YHQL YLGL YLFL — J. Caesar

Cryptography

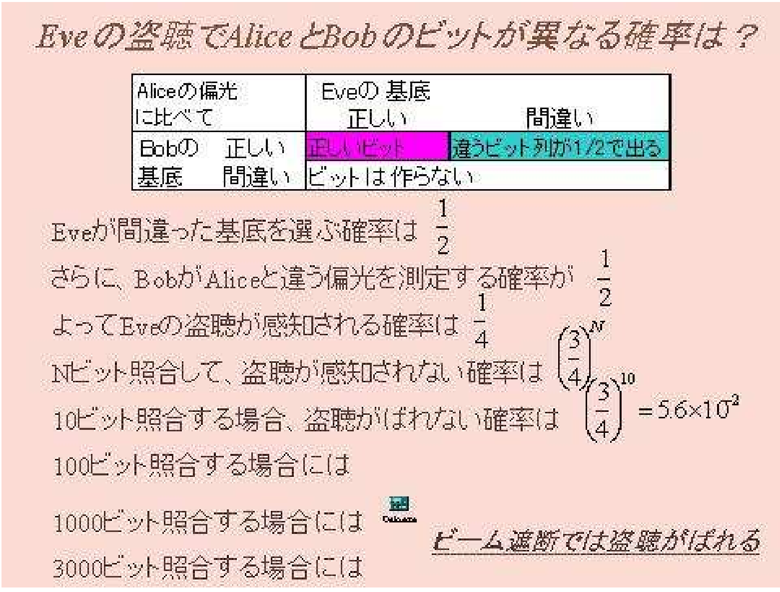
Andreas Zeller/Stephan Neuhaus
Lehrstuhl für Softwaretechnik
Universität des Saarlandes, Saarbrücken

SS 2004/CVS Revision: 1.6

The Menu

- Symmetric Crypto
- Asymmetric Crypto (aka Public-Key)
- Hashes, MICs, and MACs

Cryptography



Terminology

Encryption transforms a *message* or *plaintext* into a *cryptogram* or *ciphertext* under the control of a *key*.

Plaintext will be denoted by P. Sometimes, plaintext is available in blocks or other units; those units are then denoted P_j or p_j.

Same for ciphertext: C, C_j, or c_j.

Same for key: K, and (although this is unusual) k_j.

$$C = E_K(P); \quad P = D_K(C) \qquad c_j = E_K(p_j); \quad p_j = D_K(c_j)$$

Avoid subscript k; easily confused with subscript K.

Secret-Key and Public-Key

- In *secret-key* or *symmetric* cryptography, the participants share one key, which is used for encryption and decryption.
- Examples: DES, AES, IDEA, RC4, Blowfish, Twofish, ...
- In *public-key* or *asymmetric* cryptography, a participant's key is split in two parts: once is public and is used for encryption, one is private and is used for decryption.
- Examples: RSA, Elgamal, ECC

Block Ciphers

A block cipher is a function that takes a n-bit key K and a m-bit bit string B and either encrypts or decrypts B into an m-bit string B'.

The numbers m and n are usually fixed for each block cipher, but can vary between ciphers.

Cipher	n	m
DES	56	64
IDEA	128	64
AES	*varies*	*varies*
RSA	*varies*	*varies*

With AES, you can choose m and n independently from $\{128, 160, 192, 224, 256\}$.

Properties Of a Good Block Cipher

Two (of many) statistical properties (called “cascading” properties):

- Change one key bit and about half of the output bits will change.
- Change one plaintext bit and about half of the output bits will change.

One cryptanalytic property: There is no way to find an unknown key except by trying all keys in some order and stopping when the correct one has been found.

That’s a bit difficult to attain in practice, because we can’t see into the future!

Stream Ciphers

A stream cipher is a function that takes a n-bit key and a (potentially infinite) bit stream as input and produces a (potentially infinite) bit stream as output.

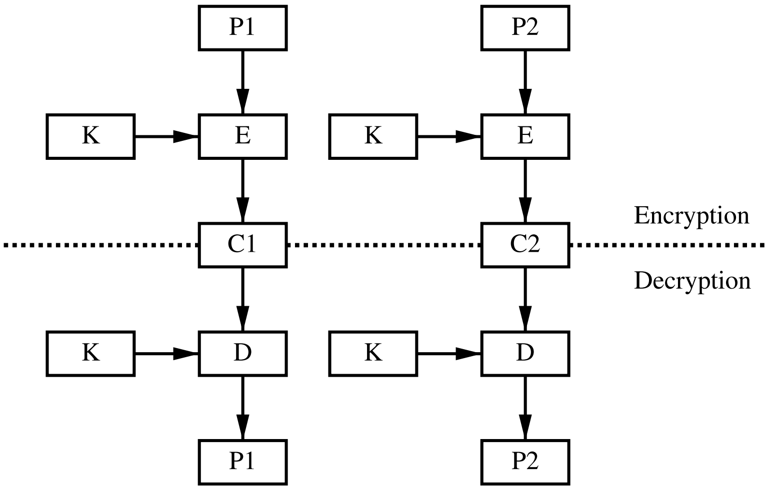
In practice, the input and output bits are grouped into larger blocks, but it’s still not a block cipher because encryption of block j depends on the encryptions of blocks 1 through $j - 1$.

Most stream ciphers work by taking the key K and generating a stream of key bits (or blocks) k_j from it, and then setting

$$c_j \leftarrow m_j \oplus k_j.$$

Decryption then generates the same key stream from K and computes $m_j = c_j \oplus k_j$. Some stream ciphers calculate k_j from k_{j-1} and m_{j-1}.

Electronic Codebook Mode (ECB)



Problems with ECB

A salary database contains salary records encrypted with a 64-bit block cipher in ECB mode.

Trudy knows her own record in plaintext; all the others are just ciphertext:

Type	Person	Contents
Plain	Trudy	`Trudy␣␣␣$20,000␣Progr␣␣␣`
Cipher	Trudy	`a67sj*7k2mlz8m/>suwopslg`
Cipher	Boss	`kdndsuye;hfd7as/8endfuah`

Trudy wants to earn as much as her boss:

`a67sj*7k;hfd7as/suwopslg`

Other Problems With ECB

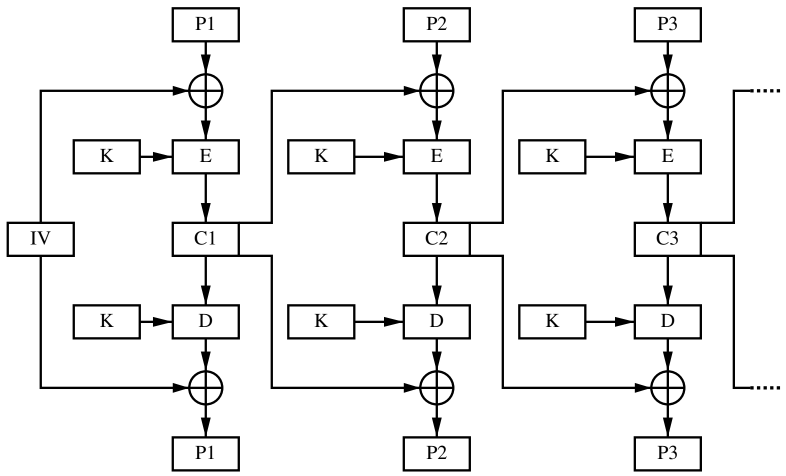
Person	Record
Trudy	`a67sj*7k2mlz8m/>suwopslg`
Boss	`kdndsuye;hfd7as/8endfuah`
CEO	`asoiwq34;hfd7as/kjsd9kjq`
Janitor	`epxn7mn-2mlz8m/>-m,39j,s`
Alice	`kmeqw9ks;hfd7as/suwopslg`

Identical plaintext blocks lead to identical ciphertext blocks.

This makes it possible to find all employees with the same salary as employee X...

. . . without breaking the encryption scheme.

Cipher Block Chaining (CBC)



The “IV” is a random initialization vector that is sent unencrypted with the message.

Features of CBC

If a ciphertext block is modified during the encryption, this will affect only two decrypted plaintext blocks (see exercises).

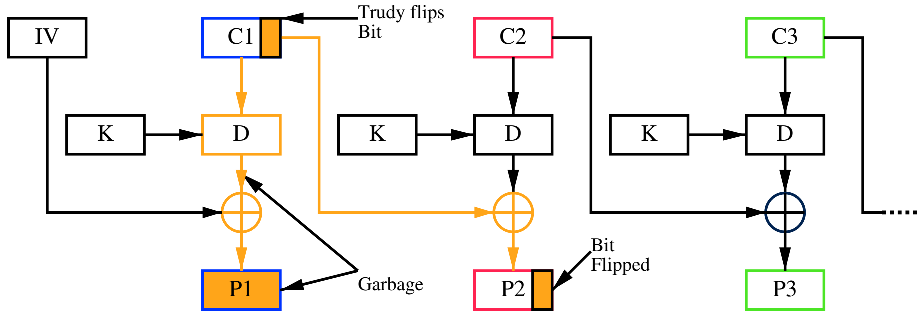
If ciphertext bits (not blocks!) are deleted or added, it will affect the rest of the message (will come out as garbage as long as block synchronization is lost).

In most cases, security is not weakened by choosing a constant IV for each message, but there are exceptions (see exercises).

Problems With CBC (1)

Assume the plaintext is “Trudy␣␣␣R&D␣␣␣$20000␣␣␣”

The character 2 has the bit representation 00110010. 3 is 00110011. Can Trudy force this single bit to change?



If Trudy flips the last bit of C_1, block 1 will decrypt as garbage, but C_2 will decrypt as R&D␣␣␣$2 $\oplus 1 =$ R&D␣␣␣$3, a 50% increase in Trudy's salary!

Problems With CBC (2)

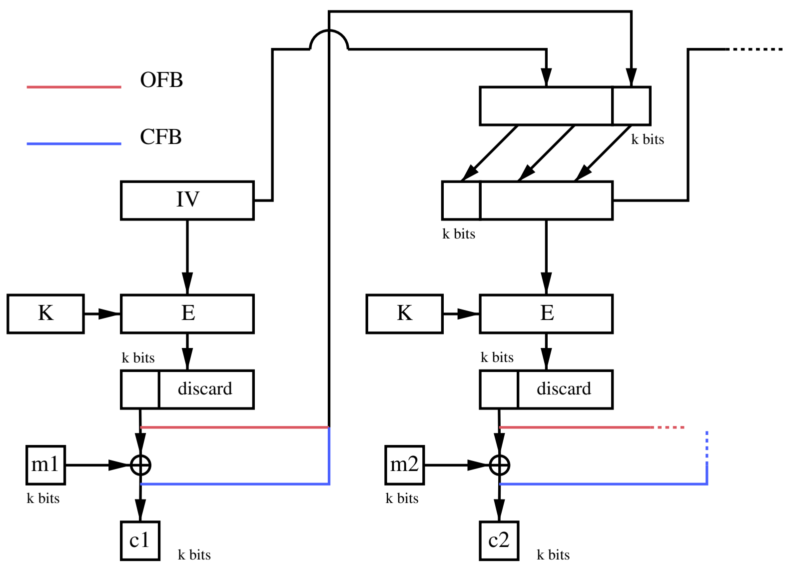
In CBC, $p_i = c_{i-1} \oplus D_K(c_i)$ where c_0 is the IV. Hence, $D(c_i) = c_{i-1} \oplus p_i$.

Therefore, if you know all the plaintext blocks and all the ciphertext blocks, you can *rearrange* the ciphertext blocks and know what the new encrypted message will decrypt to.

Arrangement	Decryption
$c_0 \vert c_1 \vert c_2 \vert c_3$	$p_1 \vert p_2 \vert p_3$
$c_1 \vert c_0 \vert c_2 \vert c_3$	$c_1 \oplus D(c_0) \vert c_0 \oplus D(c_1) \vert p_3$
$c_0 \vert c_1 \vert c_2 \vert c_2$	$p_1 \vert p_2 \vert c_2 \oplus D(c_2) = p_3 \oplus D(c_3) \oplus D(c_2)$

It is improbable that rearranged messages will decrypt to something useful, but it's still a threat.

Feedback Modes (CFB, OFB)



Feedback Modes Explained

OFB and CFB generate a *one-time pad* consisting of pseudo-random numbers from an IV and a key: $c_i = p_i \oplus k_i$, where k_i is the key stream generated by the IV and K.

OFB	CFB
Uses only key and IV to generate key stream	Also uses message
Encryption pad can be computed beforehand	Must wait for plaintext
Can generate ciphertext as fast as the plaintext appears	Can generate ciphertext as fast as plaintext appears if block sizes match

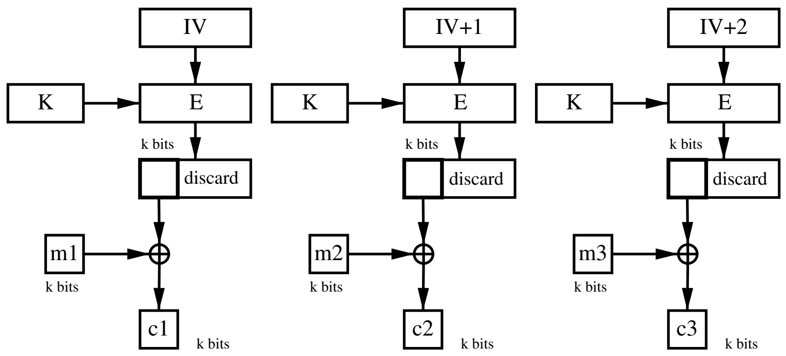
Effect of Transmission Errors and Attacks

Error	OFB Decryption	CFB Decryption
Garbled bits	Garbles rest of message	Garbles only these bits
Added ciphertext	Garbles rest of message	Will re-synchronize

If Trudy knows the one-time pad, she can alter the ciphertext to say anything she wants:

Since $p_i = c_i \oplus k_i$, we must substitute $p_i' \oplus k_i$ for c_i if we want the i-th ciphertext character to decrypt to p_i'.

Counter Mode (CTR)



Key stream can again be precomputed (like OFB) and decryption can start at any point (not just at the beginning).

Advice

Encrypt What	Recommendation
Files	CBC with a random IV (especially if you want to access the file non-sequentially). Also use a good Message Integrity Code (MIC) in order to detect modification of the ciphertext.
Net Sessions	CFB or OFB with a random IV or native stream cipher like RC4. Protect each packet with a MIC.
Short Database Fields	CBC with random IV and MIC.
Encryption Keys	ECB with MIC.

Advice on Algorithms and Key Sizes

Do not use DES (key size too short).

If you *must* use DES (and only then), **do** use 3DES (using three keys of 56 bits) or 2Key-3DES (using only two). Both(!) have an effective key size of 112 bits.

Do not just encrypt twice with DES to get longer keys!

Do choose key sizes of at least 112 bits.

Do use one of these algorithms; they are probably OK: IDEA, AES, RC4, RC5, Blowfish, Twofish.

Do not deploy any algorithm without checking whether it has been broken in the meantime. It happens.

More Advice on Algorithms

Do not use these ciphers; they are broken: GDES, DESX, (and most other DES variants), Bass-O-Matic, Khufu, Khafre, FEAL, Akelarre, SPEED, Enigma 2000, JEL, StreamBuddy, and many *many* more.

N.B.: DES is an excellent cipher; it has withstood about 30 years of cryptanalysis. The best way of attacking DES is brute force. The problem with DES is that brute force is too easy.

Why Isn't He Showing Source Code?

Never roll your own crypto algorithms!

It's *very, very* difficult to create a good crypto algorithm. Without proper education (and probably years of experience), you can't do it. The ciphertext might look "random" to you, but an experienced cryptographer can probably break it.

Never write your own crypto code!

Even when using algorithms that are known to be good, it's still bloody difficult to write correct crypto code.

Example: I've seen an application that fed the plaintext back instead of the ciphertext, turning CFB into "PFB", which exposes patterns in the input. (Code change: one identifier.)

Shortest Possible Intro to Public Key

- A *public key pair* consists of a public *encryption key* e and a private *decryption* or *signature key* d that can't easily be computed from e.
- Each key defines a function associated with that key. For the key pair belonging to Alice, we'll write $\{\cdot\}_{\text{Alice}}$ for the public encryption function and $[\cdot]_{\text{Alice}}$ for the private decryption function.
- For every message M in the domain of $\{\cdot\}_{\text{Alice}}$, we have $[\{M\}_{\text{Alice}}]_{\text{Alice}} = M$ (if $\{M\}_{\text{Alice}}$ is in the domain of $[\cdot]$), and for every message M' in the domain of $[\cdot]_{\text{Alice}}$, we have $\{[M']_{\text{Alice}}\}_{\text{Alice}} = M'$.
- It is not necessary that $\{M\}_{\text{Alice}}$ be in the domain of $[\cdot]_{\text{Alice}}$. (Signature without encryption.)

Best Known Public-Key Algorithm: RSA

> "The obvious mathematical breakthrough would be development of an easy way to factor large prime numbers." Bill Gates, *The Road Ahead*

RSA works because it is difficult (under certain circumstances) to factor large numbers that are the product of two large primes. We think.

RSA is a variable-length block cipher, where it makes no sense to employ any mode other than ECB!

There are crypto libraries out there that are so orthogonal that they allow you to specify RSA with CBC, *but that's nonsense!*

It's even more important than in the case with symmetric crypto *not to write your own RSA package*, because there are even more things that can go wrong when you don't do it right.

RSA Key Generation

The number of positive integers that are relatively prime to some positive integer x (and less than it) is written $\phi(x)$, aka Euler's Totient Function.

RSA works because of one of Euler's theorems which says that $a^{\phi(n)} \equiv 1 \pmod n$ if $\gcd(a, n) = 1$.

Let p and q be two different odd primes. Let $n = pq$. We have $\phi(n) = (p-1)(q-1)$. Choose e such that $\gcd(e, p-1) = 1$ and $\gcd(e, q-1) = 1$. Note that this means that $\gcd\left(e, \phi(n)\right) = 1$.

Compute d such that $ed \equiv 1 \pmod{\phi(n)}$.

The public key is (e, n); the private key is (d, n).

Some choices of p and q are better than others! Beware!

RSA Encryption/Decryption

To encrypt a message $0 < P < n$, compute $C = P^e \bmod n$. To decrypt a message, compute $P' = C^d \bmod n$.

$C^d \equiv (P^e \bmod n)^d \equiv P^{ed} \equiv P^{k\phi(n)+1} \equiv P^{k\phi(n)} \cdot P \equiv P \pmod n$.

When P is a multiple of p or q, things also work out. (Having $P = kp$ would expose p, because $\gcd(P^e \bmod n, n) = p$, but that is just as likely as correctly guessing p or q.)

RSA Pitfalls: Small Encryption Exponent

You want to send a message P to three participants with public keys $(3, n_1)$, $(3, n_2)$, and $(3, n_3)$. Encryption is:

$$C_j = P^3 \bmod n_j \qquad \text{for } 1 \le j \le 3.$$

By the Chinese Remainder Theorem, we can compute some x with $C_j = x \bmod n_j$ $(1 \le j \le 3)$, if the n_j are pairwise relatively prime (very likely).

This x is unique modulo $n_1 n_2 n_3$. We compute the smallest nonnegative such x.

Since $P < n_j$ for $1 \le j \le 3$, we have $x = P^3$.

$\Longrightarrow$ Compute x, take cube root, get P.

Solution: Choose $e = 65537$.

RSA Pitfalls: No Padding/Small Message

If $e = 3$ (many still are!), and if the message P is so small that $P^3 < n$, then you can simply take the e-th root of the ciphertext to get P back.

Most messages are indeed small (112-bit or 128-bit encryption keys, for example), where there's a chance that this will happen.

Solution: Pad the message on the left with nonzero (or random) bits, such that $P^e > n$.

These are just two of the easier pitfalls. There are many more (for example, the exact form of the factors p and q etc.). Therefore:

Never roll your own RSA routines!

RSA Pitfalls: Timing Attacks

If you implement $x^a \bmod n$, you'll very probably use a technique that doesn't always take the same time for every a.

Some of the most common multiplication algorithms can be expolited simply by measuring how long it takes to compute $x^a \bmod n$ when a isn't known.

That way, a (or even some bits of a) can be recovered indirectly.

Therefore:

Never roll your own RSA routines!

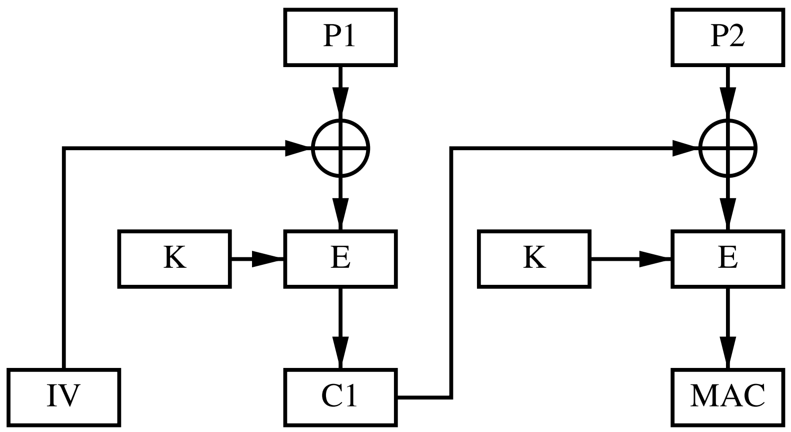
MACs and MICs

They are *cryptographic checksums*:

- They map an arbitrarily long byte sequence to a fixed (and usually rather small) number of bytes.
- Given a checksum, it is infeasible to find a message that has this checksum.
- Given a message, it is infeasible to find another message with the same checksum.
- They depend on a *key* such that the checksum will be different when different keys are used and that the checksum can't be predicted without knowing the key.

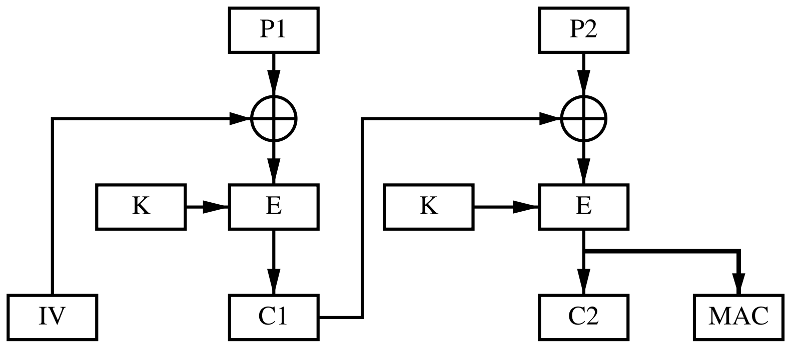
All but the last requirements are also required of hash functions.

Computing a MAC: CBC Residue



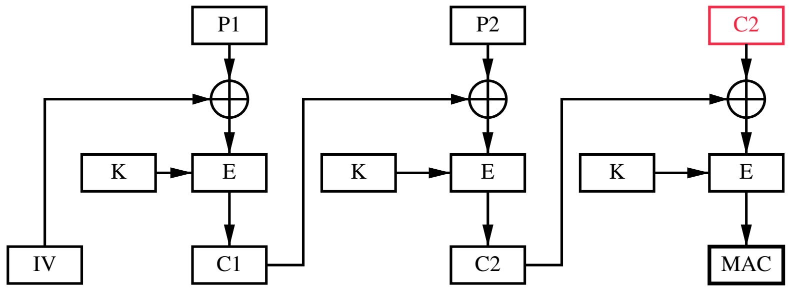
Privacy And Integrity (1)

Can we get encryption *and* integrity protection at the same time?



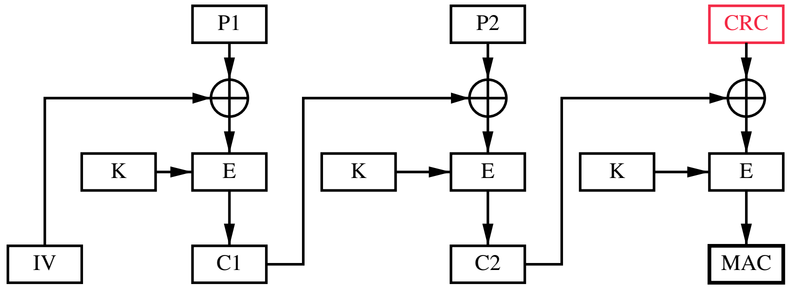
This can't be right. Simply repeating the last ciphertext block can't enhance security, because an attacker would simply modify that block twice.

Privacy And Integrity (2)



This can't be right either. No matter what the value of C_2 is, the input to the encryption function will be 0, because $x \oplus x = 0$ for all x. Therefore, the value of C_2 won't have an influence on the "MAC".

Privacy And Integrity (3)



This will *almost* work. Problems occur if the CRC is short. Longer (but non-cryptographic) checksums are also suspect.

The Moral

You might be able to get integrity and privacy protection in one pass over the data, but how to do that is still under active research.

Your best best will be to do two passes over the data; the first pass should compute a hash (or keyed hash; later), and the second pass should encrypt.

If you use a hash function, the hash should be encrypted, too. A keyed hash can be transmitted in the clear, *if* the keys used for hashing and encryption are different.

Do not try to take shortcuts in crypto!

Cryptographic Hash Functions

Cryptograhic hash functions have the following properties:

- They map an arbitrarily long byte sequence to a fixed (and usually rather small) number of bytes, called a *hash* or *message digest*.
- Given a checksum, it is infeasible to find a message that has this checksum.
- Given a message, it is infeasible to find another message with the same checksum.

Note that it cannot be *impossible* to find collisions, because of the pigeonhole principle: If you have infinitely many messages, but only finitely many hashes, some messages must hash to the same value.

How Infeasible is Finding a Collision?

Let's say the hash function is cryptographically strong, but I still want to crack it. I follow the following algorithm:

1. Set $S \leftarrow \emptyset$.
2. Generate a new, random message m and its hash $h(m)$.
3. If $(m, h(m)) \in S$, terminate the algorithm. Otherwise, set $S \leftarrow S \cup (m, h(m))$ and repeat step 2.

How often will step 2 have to be executed before the algorithm terminates? (We may assume that the messages that are generated contain no duplicates.)

Collision Probability (1)

Assume that the hash function maps messages to n-bit digests. We model the problem of finding a collision as follows:

We have an urn containing 2^n numbered balls. We draw balls from the urn, note the number on them and replace them. How often must we draw balls before a number appears that is already on our list?

What's the probability that the first k draws are all distinct? Set $N = 2^n$.

$$P(k) = \frac{N}{N} \cdot \frac{N-1}{N} \cdots \frac{N-k+1}{N} = \prod_{j=0}^{k-1} \left(1 - \frac{j}{N}\right)$$

Now we want to know the first k for which $P(k) < 0.5$.

Collision Probability (2)

$$\begin{aligned}
\prod_{j=0}^{k-1} \left(1 - \frac{j}{N}\right) &< \left(\frac{1}{k} \sum_{j=0}^{k-1} \left(1 - \frac{j}{N}\right)\right)^k \\
&= \left(1 - \frac{k-1}{2N}\right)^k \\
&\approx \left(1 - \frac{k}{2N}\right)^k \\
&< \exp(-k^2/2N).
\end{aligned}$$

To find k for which $P(k) < 0.5$, we solve $\exp(-k^2/2N) < 0.5$ for k to yield $k > \lambda\sqrt{N}$ where $\lambda = \sqrt{2 \ln 2} \approx 1.18$.

If $N = 2^n$, and if n is even, $\sqrt{N} = 2^{n/2}$. We'll leave out the factor of λ (since it's so close to 1).

Collision Probability (3)

For an n-bit hash, we have to hash about $2^{n/2}$ messages before we can expect a collision with probability at least $1/2$.

That means that

Any hash function that has less than 128 bits of hash should be considered insecure and weak *and should not be used*.

Well-Known Hash Functions

For some reason, it seems to be easier to create good hash functions than to create good encryption schemes. Some good hash functions are:

Name	Bits	Comment
MD5	128	Less fast than predecessor MD4 (*)
SHA-1	160	Standard (*)
RIPEMD-160	160	

(*) Length limited to be less than 2^{64} bits; but "If you can't say something in 2^{64} bits, you probably shouldn't say it at all".

If we could hash one Terabyte per second (which we can't), hashing the entire 2^{64} bits would take about 550,000 years to compute.

Computing MACs With Hashes

A hash function is collision resistant, so we can compute $\mathrm{hash}(m)$ for a message m and send that as the MAC.

No, we can't, because of the fourth requirement for MACs:

> They depend on a *key* such that the checksum will be different when different keys are used and that the checksum can't be predicted without knowing the key.

How can we add a key to the message digest algorithm?

MACs With Hashes And Keys (1)

Alice and Bob agree on a shared secret K_{AB}. If Alice sends a message m to Bob, she concatenates K_{AB} and m and sends $\mathrm{hash}(K_{AB}|m)$ as the MAC.

This way, the message digest depends on the secret and Eve cannot send a message that will be accepted as authentic.

Wrong.

The key to the attack is that it's possible to compute $\mathrm{hash}(x|y)$ if you know $\mathrm{hash}(x)$ and y.

That means that if Eve sees $\mathrm{hash}(K_{AB}|m)$, she can compute

$$\mathrm{hash}(K_{AB}|m|\texttt{Romeo must die})$$

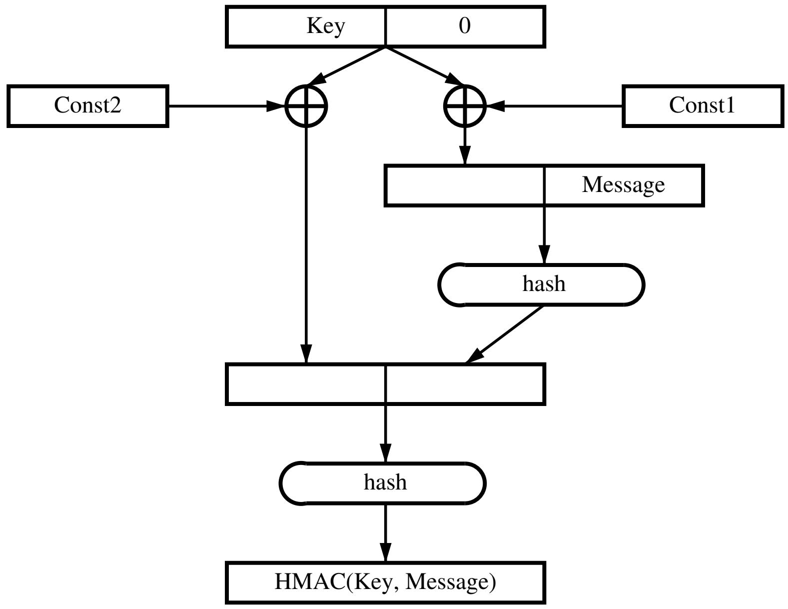
MACs With Hashes And Keys (2)

Solution: HMAC, which is becoming the standard MAC.

HMAC is provably "secure" if the underlying hash algorithm is "secure":

- It has collision resistance; and
- if the attacker doesn't know the key K, he cannot compute $\mathrm{MAC}(K, x)$ even if he sees arbitrarily many $\mathrm{MAC}(K, y)$ values.

HMAC



Libraries: OpenSSL and cryptlib (1)

	OpenSSL	cryptlib
Author	Eric Young, OpenSSL Project Team	Peter Gutmann
Since	1990's	1990's
Vuln's	several	none
Scope	wide, many OSS projects	wide, mostly non-OSS projects
Approach	bunch of functions	application support
Runs on	mostly Unix and Windows	tons of stuff: mainframes to embedded systems
License	OSS	OSS
Free?	all use	noncommercial use

Libraries: OpenSSL and cryptlib (2)

Additionally, cryptlib supports hardware encryption, PGP data formats, S/MIME enveloping, LDAP, RDBMS and ODBC keystores, and CRL checking.

It is difficult to use cryptlib in an insecure way; cryptlib checks on each operation whether it is meaningful for the participating objects.

Has many secure defaults.

Once it's set up, encrypting an email message is a matter of three lines, including S/MIME enveloping.

Summary

- Symmetric Crypto
- Asymmetric Crypto (aka Public-Key)
- Hashes, MICs, and MACs

References

- The OpenSSL Project, `http://www.openssl.org`.
- Cryptlib, `http://www.cryptlib.com`.
- Bruce Schneier, *Applied Cryptography*, John Wiley & Sons
- Charlie Kaufman, Radia Perlman, Mike Speciner, *Network Security*, Prentice-Hall