CO 487 Winter 2024:
Lecture Notes

Lecture notes taken, unless otherwise specified, by myself during the Winter 2024 offering of CO 487, taught by Alfred Menezes.

Lectures

Chapter 1

Introduction

Lecture 1
Jan 8

Cryptography is securing communications in the presence of malicious adversaries. To simplify, consider Alice and Bob communicating with the eavesdropper Eve. Communications should be:

- Confidential: Only authorized people can read it
- Integral: Ensured that it is unmodified
- Origin authenticated: Ensured that the source is in fact Alice
- Non-repudiated: Unable to gaslight the message existing

Examples: TLS for intenet browsing, GSM for cell phone communications, Bluetooth for other wireless devices.

Overview: Transport Layer Security The protocol used by browsers to visit websites. TLS assures an individual user (a client) of the authenticity of the website (a server) and to establish a secure communications session.

TLS uses symmetric-key cryptography. Both the client and server have a shared secret k called a key. They can then use AES for encryption and HMAC for authentication.

To establish the shared secret, use public-key cryptography. Alice can encrypt the session key k can be encrypted with Bob's RSA public key. Then, Bob can decrypt it with his private key.

To ensure Alice is getting an authentic copy of Bob's public key, a certification authority (CA) signs it using the CA's private key. The CA public key comes with Alice's device preinstalled.

Potential vulnerabilities when using TLS:

- Weak cryptography scheme or vulnerable to quantum computing
- Weak random number generation for the session key
- Fraudulent certificates
- Implementation bugs
- Phishing attacks
- Transmission is secured, but the endpoints are not

These are mostly the purview of cybersecurity, of which cryptography is a part. Cryptography is not typically the weakest link in the cybersecurity chain.

Chapter 2

Symmetric key encryption

2.1 Basic concepts

Lecture 2
Jan 10

Definition 2.1.1 (symmetric-key encryption scheme)

A symmetric-key encryption scheme (SKES) consists of:

- plaintext space M,
- ciphertext space C,
- key space K,
- family of encryption functions $E_k : M \to C$ for all keys $k \in K$, and
- family of decryption functions $D_k : C \to M$ for all keys $k \in K$

such that $D_k(E_k(m)) = m$ for all m and k.

For Alice to send a message to Bob:

1. Alice and Bob agree on a secret key k *somehow* (assume a secured channel)
2. Alice computes $c = E_k(m)$ and sends c to Bob
3. Bob recovers the plaintext by computing $m = D_k(c)$

Examples include the Enigma and Lorenz machines.

Cryptoscheme 2.1 (simple substitution cipher)

Let:

- M be English messages
- C be encrypted messages
- K be permutations of the English alphabet
- $E_k(m)$ apply the permutation k to m, one letter at a time
- $D_k(c)$ apply the inverse permutation k^{-1} to c, one letter at a time

We want a system to have:

1. Efficient algorithms should be known for computing (encryption and decryption)

2. Small keys but large enough to render exhaustive key search infeasible
3. Security
4. Security against its designer

To determine how secure the protocol is, we have to define security.

Definition 2.1.2 (security model)

Some parameters which define the strength of the adversary, specific interaction with the "secure" channel, and the goal of the adversary.

Some options for strength:

- Information-theoretic security: Eve has infinite resources.
- Complexity-theoretic security: Eve is a polynomimal-time Turing machine.
- Computational-theoretic security: Eve has a specific amount of computing power. In this course, Eve is computationally bounded by 6,768 Intel E5-2683 V4 cores running at 2.1 GHz at her disposal.

For the interaction:

- Ciphertext-only attack: Eve only knows the ciphertext.
- Known-plaintext attack: Eve knows some plaintext and the corresponding ciphertext.
- Chosen-plaintext attack: Eve picks some plaintext and knows the corresponding ciphertext.
- Clanedestine attack: Eve resorts to bribery, blackmail, etc.
- Side-channel attack: Eve has physical access to hardware and has some monitoring data.

And for the goal:

- Recovering the secret key k
- Systematically decrypt arbitrary ciphertexts without knowing k (total security)
- Learn partial information about the plaintext (other than the length) (semantic security)

Definition 2.1.3 (security)

An SKES is secure if it is semantically secure against a chosen-plaintext attack by a computationally bounded adversary.

Equivalently, an SKES is broken if:

1. Given a challenge ciphertext c for m generated by Alice,
2. …and access to an encryption oracle for Alice,
3. …Eve can obtain some information about m other than its length,
4. …using only a feasible amount of computation.

Note: this is IND–CPA from CO 485.

Example 2.1.4. Is the simple substitution cipher secure? What about under a ciphertext-only attack?

Solution. Under CPA, encrypt the entire alphabet. Then, the entire key k is recovered.

With a ciphertext-only attack, an exhaustive key search would take $26! \approx 2^{88}$ attempts. This would take over 1,000 years, which is pretty infeasible, so it is secure. □

Can we quantify how feasible something is?

Definition 2.1.5 (security level)

A scheme has a security level of ℓ bits if the fastest known attack on the scheme takes approximately 2^ℓ operations.

Convention. In this course:

- 40 bits is very easy to break
- 56 bits is easy to break
- 64 bits is feasible to break
- 80 bits is barely feasible to break
- 128 bits is infeasible to break

Lecture 3
Jan 12

The simple substitution cipher can be attacked by frequency analysis, since, for example, if "e" is the most common English letter, we check the ciphertext for the most common letter and identify it with "e".

Cryptoscheme 2.2 (Vigenère cipher)

Let the key K be an English word with no repeated letters, e.g., $K = \text{CRYPTO}$.

To encrypt, add letter-wise the key modulo 26, where k is K repeated until it matches the length of the message:

	$m =$	t	h	i	s	i	s	a	m	e	s	s	a	g	e
+	$k =$	C	R	Y	P	T	O	C	R	Y	P	T	O	C	R
	$c =$	V	Y	G	H	B	G	C	D	C	H	L	O	I	V

To decrypt, just take $c - k$.

This solves our frequency analysis problem. However, the Vigenere cipher is still totally insecure.

Exercise 2.1.6. Show that the Vigenere cipher is totally insecure under a chosen-plaintext attack and a ciphertext-only attack.

Cryptoscheme 2.3 (one-time pad)

The key is a random string of letters with the same length as the message.

Repeat the process for Vigenere. To encode, add each letter. To decode, subtract each letter.

Example 2.1.7. We can encrypt as follows:

	$m =$	t	h	i	s	i	s	a	m	e	s	s	a	g	e
$+$	$k =$	Z	F	K	W	O	G	P	S	M	F	J	D	L	G
	$c =$	S	M	S	P	W	Y	P	F	Q	X	C	D	R	K

This is semantically secure as long as the key is never reused. Formally, there exist keys that can decrypt the ciphertext into *anything*, so there is no way for an attacker to know the plaintext. If it is reused, i.e., if $c_1 = m_1 + k$ and $c_2 = m_2 + k$, then $c_1 - c_2 = (m_1 + k) - (m_2 + k) = m_1 - m_2$. Since this is a function only of messages, it can leak frequency information etc.

Also, since the key is never reused, this is secure against a chosen plaintext attack, since one would only recover the already used key.

Convention. From now on, messages and keys are assumed to be binary strings.

Definition 2.1.8 (bitwise exclusive or)
For two bitstrings $x, y \in \{0,1\}^n \cong \mathbb{Z}/2\mathbb{Z}^n$, the bitwise XOR $x \oplus y$ is just addition mod 2.

Unfortunately, due to Shannon, we have this theorem:

Theorem 2.1.9
A perfectly secure symmetric-key scheme must have at least as many keys as there are messages.

2.2 Stream ciphers

Instead of using a random key in the OTP, use a pseudorandom key.

Definition 2.2.1 (pseudorandomness)
A pseudorandom bit generator (PBRG) is a deterministic algorithm that takes as input a seed and outputs a pseudorandom sequence called the keystream.

Then, we can construct a stream cipher by defining the key as the seed and the ciphertext as the keystream XOR'd with the plaintext. To decrypt, use the seed to generate the same keystream and XOR with the ciphertext.

For a stream cipher to be secure, we need:

- Indistinguishability: the keystream is indistinguishable from a truly random sequence; and
- Unpredictability: given a partial keystream, it is infeasible to learn any information from the remainder of the keystream.

Remark 2.2.2. Do not use built-in UNIX `rand` or `srand` for cryptography!

Now, we introduce ChaCha20, a stream cipher actually uesd in the real world. The algorithm works entirely on words (32-bit numbers). It has no known flaws (other than people bungling the implementation).

Cryptoscheme 2.4 (ChaCha20)

First, define a helper function $QR(a, b, c, d)$ on 32-bit words:

1. $a \leftarrow a \boxplus b$, $d \leftarrow d \oplus a$, $d \leftarrow d \lll 16$
2. $c \leftarrow c \boxplus d$, $b \leftarrow b \oplus c$, $b \leftarrow b \lll 12$
3. $a \leftarrow a \boxplus b$, $d \leftarrow d \oplus a$, $d \leftarrow d \lll 8$
4. $c \leftarrow c \boxplus d$, $b \leftarrow b \oplus c$, $b \leftarrow b \lll 7$

where $\oplus$ is bitwise XOR, $\boxplus$ is addition mod 2^{32}, and $\lll$ is left bit-rotation.

Given a 256-bit key $k = (k_1, \dots, k_8)$, a selected 96-bit nonce $n = (n_1, n_2, n_3)$, a 128-bit given constant $f = (f_1, \dots, f_4)$, and 32-bit counter $c \leftarrow 0$, construct an initial state:

$$S := \begin{bmatrix} f_1 & f_2 & f_3 & f_4 \\ k_1 & k_2 & k_3 & k_4 \\ k_5 & k_6 & k_7 & k_8 \\ c & n_1 & n_2 & n_3 \end{bmatrix} = \begin{bmatrix} S_1 & S_2 & S_3 & S_4 \\ S_5 & S_6 & S_7 & S_8 \\ S_9 & S_{10} & S_{11} & S_{12} \\ S_{13} & S_{14} & S_{15} & S_{16} \end{bmatrix}$$

Keep a copy $S' \leftarrow S$, then apply:

$$QR(S_1, S_5, S_9, S_{13}), \quad QR(S_2, S_6, S_{10}, S_{14}), \quad QR(S_3, S_7, S_{11}, S_{15}), \quad QR(S_4, S_8, S_{12}, S_{16})$$
$$QR(S_1, S_6, S_{11}, S_{16}), \quad QR(S_2, S_7, S_{12}, S_{13}), \quad QR(S_3, S_8, S_9, S_{14}), \quad QR(S_4, S_5, S_{10}, S_{15})$$

ten times (for 80 total calls to QR) and output $S \oplus S'$. This gives us 64 keystream bytes.

Increment $c \leftarrow c + 1$ and repeat as necessary to generate more keystream bytes.

To encrypt, XOR the keystream with the plaintext, then append the nonce.

To decrypt, pop off the nonce, then XOR the keystream with the ciphertext.

Lecture 4
Jan 15

One must be careful never to reuse nonces, since this results in the same keystream, leading to recoverable messages. In practice, this is hard (e.g., two devices with the same key).

Miscellaneous remarks:

- Why is ChaCha20 so good? The QR function is very fast at the hardware level and there is wide adoption/standardization by experts.
- Why 10 rounds? If you do 1 or 2 rounds, there is a trivial attack. The latest theoretical attacks can attack 7 rounds (currently infeasible, but still better than exhaustive key search). So 8 rounds is secure and we do 10 to be safe.
- Is this secure forever (i.e., can we always just increase rounds)? No. Nothing in this course is. Someone could find a super crazy PMATH theorem that shows predictability of the QR scramble.

2.3 Block ciphers

Definition 2.3.1 (block cipher)
Like a stream cipher, but instead of processing one character at a time, we break up the plaintext into blocks of equal length and encrypt block-wise.

Example 2.3.2. The Data Encryption Standard (DES) is a standard 56-bit key and 64-bit blocks.

Aside: History and the NSA doing ratfuckery In 1972, the National Institute of Standards and Technology (NIST)[1] puts out an RfP for encryption algorithms.

IBM developed and proposed 64-bit DES, but then the NSA reduced it in 1975 to 56-bit so they can do some spying. This made DES feasible to break by nation-states but not smaller organizations.

The National Security Agency (NSA) is the US' signals intelligence (SIGINT; hacking foreign intelligence) and information insurance (IA; defending domestic intelligence) agency. They have a history of regulating how strong cryptoraphic products can be by banning the export of strong cryptography.

Canada has an NSA equivalent: the Communications Security Establishment (CSE). Along with the Kiwi CCSA, British GCHQ, and Australian ASD, these are the Five Eyes who spy on just about everyone.

We only really know stuff about the NSA/Five Eyes due to the Snowden leaks. For example, the SIGINT Enabling Project attempts to influence/blackmail companies to weaken their security with backdoors.

Throughout the course, we will use the NSA to mean "generic nation-state level adversary", since if you can defeat the NSA, you can defeat basically anyone.

Anyways, weakened DES was adopted by NIST in 1977 as FIPS 46 in 1977, then as a banking standard as ANSI X3.92 in 1982 (replaced by Triple-DES in 1988). From 1997–2001, a new contest developed the Advanced Encryption Standard (AES), which is the current standard block cipher.

Desired properties of block ciphers (Shannon, 1949):

1. Diffusion: Each ciphertext bit should depend on all plaintext bits.
2. Confusion: The key–ciphertext relationship should be complicated.
3. Key length: Keys should be small but not too small to be searchable.
4. Simplicity: Ease of implementation and analysis.
5. Speed: Runs quickly on all reasonable hardware.
6. Platform: Can be implemented in hardware and software.

[1] of standardized peanut butter fame

Cryptoscheme 2.5 (DES)

The design principles of DES are still classified, so we just treat it as a black box for this course. We only need to know that there is a 56-bit key and 64-bit blocks.

The DES key space is not very big. Exhaustive search on DES takes 2^{56} operations. In 1997, this took three months. In 2012, it takes 11.5 hours.

The blocks are also not very large. By the birthday paradox, there is a collision every 2^{32} blocks. This is an information leak, breaking semantic security.

These are the only (known) weaknesses in DES.

Definition 2.3.3 (multiple encryption)

Re-encrypt the ciphertext more times with different keys.

This is not always more secure. For example, in the simple substitution cipher, permutations can be composed and do not introduce more security.

Cryptoscheme 2.6 (Double-DES)

Pick a secret key $k = (k_1, k_2) \in_R \{0,1\}^{112}$.

Then, encrypt $E_{k_2}(E_{k_1}(m))$ where E is DES encryption

Likewise, decrypt $E_{k_2}^{-1}(E_{k_1}^{-1}(m))$ where E^{-1} is DES decryption.

We now have an exhaustive key search of 2^{112} operations, which is better. However, there is an attack which reduces this to breaking DES.

↓ *Lectures 5, 6, and 7 taken directly from slides* ↓ *Lecture 5 Jan 17*

Attack 2.7 (Meet-in-the-middle attack on Double-DES)

The main idea is that $c = E_{k_2}(E_{k_1}(m))$ if and only if $E_{k_2}^{-1}(c) = E_{k_1}(m)$.

Given three plaintext/ciphertext pairs (m_1, c_1), (m_2, c_2) and (m_3, c_3):

```
Create a table T of pairs sorted by first entry
for h_2 ∈ {0,1}^56 do                                  ▷ h_2 is a guess for k_2
    T.insert(E_{h_2}^{-1}(m_1), h_2)
for h_1 ∈ {0,1}^56 do                                  ▷ h_1 is a guess for k_1
    Compute E_{h_1}(m_1)
    Search for entries in T matching (E_{h_1}(m_1), −)
    for each match (−, h_2) do
        if E_{h_2}(E_{h_1}(m_2)) = c_2 then
            if E_{h_2}(E_{h_1}(m_3)) = c_3 then
                return (h_1, h_2)
```

In attack 2.7, we use three pairs. Why do we need that many?

Since the key space is smaller than the message space, there will be multiple keys that encrypt a message to the same ciphertext.

Lemma 2.3.4 (number of plaintext-ciphertext pairs needed)
Let E be a block cipher with ℓ-bit key space and L-bit plaintext/ciphertext space.

If E is a random bijection, the expected number of false keys matching t pairs is $\frac{2^{\ell}-1}{2^{Lt}}$.

Proof. We assume that E is a random bijection, so we can calculate probabilities.

Fix the true key k'. Let (m_i, c_i) for $i = 1, \dots, t$ be known plaintext-ciphertext pairs where each plaintext is distinct.

For some $k \in K$, $k \neq k'$, the probability that $E_k(m_i) = c_i$ for all i is $\underbrace{\frac{1}{2^L} \cdot \frac{1}{2^L} \cdots \frac{1}{2^L}}_{t \text{ times}} = \frac{1}{2^{Lt}}$.

Therefore, across all of $K \setminus \{k'\}$, the expected number of false keys is $\frac{2^{\ell}-1}{2^{Lt}}$. □

For Double-DES, $\ell = 112$ and $L = 64$:

- For $t = 1$, $FK \approx 2^{48}$. That is, given (m, c) the number of Double-DES keys (h_1, h_2) for which $E_{h_2}(E_{h_1}(m)) = c$ is $\approx 2^{48}$.
- For $t = 2$, $FK \approx 2^{-16}$. That is, the number of Double-DES keys (h_1, h_2) for which $E_{h_2}(E_{h_1}(m_1)) = c_1$ is $\approx 2^{48}$.

Therefore, we use three plaintext-ciphertext pairs in attack 2.7.

The time requirement of the attack is $2^{56} + 2^{57} + 2 \cdot 2^{48} \approx 2^{57}$ DES encryptions/decryptions. The size of the table is $2^{56}(64 + 56)$ bits or about 1 million TB.

Exercise 2.3.5. Modify attack 2.7 to decrease storage requirements at the expense of time. We can get down to 2^{56+s} operations and 2^{56-s} rows for $1 \leq s \leq 55$.

We can now conclude that the security level of Double-DES is 57 bits, not much better than normal DES' 56 bits.

Cryptoscheme 2.8 (Triple-DES)
Pick a secret key $k = (k_1, k_2, k_3) \in_R \{0,1\}^{168}$.

Then, encrypt $E_{k_3}(E_{k_2}(E_{k_1}(m)))$ where E is DES encryption

Likewise, decrypt $E^{-1}_{k_3}(E^{-1}_{k_2}(E^{-1}_{k_1}(m)))$ where E^{-1} is DES decryption.

As for Double-DES, the 168-bit keys are infeasible to search.

Exercise 2.3.6. Show that attack 2.7 on Triple-DES takes 2^{112} operations.

This means the security level of Triple-DES is 112 bits. We cannot *prove* Triple-DES is more secure than DES, just that it empirically feels better.

2.4 Substitution-permutation networks

Lecture 6
Jan 19

Definition 2.4.1 (substitution-permutation network)

A substitution-permutation network (SPN) is an iterated block cipher where each iteration (round) is a substitution followed by a permutation.

Formally, we have:

- a block length n, key length ℓ
- number of rounds h,
- substitution $S : \{0,1\}^b \to \{0,1\}^b$, an invertible function where $b \mid n$,
- permutation P, an invertible function $\{1,\dots,n\} \to \{1,\dots,n\}$, and
- key scheduling algorithm k_i that determines a round key for each round $i = 1,\dots,h+1$ given a key k.

Note that n, ℓ, h, S, P, and the key scheduling algorithm are public.

Then, we can write encryption as

$A \leftarrow m$
for $i = 1,\dots,h$ **do**
  $A \leftarrow A \oplus k_i$
  $A \leftarrow S(A[1:b]) \parallel S(A[b+1:2b]) \parallel \cdots \parallel S(A[n-b+1:n])$ ▷ *Apply S to each b bits*
  $A \leftarrow P(A)$
$A \leftarrow A \oplus k_{h+1}$ **return** A

and decryption is the reverse (since S and P are invertible).

The most notable SPN is the Advanced Encryption Standard (AES) which was adopted in 2001 as FIPS 197, a U.S. government standard. It uses 128-bit blocks and either 128, 192, or 256-bit keys.

As of 2024, there are no known AES attacks that are significantly faster than exhaustive key search.

Cryptoscheme 2.9 (AES)

Given a key k and block of plaintext, initialize a 4×4 byte array State containing the plaintext.

Depending on the key size (128, 192, 256), let h be (10, 12, 14). Using the key schedule, generate $h+1$ round keys $k_0, \dots, k_h$. We will need three helper functions SubBytes, ShiftRows, and MixColumns.

Then, encryption is

State $\leftarrow$ block of plaintext
$(k_0, \dots, k_h) \leftarrow$ round keys from the key schedule
State $\leftarrow$ State $\oplus\, k_0$
for $i = 1, \dots, h-1$ **do**
    State $\leftarrow$ SubBytes(State)
    State $\leftarrow$ ShiftRows(State)
    State $\leftarrow$ MixColumns(State)
    State $\leftarrow$ State $\oplus\, k_i$
State $\leftarrow$ SubBytes(State) ▷ *Note: we skip MixColumns in the last round*
State $\leftarrow$ ShiftRows(State)
State $\leftarrow$ State $\oplus\, k_h$
return State

To decrypt, do everything backwards (making calls to InvSubBytes, InvShiftRows, and InvMixColumns).

AES does a lot of math over the Galois field $\mathrm{GF}(2^8)$.

Definition 2.4.2 ($\mathrm{GF}(2^8)$)

Consider the field $\mathbb{Z}/2\mathbb{Z}[y]$ of polynomials with coefficients in $\mathbb{Z}$ mod 2.

The finite field $\mathrm{GF}(2^8) = (\mathbb{Z}/2\mathbb{Z}[y])/(y^8 + y^4 + y^3 + y + 1)$ contains those polynomials with degree at most 7.

Addition and multiplication are defined normally (mod $y^8 + y^4 + y^3 + y + 1$).

We notate elements $a(y) \in \mathrm{GF}(2^8)$ as the binary string of their coefficients.

Example 2.4.3. The string $a = 11101100 =$ `ec` is identified with $a(y) = y^7 + y^6 + y^5 + y^3 + y^2$.

Since polynomial addition is coefficient-wise and $\mathbb{Z}/2\mathbb{Z}$ is isomorphic with XOR, we can treat $\mathrm{GF}(2^8)$ addition as binary string XOR.

Example 2.4.4. Let $b = 00111011 =$ `3b`. Then, $c(y) = a(y) + b(y) = y^7 + y^6 + y^4 + y^2 + y + 1$. We could have instead found $c = 11010111 =$ `d7` by noticing that $a \oplus b =$ `ec` $\oplus$ `3b` $=$ `d7`.

Multiplication requires a long division to find the answer mod $y^8 + y^4 + y^3 + y + 1$.

Example 2.4.5. Let $d(y) \coloneqq a(y) \cdot b(y)$. Calculate:

$$\begin{aligned} d(y) &= (y^7 + y^6 + y^5 + y^3 + y^2)(y^5 + y^4 + y^3 + y + 1) \\ &= y^{12} + 2y^{11} + 3y^{10} + 2y^9 + 3y^8 + 4y^7 + 4y^6 + 2y^5 + y^4 + 2y^3 + y^2 \\ &= y^{12} + y^{10} + y^8 + y^4 + y^2 \end{aligned}$$

Then, do polynomial long division:

$$\begin{array}{r|l} & y^4 \quad + y^2 \\ \hline y^8 + y^4 + y^3 + y + 1) & y^{12} + y^{10} + y^8 + \quad y^4 \quad + y^2 \\ & \underline{- y^{12} \quad + y^8 + y^7 \quad + y^5 + y^4} \\ & \quad y^{10} \quad + y^7 \quad + y^5 \\ & \underline{- y^{10} \quad + y^6 + y^5 \quad + y^3 + y^2} \\ & \quad y^7 + y^6 \quad + y^3 \end{array}$$

to conclude that the remainder is $y^7 + y^6 + y^3$. Therefore, $\texttt{ec} \cdot \texttt{3b} = 11001000 = \texttt{c8}$.

But by the XOR trick from addition, we can do this faster using XOR.

Example 2.4.6. First, long multiply 11101100 by 00111011 using XOR to reduce:

$$\begin{array}{r} 11101100 \\ 11101100 \\ 11101100 \\ 11101100 \\ \oplus\, 11101100 \\ \hline 1010100010100 \end{array}$$

Then, do long division by $f = 100011011$ using XOR to subtract:

$$\begin{array}{rl|llll} & & & 1 & 01 & 00 \\ \hline 100011011) & 10 & 1010001 & 01 & 00 \\ & 10 & 0011011 & & & \\ \hline & 00 & 1001010 & & & \\ & & 1000110 & 11 & & \\ \hline & & 0001100 & 10 & 00 & \end{array}$$

Therefore, $\texttt{ec} \cdot \texttt{3b} = 11001000 = \texttt{c8}$.

Now, we can define the helper functions. The substitution in AES is based on the inverse in $\mathrm{GF}(2^8)$.

Definition 2.4.7 (AES S-box)

Let $p \in \{0,1\}^8$. We define $S : \{0,1\}^8 \to \{0,1\}^8$.

Considering p as an element of $\mathrm{GF}(2^8)$, let $q = p^{-1}$ (which always exists except if $p = 0$, in which case let $q = 0$). Treating q as a bit vector, compute

$$S(p) = r = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} q + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

with scalar arithmetic in $\mathbb{Z}/2\mathbb{Z}$.

Then, SubBytes just applies S to each byte of State. The decryption call InvSubBytes just multiplies by the inverse of the matrix.

The permutation takes two steps: first, ShiftRows shifts the i^{th} row left by i bits. Then, MixColumns treats each column as a polynomial in $\mathrm{GF}(2^8)[x]/(x^4 - 1)$ and multiplies it by $c(x) = \texttt{02} + \texttt{01}x + \texttt{01}x^2 + \texttt{03}x^3$.

Example 2.4.8. Let $a = \texttt{d0f112bb}$ be a column. Multiply

$$\begin{aligned} a(x) \cdot c(x) &= (\texttt{d0} + \texttt{f1}x + \texttt{12}x^2 + \texttt{bb}x^3)(\texttt{02} + \texttt{01}x + \texttt{01}x^2 + \texttt{03}x^3) \\ &= (\texttt{d0} \cdot \texttt{02}) + (\texttt{d0} \cdot \texttt{01} + \texttt{f1} \cdot \texttt{02})x + (\texttt{d0} \cdot \texttt{01} + \texttt{f1} \cdot \texttt{01} + \texttt{12} \cdot \texttt{02})x^2 \\ &\quad + (\texttt{d0} \cdot \texttt{03} + \texttt{f1} \cdot \texttt{01} + \texttt{12} \cdot \texttt{01} + \texttt{bb} \cdot \texttt{02})x^3 \\ &\quad + (\texttt{f1} \cdot \texttt{03} + \texttt{12} \cdot \texttt{01} + \texttt{bb} \cdot \texttt{01})x^4 + (\texttt{12} \cdot \texttt{03} + \texttt{bb} \cdot \texttt{01})x^5 + (\texttt{bb} \cdot \texttt{03})x^6 \\ &= \texttt{bb} + \texttt{29}x + \texttt{05}x^2 + \texttt{e5}x^3 + \texttt{a1}x^4 + \texttt{8d}x^5 + \texttt{d6}x^6 \end{aligned}$$

where coefficient arithmetic is in $\mathrm{GF}(2^8)$. Find the remainder modulo $\texttt{01}x^4 - \texttt{01}$ by replacing $x^4 \mapsto 1$:

$$\begin{aligned} r(x) &= \texttt{bb} + \texttt{29}x + \texttt{05}x^2 + \texttt{e5}x^3 + \texttt{a1} + \texttt{8d}x^2 + \texttt{d6}x^3 \\ &= \texttt{1a} + \texttt{a4}x + \texttt{d3}x^2 + \texttt{e5}x^3 \end{aligned}$$

Therefore, MixColumn($\texttt{d0f112bb}$) = $\texttt{1aa4d3e5}$.

Naturally, InvShiftRows shifts the i^{th} row right by i bits and InvMixColumns multiplies each column by $c^{-1} = \texttt{0e09d00b}$.

Finally, we can define the key schedule. For 128-bit keys, we need 11 round keys. The first round key $k_0 = (r_0, r_1, r_2, r_3)$ is the actual AES key. Then, each subsequent round key

$$k_i = (r_{4i}, r_{4i+1}, r_{4i+2}, r_{4i+3}) = (f(r_{4i-1}) \oplus r_{4i-4}, r_{4i} \oplus r_{4i-3}, r_{4i+1} \oplus r_{4i-2}, r_{4i+2} \oplus r_{4i-1})$$

where f_i maps the four bytes (a, b, c, d) to $(S(b) \oplus \ell_i, S(c), S(d), s(a))$ for some round constants ℓ_i.

Aside: Implementation This section is just me doing nerd shit trying to make Assignment 2 easier. The finite fields used in AES can be replicated in Sage or Mathematica. In Sage:

```
aes.<y> = GF(2^8, modulus=x^8+x^4+x^3+x+1) # define AES field (ℤ/2ℤ)[y]/(f(y))
aes_int = aes._cache.fetch_int             # byte to GF(2^8) element
mcf.<x> = aes[]                            # MixColumns field GF(2^8)[x]
hex_string = lambda x: bytes(u.integer_representation() for u in x.list()).hex()

# Example 2.4.5: multiply ec * 3b
a, b = aes_int(0xec), aes_int(0x3b)
r = a * b
print('r(x):', r)
print('a*b:', hex(r.integer_representation()))
# r(x): y^7 + y^6 + y^3
# a*b:  0xc8

# Example 2.4.8: MixColumn(d0f112bb)
a = [0xd0,0xf1,0x12,0xbb]
ax = mcf([aes_int(u) for u in a])
cx = (y + x + x^2 + (y+1) * x^3)
bx = (ax * cx).mod(x^4+1)
print('b(x):', bx)
print('b:', hex_string(bx))
# b(x): (y^7 + y^6 + y^5 + y^2 + 1)x^3 + (y^7 + y^6 + y^4 + y + 1)x^2 + (y^7 + y^5 + y^2)x + y^4 + y^3 + y
# b:    1aa4d3e5
```

In Mathematica (version 13.3 or later):

```
F = FiniteField[2, #^8 + #^4 + #^3 + # + 1 &];
GF[hex_] := F[FromDigits[hex, 16]];
poly[f_] := Expand@FromDigits[Reverse@f["Coefficients"], y];
hex[f_] := IntegerString[f["Index"], 16, 2];

(* Example 2.4.5: multiply ec * 3b *)
a = GF["ec"];
b = GF["3b"];
poly[a*b]       (* y^3 + y^6 + y^7 *)
hex[a*b]        (* c8 *)

(* Example 2.4.8: MixColumn(d0f112bb) *)
a = GF["d0"] + GF["f1"] x + GF["12"] x^2 + GF["bb"] x^3;
c = F[2] + F[1] x + F[1] x^2 + F[3] x^3;
b = PolynomialRemainder[a*c, x^4 - 1, x];
StringJoin[hex /@ CoefficientList[b, x]] (* 1aa4d3e5 *)
```

2.5 Block cipher modes of operation

TODO

Chapter 3

Hash functions

3.1 Definitions

Lecture 7
Jan 22

Definition 3.1.1 (hash function)

A mapping H such that

1. $H : \{0,1\}^{\leq L} \to \{0,1\}^n$ maps binary messages of arbitrary lengths $\leq L$ to outputs of a fixed length n.
2. $H(x)$ can be efficiently computed for all $x \in \{0,1\}^{\leq L}$

is an n-bit hash function. We call $H(x)$ the hash or digest of x.

We usually suppose that L is large and just write $H : \{0,1\}^* \to \{0,1\}^n$.

In a more general context, a hash function is an efficiently computable function.

Example 3.1.2. Let $H : \{0,1\}^{\leq 4} \to \{0,1\}^2$ be a hash function mapping a bitstring to its last two digits. For example, $H(1101) = 01$.

We call 1001 a preimage of 01.

The pair $(01, 1001)$ is a collision where 01 is a second preimage of 1001.

Generically, we can create a hash function given a block cipher.

Cryptoscheme 3.1 (Davies–Meyer hash function)

Let E_k be an m-bit block cipher with n-bit key k. Let IV be a fixed m-bit initializing value.

Then, to compute $H(x)$,

1. Break up $x \parallel 1$ into n-bit blocks $\bar{x} = x_1, \dots, x_t$ (padding x_t with 0s if necessary)
2. $H_0 \leftarrow \mathsf{IV}$
3. $H_i \leftarrow E_{x_i}(H_{i-1}) \oplus H_{i-1}$ for all $i = 1, \dots, t$
4. $H(x) \leftarrow H_t$

Hash functions are used basically everywhere in cryptography, mostly just because they are stupidly fast and introduce "scrambling" that, given a good enough hash function, cannot be reversed.

Definition 3.1.3 (preimage resistance)

A hash function $H = \{0,1\}^* \to \{0,1\}^n$ is preimage resistant (PR) if, given a hash value $y \in_{\mathsf{R}} \{0,1\}^n$, it is computationally infeasible to find any $x \in \{0,1\}^*$ with $H(x) = y$ with non-negligible success probability.

Note that we include disclaimers like "non-negligible success probability" since otherwise we could just use an attack like "guess! it might just work!"

This is helpful for implementing passwords. If we store $(\mathsf{password}, H(\mathsf{password}))$ with a PR hash H, then stealing the system password file does not actually reveal the passwords.

Definition 3.1.4 (2nd preimage resistance)

A hash function $H = \{0,1\}^* \to \{0,1\}^n$ is 2nd preimage resistant (2PR) if, given $x \in_{\mathsf{R}} \{0,1\}^*$, it is computationally infeasible to find any $x' \in \{0,1\}^*$ with $x' \neq x$ and $H(x') = H(x)$ with non-negligible success probability.

This is helpful for ensuring that a message is unchanged (Modification Detection Codes; MDCs). To ensure a message m is unmodified, publicize $H(m)$. Then, as long as H is 2PR and we can verify the hash, we can safely assume m is unmodified.

Definition 3.1.5 (collision resistance)

A hash function $H = \{0,1\}^* \to \{0,1\}^n$ is collision resistant (CR) if it is computationally infeasible to find distinct $x, x' \in \{0,1\}^*$ where $H(x') = H(x)$.

This allows us to optimize message signing. Instead of signing a large file x, Alice can sign $H(x)$ instead. Keeping all the desired properties of a signing scheme requires PR, 2PR, and CR.

↑ Lectures 5, 6, and 7 taken directly from slides ↑ *Lecture 8 Jan 24*

Proposition 3.1.6

If H is CR, then H is 2PR.

Proof. Take the contrapositive: H is not 2PR $\implies$ H is not CR.

Suppose H is not 2PR, i.e., we have an efficient algorithm to find a collision x' given x.

Select a random x. Get the collision x' from our algorithm. Then, we have a collision (x, x') that we found efficiently, so H is not CR. □

It will *always* be easier to do the contrapositive in this course, especially because most definitions use "it is not possible".

Proposition 3.1.7

CR does not guarantee PR.

Proof. We give a counterexample.

Suppose that $H : \{0,1\}^* \to \{0,1\}^n$ is CR.

Consider the hash function $\bar{H} = \{0,1\}^* \to \{0,1\}^{n+1}$ defined by

$$\bar{H}(x) = \begin{cases} 0 \parallel H(x) & x \notin \{0,1\}^n \\ 1 \parallel x & x \in \{0,1\}^n \end{cases}$$

where $\parallel$ denotes the concatenation operation. Then $\bar{H}$ is CR because H is.

However, $\bar{H}$ is not PR for at least half of all $y \in \{0,1\}^{n+1}$ we can efficiently find the preimage (i.e., for all the hash values beginning with 1, we can just lop off the 1 to get the original). □

Note: if we disallow pathological hash functions like this, i.e., we have some constraint on uniformity in the size of preimages, CR does guarantee PR.

Proposition 3.1.8

Suppose H is somewhat uniform, i.e., preimages are all around the same size. If H is CR, then H is PR.

Proof. Suppose that $H : \{0,1\}^* \to \{0,1\}^n$ is not PR. We must show H is not CR.

Select $x \in_R \{0,1\}^*$ and compute $y = H(x)$. Since H is not PR, we can efficiently find $x' \in \{0,1\}^*$ with $H(x') = y$. Since H is somewhat uniform, we expect that y has many preimages, so $x' \neq x$ with very high probability.

Therefore, (x, x') is a collision and H is not CR. □

Proposition 3.1.9

PR does not guarantee 2PR.

Proof. Suppose that $H : \{0,1\}^* \to \{0,1\}^n$ is PR.

Define $\bar{H} : \{0,1\}^* \to \{0,1\}^n$ by $\bar{H}(x_1 x_2 \dots x_t) = H(0 x_2 \dots x_t)$.

Then, $\bar{H}$ is PR but not 2PR. □

Proposition 3.1.10

Suppose H is somewhat uniform. If H is 2PR, then H is PR.

Proof. Suppose that $H : \{0,1\}^* \to \{0,1\}^n$ is not PR and show it is not 2PR.

Suppose we are given $x\{0,1\}^*$. Compute $y = H(x)$ and then find with our PR-breaking algorithm x' with $H(x') = y$. Since H is somewhat uniform, we expect $x \neq x'$.

Then, we have a collision x' for x and that breaks PR. □

Proposition 3.1.11

2PR does not guarantee CR.

Proof. Suppose that $H : \{0,1\}^* \to \{0,1\}^n$ is 2PR.

Consider $\bar{H} : \{0,1\}^* \to \{0,1\}^n$ defined by $\bar{H}(x) = H(x)$ except $\bar{H}(1) = H(0)$.

Then, $\bar{H}$ is not CR because $(0,1)$ is a collision.

However, we can show $\bar{H}$ is 2PR. Suppose $\bar{H}$ is *not* 2PR. We show H would also not be 2PR.

Suppose we are given some x. Since $\bar{H}$ is not 2PR, we can find $x' \neq x$ with $\bar{H}(x') = \bar{H}(x)$. With almost certain probability, we can assume $x \neq 0, 1$. Then, $\bar{H}(x) = H(x)$. If $x' \neq 1$, we have $\bar{H}(x') = H(x') = H(x)$. Otherwise, $\bar{H}(x') = \bar{H}(1) = H(0) = H(x)$ We found a second preimage x' or 0 for x, so H is not 2PR.

Therefore, by contradiction, $\bar{H}$ is 2PR. □

Theorem 3.1.12 (relation between PR, 2PR, CR)

Summarize:

If H is $\downarrow$, then it is $\rightarrow$	PR	2PR	CR
PR	–	$\not\Rightarrow$	$\not\Rightarrow$
2PR	$\Rightarrow^*$	–	$\not\Rightarrow$
CR	$\Rightarrow$	$\Rightarrow^*$	–

where $\Rightarrow^*$ means "implies under somewhat uniformity"

3.2 Attacks

Lecture 9
Jan 26

Definition 3.2.1 (generic attack)
An attack which does not exploit any specific properties of a hash function. That is, it works on any generic hash function $H : \{0,1\}^* \to \{0,1\}^n$.

To analyze a generic attack, we assume H is a random function in the sense that $y = H(x)$ can be treated as $y \in_R \{0,1\}^n$.

Attack 3.2 (generic attack for preimages)
Given $y \in_R \{0,1\}^n$, repeatedly select arbitrary x until $H(x) = y$.

This will take 2^n attempts, so as long as $n \geq 128$ we are safe.

Attack 3.3 (generic attack for collisions)
Select arbitrary $x \in \{0,1\}^*$ and store $(H(x), x)$ in a table sorted by the first entry. Repeat until a collision is found.

By the birthday paradox, the expected number of hash operations is $\sqrt{\pi 2^n/2} \approx \sqrt{2^n}$. Therefore, the attack is infeasible for $n \geq 256$.

The space complexity is also $\mathcal{O}\left(\sqrt{2^n}\right)$. This is important since, for example, $n = 128$ has a feasible runtime 2^{64} but an infeasible space requirement of 500 million TB.

We can prove that this is the optimal generic collision attack, i.e., no faster generic attack exists.

However, we can improve the space complexity.

Let $N = 2^n$. Define a sequence $(x_i)_{i \geq 0}$ by $x_0 \in_R \{0,1\}^n$ and $x_i = H(x_{i-1})$.

Since $(x_i) \subseteq \{0,1\}^n$, we will eventually get repetitions. Therefore, (x_i) is eventually periodic, i.e., we will eventually get $x_a = x_b$ for $a \neq b$. Then, we found a collision (x_{a-1}, x_{b-1}).

More formally, let j be the smallest index for which $x_j = x_i$ for some $i < j$, which must exist. Then, $x_{j+\ell} = x_{i+\ell}$ for all $\ell \geq 1$.

By the birthday paradox, $\mathrm{E}[j] \approx \sqrt{\pi N/2} \approx \sqrt{N}$. In fact, since i is a random element from before j, we can say $\mathrm{E}[i] \approx \frac{1}{2}\sqrt{N}$ and $\mathrm{E}[j-i] \approx \frac{1}{2}\sqrt{N}$.

We will store only some distinguished points, for example, elements where the top 32 bits are all 0. Let θ be the proportion of distinguished points. Here, $\theta = 2^{-32}$.

We can still tell detect a cycle as long as there is a distinguished point in the cycle. Once we detect a collision, we work through the sequence near it.

Attack 3.4 (van Oorschot–Wiener parallel collision search)

We write this attack as two stages:

```
Create a table T
procedure DetectCollision(H)
    Select x_0 ∈_R {0,1}^n
    T[x_0] ← (0, −)                          ▷ store (index, last distinguished point)
    c ← 0                                    ▷ last distinguished point
    for d = 1, 2, 3 ... do
        x_d ← H(x_{d−1})
        if x_d is distinguished then
            if T[x_d] exists as (b, x_a) then
                (a, −) ← T[x_a]              ▷ need the index of x_a
                return FindCollision(x_a, x_c, a, b, c, d)
            T[x_d] ← (d, c)
            c ← d
procedure FindCollision(x_a, x_c, a, b, c, d)
    ℓ_1 ← b − a, ℓ_2 ← d − c
    Suppose ℓ_1 ≥ ℓ_2 so k ← ℓ_1 − ℓ_2
    Compute x_{a+1}, ... , x_{a+k}
    m ← 1
    repeat
        Compute x_{a+k+m}, x_{c+m}
        m ← m + 1
    until x_{a+k+m} = x_{c+m}
    return (x_{a+k+m−1}, x_{c+m−1})          ▷ the collision is H(x_{a+k+m−1}) = H(x_{c+m−1})
```

In DetectCollision, we will call the hash function $\sqrt{\pi N/2} + \frac{1}{\theta}$ times.

In FindCollision, we perform at most $\frac{3}{\theta}$ hashes.

In total, we expect to take $\sqrt{N} + \frac{4}{\theta}$ time. But this time we only need $3n\theta\sqrt{N}$ space.

So for our $n = 128$ case with $\theta = 2^{-32}$, the expected runtime is 2^{64} hashes (feasible) and the expected storage is 192 GB (negigible).

Lecture 10
Jan 29

We can parallelize VW collision search by having each processor start on a random point and report discovered distinguished points to a central server.

For m processors, we get an expected time of $\frac{1}{m}\sqrt{N} + \frac{4}{\theta}$ hashes and space of $3n\theta\sqrt{N}$ bits. That is, we get a speedup of m times.

This is also nice because there is no communication between processors and only occasional communication with the central server (reducing the chance of race conditions and other parallelism problems).

3.3 Iterated hash functions

Cryptoscheme 3.5 (Merkle's meta method)

Fix an initializing value $\mathsf{IV} \in \{0,1\}^n$ and pick a compression function $f : \{0,1\}^{n+r} \to \{0,1\}^n$.

Given a b-bit message x, to compute $H(x)$:

1. Break up x into r-bit blocks $\bar{x} = x_1, \dots, x_t$ (padding the last block with 0s if necessary)
2. Define x_{t+1} to hold the binary representation of b (left-padding with 0s as necessary)
3. Define $H_0 = \mathsf{IV}$
4. Compute $H_i = f(H_{i-1} \parallel x_i)$ for $i = 1, \dots, t+1$
5. Return $H(x) = H_{t+1}$

Merkle also proved that collision resistance depends on f.

Theorem 3.3.1 (Merkle)

If the compression function f is collision resistant, then the iterated hash function H is also collision resistant.

Note that by thm. 3.1.12, we get PR and 2PR as well.

This feels very circular, but it can be helpful to give a proof of security given certain very precise definitions. However, the assumptions in the definitions might not be realistic.

Proof. Suppose that H is not CR. We will show that f is not CR.

Since H is ont CR, we can efficiently find messages $x, x' \in \{0,1\}^*$ with $x \neq x'$ and $H(x) = H(x')$.

Define $\bar{x} = x_1, \dots, x_t$, $b = |x|$, length block x_{t+1}, and $\bar{x}' = x'_1, \dots, x'_{t'}$, $b' = |x'|$, length block $x'_{t'+1}$.

Then, we can efficiently compute

$$\begin{aligned} H_0 &= \mathsf{IV} \\ H_1 &= f(H_0, x_1) \\ H_2 &= f(H_1, x_2) \\ H_3 &= f(H_2, x_1) \\ &\vdots \\ H_{t-1} &= f(H_{t-2}, x_{t-1}) \\ H(x) = H_t &= f(H_{t-1}, x_t) \end{aligned} \qquad \begin{aligned} H_0 &= \mathsf{IV} \\ H'_1 &= f(H_0, x'_1) \\ H'_2 &= f(H'_1, x'_2) \\ H'_3 &= f(H'_2, x'_3) \\ &\vdots \\ H'_{t'-1} &= f(H'_{t'-2}, x'_{t'-1}) \\ H(x') = H'_{t'} &= f(H'_{t'-1}, x'_{t'}) \end{aligned}$$

Since $H(x) = H(x')$, we have $H_t = H'_{t'}$.

Now, if $b \neq b'$, then $x_{t+1} \neq x'_{t'+1}$. Then, $(H_t \parallel x_{t+1}, H'_{t'} \parallel x'_{t'+1})$ is a collision for f.

Otherwise, if $b = b'$, then $t = t'$ and $x_{t+1} = x'_{t+1}$. Let i be the largest index for which $(H_i \parallel x_{i+1}) \neq (H'_i \parallel x'_{i+1})$ which must exist because $x \neq x'$.

Then, $H_{i+1} = f(H_i, x_{i+1}) = f(H'_i, x'_{i+1}) = H'_{i+1}$ and we have a collision $(H_i \parallel x_{i+1}, H'_i \parallel x'_{i+1})$.

Since we found a collision, f is not CR. □

Aside: MD5 is bad MDx is a family of iterated hash functions. MD4 was designed by Rivest in 1990 with a security level against VW of 64 bits but broken *by hand* by Wang in 2004 with an attack reducing the security level to 4 bits. MD4 preimages can also be found in 2^{102} operations, which is infeasible but also still bad.

In 1991, Rivest designed MD5, a strengthened version of MD4. The Wang attack reduced the security level of 39 bits, but modern attacks can find collisions in 2^{24} operations.

In summary, MD5 should not be used if collision resistance is required but it's *probably* preimage resistant. In fact, the Flame malware used a forged MD5-based Microsoft certificate created using an improved version of Wang's attack.

Lecture 11
Jan 31

For another example of why this is a problem, consider the fact that Crowdmark uses MD5 hashes to verify that a submitted file is the same (in case of an upload error). A student could change their answer and submit the "same" file after looking at the solutions after the deadline.

The SHA family of functions are designed by the NSA. Wang (our recurring character) attacked SHA (1993) to 39 bits and SHA-1 (1994) to 63 bits. The SHA-2 family, a variable-length output version of SHA-1, has no known weaknesses. The 224-, 256-, 384-, and 512-bit lengths are chosen so that the security levels (against VW collision finding) line up with the security levels of Triple-DES, AES-128, AES-192, and AES-256.

Cryptoscheme 3.6 (SHA-256)

SHA-256 is an iterated hash function with block length $r = 512$, hash length $n = 256$, and compression function $f : \{0,1\}^{256+512} \to \{0,1\}^{256}$.

The design principles are classified, so we can treat it as a black box.

Define $h_1, \dots, h_8$ as the fractional parts of the square roots of the first eight primes and $y_0, \dots, y_{63}$ as the fractional parts of the cube roots of the first 64 primes. Finally, let

$$f(A,B,C) = AB \oplus \overline{B}C \qquad g(A,B,C) = AB \oplus AC \oplus BC$$
$$r_1(A) = (A \hookrightarrow 2) \oplus (A \hookrightarrow 13) \oplus (A \hookrightarrow 22) \qquad r_2(A) = (A \hookrightarrow 6) \oplus (A \hookrightarrow 11) \oplus (A \hookrightarrow 25)$$
$$r_3(A) = (A \hookrightarrow 7) \oplus (A \hookrightarrow 18) \oplus (A \gg 3) \qquad r_4(A) = (A \hookrightarrow 17) \oplus (A \hookrightarrow 19) \oplus (A \gg 10)$$

To find the hash of a b-bit message x made of 32-bit words $x_0, x_1 \dots$:

1: pad x with 1 followed by 0s until the bitlength is $-64 \pmod{512}$.
2: append a 64-bit representation of $b \pmod{2^{64}}$
3: initialize $(H_1, \dots, H_8) \leftarrow (h_1, \dots, h_8)$
4: **for** $i = 0, \dots, m-1$ **do**
5: $\quad X_j \leftarrow x_{16i+j}$ **for** $0 \le j \le 15$ ▷ *copy i^{th} 16-word block into temp storage*
6: $\quad X_j \leftarrow r_4(X_{j-2}) + X_{j-7} + r_3(X_{j-15}) + X_{j-16}$ **for** $16 \le j \le 63$ ▷ *expand X to 64 words*
7: $\quad (A, B, \dots, G, H) \leftarrow (H_1, H_2, \dots, H_7, H_8)$ ▷ *initialize working variables*
8: $\quad$ **for** $j = 0, \dots, 63$ **do** ▷ *weird random shuffling*
9: $\quad\quad T_1 \leftarrow H + r_2(E) + f(E,F,G) + y_j + X_j$
10: $\quad\quad T_2 \leftarrow r_1(A) + g(A,B,C)$
11: $\quad\quad (H, G, F, E, D, C, B, A) \leftarrow (F, G, E, D + T_1, C, B, A, T_1 + T_2)$
12: $\quad (H_1, \dots, H_8) \leftarrow (H_1 + A, \dots, H_8 + H)$ ▷ *update working variables*
13: **return** $H_1 \| H_2 \| \cdots \| H_8$

It would be very profitable to crack SHA-256 since it is used to verify proof of work for Bitcoin mining. Finding messages with arbitrary numbers of 0s at the starts of hashes would print money.

Just to be sure Wang can't come back and break SHA-2 (since it is still a Merkle design). SHA-3 (Keecak, based on "sponge construction") was selected in 2012, but nobody really uses it because SHA-256 is still better.

Chapter 4

Message authentication codes

Lecture 12
Feb 2

4.1 Definitions

Definition 4.1.1 (message authentication code)

A family of functions $\text{MAC}_k : \{0,1\}^* \to \{0,1\}^n$ paramaterized by an ℓ-bit key k where each function MAC_k can be efficiently computed.

The MAC or tag of a message x is denoted $t = \text{MAC}_k(x)$.

We use MAC schemes to provide data integrity and origin verification. To do this:

1. Alice and Bob establish a secret key $k \in \{0,1\}^\ell$.
2. Alice computes the tag $t = \text{MAC}_k(x)$ of a message x and sends (x,t) to Bob.
3. Bob verifies that $t = \text{MAC}_k(x)$.

To avoid a replay attack (Eve saves a copy of a message and resends it later), add a timestamp or sequence number.

Like with encryption, we have to formulate a security definition.

Definition 4.1.2 (MAC security)

A MAC scheme is secure if it is existentially unforgeable under a chosen-message attack.

That is, for chosen messages x_i and their MACs t_i, it is computationally infeasible to find with non-negligible success probability a valid message-MAC pair (x,t) for a new message x.

Realistically, the messages x_i will have to be "harmless" messages that Alice is ordinarily willing to tag and the forgery x is a "harmful" message that Alice would ordinarily be unwilling to tag.

Definition 4.1.3

An ideal MAC scheme is one where for each key $k \in \{0,1\}^\ell$, the function $\text{MAC}_k : \{0,1\}^* \to \{0,1\}^n$ is a random function.

The naive generic attack is to just guess.

Attack 4.1 (generic attack for tags)
Select $y \in_R \{0,1\}^n$ and guess that $\text{MAC}_k(x) = y$. Keep guessing.

Assuming an ideal scheme, the success probability is $\frac{1}{2^n}$.

Attack 4.2 (generic attack for keys)
Perform the same attack as on an SKES.

Assuming an ideal scheme, the expected number of keys for which r messages verify is $1 + \mathsf{FK} = 1 + (2^\ell - 1)/2^r$. If FK is negligible, the expected number of operations is $2^{\ell-1}$.

4.2 Specific MACs

Cryptoscheme 4.3 (CBC-MAC)
Let E be an n-bit block cipher with key space $\{0,1\}^\ell$. We assume that plaintext messages all have lengths that are multiples of n. To compute $\text{CBC-MAC}_k(x)$:

1: Divide x into n-bit blocks $x_1, \ldots, x_r$.
2: $H_1 \leftarrow E_k(x_1)$
3: **for** $i = 2, \ldots, r$ **do**
4: $\quad H_i \leftarrow E_k(H_{i-1} \oplus x_i)$
5: **return** H_r

It was proven in 1994 that CBC-MAC with fixed-length messages is secure if E is ideal (i.e., $E_k : \{0,1\}^n \to \{0,1\}^n$ is random).

However, it is totally broken for variable-length messages.

Proof. Select an arbitrary 3-block message $x = (x_1, x_2, x_3)$. Obtain $t_1 = \text{CBC-MAC}_k(x_1) = E_k(x_1)$. Obtain $t_2 = \text{CBC-MAC}_k((t_1 \oplus t_2) \parallel x_3) = E_k(E_k(t_1 \oplus x_2) \oplus x_3)$. Then, (x, t_2) is a forgery. □

One way to fix this is to add one more encryption round.

Cryptoscheme 4.4 (Encrypted CBC-MAC (EMAC))
Given a second key s for E, let $\text{EMAC}_{k,s}(x) = E_s(\text{CBC-MAC}_k(x))$.

Again, it has been proven that EMAC is secure if E is ideal.

Now, consider creating a MAC based on a hash function. Let H be an iterated n-bit hash function with compression function $f : \{0,1\}^{n+r} \to \{0,1\}^n$. Pick $k \in_R \{0,1\}^n$ and let $K \in \{0,1\}^r$ be k padded with 0s.

We might propose $\text{MAC}_k(x) = H(K \parallel x)$. However, this is insecure under a length extension attack since if we know $(x, \text{MAC}_k(x))$, we can calculate $\text{MAC}_k(x \parallel y)$ for arbitrary y by resuming the hash process with the last block of x (i.e., computing $f(t \parallel y_1)$, etc.).

Exercise 4.2.1. Show that this is also insecure if messages are of arbitrary length and a length block is added to $K \parallel x$.

Securing against this attack is as simple as adding the key to each step, giving us a secure hash-based MAC ("HMAC")

Cryptoscheme 4.5 (HMAC)

Let H be a hash function. Define r-bit constants $\mathsf{opad} = \mathtt{3636\cdots36}$ and $\mathsf{ipad} = \mathtt{5c5c\cdots5c}$.

Then, $\text{HMAC}_k(x) = H(K \oplus \mathsf{opad}, H(K \oplus \mathsf{ipad}, x))$.

Lecture 13
Feb 5

The security analysis of HMAC is hard, but we can prove a theorem.

Theorem 4.2.2

Suppose that the f compression function used in H is a secure MAC with fixed-length messages and a secret IV as the key. Then, HMAC is a secure MAC scheme.

Usually, HMAC uses SHA-256. It is widely used for internet security.

HMAC is also widely used as a key derivation function. If Alice has k and needs multiple session keys sk_i, she can compute $sk_1 = \text{HMAC}_k(1)$, $sk_2 = \text{HMAC}_k(2)$, etc. Then, since HMAC is secure, Eve can intercept individual session keys and still know nothing about either the actual key k or the other session keys.

4.3 GSM

TODO

Chapter 5

Authenticated encryption

5.1 Generic schemes

We have established ways to encrypt data (e.g., AES-CBC) and ways to authenticate data (i.e., MACs) but what if we need both?

Cryptoscheme 5.1 (Encrypt-and-MAC)

Alice sends $(c, t) = (E_{k_1}(m), \text{MAC}_{k_2}(m))$ to Bob, where m is the plaintext and (k_1, k_2) is a secret key shared with Bob.

Then, Bob first decrypts c to obtain $m = E_{k_1}^{-1}(c)$ and then verifies $t = \text{MAC}_{k_2}(m)$.

This is insecure because $\text{MAC}_{k_2}(m)$ might leak information about m (MACs are generally not semantically secure).

Cryptoscheme 5.2 (Encrypt-then-MAC)

Alice sends $(c, t) = (E_{k_1}(m), \text{MAC}_{k_2}(c))$ to Bob, where m is the plaintext and (k_1, k_2) is a secret key shared with Bob.

Then, Bob first verifies that $t = \text{MAC}_{k_2}(c)$ and decrypts c to obtain $m = E_{k_1}^{-1}(c)$.

This has been proven to be secure given that E_{k_1} and MAC_{k_2} are secure.

Definition 5.1.1 (authentication encryption security)

An authentication encryption scheme is secure if

1. it is semantically secure against chosen-plaintext attacks, and
2. has ciphertext integrity, i.e., given $(m_1, c_1, t_1), \ldots, (m_\ell, c_\ell, t_\ell)$, an attacker cannot forge a valid (m, c, t).

Lecture 14
Feb 7

5.2 AES-GCM

The most popular AE scheme is AES-GCM which uses AES-CTR and GMAC.

Recall how ChaCha20 (scheme 2.4) used a key, nonce, and counter to generate a keystream. AES-CTR uses AES in a similar stream cipher paradigm.

Cryptoscheme 5.3 (AES-CTR)

Let $k \in_R \{0,1\}^{128}$ be a shared secret and $M = (M_1, \dots, M_u)$ be a message of 128-bit blocks. To encrypt:

1: Select a nonce $\mathsf{IV} \in \{0,1\}^{96}$,
2: $J_0 \leftarrow \mathsf{IV} \parallel 0^{31} \parallel 1$,
3: **for** $i = 1, \dots, u$ **do**
4: $\quad J_i \leftarrow J_{i-1} + 1$
5: $\quad C_i \leftarrow \mathrm{AES}_k(J_i) \oplus M_i$
6: **return** $(\mathsf{IV}, C_1, \dots, C_u)$

To decrypt, generate the keystream and XOR.

Since it is a counter, CTR encryption is parallelizable. We also require, as with other IV-containing schemes, that the IV be unique and never reused.

To define GMAC, recall def. 2.4.2.

Definition 5.2.1 ($\mathrm{GF}(2^{128})$)

The field $\mathbb{Z}/2\mathbb{Z}[x]$ modulo $f(x) = 1 + x + x^2 + x^7 + x^{128}$.

We associate a 128-bit block $a = a_0 \cdots a_{127}$ with the polynomial $a(x) = a_0 + a_1 x + \cdots + a_{127} x^{127}$.

As in $\mathrm{GF}(2^8)$, we define $a \cdot b = a(x) \cdot b(x) \pmod{f(x)}$.

Then, we can describe GMAC:

Cryptoscheme 5.4 (Galois Message Authentication Code (GMAC))

Let $A = (A_1, \dots, A_v)$ be the message in 128-bit blocks, L be the bitlength of A as a 128-bit block, and k be a secret key.

1: $J_0 \leftarrow \mathsf{IV} \parallel 0^{31} \parallel 1$
2: $H \leftarrow \mathrm{AES}_k(0^{128})$
3: Define $f_A(y) = A_1 y^{v+1} + A_2 y^v + \cdots + A_{v-1} y^3 + A_v y^2 + Ly \in \mathrm{GF}(2^{128})[y]$
4: $t \leftarrow \mathrm{AES}_k(J_0) \oplus f_A(H)$
5: **return** (IV, A, t)

Proposition 5.2.2

GMAC is secure.

Proof (outline). Consider the simplified tag $t' = f_A(H)$.

Then, the adversary can guess with probability $\frac{1}{2^{128}}$.

She can also guess t' by making a guess H' and computing $f_A(H')$ with success probability $\frac{v+1}{2^{128}}$.

But if Eve sees a single valid message-tag pair (A, t'), she can solve $f_A(H) = t'$ for H.

To avoid this attack, we add a one-time pad $\text{AES}_k(J_0) \oplus t$. □

Finally, we can define AES-GCM. The scheme will encrypt/authenticate a message and also authenticate (but not encrypt) an encryption context.

Cryptoscheme 5.5 (AES-GCM)

Let $M = (M_1, \dots, M_u)$ for $u \leq 2^{32} - 2$ be the message, $A = (A_1, \dots, A_v)$ be the encryption context, and $k \in_R \{0,1\}^{128}$ be a shared secret.

To encrypt/sign:

$L \leftarrow |A| \parallel |M|$
$J_0 \leftarrow \mathsf{IV} \parallel 0^{31} \parallel 1$ for unique $\mathsf{IV} \in \{0,1\}^{96}$
for $i = 1, \dots, u$ **do** ▷ *Encryption with AES-CTR*
    $J_i \leftarrow J_{i-1} + 1$
    $C_i \leftarrow \text{AES}_k(J_i) \oplus M_i$
$H \leftarrow \text{AES}_k(0^{128})$ ▷ *Authentication with GMAC*
Define $f_{A,C}(x) = A_1 x^{u+v+2} + \cdots + A_v x^{u+2} + C_1 x^{u+1} + \cdots + C_u x^2 + Lx$
$t \leftarrow \text{AES}_k(J_0) \oplus f_{A,C}(H)$
return (IV, A, C, t)

To decrypt/authenticate:

AES-GCM does both authentication and encryption, but it can be used to do just authentication by passing $M = \varepsilon$.

There are very fast hardware implementations of AES and multiplication under $\text{GF}(2^{128})$. If we use Horner's rule, we can evaluate an n-degree polynomial in $\text{GF}(2^{128})[x]$ with n PCLMUL instructions and $n-1$ XOR instructions.

We can also parallelize since we can assign blocks to other processors and split up the polynomial $f_{A,C}$ into multiple parts for evaluation.

The scheme can be used in streaming mode, which is helpful.

This is also proven to be secure.

Lecture 15
Feb 9

In general, an AES-GCM IV should never be reused.

If we reuse $(\mathsf{IV}, A_1, C_1, t_1)$ and $(\mathsf{IV}, A_2, C_2, t_2)$, then $t_1 \oplus t_2 = f_{A_1,C_1}(H) \oplus f_{A_2,C_2}(H)$. This is a polynomial function of H and can be solved. The entire keystream can then be derived from H.

5.3 Real-life example: AWS

AWS requires security for data in transit, data in use, and data at rest.

Lecture 16
Feb 12

Every data item is encrypted with a one-time data encryption key (DEK).

To upload: A client makes a request to a key management service (KMS) server, which gets the one-time key from an HSM and passes it on. Then, the client sends the ciphertext to an S3 server.

Alternatively, a client can send data directly to the S3 server, which queries the KMS (querying the HSM) for a key an doing the encryption server-side.

To keep the DEKs secure, we use a customer main key (CMK). The HSM generates a CMK K and DEK k, then sends k and $wk = E_K(A, k)$ to the KMS, which passes it to the client. Then, the client uses k and immediately deletes it, storing only wk and the ciphertext.

To decrypt: The client sends wk to the HSM (via the KMS). The HSM decrypts wk and returns k. The client uses k and immediately deletes it.

The IVs used are guaranteed to be randomly generated with collision probability better than 2^{-32}. However, by the birthday paradox, this means that for B IVs, there is a $B^2/(2 \cdot 2^{96})$ chance of collision. Therefore, never encrypt more than 2^{32} chunks with the same DEK.

This also applies to random IVs for use of CMKs, so a CMK should only be used for 2^{32} DEKs. AWS only rotates CMKs once a year, so a user can only encrypt 136 DEKs per second.

To solve this, derive key mode works by setting $wk = \text{Enc}_L(A, k)$ where $L = \text{HMAC}_K(N)$. It permits 2^{40} CMKs, each encrypting 2^{50} DEKs.

AWS charges \$1/mo for CMKs, and \$0.03/10,000 requests to the KMS API (after a free tier of 20,000). DynamoDB is a NoSQL databse with unique DEKs for each table entry, which is just kinda stupid crazy.

Lecture 17
Feb 14

CMKs are never stored (unencrypted) on disk and never transmitted off its HSM. A CMK is encrypted with a domain key (shared across HSMs in a domain) to create an exported key token (EKT) which actually put in storage.

The domain key is rotated daily. When an HSM selects the new domain key, it is encrypted with a one-time AES-GCM key ℓ. Then, ℓ is encrypted with ECC to be exported to the other HSMs, adding another layer to the long chain of

$$\text{ECC keys} \to \ell \to \text{domain key} \to \text{CMK} \to \text{DEK} \to \text{actual data}$$

to create an exported domain token (EDT), which is distributed to the domain and stored in storage.

Chapter 6

Public-key cryptography

6.1 Overview

Symmetric-key cryptography is limited because it assumes that communicating parties share the secret key beforehand. This can be established in a few ways:

1. Point-to-point distribution. Alice sends it to Bob over a secured channel, like a face-to-face meeting or a pre-secured SIM card.
2. Through a trusted third party (TTP) which acts as a key distribution centre (KDC). The KDC picks keys and sends them to each party, encrypted by pre-established keys.

These are inefficient. In a network of n users, each user needs to manage $n-1$ secret keys for a total of $\mathcal{O}(n^2)$ keys in the network.

Symmetric-key encryption also cannot be used to achieve non-repudiation (i.e., a scheme where nobody can deny they were the source of a message). This is because secrets are shared.

Definition 6.1.1 (public-key cryptography)
Schemes in which communicating parties share some authenticated (but non-secret) information.

Public-key schemes were invented by Merkle, Diffie, and Hellman in 1975.

Cryptoscheme 6.1 (Merkle puzzle)

Suppose Alice and Bob desire to establish a secret session key over an authenticated, non-secret channel.

1. Alice creates $N = 10^9$ puzzles P_i such that each takes $t = 5$ hours to solve. The solution to each P_i is a 128-bit session key sk_i and 128-bit serial number n_i.
2. Alice sends the puzzles to Bob
3. Bob picks a random $j \in_{\mathsf{R}} [1, N]$ and solves P_j to obtain sk_j and n_j.
4. Bob sends n_j to Alice.
5. The secret key is sk_j.

In this case, an eavesdropper would need to solve 500 million puzzles to find out sk_j.

For example, let $P_i = \text{AES-CBC}_{k_i}(sk_i, n_i, n_i)$ for $k_i = r_i \parallel 0^{88}$ and $r_i \in_{\mathsf{R}} \{0,1\}^{40}$. Then, the expected time to solve P_i by exhaustive key search is 2^{39} operations.

In general, for public key cryptography, each entity A generates a key pair of a public key and secret key (P_A, S_A). It should be infeasible to recover S_A from P_A. For example, $S_A = (p, q)$ and $P_A = pq$ where p and q are random, large primes. Then, the encryption/signing function takes S_A but the decryption/verifying function only requires P_A.

Public-key cryptography is good for key management, verification, and non-repudiation. However, it is usually slower than symmetric-key encryption.

To work around this, hybrid schemes exist, where public-key encryption is used to share a key used for symmetric-key encryption.

6.2 Algorithmic number theory

Lecture 18
Feb 16

Theorem 6.2.1

Every integer $n \geq 2$ has a unique prime factorization (up to permutation of the factors).

Given an integer, it is hard to find its prime factorization. However, it is easy to verify that a given list of primes is the factorization.

Definition 6.2.2 (big-$\mathcal{O}$ notation)

If $f, g : \mathbb{N} \rightrightarrows \mathbb{R}_+$, then $f(n) = \mathcal{O}(g(n))$ means that there exists a positive constant c and integer n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

Definition 6.2.3

An algorithm is a well-defined computational procedure that takes some input and halts with some output.

The running time of an algorithm is an upper bound, as a function of the input size, of the worst-case number of fixed-size operations.

An algorithm is polynomial-time if its expected running time is $\mathcal{O}(k^c)$ for input size k and a fixed positive integer c.

Then, we can figure out the expected time in bit operations for basic integer operations. Given two k-bit integers a and b (i.e., input size $\mathcal{O}(k)$ bits):

Operation	Algorithm	Running time	
Addition	Elementary school	$\mathcal{O}(k)$	
Subtraction	Elementary school	$\mathcal{O}(k)$	
Multiplication	Elementary school	$\mathcal{O}(k^2)$	Fastest known algorithm is $\mathcal{O}(k \log k)$
Division	Elementary school	$\mathcal{O}(k^2)$	
GCD	Euclidean algorithm	$\mathcal{O}(k^2)$	Fastest known algorithm is $\mathcal{O}\left(k \log^2 k\right)$

Given integers $a, b, m \in \mathbb{Z}_n$ for a k-bit integer n:

Operation	Algorithm	Running time
Addition	Elementary school	$\mathcal{O}(k)$
Subtraction	Elementary school	$\mathcal{O}(k)$
Multiplication	Elementary school	$\mathcal{O}(k^2)$
Inversion	Extended Euclidean algorithm	$\mathcal{O}(k^2)$
Exponentiation	Square-and-multiply	$\mathcal{O}(k^3)$

Lecture 19
Feb 26

The square-and-multiply algorithm reduces the time complexity of exponentiation from $\mathcal{O}(2^k k)$ (calculating a^m and modding) to $\mathcal{O}(k^3)$.

Let $m = \sum m_i 2^i$. Then,

$$a^m = a^{\sum m_i 2^i} = \prod a^{m_i 2^i} \equiv \prod_{0 \le i \le k-1, m_i = 1} a^{2^i} \pmod{n}$$

which gives us the algorithm

Algorithm 1 Repeated square-and-multiply

```
1: procedure SquareAndMultiply(a, m = (m_k ··· m_0))
2:     if m_0 = 1 then
3:         B ← a
4:     else
5:         B ← 1
6:     A ← a
7:     for i = 1, ..., k − 1 do
8:         A ← A^2 mod n
9:         if m_i = 1 then
10:            B ← B × A mod n
11:    return B
```

which takes only at most k squarings and k multiplications, for a total time of $\mathcal{O}(k^3)$.

Chapter 7

RSA

RSA (Rivest–Shamir–Adleman) was invented in 1977.

7.1 RSA encryption

Cryptoscheme 7.1 (RSA encryption)

Each entity A generates a key pair:

1: Randomly select two large, distinct primes p and q of the same bitlength
2: Compute the RSA modulus $n = pq$ and $\phi = \phi(n) = (p-1)(q-1)$
3: Select an arbitrary encryption exponent $1 < e < \phi$ with $\gcd(e, \phi) = 1$
4: Compute the decryption exponent $1 < d < \phi$ with $ed \equiv 1 \pmod{\phi}$
5: **return** $(P_A, S_A) = ((n, e), d)$

To encrypt a message for A:

1: Obtain an authenticated copy of $P_A = (n, e)$
2: Compute the ciphertext $c = m^e \bmod n$
3: **return** c

To decrypt a message as A:

1: Compute $m = c^d \pmod{n}$
2: **return** m

For example, suppose Alice selects $p = 23$ and $q = 37$. Then, $n = 851$ and $\phi(n) = 792$. She can pick $e = 631$ since it is coprime to $\phi = 792$. Then, solving $631d \equiv 1 \bmod 792$ yields $d = 487$.

To encrypt $m = 13$, for Alice, Bob obtains the public key $(n = 851, e = 631)$. Then, he computes $c = 13^{631} \bmod 851$ using square-and-multiply to get 616, and sends 616 to Alice.

To decrypt $c = 616$, Alice computes $m = 616^{487} \bmod 851$ which recovers 13.

Theorem 7.1.1 (RSA works)
For all n and $m \in [0, n-1]$, if $c = m^e \bmod n$, then $m = c^d \bmod n$.

Proof. We will prove that $m^{ed} \equiv m \pmod{n}$ for all $m \in [0, n-1]$.

Since $ed \equiv 1 \pmod{\phi}$, we can write

$$ed = 1 + k(p-1)(q-1)$$

for some $k \in \mathbb{Z}$. Since $ed > 1$ and $(p-1)(q-1) \geq 1$, we have $k \geq 1$. Claim that $m^{ed} \equiv m \pmod{p}$.

Suppose $p \mid m$. Then, $m \equiv 0 \pmod{p}$, so $m^{ed} \equiv 0^{ed} \equiv 0 \equiv m \pmod{p}$.

Otherwise, $p \nmid m$. Then, $\gcd(m, p) = 1$, so by Fermat's Little Theorem:

$$\begin{aligned} m^{p-1} &\equiv 1 && \pmod{p} \\ m \cdot m^{k(p-1)(q-1)} &\equiv m \cdot 1^{k(q-1)} && \pmod{p} \\ m^{1+k(p-1)(q-1)} &\equiv m && \pmod{p} \\ m^{ed} &\equiv m && \pmod{p} \end{aligned}$$

Symmetrically, $m^{ed} \equiv m \pmod{q}$. Since p and q are distinct primes, we have $m^{ed} \equiv m \pmod{n}$, completing the proof. □

Lecture 20
Feb 28

RSA is only secure if Eve cannot factor n. This is the only way Eve can break RSA; the RSA problem of computing d from (n, e) reduces to factoring n.

Attack 7.2 (dictionary attack)
Suppose that $m \in \mathcal{M}$ where $|\mathcal{M}|$ is small.

Then, an adversary can just encrypt all of $\mathcal{M}$ and find the one that matches.

To prevent this, add a bunch of random garage (salt) $s \in_R \{0,1\}^{128}$ to messages. That is, instead of encrypting m, encrypt $m' = s \parallel m$. Since the salt is a fixed length, Alice knows to discard it.

Attack 7.3 (chosen-ciphertext attack on Basic RSA)
Suppose Eve has access to a decryption oracle (i.e., Alice will decrypt any message except c). Then, Eve can decrypt c by:

1: Select $x \in [2, n-1]$ coprime to n
2: Compute $\hat{c} = cx^e \bmod n$. We will have $\hat{c} \neq c$ as long as $\gcd(c, n) = 1$
3: Obtain $\hat{m} = \hat{c}^d = (cx^e)^d = c^d x^{ed} = mx$ from the oracle.
4: Recover $m = \hat{m}x^{-1} \bmod n$

To prevent this, ensure that m follows prescribed format before encryption. Then, $\hat{m}$ is unlikely to follow the right format, so it cannot be decrypted.

Therefore, a secure implementation of RSA must include salting and formatting.

Lecture 21
Mar 1

Definition 7.1.2 (security of a public-key encryption scheme)

A public-key encryption scheme is secure if it is semantically secure against a chosen-ciphertext attack by a computationally bounded adversary.

That is, to break a public-key scheme, the adversary must be able to divine some information about the plaintext given feasible computation and access to a decryption oracle.

Cryptoscheme 7.4 (RSA Optimal Asymmetric Encryption Padding (RSA-OAEP))

Let k be the bitlength of n and $\ell = k - 256 - 1$. Pick hash functions $G_1 : \{0,1\}^{256} \to \{0,1\}^{\ell}$ and $G_2 : \{0,1\}^{\ell} \to \{0,1\}^{256}$.

To encrypt $M \in \{0,1\}^{\ell-256}$, pick a salt $r \in_R \{0,1\}^{256}$. Then, use RSA to encrypt $m = 0 \parallel s \parallel t$ where $s = (0^{256} \parallel M) \oplus G_1(r) \in \{0,1\}^{\ell}$ and $t = r \oplus G_2(s) \in \{0,1\}^{256}$.

To decrypt c, first use RSA to get m and parse as $0 \parallel s \parallel t$. Compute $r = G_2(s) \oplus t$ and check the first 256 bits of $G_1(r) \oplus s$. If they are all zeroes, output $M =$ the rest, otherwise reject.

Theorem 7.1.3 (Bellare & Rogaway)

Suppose that the RSA problem is intractable and that G_1, G_2 are random functions. Then, RSA-OAEP is secure.

7.2 Integer factorization

Recall from CS 136/246/240/341:

Definition 7.2.1

We write $f(n) = \mathcal{O}(g(n))$ if there exists $c > 0$ and n_0 such that $n \geq n_0$ implies $f(n) \leq cg(n)$.

We write $f(n) = o(g(n))$ if $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

An algorithm is polynomial-time if its worst-case running time is $\mathcal{O}(n^c)$ for input size n and constant c.

An algorithm is exponential-time if its worst-case running time is *not* of the form $\mathcal{O}(n^c)$ for *any* constant c. In particular, if it is of the form $\mathcal{O}(2^{cn})$ for a constant c, it is fully exponential-time. If it is of the form $\mathcal{O}\left(2^{o(n)}\right)$, it is subexponential-time.

Consider some attacks on RSA via factoring.

Attack 7.5 (trial division)

Trial divide n by all the primes $2, 3, 5, \ldots, \lfloor\sqrt{n}\rfloor$. If any divide n, then stop.

The running time of this attack is $\mathcal{O}\left(\sqrt{n}\right)$, which is not great ($2^{64}$ divisions for 128-bit keys). In fact, since the size is $\log n$, this is fully exponential-time ($\mathcal{O}\left(2^{(\log n)/2}\right)$).

For classifying subexponential times, we use $L_n[\alpha, c]$ notation.

Definition 7.2.2

If the running time is of the form

$$L_n[\alpha, c] := \mathcal{O}(\exp[(c + o(1))(\log_e n)^\alpha (\log_e \log_e n)^{1-\alpha}])$$

where $c > 0$ and $0 < \alpha < 1$, then the running time is subexponential.

Notice that if $\alpha = 0$, we have $L_n[0, c] = \mathcal{O}\left((\log n)^{c+o(1)}\right)$, which is polynomial-time. On the other edge, if $\alpha = 1$, then $L_n[1, c] = \mathcal{O}\left(n^{c+o(1)}\right)$, which is fully exponential-time.

There are a number of special-purpose factoring algorithms, like Pollard's $p-1$, Pollard's ρ, elliptic curve, etc. that work when n has special, known properties. For example, n has a small prime factor p or $p-1$ only has small prime factors.

If p and q are of the same bitlength and randomly generated, these won't get very far.

The algorithms that really matter are the general-purpose factoring algorithms which care only about the size of the number being factored. The two major advancements are

1. the quadratic sieve factoring algorithm (QS; 1982), with running time $L_n[1/2, 1]$, and
2. the number field sieve factoring algorithm (NFS; 1990), with running time $L_n[1/3, 1.923]$.

Lecture 22
Mar 4

The largest RSA challenge factored was the 2020 factorization of RSA-250 (250 decimal digits, 829 bits).

The security levels of RSA are approximately equivalent to

Security	Block cipher	Hash function	RSA
80	SKIPJACK	SHA-1	1024
112	Triple-DES	SHA-224	2048
128	AES small	SHA-256	3072
192	AES medium	SHA-384	7680
256	AES large	SHA-512	15360

under the current best known algorithms.

However, we have no actual proof that factoring is hard. Shor's algorithm can factor in $\mathcal{O}((\log n)^2)$, but nobody has built a large enough quantum computer yet.

Today, 512-bit RSA is considered insecure, 1024-bit risky, and 2048-/3072-bit secure.

7.3 Real-life example: QQ browser

7.4 RSA signatures

The building blocks of RSA encryption can be used to create a signing scheme.

Cryptoscheme 7.6 (RSA signing)

Pick a hash function H. Generate keys as in RSA encryption.

To sign a message $m \in \{0,1\}^*$ as A,

1: Compute $M = H(m)$
2: Compute the signature $s = M^d \bmod n$
3: **return** (m, s)

To verify that message (m, s),

1: Obtain an authenticated copy of $P_A = (n, e)$
2: Compute $M = H(m)$
3: Compute $M' = s^e \bmod n$
4: **return** $M = M'$

This only works if the RSA problem is intractable. If it is not, Eve can forge Alice's signature by solving $s^e = H(m) \bmod n$ for any m.

Recall that an authentication scheme is secure if it is existentially unforgeable by a computationally bounded adversary under a chosen-mesage attack. That is, Eve gets a signing oracle.

The hash function must also be PR, 2PR, and CR.

If H is not PR, then Eve can pick a random s and get $M = s^e \bmod n$. Then, use the PR oracle to get m such that $H(m) = M$.

If H is not 2PR, then given a signed message (m, s), the oracle can find another $m' \neq m$ such that s is also the signature for m'.

If H is not CR, then given a collision (m, m'), Eve can use the signing oracle to get $s = H(m)^d \bmod n$. But s is also the signature for m'.

The basic RSA signature scheme is not secure if H is SHA-256. However, if we use a full-domain hash function, i.e., $H : \{0,1\}^* \to [0, n-1]$, then we get a secure signature scheme. Usually, to implement RSA-FDH, we use

$$H(m) = \text{SHA-256}(1, m) \parallel \text{SHA-256}(2, m) \parallel \cdots \parallel \text{SHA-256}(t-1, m) \parallel \text{truncate}(\text{SHA-256}(t, m))$$

where we repeat until it has the same bitlength as n.

Cryptoscheme 7.7 (PKCS #1 v1.5 RSA)

To sign $m \in \{0,1\}^*$,

1: Compute $H(m)$, where H is a hash function from a fixed list
2: Format $M = \texttt{0001FF}\cdots\texttt{FF00} \parallel \text{hash name} \parallel h$ to have the same byte length as n.
3: Compute $s = M^d \bmod n$
4: **return** (m, s)

To verify (m, s),

1: Obtain an authentic copy of (n, e)
2: Compute $M = s^e \bmod n$
3: Check the formatting: starts with `0001`, consecutive `FF`, then `00`
4: Read the next 15 bytes to determine the hash function H
5: **return** $h = H(m)$

7.5 Bleichenbacher's attack

Lecture 23
Mar 6

Attack 7.8 (Bleichenbacher)

Suppose that $e = 3$ and the verifier does not check that there are no bytes to the right of h. WLOG, let n be 384 bytes (3072 bits) and H be SHA-1. Then, the attacker can

1: Select $m \in \{0,1\}^*$
2: Compute $h \leftarrow H(m) \in \{0,1\}^{160}$
3: $D \leftarrow \texttt{00} \parallel \text{hash name} \parallel h \in \{0,1\}^{288}$
4: $N \leftarrow 2^{288} - D$
5: Assert that $3 \mid N$. If $3 \nmid N$, modify m sightly and try again.
6: $s \leftarrow 2^{1019} - 2^{34}\frac{N}{3}$
7: **return** (m, s)

Then, the verifier will accept (m, s).

Proof. The verifier will compute

$$\begin{aligned}
M &= s^e && \pmod n \\
&= \left(2^{1019} - 2^{34}\frac{N}{3}\right)^3 && \pmod n \\
&= 2^{3057} - 2^{2072}N + \underbrace{2^{1087}\frac{N^2}{3} - \left(2^{34}\frac{N}{3}\right)^3}_{\text{garbage}} && \pmod n \\
&= 2^{3057} - 2^{2072}(2^{288} - D) + \text{garbage} && \pmod n \\
&= 2^{3057} - 2^{2360} + 2^{2072}D + \text{garbage} && (\text{no mod since this is} \ll 2^{3072} \approx n) \\
&= 2^{2360}(2^{697} - 1) + 2^{2072}D + \text{garbage} \\
&= \underbrace{\texttt{0001FF}\cdots\texttt{FF}}_{3071-2360} \parallel \underbrace{\texttt{00} \parallel \text{hash name} \parallel h}_{D} \parallel \underbrace{\text{garbage}}_{2071-0}
\end{aligned}$$

and extract $h = H(m)$, ignoring the garbage. □

The attack fails if $s^e \gg n$, so the mod cannot be ignored. In practice, $e = 2^{16} + 1 = 65537$ is used.

Chapter 8

Elliptic curve cryptography

8.1 Elliptic curves

Definition 8.1.1

Let F be a field (i.e., $\mathbb{R}$ or $\mathbb{Z}_p$ for prime $p \geq 5$). An elliptic curve E over F is given by an equation

$$E/F : Y^2 = X^3 + aX + b$$

where a and b are in F such that $4a^3 + 27b^2 \neq 0$.

Definition 8.1.2

Let E/F be an ellpitic curve. The set of F-rational points on E is

$$E(F) = \{(x, y) \in F^2 : y^2 = x^3 + ax + b\} \cup \{\infty\}$$

where ∞ is a special value called the point at infinity.

Lecture 24
Mar 8

Example 8.1.3. The set of $\mathbb{Z}_{11}$-rational points on $E/\mathbb{Z}_{11} : Y^2 = X^3 + X + 6$ is $E(\mathbb{Z}_{11}) = \{\infty, (2,4), (2,7), (3,5), (3,6), (5,2), (5,9), (7,2), (7,9), (8,3), (8,8), (10,2), (10,9)\}$.

The size of the set $\#E(\mathbb{Z}_{11}) = 13$.

Theorem 8.1.4 (Hasse)

Let $E/\mathbb{Z}_p : Y^2 = X^3 + aX + b$. Then, $(\sqrt{p} - 1)^2 \leq \#E(\mathbb{Z}_p) \leq (\sqrt{p} + 1)^2$.

This allows us to approximate $\#E(\mathbb{Z}_p) \approx p$. There is a polynomial-time algorithm to find $\#E(\mathbb{Z}_p)$.

To define addition on $E(F)$, we can imagine connecting a line together.

Definition 8.1.5 (geometric rule)

Let P and Q be points on $E(\mathbb{R})$ and let ℓ be the line through P and Q. In the case where $P = Q$, consider the tangent line.

Let $T \in E(\mathbb{R})$ be the third point of intersection of ℓ with E. If it does not exist, let $T = \infty$, since we treat ∞ as a point "at infinity" on every line.

Then, we define $P + Q$ as the reflection $-T$ in the x-axis.

Formally:

Definition 8.1.6 (addition rule)

Let E be an elliptic curve over F. Let $P = (x_1, y_1)$ and $Q = (x_2, y_2) \in E(F)$. Then:

1. $P + \infty = \infty + P = P$ for all $P \in E(F)$.
2. $-P = (x_1, -y_1)$. Also, $-\infty = \infty$. Furthermore, $P + (-P) = (-P) + P = \infty$ for all P.
3. Suppose $P \neq \pm Q$. Then, $P+Q = (x_3, y_3)$ where $x_3 = \lambda^2 - x_1 - x_2$ and $y_3 = -1 + \lambda(x_1 - x_3)$ for $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$.
4. Suppose $P \neq -P$. Then, $P + P = (x_3, y_3)$ where $x_3 = \lambda^2 - 2x_1$, $y_3 = -y_1 + \lambda(x_1 - x_3)$ for $\lambda = \frac{3x_1^2 + a}{2y_1}$.

Example 8.1.7. In $E/\mathbb{Z}_1 1 : Y^2 = X^3 + X + 6$ (from ex. 8.1.3), $(2, 4) + (2, 7) = \infty$, $(2, 4) + (8, 3) = (5, 2)$, and $(2, 4) + (2, 4) = (5, 9)$.

Proposition 8.1.8

$(E(F), +)$ is an abelian group with identity element ∞. That is,

1. $P + \infty = P$ for all P;
2. for each P, there exists Q such that $P + Q = \infty$;
3. $P + Q = Q + P$ for all P and Q; and
4. $(P + Q) + R = P + (Q + R)$ for all P, Q, and R.

Associativity is non-trivial, but we do not prove it.

We write $P - Q := P + (-Q)$. This is point subtraction.

8.2 Elliptic curve discrete log

Lecture 25
Mar 11

Definition 8.2.1 (point multiplication)

Let $P \in E(\mathbb{Z}_p)$ and $k \in N$. Then,

$$kP := \underbrace{P + P + \cdots + P}_{k}$$

Define also $0P := \infty$ and $(-k)P := -(kP)$.

Theorem 8.2.2

Suppose $n = \#E(\mathbb{Z}_p)$ is prime and $\infty \neq P \in E(\mathbb{Z}_p)$. Then:

1. $nP = \infty$; and
2. the points $\infty, P, 2P, 3P, \ldots, (n-1)P$ are distinct, so $E(\mathbb{Z}_p) = \{\infty, P, 2P, 3P, \ldots, (n-1)P\}$.

That is, P is a generator of $E(\mathbb{Z}_p)$.

Example 8.2.3. For the curve $E/\mathbb{Z}_{23} : Y^2 = X^3 + X + 4$, we have $\#E(\mathbb{Z}_{23}) = 29$ and generator $P = (0, 2)$ since:

$1P$=(0,2)	$2P$=(13,12)	$3P$=(11,9)	$4P$=(1,12)	$5P$=(7,20)	$6P$=(9,11)
$7P$=(15,6)	$8P$=(14,5)	$9P$=(4,7)	$10P$=(22,5)	$11P$=(10,5)	$12P$=(17,9)
$13P$=(8,15)	$14P$=(18,9)	$15P$=(18,14)	$16P$=(8,8)	$17P$=(17,14)	$18P$=(10,18)
$19P$=(22,18)	$20P$=(4,16)	$21P$=(14,18)	$22P$=(15,17)	$23P$=(9,12)	$24P$=(7,3)
$25P$=(1,11)	$26P$=(11,14)	$27P$=(13,11)	$28P$=(0,21)	$29P$=∞	

Problem 8.2.4 (elliptic curve discrete logarithm problem (ECDLP))

Given E, p, n, $\infty \neq P \in E(\mathbb{Z}_p)$, and $Q \in E(\mathbb{Z}_p)$, find the discrete logarithm $\ell \in [0, n-1]$ such that $Q = \ell P$.

We write $\ell =: \log_P Q$.

Example 8.2.5. In ex. 8.2.3, we can see that $\log_{(0,2)}((10, 18)) = 18$.

A brute force approach will take $\mathcal{O}(n) = \mathcal{O}(p)$ point additions, which is fully exponential-time as the input size is $\log p$.

Shank's algorithm uses the division algorithm. Let $m = \lceil \sqrt{n} \rceil$. By the division algorithm there exist unique q and r such that $\ell = qm + r$ where $0 \leq r < m$ and $0 \leq q < m$. Then, $\ell = qm - r$ and in particular $\ell P - qmP = rP$.

If $M = mP$, we can say $Q - qM = rP$. This leads us to the algorithm:

Algorithm 2 Shank's algorithm for ECDLP

```
for r ∈ [0, m − 1] do
    Compute rP and store (rP, r) in a sorted table
M ← mP
for q ∈ [0, m − 1] do
    R ← Q − qM
    if (R, r) in the table then
        return ℓ = qm + r
```

This will require $\mathcal{O}(m) = \mathcal{O}(\sqrt{p})$ point additions and storage of $\mathcal{O}(\sqrt{p})$ points.

Pollard's algorithm (no details given) also works in $\mathcal{O}(\sqrt{p})$ time but with negligible space complexity. VW parallel collision search can also parallelize Pollard's algorithm.

Some other results: If $n = p$, there exists a polynomial-time algorithm. Also, if $n \mid (p^c - 1)$ for small $c \lessapprox 100$, there is a subexponential-time algorithm. Shor's algorithm solves the ECDLP in $\mathcal{O}((\log p)^2)$ time.

8.3 Elliptic curve cryptography

ECC is better than RSA because the fastest attacks are fully exponential instead of subexponential. This means smaller parameters are needed for the same security level:

Security level	RSA bitlength	ECC bitlength
80	1,024	160
112	2,048	224
128	3,072	256
192	7,680	384
256	15,360	512

The NSA has specific curves – P-256, P-384, and P-521 – that they use for achieving 128-, 192-, and 256-bit security levels. They all use curves of the form $E/\mathbb{Z}_p : Y^2 = X^3 - 3X + b$.

The b-values for these curves were supposedly chosen at random, but there is gossip that they were picked because of attacks only known to the NSA.

Lecture 26
Mar 13

Curve25519 was selected by Bernstein and Lange in 2005, and provides a 128-bit security level. It is the curve $E/\mathbb{Z}_{2^{255}-19} : Y^2 = X^3 + 48662X^2 + X$.

SM2 is a 256-bit elliptic curve chosen by the Chinese State Cryptography Administration, which is similar to P-256 but with a slightly different prime and b.

The NSA's Commercial National Security Algorithm Suite (CNSA) uses AES-256 (encryption), SHA-384 (hashing), ECDSA with P-384/RSA-PSS with $\geq$ 3072-bit n (signature), and ECDH with P-384/RSA with $\geq$ 3072-bit n (key establishment).

The primes $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ and $2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$ in P-256/384 were chosen because reduction modulo p can be implemented on a 32-bit machine without division.

Consider $p = 2^{192} - 2^{64} - 1$. Suppose that $a, b \in [0, p-1]$ and we want to compute $c = a \times b \bmod p$. We can write $a = a_2 2^{128} + a_1 2^{64} + a_0$ as 64-bit integers and similarly for b. Then,

$$d = a \times b = d_5 2^{230} + d_4 2^{256} + d_3 2^{192} + d_2 2^{128} + d_1 2^{64} + d_0$$

is a 384-bit integer of six 64-bit words. We have that

$$\begin{aligned} 2^{192} &\equiv 2^{64} + 1 && \pmod p \\ 2^{256} &\equiv 2^{128} + 1 && \pmod p \\ 2^{320} &\equiv 2^{192} + 2^{128} \equiv 2^{128} + 2^{64} + 1 && \pmod p \end{aligned}$$

so we can write that

$$\begin{aligned} c &\equiv d && \pmod p \\ &\equiv d_5 2^{230} + d_4 2^{256} + d_3 2^{192} + d_2 2^{128} + d_1 2^{64} + d_0 && \pmod p \\ &\equiv d_5(2^{128} + 2^{64} + 1) + d_4(2^{128} + 2^{64}) + d_3(2^{64} + 1) + d_2 2^{128} + d_1 2^{64} + d_0 && \pmod p \\ &\equiv (d_5 \parallel d_5 \parallel d_5) + (d_4 \parallel d_4 \parallel 0) + (0 \parallel d_3 \parallel d_3) + (d_2 \parallel d_1 \parallel d_0) && \pmod p \end{aligned}$$

which leads us to a nice and clean algorithm:

Algorithm 3 Reduction modulo $p = 2^{192} - 2^{64} - 1$

1: $(d_5, d_4, d_3, d_2, d_1, d_0) \leftarrow a \times b$
2: $t_1 \leftarrow (d_2, d_1, d_0)$
3: $t_2 \leftarrow (0, d_3, d_3)$
4: $t_3 \leftarrow (d_4, d_4, 0)$
5: $t_4 \leftarrow (d_5, d_5, d_5)$
6: $c \leftarrow t_1 + t_2 + t_3 + t_4$
7: **repeat**
8: $\quad c \leftarrow c - p$
9: **until** $c < p$
10: **return** c

Notice that we never perform a full long division. We also have $0 \leq c < 4p$, so at most three subtractions are required on line 8.

We now have enough defined to actually do some cryptography.

Suppose two parties want to agree on a shared secret k so they can use AES-GCM or something, but over an unsecured connection. Elliptic Curve Diffie–Hellman builds on the Diffie–Hellman key agreement protocol.

Cryptoscheme 8.1 (Elliptic Curve Diffie–Hellman (ECDH))

Let E be an elliptic curve defined over $\mathbb{Z}_p$ with $n = \#E(\mathbb{Z}_p)$ prime. Let $P \in E(\mathbb{Z}_p)$ be an arbitrary base point $P \neq \infty$. Also select a key derivation function $KDF : E(\mathbb{Z}_p) \to \{0,1\}^*$. Then:

1: Alice selects $x \in_R [1, n-1]$ and computes $X \leftarrow xP$
2: Alice sends X to Bob
3: Bob selects $y \in_R [1, n-1]$ and computes $Y \leftarrow yP$
4: Bob sends Y to Alice
5: Bob computes $K \leftarrow yX$ and $k = KDF(K)$
6: Alice computes $K \leftarrow xY$ and $k = KDF(K)$

Then, Alice and Bob both have the same k since $xY = x(yP) = (xy)P = (yx)P = y(xP) = yX$. The fastest way an attacker can determine $K = xY = yX$ from only X and Y is to solve an instance of the ECDLP.

However, since ECDH does not ask for an authenticated channel, it is vulnerable to a man-in-the-middle (MITM) attack. An attacker Eve can sit between Alice and Bob and just pass along her own secrets to establish a key with Alice and a key with Bob. Then, Eve can decrypt/re-encrypt messages and pass them along without Alice or Bob knowing.

Lecture 27
Mar 15

To solve this, instead of sending just X, Alice sends X, an ECDSA signature on X, and a certificate for her public key. Then, Bob verifies the certificate and signature to know that X is legitimate.

Cryptoscheme 8.2 (Elliptic Curve Digital Signature Algorithm (ECDSA))

Let E be an elliptic curve defined over $\mathbb{Z}_p$ with $n = \#E(\mathbb{Z}_p)$ prime. Let $P \in E(\mathbb{Z}_p)$ be a generator point $P \neq \infty$. Also select a collision-resistant hash function $H : \{0,1\}^* \to \{0,1\}^n$.

Alice selects $a \in_R [1, n-1]$ and computes $A \leftarrow aP$. Her public key is A; her private key is a. To sign a message $M \in \{0,1\}^*$:

1: $m \leftarrow H(M)$
2: Select a per-mesage secret $k \in_R [1, n-1]$
3: $R \leftarrow kP$, $r \leftarrow x(R) \bmod n$. Assert $r \neq 0$.
4: $s \leftarrow k^{-1}(m + ar) \bmod n$. Assert $s \neq 0$.
5: **return** $(M, (r, s))$

Note that k should be random and only used once. To validate a signature (r, s) on M:

1: Obtain a copy of A
2: Assert that $r, s \in [1, n-1]$
3: $m \leftarrow H(M)$
4: $u_1 \leftarrow ms^{-1} \bmod n$, $u_2 \leftarrow rs^{-1} \bmod n$
5: $V \leftarrow u_1 P + u_2 A$. Assert $V \neq \infty$.
6: $v = x(V) \bmod n$
7: **return** $v = r$

ECDSA is secure assuming the ECDLP is intractable and H is secure.

We can compare 256-bit ECDSA with 3072-bit RSA:

- Usually we take $e = 3$ or $e = 2^{16} + 1$ (small), so RSA verification $s^e \bmod n$ is faster than ECDSA verification $u_1 P + u_2 A$
- RSA generation $H(M)^d \bmod n$ is slower than ECDSA generation kP
- RSA signatures are much longer (3072 bits) than ECDSA signatures (512 bits)

List of Named Results

List of Cryptoschemes and Attacks

Index of Defined Terms