Crypto 2

I DON'T ALWAYS ENCRYPT

BUT WHEN I DO, IT'S AES-CFB + HMAC_SHA256
Modern Encryption: Block cipher

- A function $E : \{0, 1\}^b \times \{0, 1\}^k \rightarrow \{0, 1\}^b$. Once we fix the key $K$ (of size $k$ bits), we get:

- $E_K : \{0,1\}^b \rightarrow \{0,1\}^b$ denoted by $E_K(M) = E(M,K)$.

- (and also $D(C,K)$, $E(M,K)$'s inverse)

Three properties:

- Correctness:
  - $E_K(M)$ is a permutation (bijective function) on $b$-bit strings
    - Bijective $\Rightarrow$ invertible

- Efficiency: computable in $\mu$sec’s

- Security:
  - For unknown $K$, “behaves” like a random permutation

- Provides a building block for more extensive encryption
DES (Data Encryption Standard)

- Designed in late 1970s
- Block size 64 bits, key size 56 bits
- NSA influenced two facets of its design
  - Altered some subtle internal workings in a mysterious way
  - Reduced key size 64 bits → 56 bits
    - Made brute-forcing feasible for attacker with massive (for the time) computational resources
- Remains essentially unbroken 40 years later!
  - The NSA’s tweaking hardened it against an attack “invented” a decade later
- However, modern computer speeds make it completely unsafe due to small key size
Today’s Go-To Block Cipher: AES (Advanced Encryption Standard)

- >20 years old, standardized >15 years ago...
- Block size 128 bits
- Key can be 128, 192, or 256 bits
  - 128 remains quite safe; sometimes termed “AES-128”, paranoids use 256b
- As usual, includes encryptor and (closely-related) decryptor
  - How it works is beyond scope of this class…
- Not proven secure
  - But no known flaws
    - The NSA uses it for Top Secret communication with 256b keys: stuff they want to be secure for 40 years including possibly unknown breakthroughs!
  - so we assume it is a secure block cipher
AES is also effectively free...

- Computational load is remarkably low
  - Partially why it won the competition:
    - There were 3 really good finalists from a performance viewpoint:
      - Rijndael (the winner), Twofish, Serpent
      - One OK: RC6
      - One ugggly: Mars
  - On any given computing platform:
    - Rijndael was *never* the fastest
  - But on every computing platform:
    - Rijndael was *always* the second fastest
    - The other two good ones always had a "this is the compute platform they are bad at"
- And now CPUs include dedicated AES support
How Hard Is It To Brute-Force 128-bit Key?

- $2^{128}$ possibilities – well, how many is that?
- Handy approximation: $2^{10} \approx 10^3$
- $2^{128} = 2^{10 \times 12.8} \approx (10^3)^{12.8} \approx (10^3)^{13} \approx 10^{39}$
How Hard Is It To Brute-Force 128-bit Key?

- $2^{128}$ possibilities – well, how many is that?
- Handy approximation: $2^{10} \approx 10^3$
- $2^{128} = 2^{10 \times 12.8} \approx (10^3)^{12.8} \leq (10^3)^{13} \approx 10^{39}$
- Say we build massive hardware that can try $10^9$ (1 billion) keys in 1 nanosecond (a billionth of a second)
  - So $10^{18}$ keys/sec
  - Thus, we’ll need $\approx 10^{21}$ sec
- **How long is that?**
  - One year $\approx 3 \times 10^7$ sec
  - So need $\approx 3 \times 10^{13}$ years $\approx 30$ trillion years
What about a 256b key in a year?

- Time to start thinking in **astronomical** numbers:
  - If each brute force device is 1mm³...
  - We will need $10^{52}$ of these things...
- $10^{43}$ cubic meters...
- Or the volume of $7 \times 10^{15}$ **suns**!
  - Yes, 7 **petasuns** worth of sci-fi nanotech!
- Brute force is **not a factor** against modern block ciphers...
  - **IF the key is actually random!**
Issues When Using the Building Block

- Block ciphers can only encrypt messages of a certain size
  - If \( M \) is smaller, easy, just pad it (more later)
  - If \( M \) is larger, can repeatedly apply block cipher
    - Particular method = a “block cipher mode”
    - Tricky to get this right!

- If same data is encrypted twice, attacker knows it is the same
  - Solution: incorporate a varying, known quantity (IV = “initialization vector”)

So enter "Modes of operation"

- We don't just run the block cipher on its own...
- But instead as part of a larger "Mode of Operation":
  - Combining the block cipher as the core of a larger function
Electronic Code Book (ECB) mode

• Simplest block cipher mode
• Split message into b-bit blocks $P_1, P_2, \ldots$
• Each block is enciphered independently, separate from the other blocks
  $C_i = E(P_i, K)$
• Since key $K$ is fixed, each block is subject to the same permutation
  • (As though we had a “code book” to map each possible input value to its designated output)
ECB Encryption

Electronic Codebook (ECB) mode encryption
ECB Decryption

Problem: Relationships between $P_i$’s reflected in $C_i$’s
IND-CPA and ECB?

- Of course not!
- \(\mathbf{M, M'}\) is 2x the block length...
  - \(\mathbf{M} = \) all 0s
  - \(\mathbf{M'} = \) 0s for 1 block, 1s for the 2nd block
- This has catastrophic implications in the real world...
Original image, RGB values split into a bunch of b-bit blocks
Encrypted with ECB and interpreting ciphertext directly as RGB
Later (identical) message again encrypted with ECB
Building a Better Cipher Block Mode

• Ensure blocks incorporate more than just the plaintext to mask relationships between blocks. Done carefully, either of these works:
  • Idea #1: include elements of prior computation
  • Idea #2: include positional information

• Plus: need some initial randomness
  • Prevent encryption scheme from determinism revealing relationships between messages
  • Introduce initialization vector (IV):
    • IV is a public nonce, a use-once unique value: Easiest way to get one is generate it randomly
Nonces

- A **nonce** is a public use-once value
  - EG, as the initialization vector
- It is critical to **never ever ever ever** reuse a nonce
  - But if the nonce is 128b or greater, generate it randomly and you are good
- Depending on the algorithm, it can be mildly bad
  - Eh, you leak a little information...
- To catastrophic, CATASTROPHIC FAILURE!
CBC Encryption

\[ E(\text{Plaintext}, K): \]

- If \( b \) is the block size of the block cipher, split the plaintext in blocks of size \( b \): \( P_1, P_2, P_3, \ldots \)
- Choose a random IV (do not reuse for other messages)
- Now compute:

\[ (\text{IV}, C_1, C_2, C_3) \]

- Final ciphertext is \((\text{IV}, C_1, C_2, C_3)\). This is what Eve sees.
**CBC Decryption**

\[ D(\text{Ciphertext}, K): \]
- Take IV out of the ciphertext
- If \( b \) is the block size of the block cipher, split the ciphertext in blocks of size \( b \): \( C_1, C_2, C_3, \ldots \)
- Now compute this:

\[
\begin{align*}
\text{Initialization Vector (IV)} & \quad \text{Ciphertext} & \quad \text{Block Cipher Decryption} \\
\text{Key} & \quad \text{Output the plaintext as the concatenation of } P_1, P_2, P_3, \ldots 
\end{align*}
\]
Original image, RGB values split into a bunch of b-bit blocks
Encrypted with CBC: Should be indistinguishable from random noise
CBC Mode...

- Widely used
- Issue: sequential encryption, can't parallelize encryption
  - **Must** finish encrypting block $b$ before starting $b+1$
  - But you can parallelize decryption
- Parallelizable alternative: CTR (Counter) mode
- Security: If no reuse of nonce, both are provably secure
  (IND-CPA) assuming the underlying block cipher is secure
And padding...

- What happens if length(M) % BlockSize != 0?
  - Need to “Pad” to add bits

- Two main options:
  - Send the length at the start of the message…
    - And then who cares what you add on at the end
  - Use a padding scheme that you can add on to the end…

- EG, PKCS#7:
  - If M % BlockSize == Blocksize - 1: Pad with 0x01
  - If M % BlockSize == Blocksize - 2: Pad with 0x02 0x02
  - If M % BlockSize == 0: Pad a \textbf{full block} with the block size (so for AES 0x20 0x20…)
CTR Mode Encryption

(Nonce = Same as IV)

Counter (CTR) mode encryption

Important that nonce/IV does not repeat across different encryptions.
Choose at random!
Counter Mode Decryption

Note, CTR decryption uses block cipher’s encryption, not decryption
Thoughts on CTR mode...

- CTR mode is actually a stream cipher (more on those later):
  - You no longer need to worry about padding which is nice
  - CTR mode is fully parallelizeable for encryption as well as decryption
  - In high performance applications you can always just throw more compute and encrypt faster
NEVER EVER EVER use CTR Mode!
(Well, if you are paranoid…)

- What happens if you reuse the IV in CBC...
  - It's bad but not catastrophic: you fail IND-CPA but the damage may be tolerable:
    - $M = \{A,A,B\}$
    - $M' = \{A,B,B\}$
    Adversary can see that the first part of $M$ and $M'$ are the same, but not the later part

- What happens if you reuse the IV in CTR mode?
  - It is exactly like reusing a one-time pad!

- An example of a system which fails badly...
  - CTR mode is theoretically as secure as CBC when used properly...
  - But when it is misused it fails catastrophically:
    Personal bias: I believe we need systems that are still robust when implemented incorrectly

F-U!
THIS IS CRYPTO!!!
This was the summer 61A exam mistake!

- They used a python AES library
  - A bad library for a whole host of reasons but...
- When they invoked CTR mode encryption...
  - They never specified an IV...
    Just assuming the library would use a RANDOM IV
  - Nope, library defaults to a 0 IV
- And since multiple different versions of the exam are all encrypted with the same key...
  - **ALL SECURITY WAS LOST!**
What To Use Then?

• What if you want a cipher mode where you don't need to pad (like CTR mode)?
  • But you want the robust to screwup properties of CBC mode?
• Idea: lets do it CTR-like (xor plaintext with block cipher output), but...
• Instead of the next block input being an incremented counter...
  have the next block be the previous ciphertext
• Still lacks integrity however, we'll fix that next time...
CFB Encryption

Cipher Feedback (CFB) mode encryption
CFB Decryption

Cipher Feedback (CFB) mode decryption
CFB doesn't need to pad...

• Since the encryption is XORed with the plaintext...
  • You can end on a "short" block without a problem
  • So more convenient than CBC mode

• But similar security properties as CBC mode
  • Sequential encryption, parallel decryption
  • Same error propagation effects
  • Effectively the same for IND-CPA

• But a bit worse if you reuse the IV
Mallory the Manipulator

- Mallory is an active attacker
  - Can introduce new messages (ciphertext)
  - Can “replay” previous ciphertexts
  - Can cause messages to be reordered or discarded

- A “Man in the Middle” (MITM) attacker
  - Can be much more powerful than just eavesdropping
Encryption Does Not Provide Integrity

• Simple example: Consider a block cipher in CTR mode...

• Suppose Mallory knows that Alice sends to Bob “Pay Mal $0100”. Mallory intercepts corresponding C

  • $M = \text{“Pay Mal $0100”}$. $C = \text{“r4ZC\#jj8qThMK”}$
  • $M_{10..13} = \text{“0100”}$. $C_{10..13} = \text{“ThMK”}$

• Mallory wants to replace some bits of C...
Encryption Does Not Provide Integrity

• Mallory computes
  • “0100” ⊕ C_{10..13}
  • Tells Mallory that section of the counter XOR:
    Remember that CTR mode computes $E_k(\text{IV}||\text{CTR})$ and XORs it with the corresponding part of the message
  • $C'_{10..13} = "9999" \oplus "0100" \oplus C_{10..13}$

• Mallory now forwards to Bob a full $C' = C_{0..9}||C'_{10..13}||C_{14...}$

• Bob will decrypt the message as "Pay Mal $9999"..."

• For a CTR mode cipher, Mallory can in general replace any known message $M$ with a message $M'$ of equal length!
Integrity and Authentication

- Integrity: Bob can confirm that what he’s received is exactly the message M that was originally sent
- Authentication: Bob can confirm that what he’s received was indeed generated by Alice
- Reminder: for either, confidentiality may-or-may-not matter
  - E.g. conf. not needed when Mozilla distributes a new Firefox binary
- Approach using symmetric-key cryptography:
  - Integrity via MACs (which use a shared secret key $K$)
  - Authentication arises due to confidence that only Alice & Bob have $K$
- Approach using public-key cryptography (later on):
  - “Digital signatures” provide both integrity & authentication together
- Key building block: cryptographically strong hash functions
Hash Functions

- Properties
  - Variable input size
  - Fixed output size (e.g., 256 bits)
  - Efficient to compute
  - Pseudo-random (mixes up input extremely well):
    A single bit changes on the input and ~1/2 the bits should change on the output

- Provides a “fingerprint” of a document
  - E.g. “shasum -a 256 <exams/mt1-solutions.pdf” prints
    0843b3802601c848f73ccb5013afa2d5c4d424a6ef477890ebf8db9bc4f7d13d
Cryptographically Strong Hash Functions

- A collision occurs if $x \neq y$ but $\text{Hash}(x) = \text{Hash}(y)$
- Since input size > output size, collisions do happen
- A cryptographically strong $\text{Hash}(x)$ provides three properties:
  - One-way: $h = \text{Hash}(x)$ easy to compute, but not to invert.
    - Intractable to find any $x'$ s.t. $\text{Hash}(x') = h$, for a given $h$
    - Also termed “preimage resistant”
Cryptographically Strong Hash Functions

• The other two properties of a cryptographically strong $\text{Hash}(x)$:
  • Second preimage resistant: given $x$, intractable to find $x'$ s.t. $\text{Hash}(x) = \text{Hash}(x')$
  • Collision resistant: intractable to find any $x, y$ s.t. $\text{Hash}(x) = \text{Hash}(y)$

• Collision resistant $\implies$ Second preimage resistant
  • We consider them separately because given Hash might differ in how well it resists each
  • Also, the Birthday Paradox means that for n-bit Hash, finding $x$-$y$ pair takes only $\approx 2^{n/2}$ hashes
    • Vs. potentially $2^n$ tries for $x'$: $\text{Hash}(x) = \text{Hash}(x')$ for given $x$

• Plus a hash function should look "random"
  • A "PRF" or Pseudo-Random Function
Cryptographically Strong Hash Functions, con’t

- Some contemporary hash functions
  - MD5: 128 bits
    - broken – lack of collision resistance
    - Collisions for the heck of it: https://shells.aachen.ccc.de/~spq/md5.gif
      An MD5 "hash quine": an animated GIF that shows its own hash
  - SHA-1: 160 bits broken spring 2017, but was known to be weak yet still used...
  - SHA-256/SHA-384/SHA-512: 256, 384, 512 bits in the SHA-2 family, at least not currently broken
  - SHA-3: New standard! Yayy!!!! (Based on Keccak, again 256b, 384b, and 512b options)

- Provide a handy way to unambiguously refer to large documents
  - If hash can be securely communicated, provides integrity
    - E.g. Mozilla securely publishes SHA-256(new FF binary)
    - Anyone who fetches binary can use “cat binary | shasum -a 256” to confirm it’s the right one, untampered

- Not enough by themselves for integrity, since functions are completely known – Mallory can just compute revised hash value to go with altered message
SHA-256...

- SHA-256/SHA-384 are two parameters for the SHA-2 hash algorithm, returning 256b or 384b hashes
- Works on blocks with a truncation routine to make it act on sequences of arbitrary length
- Is vulnerable to a *length-extension attack*: s is secret
  - Mallory knows len(s), H(s)
  - Mallory can use this to calculate H(s||M) for an M of Mallory’s construction
    - Works because *all the internal state* at the point of calculating H(s||...) is derivable from H(s) and len(s)
- New SHA-3 standard (Keccak) does not have this property
Stupid Hash Tricks:
Sample A File...

• BlackHat Dude claims to have 150M records stolen from Equifax...
  • How can I as a reporter verify this?

• Idea: If I can have the hacker select 10 **random** lines...
  • And in selecting them also say something about the size of the file...
  • Voila! Verify those lines and I now know he's not full of BS

• Can I use hashing to write a small script which the BlackHat Dude can run?
  • Where I can easily verify that the 10 lines were sampled at random, and can't be faked?
Sample a File

```python
#!/usr/bin/env python
import hashlib, sys
hashes = {}

for line in sys.stdin:
    line = line.strip()
    for x in range(10):
        tmp = "%s-%i-nickrocks" % (line, x)
        hashval = hashlib.sha256(tmp)
        h = hashval.digest()
        if x not in hashes or hashes[x][0] > h:
            hashes[x] = (h, hashval, tmp)

for x in range(10):
    h, hashval, val = hashes[x]
    print "%s="%s" % (hashval.hexdigest(), val)
```
Why does this work?

- For each x in range 0-9...
  - Calculates $H(line||x)$
  - Stores the lowest hash matching so far

- Since the hash appears random...
  - Each iteration is an *independent* sample from the file
  - The expected value of $H(line||x)$ is a function of the size of the file: More lines, and the value is smaller

- To fake it...
  - Would need to generate fake lines, *and see if the hash is suitably low*
  - Yet would need to make sure these fake lines semantically match!
  - Thus you can't just go "John Q Fake", "John Q Fakke", "Fake, John Q", etc...
Message Authentication Codes (MACs)

- Symmetric-key approach for integrity
  - Uses a shared (secret) key $K$
- Goal: when Bob receives a message, can confidently determine it hasn’t been altered
  - In addition, whomever sent it must have possessed $K$
    ($\Rightarrow$ message authentication, sorta...)

- Conceptual approach:
  - Alice sends $\{M, T\}$ to Bob, with tag $T = \text{MAC}(K, M)$
    - Note, $M$ could instead be $C = \text{E}_K'(M)$, but not required
    - When Bob receives $\{M', T'\}$, Bob checks whether $T' = \text{MAC}(K, M')$
      - If so, Bob concludes message untampered, came from Alice
      - If not, Bob discards message as tampered/corrupted
Requirements for Secure MAC Functions

- Suppose MITM attacker Mallory intercepts Alice’s \{M, T\} transmission …
  - … and wants to replace \(M\) with altered \(M^*\)
  - … but doesn’t know shared secret key \(K\)

- We have secure integrity if MAC function \(T = MAC(M, K)\) has two properties:
  - Mallory can’t compute \(T^* = MAC(M^*, K)\)
    - Otherwise, could send Bob \{\(M^*, T^*\)\} and fool him
  - Mallory can’t find \(M^{**}\) such that \(MAC(M^{**}, K) = T\)
    - Otherwise, could send Bob \{\(M^{**}, T\)\} and fool him

- These need to hold even if Mallory can observe many \{\(M_i, T_i\)\} pairs, including for \(M_i\)’s she chose
MAC then Encrypt or Encrypt then MAC

- You should **never** use the same key for the MAC and the Encryption
  - Some MACs will break completely if you reuse the key
  - Even if it is **probably** safe (e.g., AES for encryption, HMAC for MAC) it's still a bad idea

MAC then Encrypt:
- Compute $T = \text{MAC}(M, K_{\text{mac}})$, send $C = E(M||T, K_{\text{encrypt}})$

Encrypt then MAC:
- Compute $C = E(M, K_{\text{encrypt}})$, $T = \text{MAC}(C, K_{\text{mac}})$, send $C||T$
- Theoretically they are the same, but...
  - Once again, it's time for...
HTTPS Authentication in Practice

- When you log into a web site, it sets a "cookie" in your browser
  - All subsequent requests include this cookie so the web server knows who you are
- If an attacker can get your cookie...
  - They can impersonate you on the "Secure" site
- And the attacker can create multiple tries
  - On a WiFi network, inject a bit of JavaScript that repeatedly connects to the site
  - While as a man-in-the-middle to manipulate connections
The TLS 1.0 "Lucky13" Attack: "F-U, This is Cryptography"

- HTTPS/TLS uses MAC then Encrypt
  - With CBC encryption
- The Lucky13 attack changes the cipher text in an attempt to discover the state of a byte
  - But can't predict the MAC
- The TLS connection retries after each failure so the attacker can try multiple times
  - Goal is to determine the status each byte in the authentication cookie which is in a known position
- It detects the timing of the error response
  - Which is different if the guess is right or wrong
    - Even though the underlying algorithm was "proved" secure!
- So always do Encrypt then MAC since, once again, it is more mistake tolerant
The best MAC construction: HMAC

- Idea is to turn a hash function into a MAC
- Since hash functions are often much faster than encryption
- While still maintaining the properties of being a cryptographic hash
- Reduce/expand the key to a single hash block
- XOR the key with the i_pad
  - 0x363636... (one hash block long)
- Hash \((K \oplus i_{pad}) \parallel \text{message}\)
- XOR the key with the o_pad
  - 0x5c5c5c5c...
- Hash \((K \oplus o_{pad}) \parallel \text{first hash}\)

```javascript
function hmac (key, message) {
    if (length(key) > blocksize) {
        key = hash(key)
    }
    while (length(key) < blocksize) {
        key = key || 0x00
    }
    o_key_pad = 0x5c5c5c... \oplus key
    i_key_pad = 0x363636... \oplus key
    return hash(o_key_pad \parallel
                hash(i_key_pad \parallel \text{message}))
}
```
Why This Structure?

- \texttt{i\_pad} and \texttt{o\_pad} are slightly arbitrary
  - But it is necessary for security for the two values to be different
    - So for paranoia chose very different bit patterns

- Second hash prevents appending data
  - Otherwise attacker could add more to the message and the HMAC and it would still be a valid HMAC for the key
    - Wouldn’t be a problem with the key at the \texttt{end} but at the start makes it easier to capture intermediate HMACs

- Is a Pseudo Random Function if the underlying hash is a PRF
  - AKA if you can break this, you can break the hash!

```rust
function hmac (key, message) {
    if (length(key) > blocksize) {
        key = hash(key)
    }
    while (length(key) < blocksize) {
        key = key || 0x00
    }
    o_key_pad = 0x5c5c... ⊕ key
    i_key_pad = 0x3636... ⊕ key
    return hash(o_key_pad ||
        hash(i_key_pad || message))
}
```
Great Properties of HMAC...

- It is still a hash function!
  - So all the good things of a cryptographic hash: An attacker or even the recipient shouldn't be able to calculate $M$ given $HMAC(M,K)$
  - An attacker who doesn't know $K$ can't even verify if $HMAC(M,K) == M$
  - Very different from the hash alone, and potentially very useful: Attacker can't even brute force try to find $M$ based on $HMAC(M,K)$!

- It's probably safe if you screw up and use the same key for both MAC and Encrypt
  - Since it is a different algorithm than the encryption function...
  - *But you shouldn't do this anyway!*
Considerations when using MACs

- Along with messages, can use for data at rest
  - E.g. laptop left in hotel, providing you don’t store the key on the laptop
  - Can build an efficient data structure for this that doesn’t require re-MAC’ing over entire disk image when just a few files change

- MACs in general provide no promise not to leak info about message
  - Compute MAC on ciphertext if this matters
  - Or just use HMAC, which \textbf{does} promise not to leak info if the underlying hash function doesn’t

- \textbf{NEVER} use the same key for MAC and Encryption...
  - Known "FU-this-is-crypto" scenarios reusing an encryption key for MAC in some algorithms when its the same underlying block cipher for both
Plus AEAD Encryption Modes...

- The latest block cipher modes are "AEAD":
  - Authenticated Encryption with Additional Data
- Provides both integrity and confidentiality over the data
  - With integrity also provided for the "Additional Data"
- Used right, these are great
  - Assuming you use a library...
- Used wrong...
  - The AEAD modes are built for "performance", which means parallelization, which means CTR mode, which means IV reuse is a disaster!