MasterMath Cryptology 2015 - 2/2 - Cryptanalysis

Tuesday, March 17, 2015 11:55 AM

Lecture Notes v5

UU Minnaertbuilding Room 211 14:00-16:45

1. Introduction

The topic of the second half of this course is *cryptanalysis*.
Cryptanalysis is the study of the security of cryptologic primitives.
The security of cryptologic primitives is always bounded from above due to the existence of so-called *generic* attacks that work against every primitive of the same type.
Attacks that prove the security to be below this upper bound will have to use the internal structure of the primitive and are called *cryptanalytic attacks*.
Practical cryptanalytic attacks have immediate impact on the security of deployed cryptographic systems.
If a cryptanalytic attack is faster than generic attacks, yet practically infeasible, then we call it a *certificational* cryptanalytic attack.
Certificational attacks are also important as they disprove the security claims of the primitive by exposing a structural weakness, which may eventually lead to practical attacks.
Existence of certificational attacks thus indicate it's high time to start migrating to more secure primitives, especially since such migrations typically take more than a few years.

The cryptanalytic techniques used to build cryptanalytic attacks vary widely and are tailored to each specific primitive.
In this course we will visit various important cryptanalytic techniques and apply them on example (toy) primitives.

2. Challenges

The webpage for this second half will also contain challenges that are voluntary but will allow you to get more experience with the techniques from this course.
Solutions can be submitted to mastermath@marc-stevens.nl.
For each challenge, points will be awarded based on order of submission of valid solutions.
Solutions can be submitted as a group as well, the total points for its members will be averaged out.

3. Symmetric Primitives and Generic Attacks

First though we will cover symmetric primitives and some generic attacks as they themselves can be used as part of a cryptanalytic attack:

Stream cipher
Block cipher and Modes of operations
Cryptographic hash function
Message Authentication Code (MAC)

4. Stream Ciphers

OPT - One-Time-Pad cipher

The one-time-pad cipher is essentially the only encryption method that provides *perfect secrecy* or *information-theoretical secrecy*, i.e., even given infinite computational resources one cannot learn information about the enciphered plaintext given only the ciphertext other than the obvious observation about its length.

For given $m \in \{0,1\}^l$, the cipher selects *uniformly at random* a key $k \xleftarrow{r} \{0,1\}^l$ of the *same length* and outputs the ciphertext $C = m \oplus k = (m_1 \oplus k_1)|(m_2 \oplus k_2)|\cdots|(m_l \oplus k_l)$ computed as the bitwise XOR of the message and key. Decryption is simple with $m = C \oplus k$.

Perfect secrecy can be shown with the observation that for any two messages $m_1, m_2 \in \{0,1\}^l$ and a ciphertext C belonging to either m_1 or m_2:

$$\Pr[C = m_1 \oplus k] = \Pr[k = C \oplus m_1] = 2^{-n} = \Pr[k = C \oplus m_2] = \Pr[C = m_2 \oplus k]$$

Thus the ciphertext distribution perfectly statistically hides the plaintext distribution.

The OTP provides perfect secrecy, but no authenticity, i.e., any attacker within the communication channel can apply a difference D to the ciphertext C: $C' = C \oplus D$, that results directly in that the deciphered plaintext on the receiving end is also altered: $P' = C' \oplus k = C \oplus D \oplus k = m \oplus D$.

Finally, the OTP does NOT provide perfectly secrecy if:

- The key is not kept secret
- The key is not selected uniformly at random, but with some biased distribution
- The key is reused to encrypt more than one message
  (the attacker learns $C_1 \oplus C_2 = m_1 \oplus k \oplus m_2 \oplus k = m_1 \oplus m_2$ about m_1 and m_2).

Stream ciphers operate very similar to the one-time pad, adding a keystream to the messagestream to obtain a ciphertext.

Except that the large keystream, instead of being chosen uniformly at random, is generated from a relative small key (used only once).

Predominantly the streams are bitstreams, but in general, any alphabet may be used for the stream characters.

We'll assume that keys and streams consist of bits.

There are two types of stream ciphers, namely synchronous and asynchronous or self-synchronizing stream ciphers.

The first type takes as input a key to initialize its state and in each iteration will update its state and output some keystream that is added to the message to form the cipherstream.

More formally:

Init: $K \rightarrow S_0$

Update i: $S_i \rightarrow (S_{i+1}, O_i)$

Output i: $(O_i, M_i) \rightarrow C_i = O_i \oplus M_i$

The state update is thus independent from the message and cipherstream.

The second type also allows the state update to use the cipherstream from the last l iterations, that can be used to allow deciphering to recover from missing symbols.

Update: $(S_i, C_{i-1}, C_{i-2}, \ldots, C_{i-l}) \rightarrow (S_{i+1}, O_i)$

We will limit ourselves to the most common type of stream cipher, namely synchronous stream ciphers.

4.1. Security

The following security properties are required for a cryptographic stream cipher:

- *Indistinguishability*: the output keystream is indistinguishable from a uniform randomly chosen stream
- *Key Recovery Hardness*: it is hard to recover the key from the output keystream
- *State Recovery Hardness*: it is hard to recover the state from the output keystream

Indistinguishability is the most important as it implies both Key Recovery Hardness and State Recovery Hardness.
Being able to recover the key or the state using nearly all of the output keystream allows one to exactly predict the remaining portion, thus distinguishing it from random.
Also, it implicates that there are no (significant) biases for any portion of the keystream that may be exploited to obtain more information about the message stream.

Let n be the input key size, $D(n)$ be the size of the generated output keystream, $C(n)$ be the upper bound on the number of operations, A be an algorithm that on input $(D(n), C(n), O)$ reads a stream of size $\leq D(n)$ from Oracle O and performs at most $C(n)$ operations and returns either 0 or 1.
We can define two oracles, namely O_{sc} that first selects a key $k \xleftarrow{r} \{0,1\}^n$ selected uniformly at random and returns the output key stream of the stream cipher under key k.
And O_{ur} that returns a stream of bits selected independently and uniformly at random.
So we can call a stream cipher $(D(n), C(n), \epsilon(n))$-*indistinguishable* if for any algorithm A:

$$|\Pr[A(D(n), C(n), O_{sc}) = 1] - \Pr[A(D(n), C(n), O_{ur}) = 1]| \leq \epsilon(n)$$

For a stream cipher with fixed input key size n and state size l, one can require practical parameters, e.g., $D(n) = C(n) = 2^{\min(n,l)}$, $\epsilon(n) = 2^{-80}$.
For a stream cipher with variable input key size n, one can also define security against *efficient*, i.e., polynomial time, attackers:
With slight abuse of notation we set $D(n) = poly(n)$, $C(n) = poly(n)$, $\epsilon(n) = negl(n)$ to define security against adversaries that read only a polynomial (in n) size output stream and perform a polynomial number of operations which are successful with a negligible (in n) success probabilility.
Note that *information-theoretical security* with $C(n) = \infty$ is unattainable.

4.2. Generic key recovery attack

A generic key recovery attack against an observed key stream is simply an exhaustive search among all possible keys. Computing a portion of output keystream, it returns the first candidate keys whose output key stream is equal to the observed key stream, or $\perp$ otherwise.
It requires $O(2^n)$ operations and succeeds with probability 1 if the observed key stream is sufficiently long to dismiss all bad candidate keys.

4.3. Generic state recovery attack

A generic state recovery attack against an observed key stream is simply an exhaustive search among all possible states. Computing a portion of output keystream starting from that state, it returns the first candidate state whose output key stream is equal to the observed key stream, or $\perp$ otherwise.
It requires $O(2^l)$ operations where l is the size of the state and succeeds with probability 1 if the observed key stream is sufficiently long to dismiss all bad candidate states.

4.4. Generic distinguishing attacks

Both the generic key recovery and generic state recovery attacks lead to a generic distinguishing attack, where it returns 0 if the attack returned $\perp$, and 1 otherwise.
[Correction: in the lecture I discussed another common attack type based on the period of the Update function, however this is better classified as a cryptanalytic attack as it depends on the internal structure.]
The security of the stream cipher is thus upper bounded by $O\left(2^{\min(n,l)}\right)$.

4.5. Trivial malleability attack

Like the one-time-pad, secure stream ciphers provide only *privacy*, i.e., any eavesdropper does not learn any new information about the message by observing the ciphertext other than the its length.
Importantly, the one-time-pad and any stream cipher provide no security against malleability.
I.e., an active attacker may trivially add any difference to the messagestream $M' = M \oplus E$ by adding the difference to the cipherstream $C' = C \oplus E$ between sender and receiver, as then the

receiver decodes:

$$C' \oplus KS = C \oplus E \oplus KS = M \oplus KS \oplus E \oplus KS = M \oplus E$$

Note that even though the attacker is able to modify the message, he does not learn the message itself.
Such an attack can be exploited with prior knowledge about the message, say the attacker knows the format and amount of a bank transfer, then it would be simple to change a 1M$ transfer into a 2M$ transfer.
To obtain security against malleability, one may use a Message Authentication Code.

4.6. Key reuse

Note that when a key k is reused to encrypt 2 messages M and M' then the difference is leaked via $C \oplus C' = M \oplus KS \oplus M' \oplus KS = M \oplus M'$.
Hence it is important to ensure that a different key is used for every message.
To achieve this one may split up the key into a key part and a *nonce* part.
Then the key can be reused while ensuring no nonce value will be reused.
E.g., the nonce can be a message counter, or if sufficiently long, picked from random and communicated separately.

5. Block ciphers

A Block Cipher only encrypts fixed-size blocks and has three parameters, the security parameter n (e.g., 128 or variable), the key space $K(n)$ (e.g, $\{0,1\}^n$) and the block space $M(n)$ (e.g., $\{0,1\}^n$).
Then a Block Cipher is a mapping $Enc\colon K(n) \times M(n) \to M(n)$ with the constraint that for every $k \in \mathrm{K}(n)$: $Enc_k = Enc(k,\cdot)\colon M(n) \to M(n)$ is a permutation and its inverse $Dec_k = Enc_k^{-1}$ is efficiently computable.
We'll only consider Block Cipher designs with fixed n, key space and block space.

5.1. Generic key recovery attack

Suppose an attack has access to a block cipher oracle, i.e., an oracle chooses a key uniformly at random and then upon request returns plaintext-ciphertext pairs.
Depending on the allowed type of requests one can distinguish different type of attacks, e.g.:

- Known plaintext attacks: oracle returns pairs for randomly chosen plaintexts
- Chosen plaintext attacks: oracle returns pair for plaintext requested by the attacker

A generic key recovery attack against such an block cipher oracle is simply an exhaustive search among all possible keys.
The attacker first makes a few queries obtaining pairs $(p_1, c_1), \dots, (p_l, c_l)$.
Then it simply goes through all possible key candidates k, it returns the first candidate key k that satisfies $c_i = E_k(p_i)$ for $i = 1, \dots, l$, or $\perp$ otherwise.
It requires $O(|K(n)|)$ operations and succeeds with almost certainty (there is a probility that another key is found first that satisfies the pairs, but this probability decreases exponentially in the size of the list l).

5.2. Generic 1-out-of-L key recovery attack

Now consider a variation wherein the attacker has access to L block cipher oracles and his aim is to succeed to recover the key of at least one oracle.
With this variation, we can do better in a chosen plaintext setting with $O(K(n)/L)$ operations.
The attacker selects a few plaintexts $p_1, \dots, p_l$ and queries these for every oracle and obtains ciphertexts $c_{1,1}, \dots, c_{l,L}$.
Then it simply goes through all possible key candidates k, it returns the first candidate key k that satisfies $c_{i,j} = E_k(p_{i,j})$ for some $j = 1, \dots, L$ for all $i = 1, \dots, l$, or $\perp$ otherwise.

5.2. Generic L-out-of-L key recovery attack / generic non-uniform key recovery attack

There also exist generic attacks that require fewer operations than $O(|K(n)|)$ *per problem instance* using *pre-computation*.
Note that the pre-computation must cost no less than $O(|K(n)|)$, otherwise the entire attack on just one instance would be better than exhaustive key search.
In general, we can break such attacks into two parts:

- An offline part that performs a pre-computation costing at least $O(|K(n)|)$ operations
- An online part that attacks each of the L keys independently using the pre-computed data in less than $O(|K(n)|)$ operations

Note that once the pre-computed data is known, one can write down an algorithm that includes the data in its description which would be able to recover a single key in less than $O(|K(n)|)$ operations. This means that for every block cipher there exists an algorithm that can recover a key in less than $O(|K(n)|)$.
Fortunately, or unfortunately depending on your view, we, mere humans, cannot write down such algorithm without performing the pre-computation.
These kinds of algorithms where the cost of pre-computation has been hidden by including pre-computated data already in their description are called ***non-uniform****.*

An extreme example is a codebook dictionary:

1. Offline: for a given block B, compute $C_B = E_k(B)$ for every possible key k and create a sorted (ciphertext,key)-table.
2. Online: query B and lookup C_B in the table to find k.

Using *chains* one can construct Time-Memory trade-off attacks that obtain a trade-off between the online complexity (time) and the size of the table (memory).
The idea is to create an iterative function $F: K(n) \to K(n)$ out of E that 'walks' through the key-space:

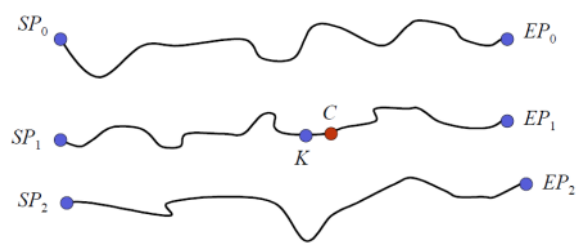
1. For a key k, first we need to obtain at least $\log_2(K(n))$ bits to be surjective, we fix blocks $B_0, B_1, \ldots$ and compute the concatenation $S = E_k(B_0)|E_k(B_1)| \ldots$
   We will assume that $K(n) = M(n)$, so we'll only need B_0.
2. We choose a fixed reduction function R that maps S into $K(n)$.

The original idea by Hellman (see http://www.cs.miami.edu/~burt/learning/Csc609.102/doc/36.pdf) was then to create a large table of m such chains, all of length t:

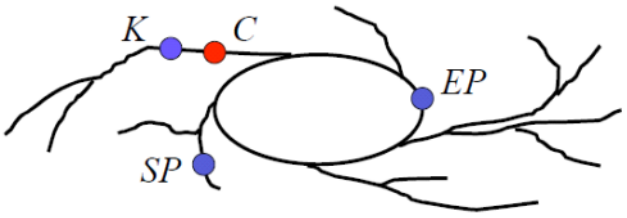
1. Pick starting points $SP_1, \ldots, SP_m$ uniformly at random from $K(n)$
2. compute $EP_i = (RE)^t(SP_i)$ for all $i = 1, \ldots, m$
3. Store (EP_i, SP_i) in a sorted table.

The attack would then proceed as follows:

1. Query $S = E_k(B_0)$ from the oracle, compute $P_0 = R(S)$
2. For $i = 0, \ldots, t-1$
   3. Lookup if P_i is an endpoint EP_j in the table
   4. If so, compute candidate key $\hat{k} = (RE)^{t-1-i}(SP_j)$, if the candidate is the sough-for key, return $\hat{k}$.
   5. Compute $P_{i+1} = (RE)(P_i)$



A graphical representation of the ideal situation with disjoint chains all of length t with start point and end point. If the unknown key k (K in the picture) is on some chain then the known P_0 (C in the picture) is the next point after k, with at most t iterations we will detect the end point EP_1. Then we can start iterating



A graphical representation of the expected situation. The iteration function (RE) behaves as a random function and has many collisions. Each collision will cause chains to merge. Even if we detect an endpoint EP, the sought-for key k might not be on the chain starting from SP, but on some other chain that merges with it. There will be a significant amount of overlap between chains,

from SP_1 to find k.	so many keys will be covered twice or more, and by extension many keys will NOT be covered. This phenomena will start once the first collision occurs which happens when $m \cdot t$ reaches size $\approx \sqrt{N}$ due to the birthday paradox (covered later on), the expected number of collisions grows quadratically in $m \cdot t$.

The attack would be successful if the sought-for key k occurred in one of the m chains and one would hope that choosing $m \cdot t = |K(n)|$ would lead to success probability 1.
Unfortunately, as (RE) behaves as a random function there will be many collisions $(RE)(CP_0) = (RE)(CP_1)$ and any two chains Q_0 and Q_1 containing such CP_0 and CP_1, respectively, will *merge* after CP_0/CP_1 and will follow the exact same walk.
Hence, the key space covered by the chains will be significantly smaller than expected.
The success probability, or the ratio of the key space covered, is close to the lower bound ($N = K(n)$):

$$P[success] \geq \frac{1}{N} \cdot \sum_{i=1}^{m} \sum_{j=0}^{t-1} \left[\frac{N - it}{N}\right]^{j+1}$$

Proof: Let A denote the set of unique keys covered by the m chains, then $P[success] = E[|A|]/N$.
Let $I\{X\}$ denote the indicater function of the *event* X, then

$$P[success] = E\left[\sum_{i=1}^{m} \sum_{j=0}^{t-1} \frac{I\{k_{ij} \text{ is new}\}}{N}\right] = \sum_{i=1}^{m} \sum_{j=0}^{t-1} \Pr[k_{ij} \text{ is new}]/N$$

Where "k_{ij} is new" denotes the event that the value of k_{ij} has not already occurred in chain i before iteration j or in any previous chain $l < i$.
By looking only at the current chain we find:

$$\begin{aligned}
&\Pr[k_{ij} \text{ is new}] \\
&\geq \Pr[k_{i0}, k_{i1}, \dots, k_{ij} \text{ are all new}] \\
&= \Pr[k_{i0} \text{ is new}] \cdot \Pr[k_{i1} \text{ is new} | k_{i0} \text{ is new}] \cdots \\
&\quad \Pr[k_{ij} \text{ is new} | k_{i0}, \dots, k_{i(j-1)} \text{ are new}] \\
&= \frac{N - |A_{i-1}|}{N} \frac{N - |A_{i-1}| - 1}{N} \cdots \frac{N - |A_{i-1}| - j}{N} \\
&\geq \left[\frac{(N - it)}{N}\right]^{j+1}
\end{aligned}$$

Where A_{i-1} denotes the set of unique keys covered by chains $l = 1, \dots, i-1$.
Clearly, each of the $(j+1)$ terms is larger than $(N - it)/N$, as $|A_{i-1}| \leq (i-1)t$.

One can approximate that for $m \cdot t^2 = N$, the success probability is about $0.80\ mt/N$.
To avoid merging to a large extent, Hellman proposed to use r tables constructed with different reduction functions R_i, more specifically he proposed to set $m = t = r = N^{\frac{1}{3}}$, such that for each individual table the success probability is $0.80mt/N = 0.80/\ N^{\frac{1}{3}}$.
Even though there will be collisions between different tables, due to the different reduction functions there will be no merging, overall we expect a success probability close to 0.80.

The total memory required is $m \cdot r = N^{\frac{2}{3}}$ (endpoint,startpoint)-pairs,
the offline complexity is $O(mtr) = O(N)$,
the online complexity is $O(rt) = O(N^{\frac{2}{3}})$.

Note that for each table it is possible that the chain starting from P_0 merges with a chain in the table, in that case we will find a *false alarm*: we find the endpoint in the table, compute the chain, but will never find the preimage to P_0 as it's not on the beginning part of chain in the table.
There are t points on the P_0-chain and mt points in the table, so the expected number F of false alarms is bounded by $\mathrm{E}[F] \leq mt \cdot t/N$.
Actually one can show a slightly tighter bound $\frac{mt(t+1)}{2N}$, see Hellman's paper.
For Hellman's parameter choices, one expects 1 false alarm per table, each costing t operations to dismiss, hence a total of $O(rt) = O(N^{\frac{2}{3}})$ operations are wasted on false alarms.

A slightly different approach is using rainbow tables. This improvement by Oechslin (see

http://lasec.epfl.ch/~oechslin/publications/crypto03.pdf) *uses, instead of r tables, for each of the t iterations a different Reduction Function. This avoids merging significantly, but the online cost increases slightly.*

5.3. Generic plaintext recovery attack

In a plaintext recovery attack the attack is tasked to decrypt a certain ciphertext block C. In a chosen plaintext or chosen ciphertext setting, the attacker is allowed to query encryptions of arbitrary plaintext blocks P or decryptions of arbitrary ciphertext blocks $C' \neq C$ different from the challenge ciphertext block C.
In the first setting, a generic attack is an exhaustive search among all plaintexts until the target ciphertext is found.
In the second setting, a generic attack is a codebook attack, where the entire codebook consisting of plaintext-ciphertext pairs for the current unknown key is created. The only entry the attacker cannot query belongs to the target ciphertext, yet he can determine the target plaintext as the only plaintext block not yet in the codebook.
Both have cost $O(|M(n)|)$.

5.4. Generic distinguishing attacks

As with the stream cipher we can define an indistinguishability notion:
a block cipher is called $(D(n), C(n), \epsilon(n))$-*indistinguishable* if for any algorithm A:

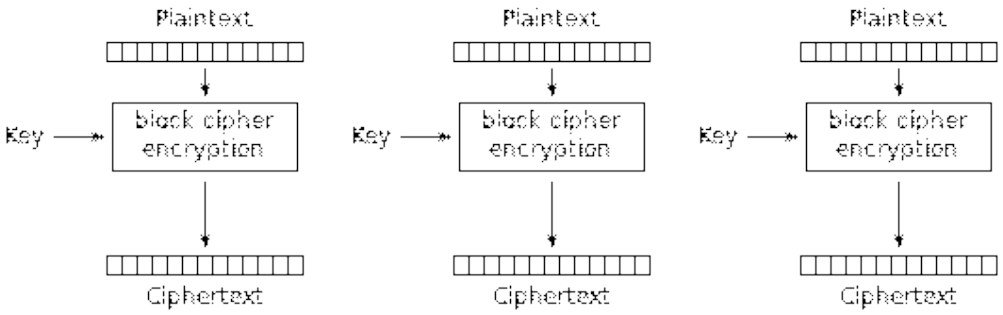
$$|\Pr[A(D(n), C(n), O_{bc}) = 1] - \Pr[A(D(n), C(n), O_{ur}) = 1]| \leq \epsilon(n)$$

where A can make $D(n)$-queries and perform $C(n)$-operations, O_{bc} is a block cipher oracle that chose a key k uniformly at random and answers encryption and decryption queries, O_{ur} is a random permutation oracle that chose a permutation π: $M(n) \to M(n)$ uniformly at random and answers encryption $\pi(\cdot)$ queries and decryption $\pi^{-1}(\cdot)$ queries.
The generic key recovery attack leads to a generic distinguishing attack, where it returns 0 if the attack returned $\perp$, and 1 otherwise, with expected complexity $O(|K(n)|)$.

6. Block cipher Modes of Operation

Using a Block cipher to encrypt a message can be done in various ways, here we will treat various modes of operation, necessarily all operations need to be reversible to be able to decrypt.
We will assume given key k and that a message M is already preprocessed with unambigious padding and split into blocks $M_1, \dots, M_l$.

6.1. ECB - Electronic Code Book



Electronic Codebook (ECB) mode encryption

This mode simply encrypts each block independently:

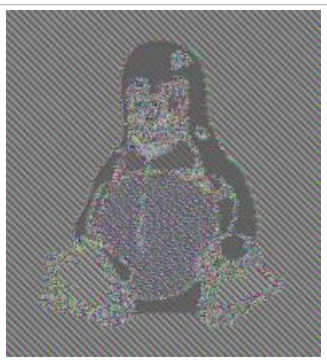
$$C_i = E_k(M_i), \qquad i = 1, \dots, l.$$

The ciphertext is simply $C_1| \dots |C_l$.

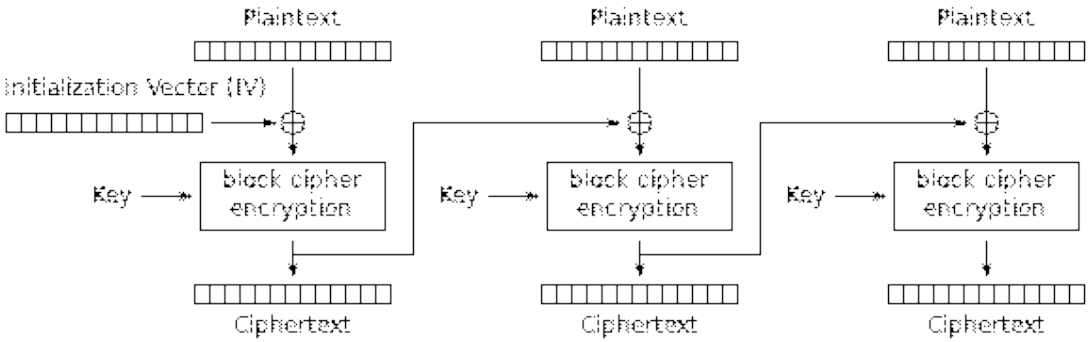
Using ECB is not recommended as there is a trivial distinguishing attack:

1. encrypt $M_1 = 0, M_2 = 0$, return 0 if $C_1 = C_2$ and 1 otherwise.

In fact, this leaks information about repetitive data in the message quite clearly:

	
Message: uncompressed image	ECB-ciphertext: repetitive 'white'-blocks clearly leak information Howto DIY encrypted image see: https://filippo.io/the-ecb-penguin/

6.2. CBC - Cipher Block Chaining



Cipher Block Chaining (CBC) mode encryption

CBC uses a random *IV (Initialization Vector*), to randomize encryptions for each encryption that has be communicated together with -- or as part of -- the ciphertext.
Thus when encrypting the same message twice, one obtains two blockwise-different ciphertexts.
Also, it uses the ciphertext in the encryption of the subsequent block to prevent the ECB-distinguishing attack.
CBC is the most commonly used mode.
Formally:

$$C_0 = IV$$
$$C_i = E_k(C_{i-1} \oplus M_i)$$

6.2.1. Padding Oracle Attack

CBC is vulnerable to a so-called *padding oracle attack* (e.g., see the recent POODLE-attack https://www.openssl.org/~bodo/ssl-poodle.pdf) that allows to recover a plaintext given a (IV,ciphertext)-pair and access to a padding oracle, i.e., an oracle that simply answers whether a given (IV,ciphertext)-pair decrypts to a plaintext that has been properly padded to a length that is a multiple of the blocksize.
Such a padding oracle may exist due to implementation flaws, e.g., distinguishing between correct and incorrect padding may be achieved via an abnormal response (error code) or a noticable difference in response time.
As the attack described below will show, it is thus important to prevent any padding oracle in implementations by ensuring no information leaks via error codes and using constant-time constant-memory code (i.e., independent of correctness of padding, decryption executes the same instructions and accesses the same memory locations).

The attack described below assumes PKCS7 padding, i.e., if $r \geq 1$ bytes need to be appended to achieve the length to be a multiple of the blocksize, then r bytes of integer value r are appended.

First note that a one-bit change to the ciphertext will cause the corresponding plaintext block to be completely different, but very importantly the only other change to the decrypted plaintext is the same one-bit change in the <u>following</u> plaintext block.

Thus one can apply arbitrary differences to the last block that contains the padding.
Let L be the blocksize in bytes, $O_{pad}: C \to \{true, false\}$ be the padding oracle, then one can recover the last plaintext block as follows:

1. For $j = 1, \dots, l$ do
    2. Set recovered last plaintext block $P_{j,recovered} = 0$
    3. For $i = 1, \dots, L$ of the last block do
        4. For byte value $b = 0, \dots, 255$ do
            5. Let block $P_{pad} = 0^{L-i}|i^i$ and block $P_{byte} = 0^{L-i}|b|0^{i-1}$
            6. If $O(C_0|\cdots\left|\left(C_{j-1} \oplus P_{j,recovered} \oplus P_{byte} \oplus P_{pad}\right)|C_j\right) = true$ then
                7. Set $P_{j,recovered} = P_{j,recovered} \oplus P_{byte}$
8. Return $P_{1,recovered}|\cdots|P_{l,recovered}$

Note that while guessing the last byte of each block there may be two possibilities:

1. we guessed correctly and the last byte is now 1, which forms a correct padding
2. the second-last plaintext byte is 2 and our guess turned the last plaintext byte in 2, and 2|2 is also a correct padding.

Continueing with the second possibility will fail quickly once all guesses fail for a byte and can then be dismissed, then one needs to go back to step 3. with $i = 1$ and now skipping the incorrect guess b.

6.2.2. Generic distinguishing attack

Despite the feed-forward of the ciphertext block into the next plaintext block there exists a generic distinguishing attack against CBC-mode with complexity $O\left(\sqrt{|K(n)|}\right)$ instead of the expected $O(|K(n)|)$ using a generic block cipher key recovery attack.
Namely, when one feeds a very large plaintext consisting of $l = 4\sqrt{|K(n)|}$ blocks B, for some fixed value B, the ciphertext blocks are:

$C_0 = IV \xleftarrow{r} M(n)$
$C_i = E_k(C_{i-1} \oplus B) =: F(C_{i-1})$

The function F is a *permutation* as both $E_k(\cdot)$ and $(\cdot \oplus B)$ are permutations, thus there can be no collisions $F(C) = F(C')$ with $C \neq C'$.
which implies that either:

1. all ciphertext blocks are different
2. or we encounter the preimage to C_0: $C_k = C_0$ for some k, then we find a loop: $C_i = C_{i-k}$ for all $i \geq k$

In contrast, if $C_0, C_1, \dots, C_l$ were produced by a random oracle then they are independently selected uniformly at random, the probability that they are all unique is:

$$\Pr\left[C_i \neq C_j \text{ for all } 0 \leq i < j \leq l\right] = 1 \cdot \frac{N-1}{N} \cdot \frac{N-2}{N} \cdots \frac{(N-l)}{N} = 1 \cdot \left(1 - \frac{1}{N}\right) \cdots \left(1 - \frac{l}{N}\right)$$

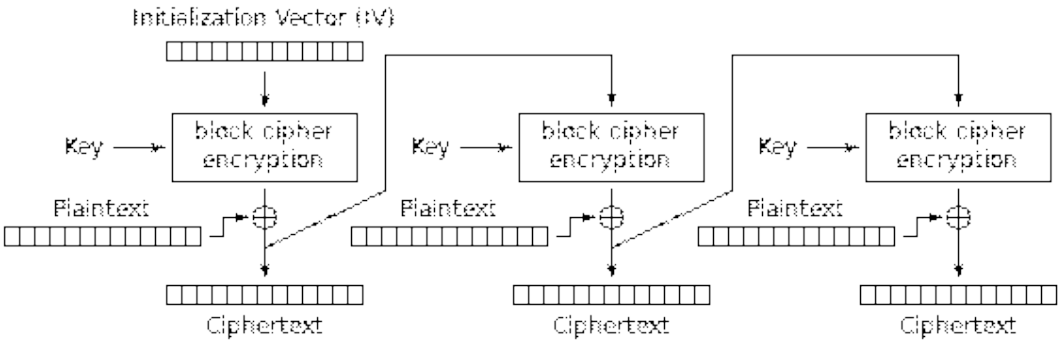
$$\approx 1 \cdot e^{-\frac{1}{N}} \cdot e^{-\frac{2}{N}} \cdots e^{-\frac{l}{N}} = e^{-\frac{1+2+\cdots+l}{N}} = e^{\frac{-l(l+1)}{2N}} = e^{\frac{-4\sqrt{N}\cdot\left(4\sqrt{N}+1\right)}{2N}} \approx e^{-\left(\frac{16}{2}\right)} = e^{-8} \approx 0.00033$$

(using the Taylor series approximation $e^x = 1 + x + \frac{x^2}{2!} + \cdots$)

This is also called the birthday paradox: namely that out of a set of size N one only needs to select approx. $\sqrt{N}$ samples uniformly at random to find a collision, thus in a group of 23 people one can expect there are 2 people with the same birthday.

Thus with overwhelming probablity there will be at least two same ciphertext blocks $C_i = C_j$ with $i \neq j$ and with overwhelming probability $i \neq 0$ and $j \neq 0$.
Hence, one can distinguish between a block cipher in CBC-mode and a random oracle by querying a message of $l = 4\sqrt{N}$ blocks B and returning 1 if there are two same ciphertext blocks $C_i = C_j$ $0 < i < j$, and returning 0 otherwise.

6.3. CFB - Cipher FeedBack



Cipher Feedback (CFB) mode encryption

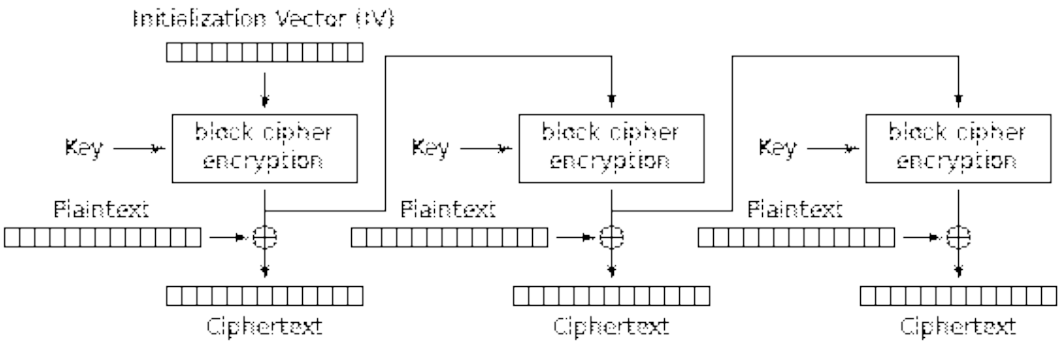
CFB, like most modes, also uses an IV, but here the input to the blockcipher is the previous ciphertext or the IV and, similar to stream ciphers, the plaintext is added to the output of the blockcipher.
Formally:

$$C_0 = IV \xleftarrow{r} M(n)$$
$$C_i = M_i \oplus E_k(C_{i-1})$$

If padding is used and a padding oracle exists then also CFB is vulnerable to a padding oracle attack similar to the one against CBC, but instead of modifying ciphertext block C_{i-1} one needs to modify C_i.
Note however, that padding is not necessary for CFB and thus most commonly not used: one can simply truncate the output of the last block cipher invocation to match the plaintext size.

CFB is also vulnerable to the same $O(\sqrt{N})$ distinguishing attack as CBC.

6.4. OFB - Output FeedBack

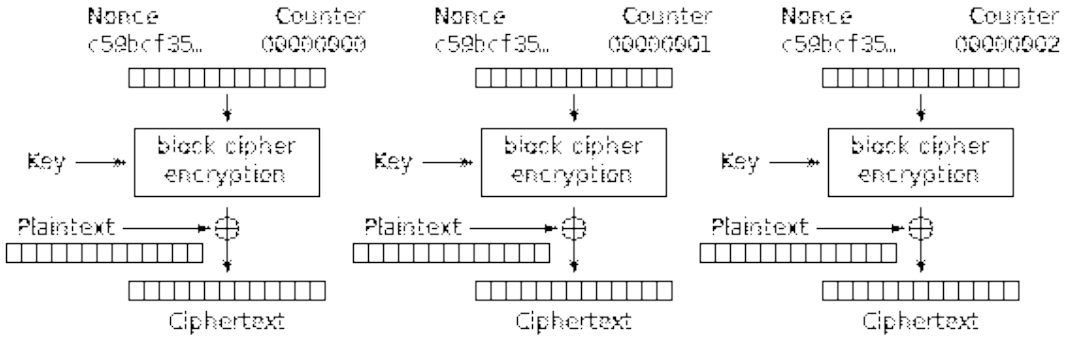


Output Feedback (OFB) mode encryption

OFB operates like a stream cipher: using an IV it generates a keystream independently of the message which are then added to each other.

Like CFB, OFB is theoretically also vulnerable to a padding oracle, but most commonly no padding is used.
And also OFB is vulnerable to the same $O(\sqrt{N})$ distinguishing attack as CBC.

6.5. CTR - Counter Mode



Counter (CTR) mode encryption

Counter Mode is another mode that operates like a stream cipher, however now all keystream blocks are generated independently by combining the IV (here called *nonce*) with a unique counter for each block.

Like CFB and OFB, CTR is theoretically vulnerable to a padding oracle, but most commonly no padding is used.
And also CTR is vulnerable to a $O(\sqrt{N})$ distinguishing attack as CBC, however for fixed plaintext block B, all ciphertext blocks C_i are all unique, i.e., no loop is possible.

6.6. Other Modes

See NIST for other modes: http://csrc.nist.gov/groups/ST/toolkit/BCM/modes_development.html

7. Cryptographic hash functions

Cryptographic hash functions map messages of arbitrary size to a fixed size *hash*, e.g. a bitstring of length 256.
For cryptographic purposes we distinguish several security properties for families $\mathcal{F}$ of hash functions with the same range $\mathcal{H}$ (see https://eprint.iacr.org/2004/035):

- *Pre (pre-image resistance)*: Given $f \xleftarrow{r} \mathcal{F}, h \xleftarrow{r} \mathcal{H}$, no algorithm A that returns $m = A(f, h)$ such that $f(m) = h$ is faster than an exhaustive search.
- *ePre (everywhere pre-image resistance):* $h \leftarrow A$, $f \xleftarrow{r} \mathcal{F}$, no algorithm A that returns $m = A(f, h)$ such that $f(m) = h$ is faster than an exhaustive search.
- *aPre (always pre-image resistance):* $f \leftarrow A, h \xleftarrow{r} \mathcal{H}$, no algorithm A that returns $m = A(f, h)$ such that $f(m) = h$ is faster than an exhaustive search.
- *Sec (second pre-image resistance) [n]*: Given $f \xleftarrow{r} \mathcal{F}, m \xleftarrow{r} \{0,1\}^{\leq n}$, no algorithm A that returns $m' = A(f, m) \neq m$ such that $f(m) = f(m')$ is faster than an exhaustive search.
- *eSec (everywhere second pre-image resistance):* $m \leftarrow A$, $f \xleftarrow{r} \mathcal{F}$, no algorithm A that returns $m' = A(f, m) \neq m$ such that $f(m) = f(m')$ is faster than an exhaustive search.
- *aSec (always second pre-image resistance) [n]:* $f \leftarrow A, m \xleftarrow{r} \{0,1\}^{\leq n}$, no algorithm A that returns $m' = A(f, m) \neq m$ such that $f(m) = f(m')$ is faster than an exhaustive search.
- *Coll (collision resistance):* $f \xleftarrow{r} \mathcal{F}$, no algorithm A that returns $(m, m') = A(f)$ with $f(m) = f(m'), m \neq m'$ is faster than a generic collision attack

The most widely used hash functions are SHA-1, SHA-2-256 and SHA-2-512, whereas SHA-3 is the future standard. SHA-1 is not collision resistant, its late predecessor and prior de facto standard MD5 is very weak.
All of these hash function standards are fixed hash functions and there is no family $\mathcal{F}$ to speak of, hence of the above security properties only aPre and aSec apply.
Note that for fixed hash functions there is no formal definition of collision resistance, in fact this seems to be fundamentaly impossible due to the *Foundations-of-Hashing dilemma* (see https://eprint.iacr.org/2006/281):

Any hash function's range is smaller than the domain, thus the pigeon-hole principle states that there exist collisions.
For a fixed hash function f consider some message space $\mathcal{M}$ with $|\mathcal{M}| > |\mathcal{H}|$ and the family of algorithms $\mathcal{A} = \{A_{m,m'} | m, m' \in \mathcal{M}\}$ consisting of trivial algorithms $A_{m,m'}$ that simply print a pair (m, m').
Given the pigeon hole principle, there exist collisions $f(m) = f(m'), m \neq m', m, m' \in \mathcal{M}$, thus inside $\mathcal{A}$ there exist trivial algorithms that return a collision for the fixed hash function.

7.1. Generic (second) pre-image attack

Given a hash function f ($\xleftarrow{r}\mathcal{F}$) and hash h ($\xleftarrow{r}\mathcal{H}$ or $= f(m)$).
Let $\mathcal{M}$ be a finite message space with $|\mathcal{M}| \gg |\mathcal{H}|$.
Then one can find a (second) pre-image using a brute force search:

1. while (true)
2. $m' \xleftarrow{r} \mathcal{M}, h' = f(m')$
3. if $h' = h$ return m'

Assuming a geometric distribution, i.e., a success probability of $1/|\mathcal{H}|$ in each iteration independent of other iterations, one expects to need $|\mathcal{H}|$ tries to succeed.

7.2. Generic low-entropy pre-image attack

Hash functions are often used to store passwords more securely by storing their hash, a given password can easily be verified against the hash.
Yet from the hash it is hard to recover the password.
However, passwords have notoriously low entropy.
Like with block ciphers one can use Time-Memory trade off attacks using Hellman's tables or Rainbow tables to precompute table(s) of chains covering a large set of passwords.
Given such tables one can recover passwords (covered by the table) with high probablity very fast.
In fact, this is the foremost practical use for this type of attack, where Rainbow tables offer some practical advantages over Hellman's tables.

To prevent this kind of practical attack to recover passwords, the proper way to store passwords is using salted hashing *(or* keyed hashing*), e.g., for each password p one selects a* salt $s \in \{0,1\}^k$ *and stores* $(s, Hash(s|p))$*. Now the attacker needs either a separate Rainbow table for each salt or a significantly bigger Rainbow table to cover pairs of (password,salt). Hence if k is large enough this will not be practical anymore compared to a simple brute force search. Note that a salt has almost no impact on a brute force search. Therefore there are special hashing algorithms that have been designed to be fast enough for a password verification, but slow enough to cause a significant factor increase in resources required for a brute force search. See also* https://password-hashing.net/

7.3. Generic collision attack

As we showed above, for a pseudo-random function with range $\mathcal{H}$ there is a high probability to see a collision among the images of $O(\sqrt{|\mathcal{H}|})$ uniformly randomly selected inputs.
To be more specific, we can compute the expected number X of inputs to find a collision, let $N = |\mathcal{H}|$:

$$E[X] = \sum_{k=1}^{\infty} k \Pr[X = k] = \sum_{k=1}^{\infty} k(\Pr[X > k-1] - \Pr[X > k]) = \sum_{k=0}^{\infty} \Pr[X > k]$$

$$(1\,(\Pr[X > 0] - \Pr[X > 1])) + (2\,(\Pr[X > 1] - \Pr[X > 2])) + \cdots$$

$$1\Pr[X > 0] - 1\Pr[X > 1] + 2\Pr[X > 1] - 2\Pr[X > 2] + 3\Pr[X > 2]$$

$$- 3\Pr[X > 3] \cdots$$

$$1\Pr[X > 0] + 1\Pr[X > 1] + 1\Pr[X > 2] + \cdots$$

Earlier we found that $\Pr[X > k] = \Pr[\text{no collision occurred within } k \text{ samples}] \approx e^{\frac{-k(k-1)}{2N}} \approx e^{-\frac{k^2}{2N}}$.

$$E[X] \approx \sum_{k=0}^{\infty} e^{-\frac{k^2}{2N}} \approx \int_0^{\infty} e^{-\frac{x^2}{2N}} dx = \sqrt{\pi N/2}.$$

A direct approach would be to compute images $h_i = f(m_i)$ for $m_i \xleftarrow{r} \mathcal{M}$ and store them until one finds a collision.
Unfortunately this would require $O(\sqrt{N})$ memory.

One can do significantly better using chains and *distinguished points*,
(see e.g., http://people.scs.carleton.ca/~paulv/papers/JoC97.pdf).
Let $\mathcal{S} \subset \mathcal{H}$ be some subset of hashes that are easily distinguished, e.g., the last l-bits are zero, we call $\mathcal{S}$ the set of distinguished points.
Then we can find collisions using the following procedure:

1. Let $T = \{\}$
2. while (true)
3. $SP \xleftarrow{r} \mathcal{M}, P = SP, l = 0$
4. while $(P \notin \mathcal{S})$
5. $P = f(P), l = l + 1$
6. For every $(SP', EP', l') \in T$ with $EP' = P$ do
7. $X = SP, r = l, X' = SP', r' = l'$
8. while $(r' > r)$
9. $X' = f(X'), r' = r' - 1$
10. while $(r > r')$
11. $X = f(X), r = r - 1$
12. while $(f(X) \neq f(X'))$
13. $X = f(X), X' = f(X')$
14. if $f(X) = f(X')$ and $X' \neq X$ return (X, X')
15. $T = T \cup (SP, P, l)$

Steps 3-5,15 produce new chains and add them to the table T.
Steps 6-14 check whether the distinguished endpoint already occurred in the table T and recomputes both chains until a collision point is found.
The average chain length is $t = \frac{|\mathcal{H}|}{|\mathcal{S}|}$, thus we can expect to store $\sqrt{\pi N/2}\ /\ t$ chains.
After there occurs a collision in step 5. with a previously computed point, it still takes t iterations on average to find a distinguished point, so the expected overall complexity to detect a collision is $\sqrt{\pi N/2}$+t function calls.
To compute the collision point itself also requires expectedly an $O(t)$ function calls.

However, there also exists the possibility that a chain never ends in a distinguished point when it enters a loop, to resolve this issue we can add the following step:

5a. If $l > 20t$ then return to step 3.

The probability that a chain's length exceeds $20t$ is $\left(1 - \frac{|\mathcal{S}|}{|\mathcal{H}|}\right)^{20t} = \left(1 - \frac{1}{t}\right)^{20t} \approx \left(e^{-1/t}\right)^{20t} = e^{-20}$, and we will waste 20 times more than average on such chains, so the ratio of wasted work is estimated by $20\ e^{-20} \approx 5 \times 10^{-8}$.
The probability that a chain enters a loop within $20t$ iterations, i.e., an internal collision, is approximately $1 - e^{-\frac{(20t)^2}{2N}}$.
Hence if $(20t)^2 \ll 2N$, i.e., $t^2 \ll N$ or $|\mathcal{S}| \gg \sqrt{N}$ then this probability is negligible.