Stream cipher syntax: Algorithm G that takes one input and produces an very long bit-string as output.

Key/Seed k: 1100..11

Typically 16 or 32 bytes.

G(k):
111110100010001110101000101000101100100111100...

⊕ 00100010011111010111011011100001010100111000...

Use G(seed) in place of pad.
Still malleable and still one-time, but key is shorter.
Addressing pad reuse: Stream cipher with a nonce

Stream cipher with a nonce: Algorithm G that takes two inputs and produces a very long bit-string as output.

Nonce IV: 1100..11  
Key/Seed k: 1100..11

G(IV,k): 1111101000100011101001000101100100111100...

- “nonce” = “number once”.
- Usually denoted IV = “initialization vector”

Security goal: When k is random and unknown, G(IV,k) should “look” random and independent for each value of IV.
Solution 1: Stream cipher with a nonce

- If nonce repeats, then pad repeats
Example of Pad Re-use: WEP

IEEE 802.11b WEP: WiFi security standard ’97-‘03

IV

IV is 24-bit wide counter

- Repeats after $2^{24}$ frames (≈16 million)
- IV is often set to zero on power cycle

Solutions: (WPA2 replacement)
- Larger IV space, or force rekeying more often
- Set IV to combination of packet number, address, etc
Example of Pad Re-use: WEP

**Warning: Broken**

IEEE 802.11b WEP: WiFi security standard ’97-‘03

### IV
- IV is 24-bit wide counter
- Repeats after $2^{24}$ frames (≈ 16 million)
- Often set to zero on reset

### Solutions: (WPA2 replacement)
- Larger IV space
- Set IV to combination of packet number, address, etc.

---

**BIZ & IT —**

**Serious flaw in WPA2 protocol lets attackers intercept passwords and much more**

KRACK attack is especially bad news for Android and Linux users.

**DAN GOODIN - 10/15/2017, 11:37 PM**

parameters to their initial values. KRACK forces the nonce reuse in a way that allows the encryption to be bypassed. Ars Technica IT editor Sean Gallagher has [much more about KRACK here](https://arstechnica.com/security/2017/04/krack-wpa-wpa2-patented-wifi-security-flaw/).
Example Stream Cipher w/ Nonces: ChaCha20

- Key-length: 256 bits
- Generates stream by applying a fixed permutation to seed and counter
- Uses “feed-forward” to break up permutation structure
ChaCha20 Block Permutation

```c
#define ROTL(a,b) (((a) << (b)) | ((a) >> (32 - (b))))
#define QR(a, b, c, d) (a += b,  d ^= a,  d = ROTL(d,16), 
                      c += d,  b ^= c,  b = ROTL(b,12), 
                      a += b,  d ^= a,  d = ROTL(d, 8), 
                      c += d,  b ^= c,  b = ROTL(b, 7))
#define ROUNDS 20

void chacha_block(uint32_t out[16], uint32_t const in[16]) {
    int i;
    uint32_t x[16];

    for (i = 0; i < 16; ++i)
        x[i] = in[i];

    // 10 loops × 2 rounds/loop = 20 rounds
    for (i = 0; i < ROUNDS; i += 2) {
        // Odd round
        QR(x[0], x[4], x[ 8], x[12]); // column 0
        QR(x[1], x[5], x[ 9], x[13]); // column 1
        QR(x[2], x[6], x[10], x[14]); // column 2
        QR(x[3], x[7], x[11], x[15]); // column 3

        // Even round
        QR(x[0], x[5], x[10], x[15]); // diagonal 1 (main diagonal)
        QR(x[1], x[6], x[11], x[12]); // diagonal 2
        QR(x[2], x[7], x[ 8], x[13]); // diagonal 3
        QR(x[3], x[4], x[ 9], x[14]); // diagonal 4
    }

    for (i = 0; i < 16; ++i)
        out[i] = x[i] + in[i];
}
```

In Assignment 2: Develop attack when a weak "statistical" stream cipher is used.
Issues with One-Time Pad

1. Reusing a pad is insecure  ✔️  Use unique nonces
2. One-Time Pad is *malleable*
3. One-Time Pad has a long key  ✔️  Use stream cipher with short key

More difficult to address; We will return to this later.
Next Up: Blockciphers

Blockciphers are a ubiquitous crypto tool applied to many different problems.

**Informal definition:** A blockcipher is essentially a substitution cipher with a very large alphabet and a very compact key. Require that efficient algorithms for forward and backward directions.

**Typical parameters:**
Alphabet = \{0,1\}^{128}
Key length = 16 bytes.

Plan: Build many higher-level protocols from a good blockcipher.

Now: Two example blockciphers, DES and AES.
Data Encryption Standard (DES)

- Originally designed by IBM
- Parameters adjusted by NSA
- NIST Standard in 1976
  - Block length $n = 64$
  - Key length $k = 56$

Parses input block into 32-bit chunks and applies 16 rounds of a “Feistel Network”
**DES is Broken**

<table>
<thead>
<tr>
<th>Attack</th>
<th>Complexity</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biham&amp;Shamir</td>
<td>$2^{47}$ encrypted blocks</td>
<td>1992</td>
</tr>
<tr>
<td>DESCHALL</td>
<td>41 days</td>
<td>1997</td>
</tr>
<tr>
<td>EFF Deepcrack</td>
<td>4.5 days</td>
<td>1998</td>
</tr>
<tr>
<td>EFF Deepcrack</td>
<td>22 hours</td>
<td>1999</td>
</tr>
</tbody>
</table>

- 3DES ("Triple DES") is still used by banks
- 3DES encrypts three times (so key length is 118)
- 3DES is not known to be broken but should be avoided
GET CRACKING

These are the types of DES cracking jobs that we support:

- Windows LM/NTLMv1 Authentication
- PRTP VPNs
- WPA-Enterprise
- des_crypt() Hashes
- DES Kerberos5
- Known Plaintext DES

NOTE: There are currently extremely high wait times. We're in the process of adding capacity to speed things up.

QUEUE WAIT TIME:
Standard 46.2 Days, ASAP 1.0 Days

---

WARNING: Charges will show up on your credit card statement as from “crack.sh” and processed through Stripe. We've experienced a high number of our charges being reported as fraudulent, so we'll be blacklisting any accounts that contest charges for jobs submitted. If you wish to cancel a job or have any issues, please email david@toorcon.org and we'll be happy to cancel and refund any charges.
- NIST ran competition to replace DES starting in 1997
- Several submissions, *Rijndael* chosen and standardized
- AES is now the gold standard blockcipher
- Very fast; Intel chips even have AES instructions
Advanced Encryption Standard (AES)

- Due to Rijmen and Daemen
  - Block length $n = 128$
  - Key length $k = 128, 192, 256$

- Different structure from DES.
- 10 rounds of “substitution-permutation network”
AES is not (know to be) broken

<table>
<thead>
<tr>
<th>Attack</th>
<th>Complexity</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bogdanov et al.</td>
<td>$\approx 2^{126.1}$</td>
<td>2011</td>
</tr>
</tbody>
</table>

- Compare to trying all keys: $2^{126.1} \approx 2^{128}/4$
- Always prefer AES for a blockcipher if setting can support it (i.e. everything except low-power hardware)
Blockcipher Security

- AES is thought to be a good “Pseudorandom Permutation”

- Outputs all look random and independent, even when inputs are maliciously controlled.
- Formal definition in CS284.
Example - AES Input/Outputs

- Keys and inputs are 16 bytes = 128 bits

- $K_1: \text{9500924ad9d1b7a28391887d95fcfbd5}$
- $K_2: \text{9500924ad9d1b7a28391887d95fcfbd6}$

$AES_{K_1}(00..00) = \text{8b805ddb39f3eef3e72b43bf95c9ce410f}$
$AES_{K_1}(00..01) = \text{9918e60f2a20b1b81674646dceebdb51}$
$AES_{K_2}(00..00) = \text{1303270be48ce8b8dd8316fdba38eb04}$
$AES_{K_2}(00..01) = \text{96ba598a55873ec1286af646073e36f6}$
So we have a blockcipher…

- Now what?

It only processes 16 bytes at a time, and I have a whole lot more data than that.

This next step is where everything flies off the rails in implementations…
Encrypting large files: ECB

- ECB = “Electronic Code Book”

\[\text{AES-ECB}_k(M)\]
- Parse \(M\) into blocks \(M_1, M_2, \ldots, M_t\)
  // all blocks except \(M_t\) are 16 bytes
- Pad \(M_t\) up to 16 bytes
- For \(i=1\ldots t:\)
  - \(C_i \leftarrow \text{AES}_k(M_i)\)
- Return \(C_1, \ldots, C_t\)
The ECB Penguin

- 16 byte chunks are consecutive pixels

Plaintext

ECB Ciphertext

- It gets even worse…
Encrypting large files, Attempt #2: CTR

- CTR = “Counter Mode”
- Idea: Build a nonce-based stream cipher from AES

\[
\text{AES-CTR}_k(\text{IV}, M) = \begin{cases} 
   M_1 \oplus \text{AES}_k(\text{IV}+1) \\
   M_2 \oplus \text{AES}_k(\text{IV}+2) \\
   \vdots \\
   M_t \oplus \text{AES}_k(\text{IV}+t) 
\end{cases}
\]

- Parse \( M \) into blocks \( M_1, M_2, \ldots, M_t \)
  // all blocks except \( M_t \) are 16 bytes
- For \( i=1\ldots t \):
  - \( C_i \leftarrow M_i \oplus \text{AES}_k(\text{IV}+i) \)
- Return \( \text{IV}, C_1, \ldots, C_t \)

Notes:
- No need to pad last block
- Must avoid reusing part of stream

When combined with authentication, CTR is a good cipher.
Penguin Sanity Check

Plaintext  ECB Ciphertext  CTR Ciphertext

Looks random
Encrypting large files, Attempt #3: CBC

- CBC = “Cipher Block Chaining”
- Nonce-based, but not a stream cipher
- Historical option (sometimes used without nonce)

AES-CBC<sup>k</sup>(IV, M)
- Parse M into blocks M<sub>1</sub>, M<sub>2</sub>, ..., M<sub>t</sub>
  // all blocks except M<sub>t</sub> are 16 bytes
- Pad M<sub>t</sub> up to 16 bytes
- C<sub>0</sub>←IV
- For i=1...t:
  - C<sub>i</sub> ← AES<sub>k</sub>(M<sub>i</sub>⊕C<sub>i-1</sub>)
- Return C<sub>0</sub>, C<sub>1</sub>, ..., C<sub>t</sub>

Decryption

\[ \text{IV} \rightarrow C_1 \rightarrow \text{AES}^{-1}_k() \rightarrow M_1 \]
\[ \text{IV} \rightarrow C_2 \rightarrow \text{AES}^{-1}_k() \rightarrow M_2 \]
Encrypting large files, Attempt #3: CBC

- CBC = “Cipher Block Chaining”
- Nonce-based, but not a stream cipher
- Historical option (sometimes used without nonce)

\[
\text{AES-CBC}_k(\text{IV}, M)
\]

- Parse \( M \) into blocks \( M_1, M_2, \ldots, M_t \)
  // all blocks except \( M_t \) are 16 bytes
- Pad \( M_t \) up to 16 bytes
- \( C_0 \leftarrow \text{IV} \)
- For \( i = 1 \ldots t \):
  - \( C_i \leftarrow \text{AES}_k(M_i \oplus C_{i-1}) \)
- Return \( C_0, C_1, \ldots, C_t \)

When combined with authentication, CBC is a good cipher.

Warning: Padding creates havoc with authentication. Very difficult to implement.
Blockcipher Encryption Summary

- AES is unbroken
- AES-CTR is most robust construction for confidentiality
- AES-CTR/AES-CBC do not provide authenticity/integrity and should almost never be used alone.
Next Up: Integrity and Authentication

- Authenticity: Guarantee that adversary cannot change or insert ciphertexts
- Achieved with MAC = “Message Authentication Code”
Integrity: Preventing message modification
Encryption Integrity: An abstract setting

Encryption satisfies **integrity** if it is infeasible for an adversary to send a new $C'$ such that $\text{Dec}_K(C') \neq \text{ERROR}$. 
AES-CTR does not satisfy integrity

M = please pay ben 20 bucks

\[ C = \text{b0595fafd05df4a7d8a04ced2d1ec800d2daed851ff509b3e446a782871c2d} \]

\[ C' = \text{b0595fafd05df4a7d8a04ced2d1ec800d2daed851ff509b3e546a782871c2d} \]

M' = please pay ben 21 bucks

Inherent to stream-cipher approach to encryption.
AES-CBC does not satisfy integrity

AES-CBC Decryption:

\[
\begin{align*}
\text{IV} & \rightarrow \text{AES}^{-1}_K() \\
C_1 & \rightarrow \text{AES}^{-1}_K() \\
\oplus & \\
M_1 & \\
\text{IV} & \rightarrow \text{AES}^{-1}_K() \\
C_2 & \rightarrow \text{AES}^{-1}_K() \\
\oplus & \\
M_2 & \\
\end{align*}
\]

Decrypts to:

\[
\begin{align*}
\text{IV}, C_1, C_2 & \\
\rightarrow & \\
\text{IV}, C_1 \oplus X, C_2 & \\
\rightarrow & \\
R, M_2 \oplus X &
\end{align*}
\]

Where \( R \) is some unpredictable block.
A message authentication code (MAC) is an algorithm that takes as input a key and a message, and outputs an "unpredictable" tag.
MAC Security Goal: Unforgeability

MAC satisfies **unforgeability** if it is unfeasible for Adversary to fool Bob into accepting $M'$ not previously sent by Alice.
MAC Security Goal: Unforgeability

Note: No encryption on this slide.

\[ M = \text{please pay ben 20 bucks} \]
\[ T = 827851dc9cf0f92ddc5c52572ff8bc \]

\[ M', T' \]
\[ M' = \text{please pay ben 21 bucks} \]
\[ T' = \text{baeaf48a891de588ce588f8535ef58b6} \]

Should be hard to predict \( T' \) for any new \( M' \).
MACs In Practice: Pretty much always use HMAC

- Don’t worry about how it works.

- Other options: Poly1305-AES or CBC-MAC (the latter is tricky)
Authenticated Encryption

Encryption that provides **confidentiality** and **integrity** is called **Authenticated Encryption**.

- Built using a good cipher and a MAC.
  - Ex: AES-CTR with HMAC-SHA2
- Best solution: Use ready-made Authenticated Encryption
  - Ex: AES-GCM is the standard
Building Authenticated Encryption

\[ \text{Encrypt}_{K_1,K_2}(M) \]

\[ M \xrightarrow{\text{Enc}_{K_1}()} C \xrightarrow{\text{MAC}_{K_2}()} (C,T) \]

Output: \((C,T)\)

\[ \text{Decrypt}_{K_1,K_2}(C,T) \]

\[ C \xrightarrow{\text{MAC}_{K_2}()} T' \xrightarrow{T'=T?} \]

\[ C \xrightarrow{\text{Dec}_{K_1}()} M' \]

Output: \(M'\) if \(T'=T\)
\(\bot\) if \(T'\neq T\)

- Summary: MAC the ciphertext, not the message
Chosen-Ciphertext Attacks (CCA) against Encryption

- Integrity + Confidentiality = security against CCAs

- Adversary provides ciphertext inputs to system
- Obtains info about decryptions of its ciphertexts

System (e.g. webserver)

\[ K \]

\[ \text{Enc}_K() \]

\[ \text{Dec}_K() \]

\[ M' \leftarrow \text{Dec}_K(C') \]

\[ \text{<info about M'>} \]

\[ C' \]
Next Up: Hash Functions

**Definition:** A hash function is a deterministic function $H$ that reduces arbitrary strings to fixed-length outputs.

- **Output length**
  - MD5: $m = 128$ bits
  - SHA-1: $m = 160$ bits
  - SHA-256: $m = 256$ bits
  - SHA-512: $m = 512$ bits
  - SHA-3: $m \geq 224$ bits

Some security goals:
- collision resistance: can’t find $M \neq M'$ such that $H(M) = H(M')$
- preimage resistance: given $H(M)$, can’t find $M$
- second-preimage resistance: given $H(M)$, can’t find $M'$ s.t. $H(M') = H(M)$

Note: Very different from hashes used in data structures!
Why are collisions bad?

The binary should hash to \text{3477a3498234f}

\text{MD5}(\text{100 001})=3477a3498234f

\text{MD5}(\text{100 001})=3477a3498234f

Hashes to \text{3477a3498234f}, so I accept.
Hash Functions are not MACs

Both map long inputs to short outputs… But a hash function does not take a key.

**Intuition**: a MAC is like a hash function, that only the holders of key can evaluate.
Hash Function Security History

- Can always find a collision in $2^{m/2}$ time ($\ll 2^m$ time). “Birthday Attack”
- MD5 (1992) was broken in 2004 - can now find collisions very quickly.
- SHA-1 (1995) was broken in 2017 - A big computer can find collisions
- SHA-256/SHA-512 (2001) are not broken
- SHA-3 (2015) is new and not broken

\[ \text{MD5(} \text{d131dd02c5e6eeec4693d9a0698aff95c 2fcab58712467eab4004583eb8fb7f89} 55ad340609f4b30283e488832571415a 085125e8f7c99fd91dbdf280373c5b \text{d8823e3156348f5bae6d4c36c919c6 dd53e2b487da03fd023963066248cda0 e99f33420f577ee8ce54b67080a8d1e c69821bcb6a8839396f9652b6ff72a70} \text{)} = \text{MD5(} \text{d131dd02c5e6eeec4693d9a0698aff95c 2fcab50712467eab4004583eb8fb7f89} 55ad340609f4b30283e488832571415a 085125e8f7c99fd91dbdf280373c5b \text{d8823e3156348f5bae6d4c36c919c6 dd53e23487da03fd023963066248cda0 e99f33420f577ee8ce54b67080280d1e c69821bcb6a8839396f9652b6ff72a70} \text{)} \]

Xiaoyun Wang (Tsinghua University), 2004
- Broken with clever techniques
- Compare to DES (broken b/c key too short)

In Assignment 2: Install and use actual attack code to see how MD5 can be abused.
MACs from Hash Functions

Goal: Build a secure MAC out of a good hash function.

- Totally insecure if $H = \text{MD5, SHA1, SHA-256, SHA-512}$
- Is secure with SHA-3

Construction: $\text{MAC}(K, M) = H(K || M)$  
**Warning: Broken**

In Assignment 2: Break this construction!

Construction: $\text{MAC}(K, M) = H(M || K)$  
**Just don’t**

Upshot: Use HMAC; It’s designed to avoid this and other issues.

Later: Hash functions and certificates
The End