November 9, 2011 - Cryptography Primer

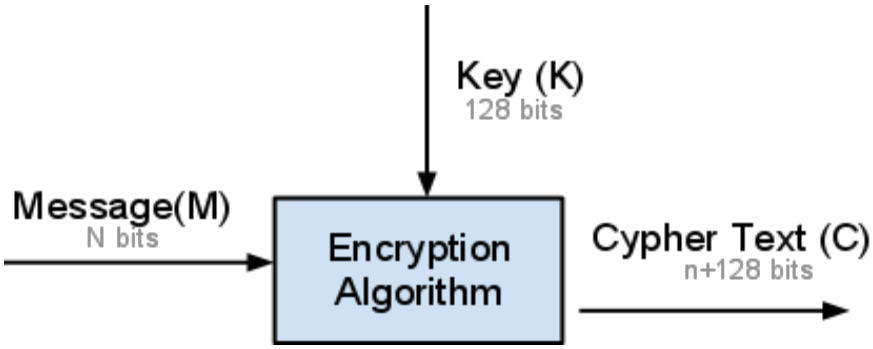
CS261 Computer Security - Taught by David Wagner

Types of Cryptography

When performing cryptography, there are two broad classes of key types (Symmetric-Key and Public-Key). Symmetric-key is faster, but requires that both parties have a shared secret. Public-Key crypto is slower, but allows one of the keys to be public, which makes key distribution substantially easier. Both types of crypto can be used to protect confidentiality or authenticity.

	Symmetric Key	Public Key
Confidentiality	AES-CBC AES-CTR 200-500 MB/S	RSA Encryption 2000-4000 Ops/S
Authenticity/Integrity	MAC CMAC (based on AES) HMAC (based on SHA-256) 200-500 MB/S	RSA Digital Signatures 200-4000 Ops/S

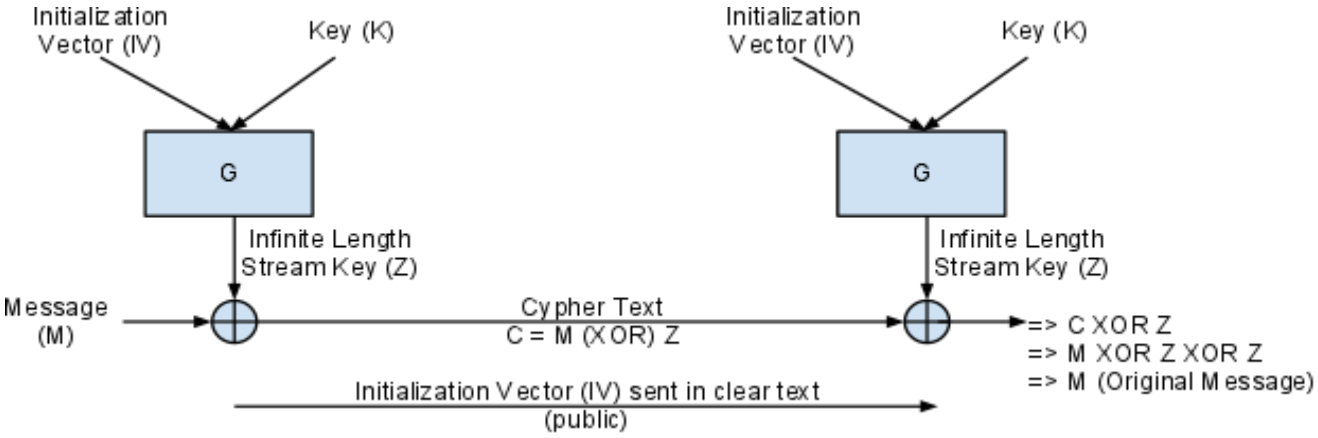
A Basic Encryption Diagram



Encryption is performed by a well-known public algorithm. The algorithm takes two inputs (a key, and a message) and encrypts the message using the provided key to produce a cypher text that can only be decrypted with the appropriate key.

Stream Cyphers

Stream cyphers use a key (and an initialization vector) to generate an infinite length stream of psudo-random bits (Z). To encrypt, take the exclusive-or (XOR) of Z and the message. To decrypt, perform the XOR again. The initialization vector is public, but since the key is private, only the parties in possession of the key (K) can encrypt/decrypt. NOTE: It is critically important that a stream cypher never re-use the stream key (Z), implying that the communicating parties should never re-use the initialization vector (IV).



Attacking a Stream Cypher

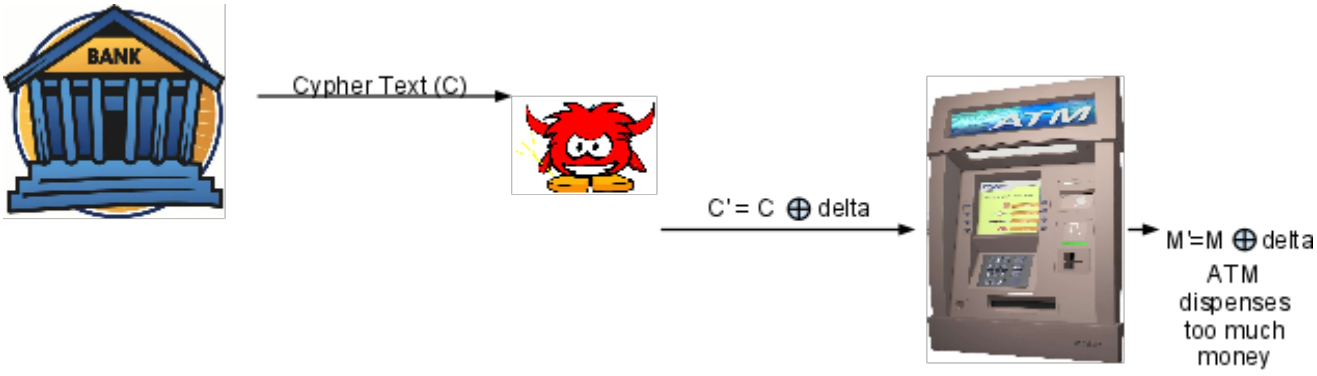
As mentioned above, it's pretty important avoid re-use of the stream key. Given two cypher texts C1 and C2, using the same stream key Z, an attacker can take the XOR of C1 and C2 to get the XOR nd of M1 aM2. It turns out, given text messages, there is approximately 1 bit of surprise per letter, which makes the XOR of M1 and M2 readily guessable, which allows attackers to compromise Z, and therefore the contents of the original messages.

$$C1 = M1 \oplus Z$$
$$C2 = M2 \oplus Z$$
$$C1 \oplus C2 = M1 \oplus Z \oplus Z \oplus M2$$
$$= M1 \oplus M2$$

Attacking Integrity of Stream Cyphers

It's also worth mentioning that stream cyphers (upper-left corner of the table) protect confidentiality, but do not protect integrity. This means an attacker may be able to mess with the message in some surprising ways.

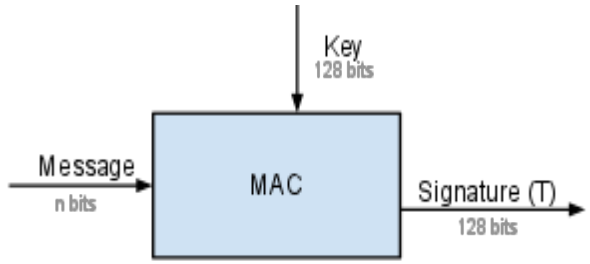
As a motivating example, let's consider an example where the bank is telling an ATM machine to dispense $100 to the user (assume the user has already authenticated themselves and the bank has deducted $100 from the user's account balance). If the attacker knows the message to the ATM says "DISPENSE 100 DOLLARS", the attacker could flip some bits in the middle of the message to modify the message to say "DISPENSE 199 DOLLARS", even if they didn't know the stream key Z. Generally, the encrypted result of a stream cypher is assumed to be arbitrarily modifiable.

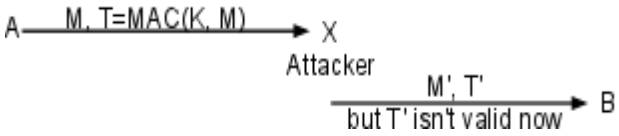


MACs & digital signatures (integrity protection)

Suppose we wanted to defend against the modification seen above. Turns out, we can do that with macs

and digital signatures. A MAC takes in a key and an arbitrary-length message, and produces a psudo-random fixed-length result. Because this result depends on both the key and the message. If the attacker tries to change the message, the old message's mac won't match the new message. Additionally, it's impossible to forge a MAC or digital signature without access to the original key. Therefore, any attempts to change the message will surely be detected.



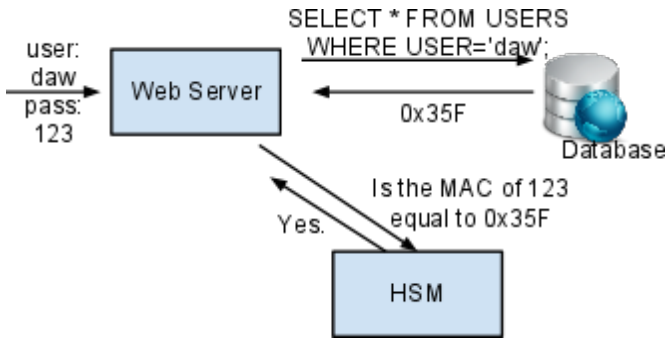


Hashes

A hash function takes in variable-length message and outputs a fixed length checksum, which can be used to verify that the message integrity, but since the hash doesn't require a key, an attacker could generate a new hash for the new message. Hashes are fast and easy to generate, so if the correct hashes are published in a trusted place, they can be used to verify the integrity of an untrusted download (ie. a big ISO downloaded from a mirror).

Hardware Security Modules

Key management can be hard. If a machine holding a key (eg. the database server) is compromised, the security of your entire system can be compromised. The solution is to use a HSM (Hardware Security Module) to manage the keys and perform the requested cryptographic operations on your behalf, without revealing the key. The HSM will also be rate limited, preventing brute force attacks, thereby minimizing the amount of damage an attacker can inflict.



To verify a user's password, we might store a MAC of the password in the database and use an HSM to generate/verify the user's password, as shown in the figure.

Common Tasks in Crypto

There are a few things you commonly want to do:

1. Setup a secure channel between A<-->B
2. Secure Storage A<-->A' (future version of yourself)

Common problems to defend against (and their solutions)

1. Dropped packetrs - use sequence numbers and acks
2. Replay attacks - use sequence numbers, timestamps, or nonces
3. Reflection - where the attacker replays A's message back to A, and A blindly assumes it was sent by B because only A and B have the key. Solution: use separate keys for each direction of communication.