Message Integrity
- **Message Authentication Code** (MAC) provides integrity. MAC is a PRF
- **Kerckhoff's Principle**:
  - Don't rely on secret function, rather a public family of functions and secret key
- Best MAC: **HMAC-SHA256**
  - Takes arbitrary length input and gives 256-bit output
- While MAC functions are similar to cryptographic hash functions, they possess different security requirements. To be considered secure, a MAC function must resist existential forgery under chosen-plaintext attacks. This means that even if an attacker has access to an oracle which possesses the secret key and generates MACs for messages of the attacker's choosing, the attacker cannot guess the MAC for other messages (which were not used to query the oracle) without performing infeasible amounts of computation.

Randomness
- PRG is secure if indistinguishable from truly random generator
- Forward secure: Mallory figures out state at time t, she cannot reconstruct output from any $t' < t$
- Generate forward secure PRG from secure PRF *f*:
  - Initialize *state = seed*
  - Generate *output f(state, 0)*
  - *state = f(state, 1),* repeat

Encryption
- Encrypting provides confidentiality
- Definition
  - Eve chooses plaintexts x & y
  - We encrypt either x or y randomly
  - Eve guesses whether we encrypted x or y
  - Secure if Eve can't do better than random guessing
- Always encrypt then provide integrity/add MAC
  - Separate keys for encryption and MAC
- **One-time Pad**
  - Random sequence of bits as long as the plaintext
  - Xor pad with message to get cipher text
- **Stream cipher**: use PRG on a unique key to create pseudorandom string that resembles one-time pad, Xor with plaintext to encrypt, do same to decrypt
  - Can never reuse key
  - Bit-flipping attack: flipping one bit of plaintext flips one bit of cipher text

Key Management
- **Diffie-Hellman**
  - Secure because of discrete log problem: takes too long to find a given $g^a \bmod p$
  - Attacks
    - Man in the Middle: Carlsen/Anand chess game
    - If Alice uses same x for all comm. with Bob
      - Eve can mount subgroup attack (element of order d where d is divisor of p-1)
      - Replace g^y with something of order d.
      - Now the key is determined by y and x mod d.
      - Eve tries all possible values for x mod d.

Block Cipher
- Encrypt fixed size blocks with reusable key
- Pseudorandom permutation: bijective function that maps each unit of input to exactly one unit of output. "Pseudorandom re-ordering"
- "Confusion": highly nonlinear: each character of cipher text should depend on more than one part of the key
- "Diffusion": change character of plaintext, multiple characters of cipher text should change and vice versa (protects from bit-flipping attack)
- **Invertible permutation from a PRF**
  - Evaluate PRF on first half of input
  - Xor with second half
  - Concatenate with second half
- **Feistel Rounds**
  - Single: not pseudorandom, one half of output is unaltered
  - Double: not pseudorandom. Attacker can do this:
    - First query: L, R
      - Response: $f(R) \oplus L$, $f(f(R) \oplus L) \oplus R$
    - Second query: R, $f(R) \oplus L$
      - Response: $f(f(R) \oplus L) \oplus R$, [some horrible thing]
    - Adversary observes that Second half of first response = First half of second response, so it's not pseudorandom
- **AES**: most used block cipher
  - Not Feistel
  - Splits key into 10 sub-keys, uses diff key for each round of chrivening
- Cipher Block Chaining Mode for message longer than block size
  - Use initialization vector IV, to encrypt first block of plaintext, use cipher text on next block of plaintext

Block Ciphers Attacks:
- Known plaintext attack: attacker given randomly chosen number of inputs and outputs corresponding to a target key
- Chosen plaintext attack: attacker chooses certain number of plaintext and gets corresponding cipher texts
- Chosen cipher text attack: attacker chooses certain number of cipher texts and gets corresponding plaintext
- Chosen plaintext/ciphertext attack: attacker can make queries of both types
- Related key attack: attacker can make queries that will be answered using keys related to the target key K
- Forgery attack: attacker deduces answer to a query he hasn't made
- Key recovery attack: attacker recovers key then attacks

Public Key Crypto

- Asymmetric cryptography. Everyone has public and private keys
  - Used to publish data to a lot of people and they all want to verify integrity
  - Receive data from many sources confidentially
- Secret key can't be derived from public key
- Encrypt with public key. Receiver decrypts with their private key.
- RSA can be used for confidentiality and integrity
  - Confidentiality: Encrypt with public key. decrypt with private key
  - Integrity: Encrypt(sign) with private key, Decrypt(verify) with public key
  - Both: do RSA twice with two different key pairs
- The key size must be large enough to make brute-force impractical, but small enough for practical encryption and decryption.

- Why do we need this complicated construction?
  - The simple attempts we saw earlier (XOR and padding) aren't secure
  - **It's possible that there's something simpler than OAEP that's secure.** But OAEP was the first proposal that came with a proof, and it became standardized. In crypto what we implement is often driven by what we can prove secure, even if that comes at the cost of a little bit of complexity or speed

- Why does the input have to be n bits long?
  - Output of OAEP has to be n bits, since that's what the RSA takes as input
  - OAEP happens to have input length equal to output length. So OAEP input length is also n bits
  - We're assuming that m has fixed length ($n-k_1-k_0$). If m is smaller than that, you need a separate padding step first to get it to this length. Pad with 1 followed by 0s
  - What if m is longer than that? Can't happen, we'll see soon.

- In reality we would use a hash function from a crypto library, and in fact, PRFs are constructed from hash functions, as we've seen, so using a PRF to construct a hash function doesn't make sense.

RSA Attacks:

- **Brute force:** trying all possible private keys
- Mathematical attacks: factoring
- **Timing attacks:** using the running time of decryption
- **Hardware-based fault attack:** induce faults in hardware to generate digital signatures
- Chosen ciphertext attack

DES Attack:
Triple DES is still used and not known to be insecure, but by the late 90s the community had agreed upon the need for a new standard with a more modern algorithm.

The attacker can then compute $ENC_{k1}(P)$ for all possible keys k_1 and then decrypt the ciphertext by computing $DEC_{k2}(C)$ for each k_2. Any matches between these two resulting sets are likely to reveal the correct keys. (To speed up the comparison, the $ENC_{k1}(P)$ set can be stored in an in-memory lookup table, then each $DEC_{k2}(C)$ can be matched against the values in the lookup table to find the candidate keys) .

This attack is one of the reasons why DES was replaced by Triple DES — "Double DES" does not provide much additional security against exhaustive key search for an attacker with 2^{56} space.[3] However, Triple DES with a "triple length" (168-bit) key is vulnerable to a meet-in-the-middle attack in 2^{56} space and 2^{112} operations.[4]

If the keysize is k, this attack uses only 2^{k+1}encryptions (and decryptions) (and $O(2^k)$ memory in case a look-up table have been built for the set of forward computations) in contrast to the naive attack, which needs $2^{2 \cdot k}$ encryptions but $O(1)$ space.

Setup: there is double-DES, where a data block is encrypted with key K_1, then again with K_2. The attacker could have access to two plaintext/ciphertext blocks: attacker knows A, B, $C = E_{K2}(E_{K1}(A))$and $D = E_{K2}(E_{K1}(B))$.

Then the attacker can compute all $E_K(A)$ for all values of K (there are 2^{56} of them). Then he can also compute all $D_{K'}(C)$ for all values of K' (again, 2^{56} keys to try). At that point, he will have about 2^{48} candidates for the (K_1,K_2) pair: these are all the pairs (K,K') for which $E_K(A) = D_{K'}(C)$. The "2^{48}" comes from the fact that we are encrypting 64-bit blocks, so about 1 block value every 256 is one of the $E_K(A)$ values, so about 1 value K' every 256 will yield a matching $D_{K'}(C)$.

The 2^{48} candidates can then be tried with the plaintext B and ciphertext D; it is expected that only the right one will pass that test.

Cost: 2^{57} invocations of DES (plus an extra average of 2^{48} for the verification of candidates, but that's negligible), and some memory storage able to hold 2^{56} intermediate values, and sorted. Since the values are sorted, each will differ from the previous by only 8 bits on average, but it must also store the corresponding K, so let's say that a practical implementation would use 10 bytes per value. That's then $10*2^{56}$ bytes of storage, which is rather large (about 650 thousands of *terabytes*) but can be envisioned with existing technology (it would not be cheap, though). The sorting step would be the biggest cost in an "academic" way (about 2^{64} comparisons) but the 2^{57} DES invocations would still be, in practice, more expensive.
I don't see any "80 bits" here. In fact, **security of Double-DES should be considered to be close to "57 bits"**. The storage requirements make it harder to break than a simple "57-bit block cipher", but nowhere near as hard as an 80-bit block cipher.

AES Attack:
All modern computers have a "cache" between main memory (RAM) and the CPU. This cache speeds up access to areas of memory that have been used recently. If an area of memory has been accessed recently, it's probably in the cache, so accessing it again will be quick. If the area hasn't been accessed in a long time, it has probably been evicted from the cache, so accessing it will be slow. The difference in time can leak information about what a process is doing.

This attack was applied to the AES cipher. Essentially, fast implementations of AES use lookup tables, which are arrays used to quickly convert one value into another. The indexes AES uses into these tables depend on the secret key, so there's a possibility that the difference in memory access time caused by the cache can leak information about the secret key. That is exactly what the authors demonstrated.

The authors demonstrate two different techniques for extracting the key. I'll give a simplified explanation of each.
In the first, called the Evict+Time attack, they evict part of one of the lookup tables from the cache, so that all of the AES lookup tables are in the cache except for one index in one table, then they run the encryption. If the encryption accesses that index that has been evicted from the cache, it will run slower, otherwise it will run at a normal speed. So, by timing how long the encryption takes, they can figure out if that table index was accessed or not. Since the table index depends on the key, this leaks information about the key.

The second type of attack, called Prime+Probe, first completely fills the cache with the attacker's data. The encryption process is run, and as it is running, the parts of the lookup table that it uses are loaded from main memory into the cache. Since the cache is full of the attacker's data, some of it will have to be evicted to make room for the part of the table. Once the encryption is done, the attacker accesses their data again (to see which parts have been evicted from the cache), and this tells them which table indexes were used by the encryption process, leaking information about the key.

Types of Attacks

- **Stream ciphers**
  - Bit-flipping attack: flipping one bit of plaintext flips one bit of cipher text

Andrew Wonnacott

Whit Diffie