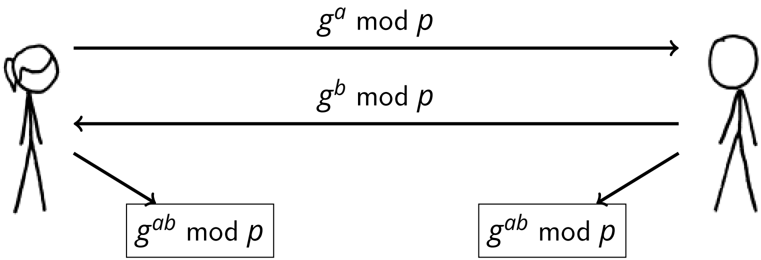
g an integer $\bmod p$

Key Exchange



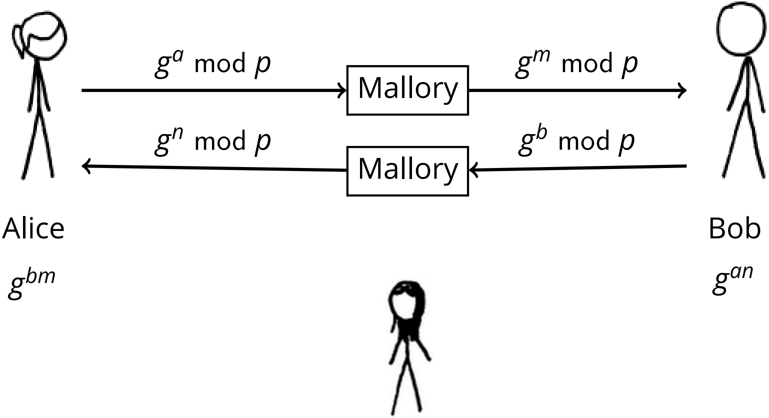
Note: $(g^a)^b \bmod p = g^{ab} \bmod p = g^{ba} \bmod p (g^b)^a \bmod p$.

Diffie-Hellman Security



- Most efficient algorithm for passive eavesdropper to break:
  Compute discrete log of public values $g^a \bmod p$ or $g^b \bmod p$.
- Parameter selection: p should be $\geq$ 2048 bits.
- Do not implement this yourself ever: discrete log is only hard for certain choices of p and g.
- Best current choice: Use elliptic curve Diffie-Hellman.
  (Similar idea, more complicated math.)

Diffie-Hellman insecure against man-in-the-middle



Active adversary can modify Diffie-Hellman messages in transit and learn both shared secrets.

Allows transparent MITM attack against later encryption.

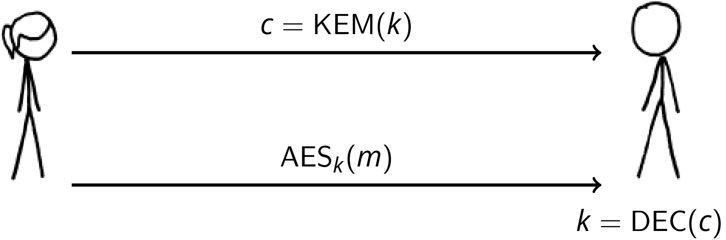
Need to authenticate messages to fix.

Computational complexity for integer problems

- Integer multiplication is efficient to compute.
- There is no known polynomial-time algorithm for general-purpose factoring.
- Efficient factoring algorithms for many types of integers. Easy to find small factors of random integers.
- Modular exponentiation is efficient to compute.
- Modular inverses are efficient to compute.

Idea # 2: Key encapsulation/public-key encryption

Solving key distribution without trusted third parties



A Method for Obtaining Digital Signatures and Public-Key Cryptosystems

R.L. Rivest, A. Shamir, and L. Adleman*

Textbook RSA Encryption

[Rivest Shamir Adleman 1977]

Public Key pk

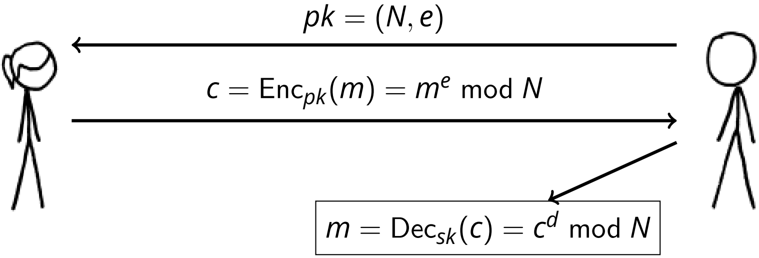
$N = pq$ modulus

e encryption exponent

Secret Key sk

p, q primes

d decryption exponent
$(d = e^{-1} \bmod (p-1)(q-1))$



$\mathrm{Dec}(\mathrm{Enc}(m)) = m^{ed} \bmod N \equiv m^{1+k\phi(N)} \equiv m \bmod N$ by Euler's theorem.

RSA Security

- Best algorithm to break RSA: Factor N and compute d.
- Factoring is not efficient in general.
- Current key size recommendations: N should be ≥ 2048 bits.
- Do not ever implement this yourself. Factoring is only hard for some integers, and textbook RSA is insecure.
- My recommendation: Use elliptic curve Diffie-Hellman instead of RSA to exchange keys.

Textbook RSA is super insecure

Unpadded RSA encryption is homomorphic under multiplication. Let's have some fun!

Attack: Malleability

Given a ciphertext $c = \mathsf{Enc}(m) = m^e \bmod N$, attacker can forge ciphertext $\mathsf{Enc}(ma) = ca^e \bmod N$ for any a.

Attack: Chosen ciphertext attack

Given a ciphertext $c = \mathsf{Enc}(m)$ for unknown m, attacker asks for $\mathsf{Dec}(ca^e \bmod N) = d$ and computes $m = da^{-1} \bmod N$.

So in practice **always use padding on messages**.

RSA PKCS #1 v1.5 padding

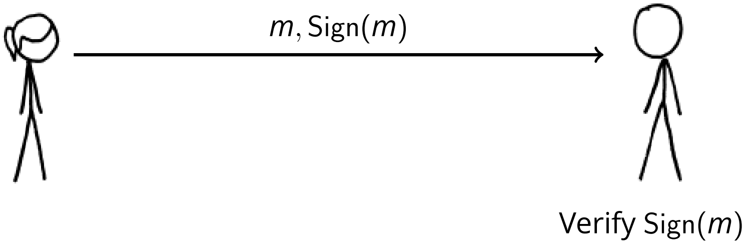
Most common implementation choice even though it is insecure

```
pad(m) = 00 02 [random padding string] 00 [m]
```

- Encrypter pads message, then encrypts padded message using RSA public key:
  $\mathrm{Enc}_{pk}(m) = pad(m)^e \bmod N$
- Decrypter decrypts using RSA private key, strips off padding to recover original data:
  $\mathrm{Dec}_{sk}(c) = c^d \bmod N = pad(m)$

PKCS#1v1.5 padding is vulnerable to a number of padding attacks. It is still commonly used in practice.

Idea #3: Digital Signatures



Bob wants to verify Alice’s signature using only a public key.

- Signature verifies that Alice was the only one who could have sent this message.
- Signature also verifies that the message hasn’t been modified in transit.

Digital Signatures

- Signing: (secret key, message) $\rightarrow$ signature

$$\mathsf{Sign}_{sk}(m) = s$$

- Verification: (public key, message, signature) $\rightarrow$ bool

$$\mathsf{Verify}_{pk}(m, s) = \text{true} \mid \text{false}$$

Digital Signatures

- Signing: (secret key, message) $\rightarrow$ signature

$$\text{Sign}_{sk}(m) = s$$

- Verification: (public key, message, signature) $\rightarrow$ bool

$$\text{Verify}_{pk}(m, s) = \text{true} \mid \text{false}$$

Signature properties:

- Verification of signed message succeeds:
  - $\text{Verify}_{pk}(m, Sign_{sk}(m)) = \text{true}$
- Unforgeability: Can't compute signature for message m that verifies with public key without corresponding secret key.
- What's the point?
  - Anybody with your public key can verify that you signed something!

Textbook RSA Signatures

[Rivest Shamir Adleman 1977]

Public Key *pk*

$N = pq$ modulus

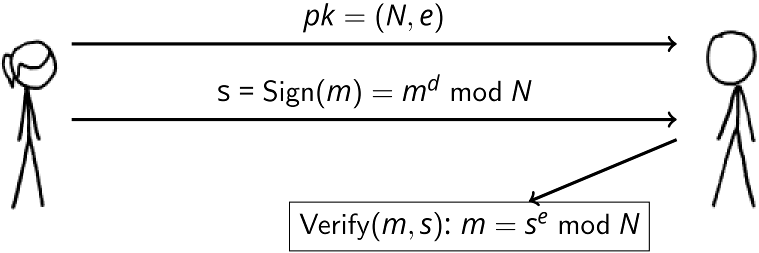
e encryption exponent

Secret Key *sk*

p, q primes

d decryption exponent

$(d = e^{-1} \bmod (p-1)(q-1))$



Works for the same reason RSA encryption does.

Textbook RSA signatures are super insecure

Attack: Signature forgery

1. Attacker wants $\mathsf{Sign}(x)$.
2. Attacker computes $z = xy^e \bmod N$ for some y.
3. Attacker asks signer for $s = \mathsf{Sign}(z) = z^d \bmod N$.
4. Attacker computes $\mathsf{Sign}(x) = sy^{-1} \bmod N$.

Countermeasures:

- **Always use padding with RSA.**
- **Sign hash of** m **and not raw message** m**.**

Positive viewpoint:

- Blind signatures: Lots of neat crypto applications.

RSA PKCS #1 v1.5 signature padding

Most widely used padding scheme in practice

```
pad(m) = 00 01 [FF FF FF ...  FF FF] 00 [data H(m)]
```

- Signer hashes and pads message, then signs padded message using RSA private key.
- Verifier verifies using RSA public key, strips off padding to recover hash of message.

Q: What happens if a decrypter doesn't correctly check padding length?

RSA PKCS #1 v1.5 signature padding

Most widely used padding scheme in practice

```
pad(m) = 00 01 [FF FF FF ...  FF FF] 00 [data H(m)]
```

- Signer hashes and pads message, then signs padded message using RSA private key.
- Verifier verifies using RSA public key, strips off padding to recover hash of message.

Q: What happens if a decrypter doesn't correctly check padding length?

A: Bleichenbacher low exponent signature forgery attack.

Bleichenbacher RSA Signature Forgery

```
pad(m) = 00 01 [FF FF FF ...  FF FF] 00 [data H(m)]
```

If victim shortcuts padding check: just looks for padding format but doesn't check length, and signature uses $e = 3$:

1. Construct a perfect cube over the integers, ignoring N, such that

$$s = 0001FF \dots FF00[\text{hash of forged message}][\text{garbage}]$$

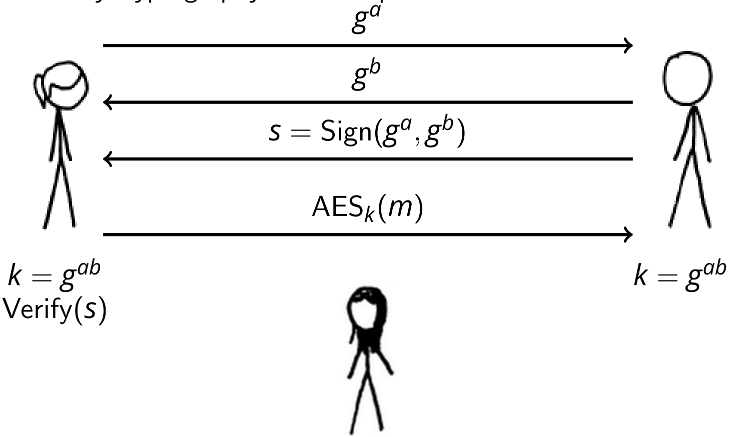
2. Compute x such that $x^3 = s$.
   (Easy way: $x = \lceil [\text{desired values}]000 \dots 0000 \rceil^{1/3}$.)
3. Lazy implementation validates bad signature!

Security for RSA signatures

- Same as RSA encryption.
- My recommendation: Use ECDSA or ed25519 instead.

Putting it all together

How public-key cryptography is used in practice



- Diffie-Hellman used to negotiate shared session key.
- Alice verifies Bob's signature to ensure that key exchange was not man-in-the-middled.
- Shared secret used to symmetrically encrypt data.

Public-key cryptography and quantum computers

Right now, all public-key cryptography used in the real world involves three “hard” problems:

- Factoring
- Discrete log mod primes
- Elliptic curve discrete log

All of these problems can be solved efficiently by a general-purpose quantum computer.

Big standardization effort now to develop replacements:

- Lattice-based cryptography
- Multivariate cryptography
- Hash-based signatures
- Supersingular isogeny Diffie-Hellman

These will likely be used more in the real world in the next few years.