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):

```
1:  Create a table T of pairs sorted by first entry
2:  for h_2 ∈ {0,1}^56 do                          ▷ h_2 is a guess for k_2
3:      T.insert(E_{h_2}^{-1}(m_1), h_2)
4:  for h_1 ∈ {0,1}^56 do                          ▷ h_1 is a guess for k_1
5:      Compute E_{h_1}(m_1)
6:      Search for entries in T matching (E_{h_1}(m_1), −)
7:      for each match (−, h_2) do
8:          if E_{h_2}(E_{h_1}(m_2)) = c_2 then
9:              if E_{h_2}(E_{h_1}(m_3)) = c_3 then
10:                 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 $i = 1, \ldots, t$ 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 $d(y) := a(y) \cdot b(y)$. 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:

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

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:

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

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 ciphers

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 $y \in_R \{0,1\}^n$, 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 (2^{nd} preimage resistance)

A hash function $H = \{0,1\}^* \to \{0,1\}^n$ is 2^{nd} preimage resistant (2PR) if, given $x \in_R \{0,1\}^*$, 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 Theorem 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.

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.

List of Named Results

List of Cryptoschemes and Attacks

Index of Defined Terms