Selected Areas in Cryptology
MasterMath – Spring 2021
Part II - Cryptanalysis

dr.ir. Marc Stevens
https://homepages.cwi.nl/~stevens/mastermath/2021

April 12, 2021

Contents

1 Introduction: Cryptanalysis

The topic of the first 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 *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.

These lecture notes will cover symmetric primitives:

- Stream ciphers
- Block ciphers

- Modes of operations on Block ciphers
- Cryptographic hash functions
- Message Authentication Codes (MAC)

We will first treat generic attacks. 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.

Security of cryptographic primitives can be analyzed in three notions:

- *Information-theoretic security*, also called perfect security, using adversaries with *infinity* computational resources;
- *Asymptotic computational security*, using adversaries that can perform only *polynomial time* computations where security relies on asymptotic computationally hard problems;
- *Real world security*, using adversaries that can perform only a realistic number of computations where security relies on real world computational resource bounds.

1.1 Notation

We will use the following style in these lecture notes:

- lower capitalized names for variables taking values in $\mathbb{Z}$, e.g., interval variables i, j, number of key bits k, number of message bits n, number of message blocks l;
- upper capitalized names for other variables, e.g., the key K, plaintext P, message M, ciphertext C, algorithm A;
- calligraphic names for sets and oracles, e.g., key space $\mathcal{K}$, message (block) space $\mathcal{M}$, oracle $\mathcal{O}$;
- selecting a value X uniformly at random from the set $\mathcal{X}$ is denoted as $X \xleftarrow{r} \mathcal{X}$;
- $\Pr[event]$ denotes the probability of *event*, $E[X]$ denotes the expected value of the random variable X;
- $poly(n)$ is the set of polynomial functions $f(n)$ in n;
- $negl(n)$ is the set of *negligible functions* $g(n)$ in n, i.e., functions $g(n)$ that go faster to 0 than any polynomial $f(n)$ when n goes to infinity: $\forall f(n) \in poly(n) : \lim_{n\to\infty} f(n)g(n) \to 0$.

1.2 Exercises

The webpage for this course half will contain exercises to practice examined course material. These are recommended. For verification, solutions can be submitted to marc AT marc-stevens DOT nl.

1.3 Challenges

The webpage for this course 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 marc AT marc-stevens DOT nl.

2 One Time Pad

The one-time-pad (OTP) 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* l and outputs the ciphertext $C = M \oplus K$ 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:

$$\Pr[C = M_1 \oplus K] = \Pr[K = C \oplus M_1] = 2^{-l} = \Pr[K = C \oplus M_2] = \Pr[C = M_2 \oplus K].$$

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: $M' = C' \oplus K = D \oplus C \oplus K = M \oplus D$.

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. Then 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.

3 Stream ciphers

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:

- Initialization: generate large state from input small key K:

$$S_0 \leftarrow \mathbf{Init}(K)$$

- Iteration $i = 0, \ldots, |M| - 1$:
  1. first do state update & produce key stream symbol:

$$(S_{i+1}, O_i) \leftarrow \mathbf{Update}(S_i)$$

  2. then encryption:

$$C_i \leftarrow O_i \oplus M_i$$

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:

$$(S_{i+1}, O_i) \leftarrow \mathbf{Update}(S_i, C_{i-1}, \ldots, C_{i-l}).$$

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

3.1 Security

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

- *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*: the output keystream is indistinguishable from a uniform randomly chosen stream

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.

3.1.1 Indistinguishability definition

Let k be the input key size in bits, $d(k)$ be an upper bound on the generated output keystream length, $c(k)$ be the upper bound on the number of operations. We denote by $A^{(\mathcal{O},d(k),c(k))}$ an algorithm with access to a keystream oracle $\mathcal{O}$ that reads a stream of size $\leq d(k)$ from Oracle $\mathcal{O}$ and performs at most $c(k)$ operations and returns either 0 or 1.

We can define two oracles. Firstly, the oracle $\mathcal{O}_{sc}$ that first selects a key $K \xleftarrow{r} \{0,1\}^k$ selected uniformly at random and returns the output key stream of the stream cipher under key K. Secondly, the oracle $\mathcal{O}_{ur}$ that returns a stream of bits selected independently and uniformly at random. So we can call a stream cipher $(d(k), c(k), \epsilon(k))$-*indistinguishable* if *for any algorithm* A:

$$\left|\Pr[A^{(\mathcal{O}_{sc},d(k),c(k))} = 1] - \Pr[A^{(\mathcal{O}_{ur},d(k),c(k))} = 1]\right| \leq \epsilon(k).$$

For a stream cipher with fixed input key size k and state size l, one can require practical parameters, e.g., $d(k) = c(k) = 2^{\min(k,l)}$, $\epsilon(k) = 2^{-80}$.

For a stream cipher with variable input key size k, one can also define security against *efficient*, i.e., probabilistic polynomial time (PPT), attackers: With slight abuse of notation we set $d(k) = poly(k)$, $c(k) = poly(k)$, $\epsilon(k) = negl(k)$ to define security against adversaries that read only a polynomial (in k) size output stream and perform a polynomial number of operations which are successful with a negligible (in k) success probability.

Note that *information-theoretical security* with $c(k) = \infty$ is unattainable.

3.2 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 $O(2^k)$ operations and succeeds with probability 1 if the observed key stream is sufficiently long to dismiss all bad candidate keys.

3.3 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.

3.4 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 , and 1 otherwise. The security of the stream cipher is thus upper bounded by $O(2^{\min(k,l)})$.

3.5 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 $M' = M \oplus D$ by adding the difference to the cipherstream $C' = C \oplus D$ between sender and receiver, as then the receiver decodes:

$$C' \oplus K = C \oplus D \oplus K = M \oplus K \oplus D \oplus K = M \oplus D.$$

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.

3.6 Key reuse

Note that when a key k is reused to encrypt 2 messages M and M' with the same keystream K then the difference is leaked via $C \oplus C' = M \oplus K \oplus M' \oplus K = M \oplus M'$. Hence it is important to ensure that a different key is used for every message. To achieve this one may split up the full key into a private key part and a public *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.

4 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 $\mathcal{K}(n)$ (e.g, $\{0,1\}^n$) and the block space $\mathcal{M}(n)$ (e.g., $\{0,1\}^n$).

Then a Block Cipher is a mapping $Enc : \mathcal{K}(n) \times \mathcal{M}(n) \to \mathcal{M}(n)$ with the constraint that for every $K \in \mathcal{K}(n)$: $Enc_K = Enc(K, \cdot) : \mathcal{M}(n) \to \mathcal{M}(n)$ is a permutation and its inverse $Dec_K = Enc_K{-}1$ is efficiently computable. For easy of exposition, we'll only consider Block Cipher designs with fixed n, key space $\mathcal{K} = \mathcal{K}(n)$ and block space $\mathcal{M} = \mathcal{M}(n)$.

4.1 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 pairs for plaintexts 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 $(P_1, C_1), \ldots, (P_l, C_l)$. Then it simply goes through all possible key candidates K, it returns the first candidate key K that satisfies $C_i = Enc_K(P_i)$ for $i = 1, \ldots, l$, or $\perp$ otherwise. It requires $O(|\mathcal{K}|)$ operations and succeeds with almost certainty (there is a probability that another key is found first that satisfies the pairs, but this probability decreases exponentially in the size of the list l).

4.2 Generic 1-out-of-L key recovery attack

Now consider a variation wherein the attacker has access to L block cipher oracles, each with its own secret key, and the attackers 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 $O(|\mathcal{K}|/L)$ operations.

The attacker selects a few plaintexts $P_1, \ldots, P_l$ and queries these for every oracle and obtains ciphertexts $C_{1,1}, \ldots, C_{l,L}$. Then it simply goes through all possible key candidates K, it returns the first candidate key K that satisfies $C_{i,j} = Enc_K(P_{i,j})$ for some $j = 1, \ldots, L$ for all $i = 1, \ldots, l$, or $\perp$ otherwise.

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

There also exist generic attacks that require fewer operations than $O(|\mathcal{K}|)$ *per problem instance* using *pre-computation*. Note that the pre-computation must cost no less than $O(|\mathcal{K}|)$, 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:

1. An offline part that performs a pre-computation costing at least $O(|\mathcal{K}|)$ operations;
2. An online part that attacks each of the L keys independently using the pre-computed data in less than $O(|\mathcal{K}|)$ operations.

An extreme example of such an attack is a codebook dictionary:

1. Offline: for a given block B, compute $C_K = Enc_K(B)$ for every possible key K and create a sorted (C_K, K)-table.
2. Online: query B to get a ciphertext C and lookup the entry (C_K, K) in the table with $C_K = C$.

Note that the online algorithm uses $O(|\mathcal{K}|)$ memory and $O(\log |\mathcal{K}|)$ time (or $O(1)$ time if using hash tables).

> Note that once the pre-computed data is known, one can write down an algorithm that includes the data in its description. This algorithm would be able to recover a single key in less than $O(|\mathcal{K}|)$ operations. This means that for every block cipher there exists an algorithm that can recover a key in less than $O(|\mathcal{K}|)$ operations. 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*.

4.3.1 Hellman's Time-Memory Trade Off attack

An algorithm that uses less memory, but more computational time, is Hellman's Time-Memory Trade Off attack[1]. This attack uses an iterative function that 'walks' through the key-space to

[1] `http://www.cs.miami.edu/home/burt/learning/Csc609.102/doc/36.pdf`

create a table of many chains or walks. One can trade-off between the online complexity (time) and the size of the table (memory).

We assume $\mathcal{M} = \mathcal{K}$ so we only need one message block B to create an iterative function $F : \mathcal{K} \to \mathcal{K}$ out of Enc that walks through the key-space. Let $R : \mathcal{M} \to \mathcal{K}$ be a fixed bijective reduction function, then we define $F(K) = R(Enc_K(B))$.

The main idea is to find the unknown key $\widehat{K}$ for a given ciphertextblock $C = Enc_{\widehat{K}}(B)$ by comparing the outcome $F^j(R(C))$ of repeat application of F with stored walks $(k_i, F^t(k_i))$ in a table. We want all walks to cover essentially the entire key space, so that $\widehat{K} = F^{j'}(k_i)$ for some i and $j' < t$. As $R(C) = F(\widehat{K})$, it follows that then $F^{t-j'-1}(R(C)) = F^{t-j'}(\widehat{K}) = F^t(k_i)$. So whenever $F^j(R(C))$ equals an stored walk endpoint $F^t(k_i)$ for some i and $j < t$ we have potentially found a walk that contains the key. When additionally $R(C)$ is actually in the walk from k_i to $F^t(k_i)$, i.e., $R(C) = F^{t-j}(k_i)$, then $\widetilde{K} = F^{t-j-1}(k_i)$ satisfies $Enc_{\widetilde{K}}(B) = C$ and is thus a key candidate for the sought $\widehat{K}$.

Because we expect F to behave as a random function, there should exist many collisions $F(x) = F(y)$ for $x \neq y$. And thus many collisions $F^t(x) = F^t(y)$ for $x \neq y$: the walks from x and y have merged. This implies that we must deal with false positives: $F^j(R(C))$ matches an endpoint, but is not actually in the walk. Moreover, these merges make it hard to effectively cover the entire key space with a single table based on a single F. That's why the following algorithmic description of the attack generates several tables, each using a different bijective reduction function R.

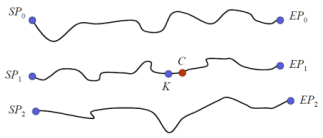
Offline attack: we use F to create a large table of m chains, all of length t:

1. Pick distinct starting points $SP_1, \ldots, SP_m$ uniformly at random from $\mathcal{K}$;
2. Compute $EP_i = F^t(SP_i)$ for all $i = 1, \ldots, m$;
3. Store (EP_i, SP_i) in a sorted table.

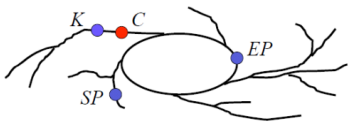
Online attack given $C = Enc_{\hat{K}}(B)$ for some unknown key $\hat{K}$:

1. $P_0 = R(C) (= F(\hat{K}))$
2. For $i = 0, \ldots, t-1$:
   2.1. If $P_i = EP_j$ for some j then let $\tilde{K} = F^{t-i-1}(SP_j)$, if $Enc_{\tilde{K}}(B) = C$ then return $\tilde{K}$
   2.2. Compute $P_{i+1} = F(P_i)$
3. Return $\perp$

Below we analyze the complexity of this algorithm.



Ideally, the chains (all of length t) would be disjoint as depicted above. If the unknown key K 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 from SP_1 to find K.



However, the iteration function ($F = R \cdot Enc$) behaves as a random function and has many collisions as depicted above. 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, 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 $\approx \sqrt{|\mathcal{K}|}$ 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 $m \cdot t = |\mathcal{K}|$ would lead to success probability 1. Unfortunately, as F behaves as a random function there will be many collisions $F(P_1) = F(P_2)$ and any two chains Q_1 and Q_2 containing such P_1 and P_2, respectively, will *merge* after P_1/P_2 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 following lower bound (here $N = |\mathcal{K}|$):

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

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

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

Where "k_{ij}" denotes the event that the value of k_{ij} of chain i at iteration j 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:

$$\begin{aligned}
&\Pr[k_{ij} \text{ is new}] \\
&\quad \geq \quad \Pr[k_{i0}, k_{i1}, \ldots, k_{ij} \text{ are all new}] \\
&\quad = \quad \Pr[k_{i0} \text{ is new}] \cdot \Pr[k_{i1} \text{ is new} \mid k_{i0} \text{ is new}] \cdots \\
&\qquad \cdots \Pr[k_{ij} \text{ is new} \mid k_{i0}, \ldots, k_{i(j-1)} \text{ are new}] \\
&\quad = \quad \frac{N - |\mathcal{A}_{i-1}|}{N} \cdot \frac{N - |\mathcal{A}_{i-1}| - 1}{N} \cdots \frac{N - |\mathcal{A}_{i-1}| - j}{N} \\
&\quad \geq \quad \left(\frac{N - i \cdot t}{N} \right)^{j+1}.
\end{aligned}$$

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

One can approximate that for $m \cdot t^2 = N$, the success probability is about $0.80mt/N$, see Hellman's paper. To avoid merging to a large extent, Hellman proposed to use r tables constructed with different reduction functions R_s, more specifically he proposed to set $m = t = r = N^{1/3}$, such that for each individual table the success probability is $0.80mt/N = 0.80N^{-1/3}$. Even though there will be collisions between different tables, due to the different reduction functions there will be no merging. Thus overall we expect a success probability close to 0.80. The total memory required is $m \cdot r = N^{2/3}$ (endpoint,startpoint)-pairs, the offline complexity is $O(mtr) = O(N)$, the online complexity, excluding the cost evaluating candidate keys, is $O(rt) = O(N^{2/3})$.

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, with each independent pair of points having probability $\frac{1}{N}$ to collide. So the expected number Z of false alarms is bounded

by $E[Z] \leq mt \cdot t \cdot \frac{1}{N}$. Actually one can show a slightly tighter bound $mt(t+1)/2N$, see Hellman's paper. For Hellman's parameter choices, one expects 1 false alarm per table, each costing at most t operations to dismiss, hence a total of $O(rt) = O(N^{2/3})$ operations are wasted on evaluating false key candidates.

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.

4.4 Generic distinguishing attack

As with stream ciphers we can define an indistinguishability notion: a block cipher is called $(d(n), c(n), \epsilon(n))$-*indistinguishable* if for any algorithm $A^{(\mathcal{O}, d(n), c(n))}$:

$$|\Pr[A^{(\mathcal{O}_{bc}, d(n), c(n))} = 1] - \Pr[A^{(\mathcal{O}_{ur}, d(n), c(n))} = 1]| \leq \epsilon(n).$$

where A can make $d(n)$-queries and perform $c(n)$-operations, $\mathcal{O}_{bc}$ is a block cipher oracle that chose a key uniformly at random and answers encryption and decryption queries, $\mathcal{O}_{ur}$ is a random permutation oracle that chose a permutation $\pi : \mathcal{M}(n) \to \mathcal{M}(n)$ 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 $O(|\mathcal{K}(n)|)$.

5 Block cipher Modes of Operation

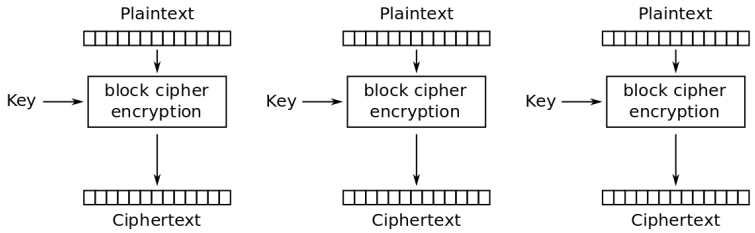
Let $Enc : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a block cipher with k-bit keys and n-bit message blocks. To use block cipher *Enc* to encrypt an arbitrary length message can be done in various ways, here we will treat various modes of operation, necessarily all operations need to be reversible 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 $M_1, \ldots, M_l$.

5.1 ECB - Electronic Code Book

This mode simply encrypts each block independently:

$$C_i = Enc_k(M_i), \quad i = 1, \ldots, l.$$

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



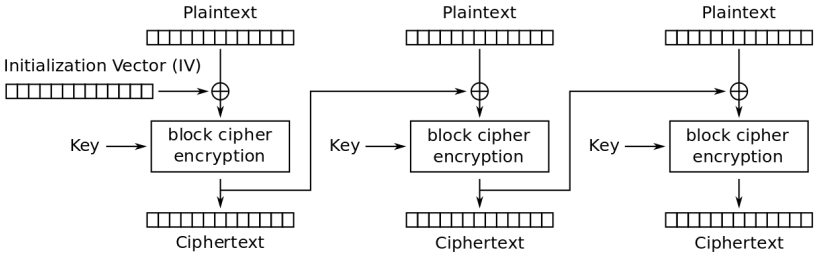
Electronic Codebook (ECB) mode encryption

5.2 CBC - Cipher Block Chaining

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:

$$C_0 = IV, \quad C_i = Enc_k(C_i \oplus M_i).$$

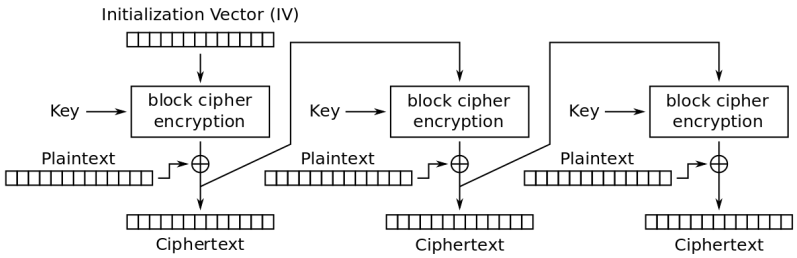


Cipher Block Chaining (CBC) mode encryption

5.3 CFB - Cipher Feedback

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:

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

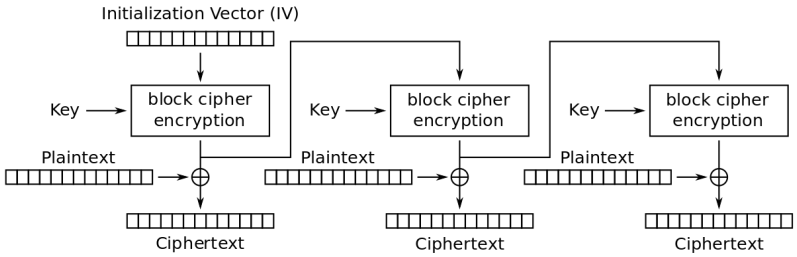


Cipher Feedback (CFB) mode encryption

5.4 OFB - Output FeedBack

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

$$O_0 = IV \xleftarrow{r} M(n), \quad O_i = Enc_k(O_{i-1}), \quad C_i = M_i \oplus O_i.$$

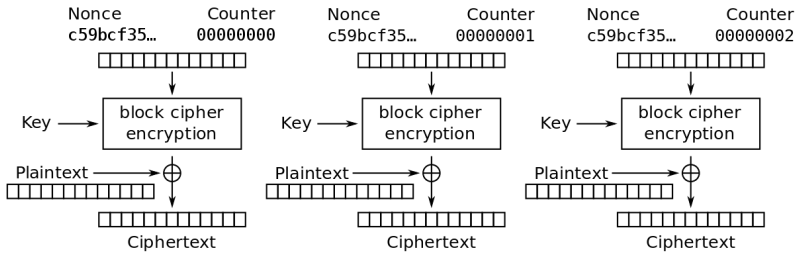


Output Feedback (OFB) mode encryption

5.5 CTR - Counter Mode

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

$$O_i = Enc_k(nonce||i), \quad C_i = M_i \oplus O_i.$$



Counter (CTR) mode encryption

5.6 Other Modes

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

5.7 Generic attacks

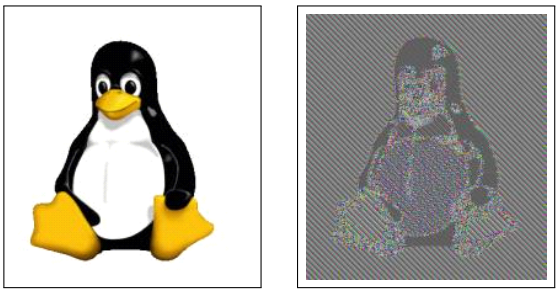
As with stream ciphers and block ciphers there exist generic key recovery attacks with complexity $O(2^k)$ by exhaustive search. However, for all these modes there exist distinguishing attacks independent of the used block cipher with complexity $O(2^{n/2})$ or less. CBC mode has a particular vulnerability because it is a mode that requires padding and a simple change in the ciphertext causes a simple change in the decrypted plaintext. These attacks are described in the following sections.

Mode	Generic key recovery	Generic distinguishing	Padding oracle attacks
ECB	$O(2^k)$	$O(1)$	N/A
CBC	$O(2^k)$	$(O(2^{n/2})$	$O(256 \cdot (n/8) \cdot l)$
CFB	$O(2^k)$	$(O(2^{n/2})$	only if padding used
OFB	$O(2^k)$	$(O(2^{n/2})$	only if padding used
CTR	$O(2^k)$	$(O(\min(2^c, 2^{n/2}))$	only if padding used

5.8 Trivial ECB distinguishing attack

Using ECB is not recommended as there is a trivial distinguishing attack. Encrypt $M = 0||0$, i.e., $M_1 = 0$, $M_2 = 0$, return 0 if $C_1 = C_2$ and 1 otherwise. In fact this mode leaks information about repetitive data in the message quite clearly[2]:

[2] `https://blog.filippo.io/the-ecb-penguin/`



5.9 $O(2^{n/2})$ distinguishing attacks

This section illustrates a distinguishing attack against CBC with complexity $O(\sqrt{|\mathcal{M}|}) = O(\sqrt{2^n})$ instead of the expected $O(|\mathcal{K}|) = O(2^k)$ using a generic block cipher key recovery attack. This shows that *long* encrypted plaintexts are distinguishable from random bitstrings. This attack can be modified to work against modes CFB, OFB, and CTR as well.

Let us denote $C = CBCEnc_K(P)$ as the ciphertext C resulting from encrypting P using CBC and key K. We assume the ciphertext bit length only depends on the plaintext bit length, so let $\ell : \mathbb{N} \to \mathbb{N}$ be the ciphertext bit length function: $\ell(|P|) = |CBCEnc_K(P)|$ for all P and K.

Formally we first have to define a similar, but ideal, encryption primitive *IEnc*. There are only 2 requirements for this ideal primitive:

- For any $P \in \{0,1\}^*$ the corresponding ciphertext has the same length $\ell(|P|) = |IEnc(P)|$ as for CBC;
- The encryption function is injective: $IEnc(P_1) = IEnc(P_2) \Leftrightarrow P_1 = P_2$. This ensures every valid ciphertext has an unique preimage (plaintext);

To ensure ciphertexts offer no information other than the unavoidable implied information about the plaintext length, we would like to choose *IEnc* uniformly at random from all functions $f : \{0,1\}^* \to \{0,1\}^*$ that fulfill these requirements. However this cannot be done directly, due to the issue of uniformly at random sampling from an infinite set. Therefore we partition the plaintext input domain of *IEnc* based on the resulting ciphertext length into finite subdomains. More specifically for every possible ciphertext length $t \in \ell(\mathbb{N})$, we define the *finite* set of plaintexts with ciphertexts of length t is $\mathcal{P}_t = \{P \in \{0,1\}^* \mid \ell(|P|) = t\}$ and choose a function $IEnc_t$ uniformly at random from the set of all injective functions $f : \mathcal{P}_t \to \{0,1\}^t$.

It should be easily verifiable that the implied encryption primitive $IEnc(P) = IEnc_{\ell(|P|)}(P)$ indeed fulfills the ciphertextlength and injectivity requirements. Importantly, one should note that sampling *IEnc* and then quering only one plaintext $C = IEnc(P)$ is statistically equivalent to simply sampling a bit string $C \xleftarrow{r} \{0,1\}^{\ell(|P|)}$ of the correct length uniformly at random. More specifically, since C is a sequence of block $C_0, \ldots, C_l$, it is statistically equivalent to sampling $l+1$ blocks $C_0, \ldots, c_L$ independently and uniformly at random from $\{0,1\}^n$. The objective below is to distinguish the output of a single CBC query $P \in \mathcal{P}_t$ under an unknown random key from t independently and uniformly at random chosen blocks.

This distinguishing attack works by asking for the encryption of a very large plaintext consisting of $l = 4\sqrt{2^n}$ blocks B, for some fixed value B, then the ciphertext blocks are:

$$C_0 = IV \xleftarrow{r} \{0,1\}^n, \quad C_i = Enc_K(C_{i-1} \oplus B) =: F(C_{i-1})$$

The function F is a *permutation* as both $Enc_K(\cdot)$ and $(\cdot \oplus B)$ are permutations, thus there can be no collisions $F(C) = F(C')$ with $C \neq C'$. This implies that any CBC-encryption oracle returns a ciphertext with all distinct ciphertext blocks, or one finds a cycle of length s, e.g. $C_{s+i} = C_{0+i}$ for all i. Whereas a random oracle returns a ciphertext with independent uniform randomly chosen

ciphertext blocks. The probability that they are all unique is:

$$\begin{aligned}
&\Pr[C_i \neq C_j, \quad 0 \leq i < j \leq l] \\
&= \quad 1 \cdot \frac{N-1}{N} \cdots \frac{N-l}{N} = 1 \cdot (1-(1/N)) \cdots (1-(l/N)) \\
&\approx \quad 1 \cdot e^{-1/N} \cdots e^{-l/N} = e^{-(1+2+\cdots+l)/N} = e^{-l(l+1)/2N} = e^{-4\sqrt{N}(4\sqrt{N}+1)/2N} \\
&\approx \quad e^{-8} \approx 0.00033
\end{aligned}$$

(using $N = |\mathcal{M}| = 2^n$ and the Taylor series approximation $e^x = 1 + x + x^2/2! + \cdots$) 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 probability 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 $C_i = C_j$, $0 < i < j < l$, but $C_{i+1} \neq C_{j+1}$ and returning 0 otherwise.

5.10 CBC Padding Oracle Attack

CBC is vulnerable to a so-called padding oracle attack(e.g., see the 2014 POODLE-attack[3] against SSLv3 and some implementations TLS) 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. Below we conveniently describe blocks as concatenation of byte values, e.g., $0^{L-i}||i^i$ means the L-byte block consisting of $L-i$ bytes of value 0 followed by i bytes of value i.

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 following *plaintext block.*

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

[3] `https://www.openssl.org/~bodo/ssl-poodle.pdf`

For block $j = 1, \ldots, l$ do:
  Set recovered last plaintext block $P_{j,rec} = 0$
  For byte $i = 1, \ldots, L$ of the block do:
    For byte value $b = 0, \ldots, 255$ do:
      Let block $P_{pad} = 0^{L-i}||i^i$ and block $P_{guess} = 0^{L-i}||b||0^{i-1}$
      Let $\widehat{C}_{j-1} = C_{j-1} \oplus P_{j,rec} \oplus P_{guess} \oplus P_{pad}$
      If $\mathcal{O}_{pad}(C_0|| \cdots |C_{j-2}||\widehat{C}_{j-1}||C_j) = true$ then
        Set $P_{j,rec} = P_{j,rec} \oplus P_{guess}$
        Continue with next i
Return $P_{1,rec}|| \cdots ||P_{l,rec}$

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.

Continuing 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 the previous i and now skipping the incorrect guess b.

6 Linear cryptanalysis

This section is an introduction to linear cryptanalysis. We'll explain the basic techniques by demonstrating how to use them against a toy cipher built similar to modern block ciphers. The examples here follow those from `http://www.engr.mun.ca/~howard/PAPERS/ldc_tutorial.pdf`. Modern block ciphers (like AES) proceed in rounds. Every round uses a separate *round key* (also called *subkey*) derived from the (master) key using an algorithm called the *key schedule*. A simple structure commonly used to design block ciphers is the *substitution-permutation-network (SPN)*.

6.1 Substitution-Permutation Networks Ciphers

SPN Ciphers consist of a number of rounds. Every round consists of the following operations:

- *key mixing*: add the round key to the current state (linear operation)
- *substitutions*: the state is split into words, each word is substituted using a one-to-one value mapping called an SBox. (non-linear operation)
- *permutation*: a linear operation performing inter-word mixing, e.g., a simple permutation over the bits of the state. (linear operation)

The very last round does no permutation, but instead does a *final key mixing*. A permutation does not add security there as any attacker can immediately revert it independently of the key. Without the final key mixing, the same would hold for the substitution.

As an example for exposition of the techniques we will consider a SPN Cipher of the following bit-oriented form (e.g., in contrast AES is byte-oriented as it uses $GF(2^8)$). Let $w, s, r \in \mathbb{N}^+$, where w is the word size in bits, s is the number of words in the state, and r is the number of rounds. Then the block size, and thus state size, is $w \cdot s$. For simplicity we will use only one SBox for all substitution:

$$\pi_S : \{0,1\}^w \to \{0,1\}^w, \quad \pi_S \in Sym(\{0,1\}^w)$$

and one state bit permutation:

$$\pi_P : \{1, \ldots, w \cdot s\} \to \{1, \ldots, w \cdot s\}, \quad \pi_P \in Sym(\{1, \ldots, w \cdot s\}).$$

Also, we will assume independent random keys $K_1, \ldots, K_{r+1}$ for every round, instead of round keys derived from a master key. The SPN Cipher takes as input a plaintext $P \in \{0,1\}^{ws}$ which is used as the initial state $S_0 = P$. For round $i = 1, \ldots, r$:

1. XOR round key K_i into state: $S_i^{(K)} = S_{i-1} \oplus K_i$;
2. Apply SBox on word-splitted state $S_i^{(K)} = S_{i,1}^{(K)} || \cdots || S_{i,s}^{(K)}$:

$$S_i^{(KS)} = \pi_S(S_{i,1}^{(K)}) \;||\; \pi_S(S_{i,2}^{(K)}) \;||\; \cdots \;||\; \pi_S(S_{i,s}^{(K)}).$$

3. Permute state bits of $S_i^{(KS)} = b_0 || b_1 || \cdots || b_{ws}$:

$$S_i = S_i^{(KSP)} = b_{\pi_P(1)} || b_{\pi_P(2)} || b_{\pi_P(3)} || \cdots || b_{\pi_P(ws)}.$$

Note that in the last round the state bits permutation is typically skipped for efficiency, but includes a final key mixing before outputting the state as the ciphertext block:

$$C = S_r = S_r^{(KS)} \oplus K_{r+1}$$

6.2 Toy Cipher

In this course we will use the following toy cipher to demonstrate linear cryptanalysis as well as differential cryptanalysis in the next section.

This toy cipher has a state of 16 bits $(b_1 \cdots b_{16})$ split into $s = 4$ words of $w = 4$ bits:

$$W_1 = b_1 b_2 b_3 b_4, \quad W_2 = b_5 b_6 b_7 b_8, \quad W_3 = b_9 b_{10} b_{11} b_{12}, \quad W_4 = b_{13} b_{14} b_{15} b_{16}.$$

The cipher only has $r = 4$ rounds using the following SBox π_S on $\{0,1\}^4$:

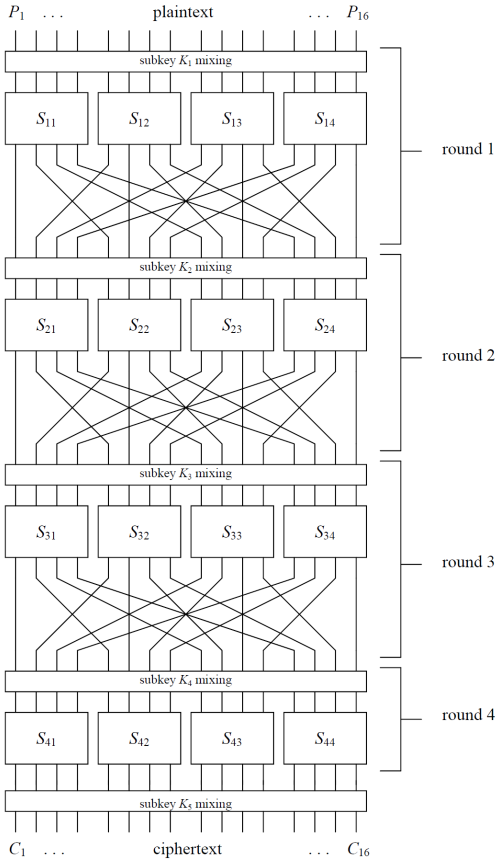
x	0	1	2	3	4	5	6	7
$\pi_S(x)$	14	4	13	1	2	15	11	8
x	0000	0001	0010	0011	0100	0101	0110	0111
$\pi_S(x)$	1110	0100	1101	0001	0010	1111	1011	1000
x	8	9	10	11	12	13	14	15
$\pi_S(x)$	3	10	6	12	5	9	0	7
x	1000	1001	1010	1011	1100	1101	1110	1111
$\pi_S(x)$	0011	1010	0110	1100	0101	1001	0000	0111

Note: here 4 bits are represented as integers, where the *most significant bit* corresponds to the *left most bit*, e.g., bit b_1 in W_1.

The cipher uses the following state bits permutation π_P on $\{1, \ldots, 16\}$:

x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\pi_P(x)$	1	5	9	13	2	6	10	14	3	7	11	15	4	8	12	16

The toy cipher is visually depicted below:



As you can see, SPN Ciphers are simple and can be very efficient, especially in hardware. Decryption is analogous to encryption, each operation is simply reversed.

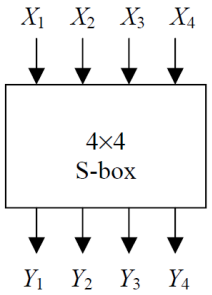
6.3 Linear cryptanalysis

Linear cryptanalysis tries to linearly approximate the cipher, in particular that means a linear approximation of the SBoxes as these are the only non-linear component. The main attack idea is to find a linear relation that holds with a probability that has a significant bias away from the expected probability 1/2. Using linear relations between a subset of the plaintext bits and a subset of the bits of the state $S_r^{(K)}$, i.e., the state before the last substitution, one can try to recover bits of the last round key.

As we show later on, the overall attack uses the probability bias of the linear relation to statistically distinguish when the relation holds (in order to derive information about a subset of the final round key bits). Therefore the attack will require a lot of plaintext-ciphertext pairs in a known-plaintext scenario. The bigger the magnitude of the bias, the fewer pairs are required.

6.4 SBox Linear Approximation Table (LAT)

Let X_1, X_2, X_3, X_4 be random variables for the input bits assumed to be independent and uniformly random distribution and let Y_1, Y_2, Y_3, Y_4 be random variables for the output bits:



Thus for the toy cipher's SBox:

$X_1X_2X_3X_4$	0000	0001	0010	0011	0100	0101	0110	0111
$Y_1Y_2Y_3Y_4$	1110	0100	1101	0001	0010	1111	1011	1000
$X_1X_2X_3X_4$	1000	1001	1010	1011	1100	1101	1110	1111
$Y_1Y_2Y_3Y_4$	0011	1010	0110	1100	0101	1001	0000	0111

We are interested in linear relations for the SBox, i.e., relations of the form

$$\sum_i a_i X_i \oplus \sum_j b_j Y_j = c,$$

where in the current case of $\mathbb{F}_2$ (bits): $a_i, b_j, c \in \{0, 1\}$. There are $2^4 = 16$ different values for $X_1X_2X_3X_4$ all equally likely (by assumption). Hence, the probability that such a linear relation holds can be determined by counting the number of input-output pairs $(X_1X_2X_3X_4, Y_1Y_2Y_3Y_4)$ that satisfy the relation, divided by 16.

E.g., for the relations $X_2+X_3+Y_1+Y_3+Y_4 = 0$, $X_1+X_4+Y_2 = 0$ and $X_3+X_4+Y_1+Y_4 = 0$:

$X_1X_2X_3X_4$	$Y_1Y_2Y_3Y_4$	X_2+X_3	$Y_1+Y_3+Y_4$	X_1+X_4	Y_2	X_3+X_4	Y_1+Y_4
0000	1110	0	0	0	1	0	1
0001	0100	0	0	1	1	1	0
0010	1101	1	0	0	1	1	0
0011	0001	1	1	1	0	0	1
0100	0010	1	1	0	0	0	0
0101	1111	1	1	1	1	1	0
0110	1011	0	1	0	0	1	0
0111	1000	0	1	1	0	0	1
1000	0011	0	0	1	0	0	1
1001	1010	0	0	0	0	1	1
1010	0110	1	1	1	1	1	0
1011	1100	1	1	0	1	0	1
1100	0101	1	1	1	1	0	1
1101	1001	1	0	0	0	1	0
1110	0000	0	0	1	0	1	0
1111	0111	0	0	0	1	0	1

Thus:

$$\Pr[X_2 + X_3 + Y_1 + Y_3 + Y_4 = 0] = 12/16 = 0.75, \quad \Pr[X_1 + X_4 + Y_2 = 0] = 8/16 = 0.5,$$

$$\Pr[X_3 + X_4 + Y_1 + Y_4 = 0] = 2/16 = 0.125.$$

For *independent* X_i and Y_j a linear relation has probability 0.5 to hold. The stronger the probability bias away from 0.5 for a relation, the more useful the relation is. We can describe the probability bias for all possible relations in the *Linear Approximation Table (LAT)*, which is a table whose rows (respectively columns) describe the possible input (respectively output) variable

sums. The cell at row I and column O contains the number of matches between the sum of input bits and the sum of output bits:

$$\sum_{i \in I} X_i = \sum_{j \in O} Y_j$$

minus half the number of possible input values (for a 4-bit SBox: $0.5 \cdot 2^4 = 8$):

		Output sum															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Input sum	0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	-2	-2	0	0	-2	6	2	2	0	0	2	2	0	0
	2	0	0	-2	-2	0	0	-2	-2	0	0	2	2	0	0	-6	2
	3	0	0	0	0	0	0	0	0	2	-6	-2	-2	2	2	-2	-2
	4	0	2	0	-2	-2	-4	-2	0	0	-2	0	2	2	-4	2	0
	5	0	-2	-2	0	-2	0	4	2	-2	0	-4	2	0	-2	-2	0
	6	0	2	-2	4	2	0	0	2	0	-2	2	4	-2	0	0	-2
	7	0	-2	0	2	2	-4	2	0	-2	0	2	0	4	2	0	2
	8	0	0	0	0	0	0	0	0	-2	2	2	-2	2	-2	-2	-6
	9	0	0	-2	-2	0	0	-2	-2	-4	0	-2	2	0	4	2	-2
	10	0	4	-2	2	-4	0	2	-2	2	2	0	0	2	2	0	0
	11	0	4	0	-4	4	0	4	0	0	0	0	0	0	0	0	0
	12	0	-2	4	-2	-2	0	2	0	2	0	2	4	0	2	0	-2
	13	0	2	2	0	-2	4	0	2	-4	-2	2	0	2	0	0	2
	14	0	2	2	0	-2	-4	0	2	-2	0	0	-2	-4	2	-2	0
	15	0	-2	-4	-2	-2	0	2	0	0	-2	4	-2	-2	0	2	0

Here, as before, input and output variable sums are described in decimal, e.g.,

$$X_2 + X_3 = 0X_1 + 1X_2 + 1X_3 + 0X_4 \rightarrow 0110 \rightarrow 6$$

$$Y_1 + Y_3 + Y_4 \rightarrow 1Y_1 + 0Y_2 + 1Y_3 + 1Y_4 \rightarrow 1011 \rightarrow 11.$$

Each table cell contains the count of inputs for which the corresponding linear approximation holds minus 8. Note that the sum for each row and column is either +8 or -8.

Thus the probability bias for any relation can be found by looking up the corresponding number in the LAT and dividing by 16. Such a linear approximation table can be easily computed using SAGE:

```
sage:  S=sage.crypto.sbox.SBox(14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7);
sage:  S.linear_approximation_table()
```

6.5 Piling-Up Lemma

In the attack we will combine several linear relations over individual Sboxes to obtain a linear relation over the first 3 rounds. To determine the probability bias for the resulting linear relation we can use the Piling-Up Lemma (see the original linear cryptanalysis paper by Matsui[4].

Let X_1, X_2 be two independent binary random variables and $p_1 = \Pr[X_1 = 0]$, $p_2 = \Pr[X_2 = 0]$, then:

$\Pr[X_1 \oplus X_2 = 0] = \Pr[X_1 = X_2] = \Pr[X_1 = 0 \wedge X_2 = 0] + \Pr[X_1 = 1 \wedge X_2 = 1] = p_1 p_2 + (1-p_1)(1-p_2)$.

Let $\epsilon_1 = p_1 - 0.5$ and $\epsilon_2 = p_2 - 0.5$ be the probability biases of $X_1 = 0$ and $X_2 = 0$, then:

$$\Pr[X_1 \oplus X_2 = 0] = 0.5 + 2\epsilon_1\epsilon_2.$$

Hence, the probability bias of $X_1 \oplus X_2 = 0$ is $\epsilon_{12} = 2\epsilon_1\epsilon_2$. This can be generalized to n independent binary random variables:

[4] http://link.springer.com/chapter/10.1007%2F3-540-48285-7_33

Lemma 1 (Piling-Up Lemma (Matsui)). *For n independent binary random variables $X_1, \ldots, X_n$ with probability biases $\epsilon_i = \Pr[X_i = 0] - 0.5$:*

$$\Pr[X_1 \oplus \cdots \oplus X_n = 0] = 0.5 + 2^{n-1} \prod_{i=1}^{n} \epsilon_i.$$

Or equivalently,

$$\epsilon_{1,\ldots,n} = 2^{n-1} \prod_{i=1}^{n} \epsilon_i.$$

Note that if $p_i = 0$ for all i then $\Pr[X_1 \oplus \cdots X_n = 0]$ is either 0 or 1. Also, if there is at least one $p_i = 0.5$ then $\Pr[X_1 \oplus \cdots X_n = 0] = 0.5$.

6.6 Linear approximation over multiple rounds

Using the SBox LAT and the piling-up lemma we will show an example how to use them to obtain a linear relation over the first three rounds of the toy cipher, i.e., only the first three key-mixings, substitutions and permutations, and not the last round key- mixing, substitution and final key-mixing. We will use P_i to denote the i-th bit of the plaintext, $K_{j,i}$ to denote the i-th bit of the round key K_j, $X_{j,i}$ and $Y_{j,i}$ to denote the i-th input and output bit of SBox $j \in \{11, \ldots, 14, 21, \ldots, 24, 31 \ldots, 34, 41 \ldots, 44\}$.

1. We start with the linear approximation $X_{12,1} \oplus X_{12,3} \oplus X_{12,4} \oplus Y_{12,2} = 0$ over SBox S_{12} which has probability bias $+1/4$. Note that $X_{12,1} = P_5 \oplus K_{1,5}$, $X_{12,3} = P_7 \oplus K_{1,7}$ and $X_{12,4} = P_8 \oplus K_{1,8}$. Although key bits are unknown, for a given problem instance they're fixed. Therefore we gather sums of key bits on the right-hand side:

$$P_5 \oplus P_7 \oplus P_8 \oplus Y_{12,2} = K_{1,5} \oplus K_{1,7} \oplus K_{1,8} \quad \text{with bias } +1/4. \tag{1}$$

2. Output bit $Y_{12,2}$ is the 6-th state bit remains the 6-th state bit through the permutation. Then key mixing is applied and $Y_{12,2} \oplus K_{2,6} = X_{22,2}$, where $X_{22,2}$ is the 2-nd input bit of SBox S_{22}.

3. We use the linear approximation $X_{22,2} \oplus Y_{22,2} \oplus Y_{22,4} = 0$ over SBox S_{22} with bias $-1/4$ and obtain:

$$Y_{12,2} \oplus Y_{22,2} \oplus Y_{22,4} = K_{2,6} \quad \text{with bias } -1/4. \tag{2}$$

4. With the Piling-Up Lemma we can combine Equation 1 and Equation 2 to obtain:

$$P_5 \oplus P_7 \oplus P_8 \oplus Y_{22,2} \oplus Y_{22,4} = K_{1,5} \oplus K_{1,7} \oplus K_{1,8} \oplus K_{2,6} \quad \text{with bias } 2(1/4)(-1/4) = -1/8. \tag{3}$$

   Note that we make an assumption of independence here which isn't necessarily true. In truth we simple hope that in reality they behave close enough to independence such that the bias is very close or equal to $-1/8$. In practice it turns out to work quite well.

5. Output bits $Y_{22,2}$ and $Y_{22,4}$ pass through another permutation and key-mixing such that

$$Y_{22,2} \oplus K_{3,6} = X_{32,2} \quad \text{and} \quad Y_{22,4} \oplus K_{3,14} = X_{34,2}.$$

6. For both SBoxes S_{32} and S_{34}, we use the same linear relation as for SBox S_{22}:

$$X_{32,2} \oplus Y_{32,2} \oplus Y_{32,4} = 0 \quad \text{with bias } -1/4$$
$$X_{34,2} \oplus Y_{34,2} \oplus Y_{34,4} = 0 \quad \text{with bias } -1/4$$

   Substituting the input bits with the previous output bits we get:

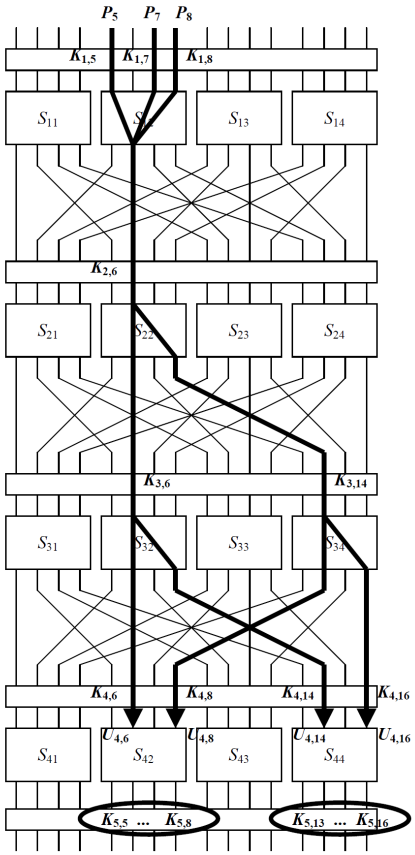
$$Y_{22,2} \oplus Y_{32,2} \oplus Y_{32,4} = K_{3,6} \quad \text{with bias } -1/4 \tag{4}$$
$$Y_{24,2} \oplus Y_{34,2} \oplus Y_{34,4} = K_{3,14} \quad \text{with bias } -1/4 \tag{5}$$

7. Applying the Piling-Up Lemma we can combine three linear relations (Eqs. 3, 4, 5) to obtain:

$$P_5 \oplus P_7 \oplus P_8 \oplus Y_{32,2} \oplus Y_{32,4} \oplus Y_{34,2} \oplus Y_{34,4} = K_{1,5} \oplus K_{1,7} \oplus K_{1,8} \oplus K_{2,6} \oplus K_{3,6} \oplus K_{3,14}, \quad (6)$$

with the following bias: $4(-1/8)(-1/4)(-1/4) = -1/32$.

This linear relation for the toy cipher is visually depicted below:



Note that the final bias of $-1/32$ depends on the biases of the individual relations and the number of SBoxes actively involved in the relation. To strengthen a cipher one can thus try to use SBoxes that have very small biases and a structure that tries to maximize the minimum number of active Sboxes.

6.7 Extracting final round key bits

Once we have a linear approximation over all but the last round for a given cipher that has a suitably large enough probability bias, we can try to exploit it to recover some bits of the final round key bits from the final key-mixing. The linear approximation we obtained above

$$P_5 \oplus P_7 \oplus P_8 \oplus Y_{32,2} \oplus Y_{32,4} \oplus Y_{34,2} \oplus Y_{34,4} = K_{1,5} \oplus K_{1,7} \oplus K_{1,8} \oplus K_{2,6} \oplus K_{3,6} \oplus K_{3,14}$$

can be extended with another key-mixing $Y_{32,2} \oplus K_{4,6} = X_{42,2}$, $Y_{32,4} \oplus K_{4,8} = X_{42,4}$, $Y_{34,2} \oplus K_{4,14} = X_{44,2}$, and $Y_{34,4} \oplus K_{4,16} = X_{44,4}$:

$$\begin{aligned} & P_5 \oplus P_7 \oplus P_8 \oplus X_{42,2} \oplus X_{42,4} \oplus X_{44,2} \oplus X_{44,4} \\ = \quad & K_{1,5} \oplus K_{1,7} \oplus K_{1,8} \oplus K_{2,6} \oplus K_{3,6} \oplus K_{3,14} \oplus K_{4,6} \oplus K_{4,8} \oplus K_{4,14} \oplus K_{4,16} \end{aligned}$$

Let's replace the sum of key bits by zero:

$$P_5 \oplus P_7 \oplus P_8 \oplus X_{42,2} \oplus X_{42,4} \oplus X_{44,2} \oplus X_{44,4} = 0.$$

Now this linear relation holds with bias $-1/32$ if the sum of the involved key bits indeed equals 0, and with bias $1/32$ otherwise. Note that only the sign of the bias changes, the magnitude remains the same. In our attack the sign has no importance, only the magnitude is important to be able to statistically distinguish a correct guess. Hence we can simplify to:

$$P_5 \oplus P_7 \oplus P_8 \oplus X_{42,2} \oplus X_{42,4} \oplus X_{44,2} \oplus X_{44,4} = 0 \quad \text{with bias } \pm 1/32.$$

6.7.1 Attack

Given oracle-access to the toy cipher for unknown $K_1, \ldots, K_5$, the attack proceeds as follows:

- Sample n (plaintext,ciphertext)-pairs $(P^1, C^1), \ldots, (P^n, C^n)$ by querying the encryption of uniformly random chosen plaintexts $P^1, \ldots, P^n$;
- For each possible value $\widehat{K}$ for the 8 bits $K_{5,5-8|13-16}$ of K_5 (those K_5 bits that are involved with SBoxes S_{42} and S_{44}, whose input bits are part of the linear relation) we do the following:
  - For $i = 1, \ldots, n$ compute 4-bit words X^i_{42} and X^i_{44} by partially decrypting C^i:

$$X^i_{42} = \pi_S^{-1}(C^i_{5-8} \oplus \widehat{K}_{5-8}), \quad X^i_{44} = \pi_S^{-1}(C^i_{13-16} \oplus \widehat{K}_{13-16});$$

  - Determine $Count(\widehat{K}) := \{\#i \mid P^i_5 \oplus P^i_7 \oplus P^i_8 \oplus X^i_{42,2} \oplus X^i_{42,4} \oplus X^i_{44,2} \oplus X^i_{42,4} = 0\}$;
- The attack's guessed value K^{guess} is the $\widehat{K}$ with largest sampled bias $Bias(\widehat{K}) = |Count(\widehat{K}) - (n/2)|$.

key guess $K_{5,5-8\|13-16}$	$\|Bias\|$	key guess $K_{5,5-8\|13-16}$	$\|Bias\|$
1C	0.0031	2A	0.0044
1D	0.0078	2B	0.0186
1E	0.0071	2C	0.0094
1F	0.0170	2D	0.0053
20	0.0025	2E	0.0062
21	0.0220	2F	0.0133
22	0.0211	30	0.0027
23	0.0064	31	0.0050
24	**0.0336**	32	0.0075
25	0.0106	33	0.0162
26	0.0096	34	0.0218
27	0.0074	35	0.0052
28	0.0224	36	0.0056
29	0.0054	37	0.0048

Experimental results for the linear attack with 10,000 (plaintext,ciphertext)-pairs. It shows partial subkey guesses and the determined $|Bias| = |count - 5,000|/10,000$, where the 8-bit partial subkeys are written down in hexidecimal. The correct partial subkey in this case is listed in bold and has the highest bias magnitude, not only in the showed listing, but among all guesses for the partial subkey.

6.7.2 Number of samples required

For a linear relation with absolute bias $\epsilon > 0$, the number of samples n for the attack to work well is a small factor times ϵ^{-2} which can be seen as follows.

For the correct key guess we expect that $Count(\widehat{K})$ has Binomial Distribution with n samples and probability $p = 0.5 + \epsilon$ or probability $p = 0.5 - \epsilon$. Thus the expectation of the measured $Count(\widehat{K})$ will be $n \cdot p = (n/2) + n \cdot \epsilon$ or $(n/2) - n \cdot \epsilon$, both leading to measured absolute bias $n \cdot \epsilon$.

For wrong key guesses we expect that $Count(\widehat{K})$ has Binomial Distribution with n samples and probability $p = 0.5$. Thus the expectation of the measured $Count(\widehat{K})$ will be $n/2$, leading to measured absolute bias 0. However, this 'incorrect guess' discrete distribution can be well approximated with the continuous Normal distribution with mean $n/2$ and variance $n \cdot p \cdot (1-p) = n/4$, and thus with standard deviation $SD = \sqrt{n/4}$. This means we can use the following rules of thumb for bounds on the measured absolute biases:

- for 68.3% of incorrect key guesses the measured absolute bias will be smaller than the standard deviation: $Bias(\widehat{K}) < SD$;
- for 95.4% of incorrect key guesses the measured absolute bias will be smaller than twice the standard deviation: $Bias(\widehat{K}) < 2 \cdot SD$;
- for 99.73% of incorrect key guesses the measured absolute bias will be smaller than three times the standard deviation: $Bias(\widehat{K}) < 3 \cdot SD$.
- for 99.9937% of incorrect key guesses the measured absolute bias will be smaller than four times the standard deviation: $Bias(\widehat{K}) < 4 \cdot SD$.
- for 99.999943% of incorrect key guesses the measured absolute bias will be smaller than five times the standard deviation: $Bias(\widehat{K}) < 5 \cdot SD$.
- for 99.99999980% of incorrect key guesses the measured absolute bias will be smaller than six times the standard deviation: $Bias(\widehat{K}) < 6 \cdot SD$.

Note that the expected fraction z of measured absolute biases smaller than $x \cdot SD$ is $z = erf(x/\sqrt{2})$, given by the error-function $erf(x)$. The expected fraction $1 - z$ of measured absolute biases larger than $x \cdot SD$ is bounded as follows:

$$1 - z = 1 - erf(x/\sqrt{2}) = erfc(x/\sqrt{2}) \leq e^{-(x/\sqrt{2})^2} = e^{-x^2/2}, \quad x > 0.$$

That means that with $x = 4$ we can expect in about 1 out of 15787 cases to find an incorrect key guesses' measured absolute bias to be larger than $4 \cdot SD$. With only $2^8 - 1 = 255$ possible incorrect key guesses, we expect all their measured absolute biases to be smaller than $4 \cdot SD$ with very high probability.

Thus for the measured absolute bias of the correct key guess to stand out as the largest with high probability, we need $n \cdot \epsilon > 4 \cdot SD = 4 \cdot \sqrt{n}/2$ which can be rewritten to $n > 4 \cdot \epsilon^{-2}$. Since the blocksize of the toy cipher is 16 bits, there are only 2^{16} possible samples. That means this attack only works for linear relations with absolute bias at least $\epsilon > \sqrt{2^{-16}} = 2^{-8}$, otherwise there aren't enough possible samples for the attack to work.

6.7.3 Extending the attack

Once we have determined the correct value for the final round subkey bits (or at least a short list of highly-likely values that we can go through), we can continue in the following manner:

1. Try to exploit other linear relations in order to obtain all final round key bits, guess all remaining unknown final round key bits;
2. Strip the last round of all known (plaintext,ciphertext)-pairs, i.e., revert the last cipher operations till the second-last key-mixing;
3. One can view the cipher as having $r - 1$ rounds, continue to attack the cipher using a linear relation over $r - 2$ rounds;
4. Iterate 1-3 until all round keys have been broken.

6.7.4 Using SAGE

To experiment with linear cryptanalysis on this toy cipher one can use SAGE and the sage files found on the course website. Install SAGE from `http://www.sagemath.org/doc/installation/` or use SAGE in the cloud `https://cloud.sagemath.com/`.

The files `ptctlist.txt` and `ptctlist.sobj` (SAGE binary format) contain a plaintext-ciphertext list generated with the following commands:

```
sage: load('toycipher_definition.sage')
sage: load('toycipher_gendata.sage')
```

Be careful: executing this yourself will overwrite the file `ptctlist.sobj`.

To execute a linear cryptanalysis attack using the above linear relation to recover bits 5,6,7,8,13,14,15,16 of final round key K5, do the following in sage with access to the `*.sage` and `ptctlist.sobj` files:

```
sage: load('toycipher_definition.sage')
sage: load('linearcryptanalysis.sage')
sage: result
[[1,167]
...
,[173,19]
,[220,200]]
sage: int2K5sub(200)
[0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
```

Read the comments in the `*.sage` files to see what's exactly happening. 'result' is a list of $(|bias|, k5sub)$ sorted for increasing bias. The function `int2K5sub` maps an 8-bit integer to the corresponding guess of bits 5,6,7,8,13,14,15,16 of K5, other bits of K5 are set to 0. In this case the largest bias indeed belongs to the correct K5 sub key, this may not always be the case.

6.8 Space of all linear relations

It is easy to build other linear relations and naturally we can consider the entire space $\mathcal{L}_r$ of all r-round linear relations with non-zero bias for the toy cipher.

6.8.1 Using multiple linear relations

A simple variation on the above linear relation is to drop P_8 and $K_{1,8}$ and use the linear relation $X_{12,1} \oplus X_{12,3} \oplus Y_{12,2} = 0$ over SBox S_{12} which has bias $-4/16$ instead of $+4/16$, hence this leads to the linear relation with bias $+1/32$:

$$P_5 \oplus P_7 \oplus X_{42,2} \oplus X_{42,4} \oplus X_{44,2} \oplus X_{44,4} = K_{1,5} \oplus K_{1,7} \oplus K_{2,6} \oplus K_{3,6} \oplus K_{3,14}.$$

Note that we now have 2 relations that have overlapping plaintext bits and the same round-4 SBoxes. We can execute the attack in section 8.6 for each attack individually and obtain the measured bias for each subkey guess. As mentioned, the correct subkey guess will hopefully show the predicted bias. However the measured bias is a random variable itself, so other incorrect subkey guesses may show random biases possibly of the same magnitude depending on the number of samples.

E.g., the top 5 measured biases for the SAGE example data and the first linear relation, written as $(keyguess, |bias| * 8192)$, are:

$$(60, 237), (236, 192), (190, 164), (252, 149), (44, 142)$$

For the second linear relation above these are:

$$(60, 342), (236, 267), (252, 234), (44 : 231), (226, 204)$$

The predicted absolute bias is $1/32 \cdot 8192 = 256$, so we indeed observe random biases of the same magnitude, we're rather lucky that in both cases the correct keyguess (60) has the highest bias.

In such a case we can use multiple linear relations to obtain an attack with increased success probability in obtaining the correct key guess (by taking the key guess with highest total absolute bias, e.g., 60:237+342=579), or an attack that requires fewer samples for the same success probability.

6.8.2 Interference

Note that in the attack we're trying to find the correct key guess by counting for how many samples $P^i_5 \oplus P^i_7 \oplus P^i_8 \oplus X^i_{42,2} \oplus X^i_{42,4} \oplus X^i_{44,2} \oplus X^i_{42,4} = 0$ holds. We use the single linear relation with large bias as a first order approximation of the real bias. However there may be multiple linear relations that involve exactly the same plaintext bits and round-4 SBox input bits having different biases and/or different involved key bits.

When one linear relation has a large bias and the remaining linear relations have small bias, it should not be a problem to use the large bias as a first order approximation of the real bias in practice.

However, when two or more linear relations have about the same absolute bias then depending on the value of the involved key bits, the biases of these linear relations can either cancel or enlarge each other in the real bias. That means that for half the keys the real bias will be small, whereas for the other half of the keys the real bias will be even larger.

E.g., consider the following two linear relations:

- One having active SBoxes S_{11} (LAT(10,6)), S_{14} (LAT(10,6)), S_{22} (LAT(9,8)), S_{23} (LAT(9,8)), S_{31} (LAT(6,11)) and bias $2^4(2/16)^2(-4/16)^2(4/16) = 1/256$:

$$\begin{aligned} & P_1 \oplus P_3 \oplus P_{13} \oplus P_{15} \oplus Y_{31,1} \oplus Y_{31,3} \oplus Y_{31,4} \\ = \quad & K_{1,1} \oplus K_{1,3} \oplus K_{1,13} \oplus K_{1,15} \oplus K_{2,5} \oplus K_{2,8} \oplus K_{2,9} \oplus K_{2,12} \oplus K_{3,2} \oplus K_{3,3} \end{aligned}$$

- And one having active SBoxes S_{11} (LAT(10,12)), S_{14} (LAT(10,12)), S_{21} (LAT(9,8)), S_{22} (LAT(9,8)), S_{31} (LAT(6,11)) and bias $2^4(2/16)^2(-4/16)^2(4/16) = 1/256$:

$$\begin{aligned} & P_1 \oplus P_3 \oplus P_{13} \oplus P_{15} \oplus Y_{31,1} \oplus Y_{31,3} \oplus Y_{31,4} \\ = \quad & K_{1,1} \oplus K_{1,3} \oplus K_{1,13} \oplus K_{1,15} \oplus K_{2,1} \oplus K_{2,4} \oplus K_{2,5} \oplus K_{2,8} \oplus K_{3,1} \oplus K_{3,2} \end{aligned}$$

In the attack described in section 8.6., we ignored the sum of the key bits as that sum is fixed for fixed key and only affects the sign of the bias. But when multiple linear relations have the same input (plaintext) and output (round-3 sbox) bits their biases will interact:

- They may partially cancel when they have opposing sign, i.e., when

$$\begin{aligned} & K_{1,1} \oplus K_{1,3} \oplus K_{1,13} \oplus K_{1,15} \oplus K_{2,5} \oplus K_{2,8} \oplus K_{2,9} \oplus K_{2,12} \oplus K_{3,2} \oplus K_{3,3} \\ \neq \quad & K_{1,1} \oplus K_{1,3} \oplus K_{1,13} \oplus K_{1,15} \oplus K_{2,1} \oplus K_{2,4} \oplus K_{2,5} \oplus K_{2,8} \oplus K_{3,1} \oplus K_{3,2} \\ \Leftrightarrow \quad & 1 = K_{2,1} \oplus K_{2,4} \oplus K_{2,9} \oplus K_{2,12} \oplus K_{3,1} \oplus K_{3,3} \end{aligned}$$

- Or a greater bias can be observed when they have identical sign, i.e., when

$$\begin{aligned} & K_{1,1} \oplus K_{1,3} \oplus K_{1,13} \oplus K_{1,15} \oplus K_{2,5} \oplus K_{2,8} \oplus K_{2,9} \oplus K_{2,12} \oplus K_{3,2} \oplus K_{3,3} \\ = \quad & K_{1,1} \oplus K_{1,3} \oplus K_{1,13} \oplus K_{1,15} \oplus K_{2,1} \oplus K_{2,4} \oplus K_{2,5} \oplus K_{2,8} \oplus K_{3,1} \oplus K_{3,2} \\ \Leftrightarrow \quad & 0 = K_{2,1} \oplus K_{2,4} \oplus K_{2,9} \oplus K_{2,12} \oplus K_{3,1} \oplus K_{3,3} \end{aligned}$$

In such a case, one can use the two linear relations as a single linear relation that has a higher bias for half of the keys, and a lower bias for the other half of the keys. Most often there will only be one linear relation with a high bias, and all other relations with same input and output bits have significantly lower or zero bias, therefore the high bias can be used as a first order approximation.

7 Differential cryptanalysis

In differential cryptanalysis we simultaneously consider two encryptions $C = Enc_K(P)$ and $C' = Enc_K(P')$ and look at all differences between their computations. For any variable X related to (P, C), we denote by X' the corresponding variable related to (P', C') and denote $\Delta X = X \oplus X'$ as the XOR difference. A key property we will be using is that key additions cancel out:

$$(Y = X \oplus K) \ \wedge \ (Y' = X' \oplus K) \quad \Rightarrow \quad \Delta Y = X \oplus K \oplus X' \oplus K = \Delta X$$

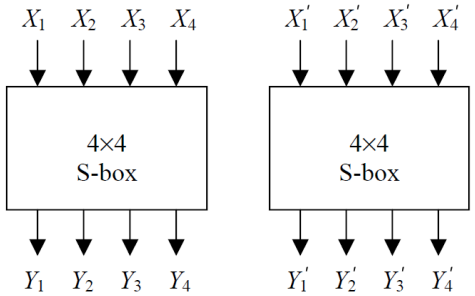
Consider input difference ΔP and output difference ΔC then we call the pair $(\Delta P, \Delta C)$ a *differential*. For an ideally randomizing cipher, the expected probability that ΔC occurs given that ΔP holds is 2^{-n} where n is the plaintext block bit length. Whereas linear cryptanalysis uses the probability bias, for differential cryptanalysis we will use the probability itself:

$$p_{\Delta P, \Delta C} = \Pr[\Delta C \mid \Delta P] = \Pr_P[Enc_K(P) \oplus Enc_K(P \oplus \Delta P) = \Delta C].$$

Differential cryptanalysis tries to exploit high probability occurrences of certain input and output differences of the SBoxes. Very similar to linear cryptanalysis, we try to construct a differential relating plaintext differences to second-last round output differences with a sufficiently high probability. Similar to linear cryptanalysis, we can then again try final round subkey values and check for which fraction of (P, C, P', C')-pairs the differential holds in order to distinguish the correct final round subkey value. The attack is a chosen-plaintext attack as we need to be able to query encryptions of P and $P' = P \oplus \Delta P$.

7.1 SBox Difference Distribution Table (DDT)

Let X_1, X_2, X_3, X_4 be random variables for the input bits assumed to be independent and uniformly random distribution and let Y_1, Y_2, Y_3, Y_4 be random variables for the output bits of one instance. Similarly, let X'_1, X'_2, X'_3, X'_4 and Y'_1, Y'_2, Y'_3, Y'_4 for the second instance.



We are interested in differentials for the SBox of the form

$$(\Delta X_1 \ \Delta X_2 \ \Delta X_3 \ \Delta X_4, \Delta Y_1 \ \Delta Y_2 \ \Delta Y_3 \ \Delta Y_4).$$

There are $2^4 = 16$ different values for $X_1X_2X_3X_4$ all equally likely (by assumption). Hence the probability that such a differential holds can be determined by counting the number of values for $X_1X_2X_3X_4$ such that $(X_1X_2X_3X_4, Y_1Y_2Y_3Y_4) \oplus (X'_1X'_2X'_3X'_4, Y'_1Y'_2Y'_3Y'_4)$ satisfies the differential,

divided by 16. Here the Y_i's are determined through the SBox with input $X_1X_2X_3X_4$, where the Y_i''s are determined through the SBox with input $X_1'X_2'X_3'X_4' = X_1X_2X_3X_4 \oplus \Delta X_1 \Delta X_2 \Delta X_3 \Delta X_4$.

E.g., consider the output difference for input differences $\Delta X_1 \Delta X_2 \Delta X_3 \Delta X_4 = 1011$, 1000 and 0100:

$X_1X_2X_3X_4$	$Y_1Y_2Y_3Y_4$	$\Delta Y \mid \Delta X = 1011$	$\Delta Y \mid \Delta X = 1000$	$\Delta Y \mid \Delta X = 0100$
0000	1110	0010	1101	1100
0001	0100	0010	1110	1011
0010	1101	0111	1011	0110
0011	0001	0010	1101	1001
0100	0010	0101	0111	1100
0101	1111	1111	0110	1011
0110	1011	0010	1011	0110
0111	1000	1101	1111	1001
1000	0011	0010	1101	0110
1001	1010	0111	1110	0011
1010	0110	0010	1011	0110
1011	1100	0010	1101	1011
1100	0101	1101	0111	0110
1101	1001	0010	0110	0011
1110	0000	1111	1011	0110
1111	0111	0101	1111	1011

Thus note that $p_{1011,0010} = 8/16 = 0.5$, $p_{1000,1011} = 4/16 = 0.25$, and $p_{0100,0110} = 6/16 = 0.375$. Ideally, a SBox would have probability 1/16 for every differential, which is impossible, so a good SBox comes as close as possible.

We can describe the probability for all possible differentials in the *Difference Distribution Table (DDT)*, which is a table whose rows (respectively columns) describe the possible input (respectively output) difference:

		Output sum															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Input sum	0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	0	2	0	0	0	2	0	2	4	0	4	2	0	0
	2	0	0	0	2	0	6	2	2	0	2	0	0	0	0	2	0
	3	0	0	2	0	2	0	0	0	0	4	2	0	2	0	0	4
	4	0	0	0	2	0	0	6	0	0	2	0	4	2	0	0	0
	5	0	4	0	0	0	2	2	0	0	0	4	0	2	0	0	2
	6	0	0	0	4	0	4	0	0	0	0	0	0	2	2	2	2
	7	0	0	2	2	2	0	2	0	0	2	2	0	0	0	0	4
	8	0	0	0	0	0	0	2	2	0	0	0	4	0	4	2	2
	9	0	2	0	0	2	0	0	4	2	0	2	2	2	0	0	0
	10	0	2	2	0	0	0	0	0	6	0	0	2	0	0	4	0
	11	0	0	8	0	0	2	0	2	0	0	0	0	0	2	0	2
	12	0	2	0	0	2	2	2	0	0	0	0	2	0	6	0	0
	13	0	4	0	0	0	0	0	4	2	0	2	0	2	0	2	0
	14	0	0	2	4	2	0	0	0	6	0	0	0	0	0	2	0
	15	0	2	0	0	6	0	0	0	0	4	0	2	0	0	2	0

Each cell at row I and column O contains the number of matches between the input difference and output difference:

$$SBox(X) \oplus SBox(X \oplus \Delta I) = \Delta O.$$

Here, as before, input and output variable sums are described in decimal, e.g.,

$$X_2 + X_3 = 0X_1 + 1X_2 + 1X_3 + 0X_4 \rightarrow 0110 \rightarrow 6$$

$$Y_1 + Y_3 + Y_4 \to 1Y_1 + 0Y_2 + 1Y_3 + 1Y_4 \to 1011 \to 11.$$

Each table cell contains the count of inputs for which the corresponding differential characteristic holds. Note that the sum for each row and column is always 16.

Thus the probability bias for any relation can be found by looking up the corresponding number in the LAT and dividing by 16. Such a linear approximation table can be easily computed using SAGE:

```
sage:  S=sage.crypto.sbox.SBox(14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7);
sage:  S.difference_distribution_table()
```

7.2 Piling-Up Differentials

Piling up differentials is very similar to piling-up linear relations, but instead of combining probability biases we can multiply probabilities directly. However, differentials cannot be arbitrarily combined as they can contradict each other, whereas linear relations can simply be added to each other. So to simplify notation we will construct 1-round differentials over the entire state from differentials over SBoxes in the same round, and we will construct $(r + s)$-round differentials by combining a r-round differential and a s-round differential. One can also take a similar view for linear cryptanalysis.

7.2.1 Constructing 1-round differentials

Consider the toy cipher from chapter 8. Let $(\Delta X_{1,2,3,4}, \Delta Y_{1,2,3,4})$, $(\Delta X_{5,6,7,8}, \Delta Y_{5,6,7,8})$, $(\Delta X_{9,10,11,12}, \Delta Y_{9,10,11,12})$, and $(\Delta X_{13,14,15,16}, \Delta Y_{13,14,15,16})$ be differentials for SBoxes S_{r1}, S_{r2}, S_{r3}, and S_{r4}, respectively, for some round r with probabilities p_{r1}, p_{r2}, p_{r3}, and p_{r4}, respectively. We combine these to obtain a differential over the entire substitution step:

$$(\Delta X_1 \dots \Delta X_{16}, \Delta Y_1 \dots \Delta Y_{16}) \quad \text{with probability } p_{subst} = p_{r1} \cdot p_{r2} \cdot p_{r3} \cdot p_{r4}.$$

Let I_r and O_r denote the input and output state of round r. Then $X_i = I_{r,i} \oplus K_{r,i}$, thus $\Delta I_{r,i} = \Delta X_i$ for $i = 1, \dots, 16$. Also, $O_{r,\pi_P(i)} = Y_i$, thus $\Delta O_{r,j} = \Delta Y_{\pi_P^{-1}(j)}$ for $j = 1, \dots, 16$. Therefore:

$$(\Delta X_1 \dots \Delta X_{16}, \Delta Y_1 \Delta Y_5 \Delta Y_9 \Delta Y_{13} \dots \Delta Y_{12} \Delta Y_{16})$$

is a 1-round differential with probability p_{subst}.

7.2.2 Concatenating a r-round and a s-round differential

Let $(\Delta I_1, \Delta O_1)$ be a r-round differential with probability p_1. Let $(\Delta I_2, \Delta O_2)$ be a s-round differential with probability p_2. If $\Delta I_2 = \Delta O_1$ then $(\Delta I_1, \Delta O_2)$ is a $(r + s)$-round differential with probability $p = p_1 \cdot p_2$.

7.3 Constructing a 3-round differential for the Toy Cipher

Let I_r and O_r denote the input and output state of round r, and X_r and Y_r the state before and after the substitution step of round r. As an example: we start with a differential over SBox S_{12} : $(\Delta X, \Delta Y) = (1011, 0010)$ which has probability 8/16 (lookup row 11=1011 column 2=0010 in the DDT). With zero differences for the other SBoxes in round 1, this translates to a differential

$$(\Delta P, \Delta Y_1) = (\Delta X_1, \Delta Y_1) = (0000\ 1011\ 0000\ 0000, 0000\ 0010\ 0000\ 0000), \quad p = 1/2,$$

and we get the round 1 differential by applying the permutation π_P on Y_1:

$$(\Delta P, \Delta O_1) = (0000\ 1011\ 0000\ 0000, 0000\ 0000\ 0100\ 0000), \quad p = 1/2,$$

For round 2 we only have a non-zero difference for SBox S_{23} and we use differential $(\Delta X, \Delta Y) = (0100, 0110)$ with probability 6/16 (DDT row 4 column 6). This leads to

$$(\Delta X_2, \Delta Y_2) = (0000\ 0000\ 0100\ 0000, 0000\ 0000\ 0110\ 0000), \quad p = 3/8,$$

and round-2 differential

$$(\Delta I_2, \Delta O_2) = (0000\ 0000\ 0100\ 0000, 0000\ 0010\ 0010\ 0000), \quad p = 3/8.$$

Combining the round-1 differential and round-2 differential we get

$$(\Delta P, \Delta O_2) = (0000\ 1011\ 0000\ 0000, 0000\ 0010\ 0010\ 0000), \quad p = 3/16.$$

For round 3 we have a non-zero difference for Sboxes S_{32} and S_{33}, we'll use the same differential $(\Delta X, \Delta Y) = (0010, 0101)$ with probability 6/16 for both. This leads to

$$(\Delta X_3, \Delta Y_3) = (0000\ 0010\ 0010\ 0000, 0000\ 0101\ 0101\ 0000), \quad p = 9/64,$$

and round-3 differential

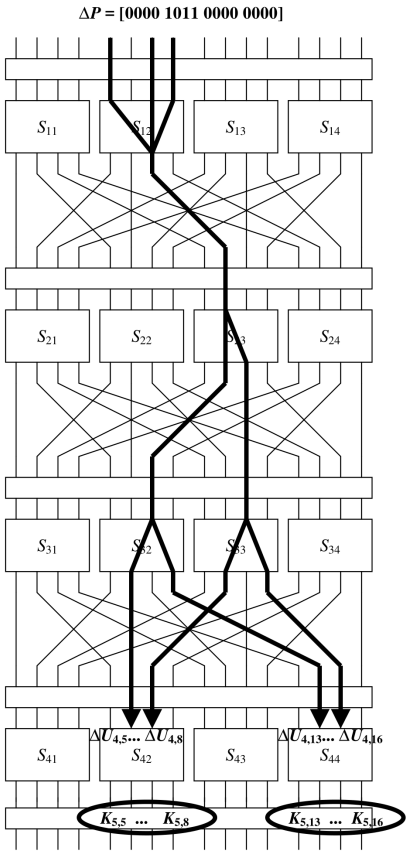
$$(\Delta I_3, \Delta O_3) = (0000\ 0010\ 0010\ 0000, 0000\ 0110\ 0000\ 0110), \quad p = 9/64.$$

Combining the earlier 2-round differential and this round-3 differential we get

$$(\Delta P, \Delta O_3) = (0000\ 1011\ 0000\ 0000, 0000\ 0110\ 0000\ 0110), \quad p = 27/1024,$$

which is a 3-round differential.

This differential for the toy cipher is visually depicted below:



7.4 Extracting final round key bits

The attack procedure is very similar to that for linear cryptanalysis. Once we have a differential over all but the last round for a given cipher that has a suitably large enough probability, we can try to exploit it to recover some bits of the final round key bits from the final key-mixing. The differential and its probability we obtained above

$$(\Delta P, \Delta O_3) = (0000\ 1011\ 0000\ 0000, 0000\ 0110\ 0000\ 0110), \quad p = 27/1024,$$

can be trivially extended with the key-mixing with K_4:

$$(\Delta P, \Delta X_4) = (0000\ 1011\ 0000\ 0000, 0000\ 0110\ 0000\ 0110), \quad p = 27/1024.$$

We will assume that we have access to a large amount of n samples of (P, P', C, C')-tuples where $P' = P \oplus \Delta P$, $C = Enc_K(P)$, $C' = Enc_K(P')$, that were obtained in a chosen-plaintext attack scenario. To be able to check whether the differential holds we need to guess the key bits corresponding to the output bits of the two active SBoxes S_{42} and S_{44} in round 4: $K_{5,\{5,6,7,8,13,14,15,16\}}$, we call these bits the final round *subkey*. The idea is to guess the final round subkey bits and that we hope that for the correct guess the measured differential probability is what is expected, whereas for incorrect guesses we hope to see a measured differential probability very close to 2^{-8}, since we check the difference over just 8 bits instead of 16 bits.

For wrong key guesses we assume a Binomial distribution with parameter 2^{-8}. The measured count of correct differences can be approximated for a large number of samples with a Normal distribution with mean $n \cdot 2^{-8}$ and standard deviation (SD) $\sqrt{n \cdot 2^{-8} \cdot (1 - 2^{-8})} \approx \sqrt{n \cdot 2^{-8}}$. Again we can use an upperbound of 4 times the SD: a measured count for a wrong key guess will exceed $n \cdot 2^{-8} + 4 \cdot \sqrt{n \cdot 2^{-8}}$ only with probability 0.000063. For the right key guess we assume a Binomial distribution with parameter p, so for n samples the expected count of correct differences is $n \cdot p$. We need at least $n \geq c/p$, for some small constant $c \geq 1$, say $c = 6$, to ensure this expected count is sufficiently large to observe and larger than the above upper bound.

For $n = 228 \approx 6/p$, we can clearly distinguish the right key guess from all wrong key guesses with near certainty:

$$n \cdot p \approx 6 > 4.67 \approx n \cdot 2^{-8} + 4 \cdot \sqrt{n \cdot 2^{-8}}.$$

Hence, for differential cryptanalysis the number of required (P, P', C, C')-tuples for the attack is mostly proportional to $1/p$. In practice, it is generally reasonable to use a small multiple of $1/p$ tuples.

Once we have determined the correct value for the final round subkey bits (or at least a short list of highly-likely values that we can go through), we can continue in the following manner identical as for linear cryptanalysis:

1. Try to exploit other differentials in order to obtain all final round key bits, guess all remaining unknown final round key bits;
2. Strip the last round of all known (plaintext,ciphertext)-pairs, i.e., revert the last cipheroperations till the second-last key-mixing;
3. One can view the cipher as having 1 less round, continue to attack the cipher using a differential over fewer rounds;
4. Iterate 1-3 until all round keys have been broken.

7.5 Assumption of independence and using multiple different differentials

For differential cryptanalysis we make the same kind of assumption of independence between the SBox differentials as between the linear relations in linear cryptanalysis. Hence, for differential

cryptanalysis we hope that the predicted differential probability is a very good approximation for the real probability.

There may exist several ways to construct the same 3-round differential whose probabilities always add up as they are so-called *disjoint probability events*. So unlike with linear cryptanalysis, there can be no interference between differentials.

E.g., consider just two distinct SBox differentials with the same input difference $(\Delta X, \Delta Y_1)$ and $(\Delta X, \Delta Y_2)$ with probabilities p_1 and p_2. Now depending on the bits $X_1X_2X_3X_4$ we'll see output difference either ΔY_1 with probability p_1, or ΔY_2 with probability p_2 or some other difference with probability $1 - p_1 - p_2$. But it is impossible to obtain both ΔY_1 and ΔY_2 simultaneously.

7.6 Truncated differential cryptanalysis

The fact that probabilities of differentials with the same input difference can be added, is used in so-called *truncated differential cryptanalysis* where a characteristic is defined over a set of input differences and a set of output differences.

E.g., the following SBox truncated differential $(\{3, 7, 14\}, \{2, 4\})$ has probability 4/16: given input difference 3, 7, or 14, one obtains output difference in the set $\{2, 4\}$ with probability 4/16. This is because differentials $(3, 2)$, $(3, 4)$, $(7, 2)$, $(7, 4)$, $(14, 2)$, $(14, 4)$ all have probability 2/16, so, e.g., for input difference 14 the probability of an output difference in the set $\{2, 4\}$ is 2/16+2/16=4/16. As you can see using truncated differential cryptanalysis we can obtain significantly higher differential probabilities.

7.7 Impossible differential cryptanalysis

Another advanced form of differential cryptanalysis is *impossible differential cryptanalysis*[5] In impossible differential cryptanalysis one tries to exploit a differential $(\Delta P, \Delta O_3)$ that has probability zero, i.e., a 3-round differential for which there exist no construction that leads to a non-zero predicted probability. Thus it does not suffice to consider only one non-zero probability construction for $(\Delta P, \Delta O_3)$, but all constructions that lead to $(\Delta P, \Delta O_3)$ must have predicted probability zero.

An example impossible differential is

$$(\Delta P, \Delta O_3) = (1000\ 0000\ 0000\ 0000, 1000\ 0000\ 0000\ 0000).$$

One way to construct this differential to simply concatenate the following 1-round differential 3 times:

$$(\Delta I, \Delta O) = (1000\ 0000\ 0000\ 0000, 1000\ 0000\ 0000\ 0000).$$

This 1-round differential has probability 0 as for the only active SBox S_{r1} table cell $(8, 8)$ of the DDT is 0.

But we can actually prove there exist no construction with non-zero probability that leads to the above 3-round differential. First off, we will exclude all constructions that use differentials of the form $(x, 0)$ or $(0, x)$ (either input or output difference zero) with non-zero x as they have probability zero anyway. Then note that in round 1 the only active SBox is SBox S_{11} and that its output bits are all mapped to the first input bit of round 2 SBoxes S_{21}, S_{22}, S_{23}, S_{24}. This implies that the output mask determines which round 2 SBoxes are active and that the active round 2 SBoxes have input difference 8(=1000) (difference only in the first bit). E.g., consider Sbox differential $(8, 11) = (1000, 1011)$ for S_{11}, then output difference 11 (=1011) implies that S_{21}, S_{23} and S_{24} are active and have input difference 8 (=1000).

On the other hand, in round 3 the only active SBox is SBox S_{31} and its input bits are all mapped to the first output bit of the round 2 Sboxes S_{21}, S_{22}, S_{23}, S_{24}. This implies that in order to

[5] E.g., see `http://link.springer.com/chapter/10.1007/3-540-48910-X_2`.

obtain $\Delta O_3 = 1000\ 0000\ 0000\ 0000$, the output bits of SBoxes S_{21}, S_{22}, S_{23}, S_{24} that map to the other SBoxes S_{32}, S_{33}, S_{34} should all have difference zero. In other words, the active round 2 SBoxes should have output difference 8=(1000).

To conclude, the active round 2 SBoxes must have input difference 8 and output difference 8, but SBox characteristic $(8, 8)$ has probability 0, so this will always lead to a zero-probability characteristic.

To exploit an impossible differential we can use a procedure very similar to the one from section 9.5 with the following modification:

1. For every guess for the entire round key K_5 we do
   1.1. We go over all (P, P', C, C')-tuples with plaintext difference $\Delta P = 1000\ 0000\ 0000\ 0000$.
   1.2. We partially decrypt the ciphertexts with K_5 and determine ΔO_3.
   1.3. As soon as we find a tuple for which $\Delta O_3 = 1000\ 0000\ 0000\ 0000$ we can cross off this key guess.

Note that even for incorrect K_5 guesses the probability of running into the target ΔO_3 is 2^{-16}, so we can expect many key guesses left that pass this filter. So in order to have only the correct key guess remaining, we'll need many more impossible differentials and use each one to filter out bad key guesses.

7.8 Boomerang distinguishers

This attack by David Wagner[6] effectively doubles the range of differentials and was designed as a distinguisher attack (remember: an attack that distinguishes a cipher with a random key from a random permutation). The idea is to split the cipher into 2 parts (e.g., rounds 1&2 and rounds 3&4) and determine a high probability differential over each part independently and then to find a quartet of plaintext-ciphertext pairs that satisfies these differentials. Thus we'll have a differential over say rounds 1&2: $(\Delta P, \Delta O_2)$ with probability p_1 and a differential over the remaining rounds: $(\Delta I_3, \Delta C)$ with probability p_2.

For the toy cipher we can use almost the same differential for both parts, where the only difference is caused due to the fact that there is no permutation step in the final round. We use differential $(11, 2)$ for SBox S_{13} and differential $(2, 5)$ for SBox S_{23}:

$$(\Delta P, \Delta O_1) = (0000\ 0000\ 1011\ 0000, 0000\ 0000\ 0010\ 0000), \quad \text{probability } 1/2$$

$$(\Delta I_2, \Delta O_2) = (0000\ 0000\ 0010\ 0000, 0000\ 0010\ 0000\ 0010), \quad \text{probability } 3/8$$

This combines to:

$$(\Delta P, \Delta O_2) = (0000\ 0000\ 1011\ 0000, 0000\ 0010\ 0000\ 0010), \quad \text{probability } 3/16.$$

Also, we use differential $(11, 2)$ for SBox S_{33} and differential $(2, 5)$ for SBox S_{43}:

$$(\Delta I_3, \Delta O_3) = (0000\ 0000\ 1011\ 0000, 0000\ 0000\ 0010\ 0000), \quad \text{probability } 1/2$$

$$(\Delta I_4, \Delta O_4) = (0000\ 0000\ 0010\ 0000, 0000\ 0000\ 0101\ 0000), \quad \text{probability } 3/8$$

This combines to:

$$(\Delta I_3, \Delta O_4) = (0000\ 0000\ 1011\ 0000, 0000\ 0000\ 0101\ 0000), \quad \text{probability } 3/16.$$

Given the two separate differentials we try to find a quartet of plaintext-ciphertext pairs $(P_1, P_2, P_3, P_4, C_1, C_2, C_3, C_4)$ that satisfy the following properties:

[6] See `http://link.springer.com/chapter/10.1007%2F3-540-48519-8_12`

- $P_1 \oplus P_2 = \Delta P$
- $P_3 \oplus P_4 = \Delta P$
- $C_1 \oplus C_3 = \Delta C$
- $C_2 \oplus C_4 = \Delta C$

This can be done in the following manner:

1. Select P_1 at random, determine $P_2 = P_1 \oplus \Delta P$;
2. Ask to encrypt P_1 and P_2: $C_1 = Enc_K(P_1)$, $C_2 = Enc_K(P_2)$;
3. Determine $C_3 = C_1 \oplus \Delta C$ and $C_4 = C_2 \oplus \Delta C$;
4. Ask to decrypt C_3 and C_4: $P_3 = Dec_K(C_3)$, $P_4 = Dec_K(C_4)$;
5. If $P_3 \oplus P_4 \neq \Delta P$ then go back to step 1;
6. Return $(P_1, P_2, P_3, P_4, C_1, C_2, C_3, C_4)$.

The predicted success probability for each iteration is $p_1^2 \cdot p_2^2$, hence the expected number of tries required is $p_1^{-2} \cdot p_2^{-2}$, which in the example is $(3/16)^4 \approx 809$.

However the success probability may be amplified by other differentials that only differ in ΔO_2 or ΔI_3. Given a set $\mathcal{R}_1$ of round-1,2 differentials with identical ΔP and a set $\mathcal{R}_2$ of round-3,4 differentials with identical ΔC, the total predicted success probability is:

$$p_{success} = \left(\sum_{(\Delta P, \Delta O_2) \in \mathcal{R}_1} \Pr[(\Delta P, \Delta O_2)]^2 \right) \cdot \left(\sum_{(\Delta I_3, \Delta C) \in \mathcal{R}_2} \Pr[(\Delta I_3, \Delta C)]^2 \right).$$

In theory also sets of characteristics over round 1 and round 2-4 increase the total success probability. But note that a quartet may satisfy both 1-2/3-4-round split differentials and 1/2-4-round split differentials, so these events are not disjoint and therefore we may not simply add success probabilities of differential-pairs with different round splits.

The measured expected number of tries for the above is actually about 100, about 8 times less than expected...

Finding such a quartet against a uniformly selected random permutation instead of a block cipher has a success probability of 2^{-N}, where N is the state size in bits. So when $p_1^{-2} p_2^{-2} << 2^N$, we can distinguish the cipher from a uniformly selected random permutation by trying to find such a quartet in a number of tries significantly smaller than 2^N.

8 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)*: no algorithm A, given randomly selected inputs $f \xleftarrow{r} \mathcal{F}, H \xleftarrow{r} \mathcal{H}$, that returns $M = A(f, H)$ such that $f(M) = H$ is faster than exhaustive search.
- *ePre (everywhere pre-image resistance)*: no algorithm A, that chooses a hash $H \leftarrow A$ before $f \xleftarrow{r} \mathcal{F}$ is selected at random, that returns $M = A(f, H)$ such that $f(M) = H$ is faster than exhaustive search.
- *aPre (always pre-image resistance)*: no algorithm A, that chooses a hash function $f \leftarrow A$ before $H \xleftarrow{r} \mathcal{H}$ is selected at random, that returns $M = A(f, H)$ such that $f(M) = H$ is faster than an exhaustive search.

- *Sec (second pre-image resistance) [n]*: no algorithm A, given randomly selected inputs $f \xleftarrow{r} \mathcal{F}$, $M \xleftarrow{r} \{0,1\}^{\leq n}$, that returns $M' = A(f, M)$ such that $f(M) = f(M')$ and $M' \neq M$ is faster than exhaustive search.
- *eSec (everywhere second pre-image resistance)*: no algorithm A, that chooses a message $M \leftarrow A$ before $f \xleftarrow{r} \mathcal{F}$ is selected at random, that returns $M' = A(f, M)$ such that $f(M) = f(M')$ and $M' \neq M$ is faster than exhaustive search.
- *aSec (always second pre-image resistance) [n]*: no algorithm A, that chooses a hash function $f \leftarrow A$ before $M \xleftarrow{r} \{0,1\}^{\leq n}$ is selected at random, that returns $M' = A(f, M)$ such that $f(M) = f(M')$ and $M' \neq M$ is faster than exhaustive search.
- *Coll (collision resistance)*: no algorithm A, given randomly selected hash function $f \xleftarrow{r} \mathcal{F}$, that returns (M, M') with $f(M) = f(M')$ and $M \neq M'$ 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. Collisions have been found for SHA-1 this year[7] and is thus not collision resistant, while its late predecessor and prior de facto standard MD5 is very weak[8]. 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 facts this seems to be fundamentally impossible due to *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 $\mathcal{A} = \{A_{M,M'} \mid M, M' \in \mathcal{M}'\}$ 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. This is another example of non-uniform algorithms.

8.1 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 $|\mathcal{M}| >> |\mathcal{H}|$. Then one can find a (second) pre-image using a brute force search:

While (true)
  $M' \xleftarrow{r} \mathcal{M}, H' = f(M')$
  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.

8.2 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.

[7] `https://shattered.io`
[8] `https://www.win.tue.nl/hashclash/rogue-ca`

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 $(S, f(S||P))$. 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/`

8.3 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 be the number of hash function output bits, i.e., $|\mathcal{H}| = 2^n$.

$$\begin{aligned}
E[X] &= \sum_{k=1}^{\infty} k \Pr[X = k] \\
&= \sum_{k=1}^{\infty} k(\Pr[X > k-1] - \Pr[X > k]) \\
&= 1(\Pr[X > 0] - \Pr[X > 1]) + 2(\Pr[X > 1] - \Pr[X > 2]) + 3(\Pr[X > 2] - \Pr[X > 3]) + \dots \\
&= 1\Pr[X > 0] + (2-1)\Pr[X > 1] + (3-2)\Pr[X > 2] + (4-3)\Pr[X > 3] + \dots \\
&= \sum_{k=0}^{\infty} \Pr[X > k]
\end{aligned}$$

Earlier we found that:

$$\Pr[X > k] = \Pr[\text{no collision occurred within } k \text{ samples}] \approx e^{-k(k-1)/(2 \cdot 2^n)} \approx e^{-k^2/(2 \cdot 2^n)}.$$

Now we can approximate:

$$E[X] = \sum_{k=0}^{\infty} \Pr[X > k] \approx \sum_{k=0}^{\infty} e^{-k^2/(2 \cdot 2^n)} \approx \int_0^{\infty} e^{-k^2/(2 \cdot 2^n)} = \sqrt{\pi/2} \cdot 2^{n/2}.$$

A direct approach to implement a generic collision attack 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 $O(2^{n/2})$ memory.

Assuming an embedding $\mathcal{H} \subset \mathcal{M}$, one can do significantly better using chains $EP = f^l(SP)$ 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, which we use to generate variable length chains that end in a distinguished point, i.e., $EP = f^l(SP) \subset \mathcal{S}$. Then we can find collisions using the following procedure:

Birthday search with distinguished points

1. Let $T = \emptyset$
2. While (true)
3. $SP \xleftarrow{r} \mathcal{H}$, $P = SP$, $l = 0$
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 $t = |\mathcal{H}|/|\mathcal{S}|$, thus we can expect to store $\sqrt{\pi/2} \cdot 2^{n/2}/t$ chains.

Once a collision occurs, i.e., two chains containing the same point, then those two chains merge and stop at the same distinguished endpoint. Note that once a collision occurs 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 $\sqrt{\pi/2} \cdot 2^{n/2} + t$ function calls. To compute the collision point itself also requires expectedly an $2.5t$ function calls, resulting in a total complexity of $\sqrt{\pi/2} \cdot 2^{n/2} + 3.5t$.

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.

To estimate the amount of wasted work due to stopping with too long chains, we look at two cases. First, for legitimate chains whose length exceeds $20t$ without reaching a loop. The probability p_1 for this is

$$p_1 = \left(1 - \frac{|\mathcal{S}|}{|\mathcal{H}|}\right)^{20t} = \left(1 - \frac{1}{t}\right)^{20t} \approx (e^{-1/t})^{20t} = e^{-20}.$$

Second, the case where a chain enters a loop within $20t$ iterations. The probability p_2 of this case, i.e., when an internal collision occurs, is approximately $p_2 \approx 1 - e^{-(20t)^2/(2 \cdot 2^n)}$.

Together the expected work lost because of these two cases equals $E[\text{lost work per trail}] = (p_1 + p_2) \cdot 20t$, where as $E[\text{work per trail}] = t$. Hence, the expected fraction of lost work is $(p_1 + p_2) \cdot 20 \cdot t/t$, which we want to be negligble. In particular, as $p_1 = e^{-20}$ is already sufficiently small, we need $(20t)^2 << 2^n$ to ensure p_2 is also sufficiently small.

9 Hash Function Cryptanalysis

Cryptographic hash functions map messages of arbitrary size to a fixed size *hash*, e.g., a bitstring of length 256. The most widely used hash functions are SHA-1, SHA-2-256 and SHA-2-512, with

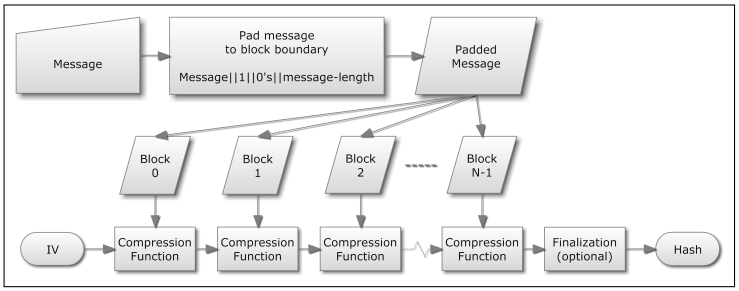
SHA-3-256 and SHA-3-512 being the most recent standard. SHA-1 is not collision resistant, its late predecessor and prior de facto standard MD5 is very weak. All of these hash function standards are fixed hash functions and there is no family $\mathcal{F}$ to speak of, hence of the security properties discussed in Section 7 only aPre and aSec apply.

Remember that for fixed hash functions there is no formal definition of collision resistance due to the *Foundations-of-Hashing dilemma*. Nevertheless many real world cryptographic systems depend on the *informal* collision resistance definition that for these fixed hash functions no one can find collisions faster than the birthday search. Note that hash functions with n-bit output claim n-bit security against aPre, aSec(, Pre, ePre, Sec, eSec), but only $(n/2)$-bit security against collision resistance.

9.1 Hash Function Construction

Hash functions can process messages of arbitrary size, but do so by splitting the message into pieces and iteratively update its internal state using one piece. The two most well-known methods are the Merkle-Damgard construction (independently proposed by Merkle and Damgard in 1989) used by MD5, SHA-1 and SHA-2-224,256,384,512 and the Sponge construction (see `http://www.infosec.sdu.edu.cn/cans2011/Joan%20SpongeCANS2011.pdf`) known from SHA-3.

9.2 Merkle-Damgard construction



To build a hash function $H : \{0,1\}^* \to \{0,1\}^n$, the Merkle-Damgard construction uses a fixed-size *compression function*

$$f : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}^n$$

that takes as input the current n-bit state and a m-bit message piece and outputs the updated n-bit state. The construction itself first unambigiously pads the message with 1s and 0s and the message bitlength, so that the total bit length is a multiple of m, and splits the padded message $pad(M) = M_1||M_2||\cdots||M_l$ into l pieces M_i of bitlength m.

It then starts with an internal state CV_0 called the *chaining value* initialized to a fixed known value called the *initial value*: $CV_0 = IV$. Then for each message block it calls the compression function together with the last chaining value to produce the next chaining value:

$$CV_i = f(CV_{i-1}, M_i), \quad \text{for } i = 1, \ldots, l.$$

Finally it outputs the last chaining value CV_l as the hash (thus a trivial finalization).

9.2.1 Collision Reduction

For the Merkle-Damgard construction one can prove that any collision for the hash function implies a collision for the compression function. Thus one can argue that if one cannot find collisions for the compression function faster than the birthday search, then the same holds for the hash function.

Proof. Let M and M' be two messages with the same hash $H(M) = H(M')$. If M and M' are of different length then their last message pieces M_l and $M'_{l'}$ are different and together with the second last chaining values form a collision of f:

$$f(CV_{l-1}, M_l) = f(CV'_{l'-1}, M'_{l'}), \quad (CV_{l-1}, M_l) \neq (CV'_{l'-1}, M'_{l'}).$$

Otherwise M and M' have the same bitlength and same number of message pieces l, note that $CV_l = CV'_l$. Now consider the smallest $i \in \{0, \ldots, l\}$ such that $(CV_{j-1}, M_j) = (CV'_{j-1}, M'_j)$ for all $i < j \leq l$. If $i = 0$ then M and M' are equal, which is a contradiction. So finally we have $(CV_{i-1}, M_i) \neq (CV'_{i-1}, M'_i)$ and $CV_i = CV'_i$, thus $(CV_{i-1}, M_i), (CV'_{i-1}, M'_i)$ is a collision pair of f. □

9.2.2 Weaknesses: length extension

A rather trivial weakness is that given any message M and hash $H(M)$, it is possible to compute the hash of the extended message $M||padding||S$ in time $O(|S|)$. One can simply continue hashing given $H(M)$ using the pieces of S, naturally the final padding must use the total length including M, the padding of M and S.

This forms a problem for very simple *Message Authentication Codes* (MAC), where one computes a *tag* for a given message using a secret key. Appending this tag to the message allows the receiver with knowledge of the secret key to verify that the message is authentic and has not been tampered with. A very simple MAC would initialize CV_0 with the key, or equivalently output $H(key||message)$, and this would be secure if the hash function behaves as a random function. However, due to the length extension attack, anyone can compute another valid $(message', tag')$-pair given $(message, tag)$-pair directly using the above length extension attack.

Although by some it may be considered a feature, not a bug, of the Merkle-Damgard based hash functions, it forces some (unexpected) care in constructing MACs from Merkle-Damgard based hash functions.

9.2.3 Weaknesses: Multi-collisions

Another weakness was exposed by Joux called the *multi-collision attack*. The idea is that one can use the birthday search to find a one-block collision for the hash function:

$$CV_1 = f(IV, M_1) = f(IV, M'_1) \quad \Rightarrow \quad H(M_1) = H(M'_1).$$

And of course we can find another one-block collision using chaining value CV_1:

$$CV_2 = f(CV_1, M_2) = f(CV_1, M'_2) \quad \Rightarrow \quad H(M_1||M_2) = H(M_1||M'_2).$$

But as $f(IV, M_1) = f(IV, M'_1)$, we actually have

$$H(M_1||M_2) = H(M'_1||M_2) = H(M_1||M'_2) = H(M'_1||M'_2).$$

Now iteratively construct one-block collisions in this manner:

$$H(M_1||\cdots||M_{i-1}||M_i) = H(M_1||\cdots||M_{i-1}||M'_i) \quad \text{for } i = 1, \ldots, t.$$

For each of the t collisions one can choose either block M_i or M'_i, leading to a total of 2^t different messages that all hash to the same hash value.

9.2.4 Weaknesses: Faster second preimages against long messages

Against very long message X of blocklength 2^l, a second preimage can be constructed in complexity $O(\max(2^{n/2+l}, 2^{n-l}))$ using the following procedure by Kelsey and Schneier[9]:

[9] `https://eprint.iacr.org/2004/304.pdf`

First, construct an *expandable message*, i.e., a variant on the above multi-collision where the i-th collision consist of a single block part $|M_i| = m$ and a $(2^{i-1}+1)$-block part $|M'_i| = (2^{i-1}+1)m$. Here we actually need *internal collisions*, i.e., a collision in the chaining value after processing only the message itself without any padding: all 2^t messages hash without padding to the same chaining value $\widehat{CV}$. Also, all 2^t different messages that can be obtained have a different blocklength between t-blocks and $(t+2^t-1)$-blocks. In particular, for any blocklength $r \in [t, t+2^t-1]$ we can construct a message P of blocklength r that hashes without padding to $\widehat{CV}$. This step constructs $t = l$ collisions and requires finding l collisions of at most $2^{l-1}+1$ blocks, thus on average $O(l(2^{l-1}+2)2^{n/2})$ compression function calls.

Now, the original message X is of blocklength 2^l, thus there are 2^l chaining values $CV_1, \ldots, CV_{2^l}$. The idea is to find a block B such that $f(\widehat{CV}, B) \in \{CV_{l+1}, \ldots, CV_{2^l}\}$, this requires on average about $O(2^{n-l})$ attempts for large l (such that $(l+1)/2^l$ is negligible).

Say we have found $f(\widehat{CV}, B) = CV_{r+1}$, $l+1 \leq r+1 \leq 2^l$, then we can construct another message X' that hashes to the same hash as X:

- For blocks 1 up to $r \in [l, 2^l-1]$, we choose the appropriate expandable message of length r that hashes without padding to $\widehat{CV}$: $CV'_r = \widehat{CV}$.
- Block B will be the $(r+1)$-th block of X', so chaining value $CV'_{r+1} = f(CV'_r, B) = CV_{r+1}$.
- Blocks $r+2$ up to 2^l of X' are identical to those of X, thus $CV'_j = CV_j$ for $j = r+2, \ldots, 2^l$.

As X and X' have the same length $|X'| = |X|$, it follows that X' has the same hash as X.

9.2.5 Wide-pipe Merkle-Damgard

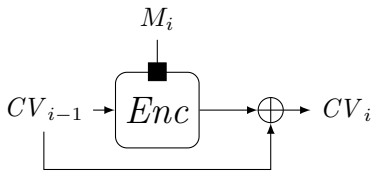
To counteract the above three weaknesses one can make a simple modification to the Merkle-Damgard construction. As the above weaknesses depend either on the knowledge of the internal state or on collision attacks against the internal state, it suffices to make the internal state bitlength twice as big as the hash bitlength. The so-called wide-pipe Merkle-Damgard construction uses an internal state twice as big as the output hash and uses a finalization function to produce the hash from the last internal state, which can be a simple truncation. The length extension attacks against simple MACs don't work anymore as the attacker only knows the output hash and therefore has only partial knowledge about the last internal state that depends on the secret key. Both the multi-collision attack and the second-preimage attack against long messages also don't work as they require more than $O(2^n)$ operations to find collisions in the internal state.

9.2.6 Instantiation using a block cipher

A block cipher $C = Enc_K(B)$ can be used to instantiate the compression function. A trivial way would be to use a message block as the key input K, the current chaining value as the plaintext input B and use the output ciphertext C as the updated chaining value. However, this is insecure and leads to a second preimage attack of complexity $O(2^{n/2})$ using a birthday search in a meet-in-the-middle attack.

Black, Rogaway and Shrimpton have proven that there are 12 basic ways of instantiating a compression function from a block cipher that leads to a secure hash function (see `http://link.springer.com/content/pdf/10.1007%2F3-540-45708-9_21.pdf`). The most common secure way to do so is the Davies-Meyer construction:

$$CV_i = f(CV_{i-1}, M_i) := Enc_{M_i}(CV_{i-1}) \oplus CV_{i-1}.$$



9.3 A very broken hash function: MD5

9.3.1 Notation

MD5 is a hash function designed by Ron Rivest in 1991 and has been used a the de facto hash function for digital signatures up to 2004. It has a hash size of 128 bits and thus was designed to offer 64-bit security against collisions and 128-bit security against (second) pre-image attacks. As we will show in these lecture notes below, MD5 is weak and differential cryptanalysis can be used to construct collisions for MD5 within a second on modern PCs.

Note that 64-bit and even 80-bit security is not sufficient in practice: The combined Bitcoin network, at the time of writing, performs $2^{62.1}$ SHA-2 hashes per second or $2^{87.0}$ SHA-2 hashes per year in Bitcoin mining. See `https://blockchain.info/charts/hash-rate`.

MD5 uses 32-bit words X which can be seen as an integer modulo 2^{32} and simultaneously as a 32-bit string:

$$X: \quad x_{31}x_{30}\cdots x_1x_0 \leftrightarrow \sum_{i=0}^{31} x_i 2^i.$$

We will write 32-bit words using hexidecimal notation, and use *word* when we mean a 32-bit word. It uses the following operations on 32-bit words X:

- Addition and subtraction modulo 2^{32};
- Bitwise AND ($X \wedge Y$), XOR ($X \oplus Y$), OR ($X \vee Y$), NOT ($\bar{X}$) on words X and Y;
- Cyclic bitwise left rotation $RL(X, n)$ of a word X by n bit positions:

$$RL(x_{31}x_{30}\cdots x_1x_0, 1) = x_{30}\cdots x_1x_0x_{31};$$

- Bit index: $X[i]$ denotes the i-th bit of X.

It uses *Little Endian* ordering to parse a bit sequence or byte sequence into words, see `http://en.wikipedia.org/wiki/Endianness`. In particular it means that a byte sequence $b_0b_1b_2b_3b_4b_5b_6b_7\ldots$ (with integers $b_i \in [0, 255]$) is parsed into a word sequence $w_0w_1w_2\ldots$ using:

$$w_i = \sum_{j=0}^{3} b_{4i+j}2^{8j}.$$

E.g., $w_0 = b_0 2^0 + b_1 2^8 + b_2 2^{16} + b_3 2^{24}$, $w_1 = b_4 2^0 + b_5 2^8 + b_6 2^{16} + b_7 2^{24}$.

9.3.2 Compression function

The initial value is defined as the word tuple: $IV = (\texttt{67452301}_x, \texttt{efcdab89}_x, \texttt{98badcfe}_x, \texttt{10325476}_x)$. The compression function takes as input a 4-word tuple $CV_{in} = (A, B, C, D)$ and a 16-word tuple $(W_0, \ldots, W_{15})$ and outputs a 4-word tuple CV_{out} that is computed as follows:

1. Let $W_{16}, \ldots, W_{63}$ and $BF_i(B, C, D)$ be defined as follows:

	$BF_i(B,C,D) = (B \wedge C) \vee (\bar{B} \wedge D)$	for $i \in [0, 15]$
$W_i = W_{1+5i \bmod 16}$	$BF_i(B,C,D) = (D \wedge B) \vee (D \wedge \bar{C})$	for $i \in [16, 31]$
$W_i = W_{5+3i \bmod 16}$	$BF_i(B,C,D) = B \oplus C \oplus D$	for $i \in [32, 47]$
$W_i = W_{7i \bmod 16}$	$BF_i(B,C,D) = C \oplus (B \vee \bar{D})$	for $i \in [48, 63]$

2. Let the addition constants be $AC_i = \lfloor 2^{32} \sin(i+1) \rfloor$ for $i \in [0, 63]$;

3. Let the rotation constants RC_i be defined as follows

$$\begin{array}{l|l} (RC_i, RC_{i+1}, RC_{i+2}, RC_{i+3}) = (7, 12, 17, 22) & \text{for } i = 0, 4, 8, 12 \\ (RC_i, RC_{i+1}, RC_{i+2}, RC_{i+3}) = (5, 9, 14, 20) & \text{for } i = 16, 20, 24, 28 \\ (RC_i, RC_{i+1}, RC_{i+2}, RC_{i+3}) = (4, 11, 16, 33) & \text{for } i = 32, 36, 40, 44 \\ (RC_i, RC_{i+1}, RC_{i+2}, RC_{i+3}) = (6, 10, 15, 21) & \text{for } i = 48, 52, 56, 60 \end{array}$$

4. Let $(A, B, C, D) = CV_{in}$

5. For $i = 0, \ldots, 63$ do

   5.1. $F_i = BF_i(B, C, D)$

   5.2. $T_i = A + F_i + W_i + AC_i \mod 2^{32}$

   5.3. $R_i = RL(T_i, RC_i)$

   5.4. $(A, B, C, D) = (D, B + R_i, B, C)$

6. Return $CV_{out} = CV_{in} + (A, B, C, D)$

For convenience, here are the full tables with AC_i, RC_i and W_i:

i	AC_i	RC_i	W_i
0	$\texttt{d76aa478}_x$	7	W_0
1	$\texttt{e8c7b756}_x$	12	W_1
2	$\texttt{242070db}_x$	17	W_2
3	$\texttt{c1bdceee}_x$	22	W_3
4	$\texttt{f57c0faf}_x$	7	W_4
5	$\texttt{4787c62a}_x$	12	W_5
6	$\texttt{a8304613}_x$	17	W_6
7	$\texttt{fd469501}_x$	22	W_7
8	$\texttt{698098d8}_x$	7	W_8
9	$\texttt{8b44f7af}_x$	12	W_9
10	$\texttt{ffff5bb1}_x$	17	W_{10}
11	$\texttt{895cd7be}_x$	22	W_{11}
12	$\texttt{6b901122}_x$	7	W_{12}
13	$\texttt{fd987193}_x$	12	W_{13}
14	$\texttt{a679438e}_x$	17	W_{14}
15	$\texttt{49b40821}_x$	22	W_{15}

i	AC_i	RC_i	W_i
16	$\texttt{f61e2562}_x$	5	W_1
17	$\texttt{c040b340}_x$	9	W_6
18	$\texttt{265e5a51}_x$	14	W_{11}
19	$\texttt{e9b6c7aa}_x$	20	W_0
20	$\texttt{d62f105d}_x$	5	W_5
21	$\texttt{02441453}_x$	9	W_{10}
22	$\texttt{d8a1e681}_x$	14	W_{15}
23	$\texttt{e7d3fbc8}_x$	20	W_4
24	$\texttt{21e1cde6}_x$	5	W_9
25	$\texttt{c33707d6}_x$	9	W_{14}
26	$\texttt{f4d50d87}_x$	14	W_3
27	$\texttt{455a14ed}_x$	20	W_8
28	$\texttt{a9e3e905}_x$	5	W_{13}
29	$\texttt{fcefa3f8}_x$	9	W_2
30	$\texttt{676f02d9}_x$	14	W_7
31	$\texttt{8d2a4c8a}_x$	20	W_{12}

i	AC_i	RC_i	W_i
32	$\texttt{fffa3942}_x$	4	W_5
33	$\texttt{8771f681}_x$	11	W_8
34	$\texttt{6d9d6122}_x$	16	W_{11}
35	$\texttt{fde5380c}_x$	23	W_{14}
36	$\texttt{a4beea44}_x$	4	W_1
37	$\texttt{4bdecfa9}_x$	11	W_4
38	$\texttt{f6bb4b60}_x$	16	W_7
39	$\texttt{bebfbc70}_x$	23	W_{10}
40	$\texttt{289b7ec6}_x$	4	W_{13}
41	$\texttt{eaa127fa}_x$	11	W_0
42	$\texttt{d4ef3085}_x$	16	W_3
43	$\texttt{04881d05}_x$	23	W_6
44	$\texttt{d9d4d039}_x$	4	W_9
45	$\texttt{e6db99e5}_x$	11	W_{12}
46	$\texttt{1fa27cf8}_x$	16	W_{15}
47	$\texttt{c4ac5665}_x$	23	W_2

i	AC_i	RC_i	W_i
48	$\texttt{f4292244}_x$	6	W_0
49	$\texttt{432aff97}_x$	10	W_7
50	$\texttt{ab9423a7}_x$	15	W_{14}
51	$\texttt{fc93a039}_x$	21	W_5
52	$\texttt{655b59c3}_x$	6	W_{12}
53	$\texttt{8f0ccc92}_x$	10	W_3
54	$\texttt{ffeff47d}_x$	15	W_{10}
55	$\texttt{85845dd1}_x$	21	W_1
56	$\texttt{6fa87e4f}_x$	6	W_8
57	$\texttt{fe2ce6e0}_x$	10	W_{15}
58	$\texttt{a3014314}_x$	15	W_6
59	$\texttt{4e0811a1}_x$	21	W_{13}
60	$\texttt{f7537e82}_x$	6	W_4
61	$\texttt{bd3af235}_x$	10	W_{11}
62	$\texttt{2ad7d2bb}_x$	15	W_2
63	$\texttt{eb86d391}_x$	21	W_9

Another way to define the compression function that unrolls the cyclic state (step 5.3) is by replacing the following steps:

4. Let $(Q_0, Q_{-3}, Q_{-2}, Q_{-1}) = (A, B, C, D) = CV_{in}$
5. For $i = 0, \ldots, 63$ do
   5.1. $F_i = BF_i(Q_i, Q_{i-1}, Q_{i-2})$
   5.2. $T_i = Q_{i-3} + F_i + W_i + AC_i \mod 2^{32}$
   5.3. $R_i = RL(T_i, RC_i)$
   5.4. $Q_{i+1} = Q_i + R_i$
6. Return $CV_{out} = CV_{in} + (Q_{61}, Q_{64}, Q_{63}, Q_{62})$

This description simplifies the cryptanalysis and is what we'll use. We'll leave it to the reader to verify the equivalence of these two description.

9.3.3 Properties of the compression function

- Each message word is used four times, which makes it very hard given arbitrary values for CV_{in} and CV_{out} to find a message block B that hashes CV_{in} to $CV_{out} = f(CV_{in}, B)$.
- Each step is a permutation on the state: for fixed $Q_{i-2}, Q_{i-1}, Q_i, W_i$ there is a bijective mapping between Q_{i-3} and Q_{i+1}.
- A consequence of 2. is that the 64 steps are a permutation on the state: for fixed message block, there is a bijective mapping between $(Q_0, Q_{-1}, Q_{-2}, Q_{-3})$ and $(Q_{64}, Q_{63}, Q_{62}, Q_{61})$. This means you can indeed view MD5's compression function as a block cipher in Davies-Meyer mode. Note that decryption (the inverse mapping) is as fast as encryption.
- In each step the message word fully contributes to the output, i.e.: for fixed $Q_{i-3}, Q_{i-2}, Q_{i-1}, Q_i$, there is a bijective mapping between W_i and Q_{i+1}.
- For the 'decryption' the same holds, i.e.: for fixed $Q_{i-2}, Q_{i-1}, Q_i, Q_{i+1}$, there is a bijective mapping between W_i and Q_{i-3}.

9.4 Modular differential cryptanalysis

One of the first attempts to break MD5 was using modular differential cryptanalysis, i.e., differential cryptanalysis using difference $\delta X = X' - X \mod 2^{32}$. (e.g., see `http://link.springer.com/content/pdf/10.1007%2F3-540-47555-9_6.pdf`). Den Boer and Bosselaers (see `http://link.springer.com/content/pdf/10.1007%2F3-540-48285-7_26.pdf`) showed in 1993 that one can find collisions for the compression function of MD5 of the form:

$$f(CV, M) = f(CV', M) \text{ with } \delta CV = (2^{31}, 2^{31}, 2^{31}, 2^{31}).$$

The idea is that for all steps $i = 0, \ldots, 63$, given $\delta Q_i = \delta Q_{i-1} = \delta Q_{i-2} = \delta Q_{i-3} = 2^{31}$, it holds that

$$\delta F_i = BF_i(Q'_i, Q'_{i-1}, Q'_{i-2}) - BF_i(Q_i, Q_{i-1}, Q_{i-2}) = 2^{31}$$

with probability 1 (for $i = 32, \ldots, 47$) or probability 0.5 (otherwise). Note that as there is no difference in the boolean function inputs in bit positions 0,...,30, there can also be no output difference in bit positions 0,...,30, i.e., $\delta F_i \in \{2^{31}, 0\}$.

For each bit position we can view the boolean function as a 3-to-1 substitution box and thus determine from DDTs that $\delta F_i = 2^{31}$ with probability 1 for $i = 32, \ldots, 47$ and probability 0.5 otherwise.

In which case it follows that:

$$\begin{aligned}
\delta F_i &= 2^{31} \\
\delta T_i &= \delta Q_{i-3} + \delta F_i + \delta W_i + \delta AC_i = 2^{31} + 2^{31} + 0 + 0 \bmod 2^{32} = 0 \\
\delta R_i &= RL(T_i', RC_i) - RL(T_i, RC_i) = RL(T_i, RC_i) - RL(T_i, RC_i) = 0 \\
\delta Q_{i+1} &= \delta Q_i + \delta R_i = 2^{31} + 0 = 2^{31}
\end{aligned}$$

Then after 64 steps we find a collision:

$$\begin{aligned}
\delta A + \delta Q_{61} &= 2^{31} + 2^{31} \bmod 2^{32} = 0 \\
\delta B + \delta Q_{62} &= 2^{31} + 2^{31} \bmod 2^{32} = 0 \\
\delta C + \delta Q_{63} &= 2^{31} + 2^{31} \bmod 2^{32} = 0 \\
\delta D + \delta Q_{64} &= 2^{31} + 2^{31} \bmod 2^{32} = 0
\end{aligned}$$

For random CV_{in} and message block B, the success probability is 2^{-48}, requiring 2^{48} attempts and 2^{49} compression function calls.

9.5 Speed-up using simple message modification

However, an important difference between block cipher cryptanalysis and hash function cryptanalysis, is that in hash function cryptanalysis the attacker knows all the intermediate computations as there is no secret key. This can provide an immediate speed-up by using a single partial solution to generate many partial solutions, essentially reducing the cost of finding partial solutions significantly.

Assume we have found a (CV, B)-pair (with implied $CV' = CV + \delta CV$) such that $\delta T_0 = \cdots = \delta T_{15} = 0$ as desired, requiring on average 2^{16} attempts. Then note that by changing W_{15} to another value, we do not change δF_{15} and thus δT_{15} also doesn't change. As earlier steps aren't affected at all, so in fact we now know 2^{32} different (CV, B)-pairs with $\delta T_0 = \cdots = \delta T_{15} = 0$. For these pairs the probability that also $\delta T_{16} = \cdots = \delta T_{63} = 0$ is 2^{-32}, thus among those 2^{32} pairs we can expect to find one that leads to a collision. The expected cost is now $2^{17} + 2^{33}$ compression function calls. By applying this idea also to the first 15 steps one can further reduce the expected cost to $32 + 2^{33}$.

Although den Boer and Bosselaers were able to find collisions for the compression function, the attack did not produce collisions for MD5, as finding message blocks (M_1, M_1') leading to the required chaining value difference $\delta CV_1 = (2^{31}, 2^{31}, 2^{31}, 2^{31})$ remained an open problem. Let's call this type of collision function collision *dBB-collisions*.

9.6 Combined modular and bitwise differential cryptanalysis

The first successful collision attack on MD5 was due to Xiaoyun Wang and her team. See `http://link.springer.com/content/pdf/10.1007%2F11426639_2.pdf` for the original paper, however another paper provides a significant better exposure how the attack works: `https://eprint.iacr.org/2004/264.pdf`.

They used a number of key ingredients to make their attack successful:

1. They used modular and bitwise differentials together to tackle both modular additions and the bitwise operations with high probabilities, i.e., they used the difference

   $$\Delta X = (X'[i] - X[i])_{i=0}^{31} \quad \text{(bitwise signed difference between the bits of } X' \text{ and } X)$$

   for variables that enter the boolean function, i.e., $Q_{-2}, \ldots, Q_{63}$, and the modular difference for other variables: W_i, F_i, T_i, R_i, Q_{-3}, Q_{64}.

2. They chose very specific δM to make their attack work, assume these fixed for now. They were chosen to create a high probability differential path over steps $16, \ldots, 63$. That means that over steps $0, \ldots, 15$ one must create a more complex path that starts with IV and ends with the state difference at step 16 required by the high probability differential path.
3. Given an exact differential path, i.e., given all (bitwise) differences δW_i, ΔQ_i, δF_i, δT_i, δR_i, one can determine sufficient conditions only on variables Q_i and T_i related to M (and thus not on variables Q'_i, T'_i related to M'). If the sufficient conditions are satisfied and hold then the different path is followed exactly obtaining $\delta CV_{out} = \delta CV_{in} + (\delta Q_{61}, \delta Q_{64}, \delta Q_{63}, \delta Q_{62})$.
4. They use speed-ups that use (more complex) message modifications. Given a partial solution (CV, W) (and thus $(CV', W') = (CV, W) + (\delta CV, \delta W)$) that satisfies the sufficient conditions up to Q_t and T_t for some t, one may make small changes to W into $\widehat{W}$ such that $(CV, \widehat{W})$ is also a partial solution that satisfies the sufficient conditions up to Q_t and T_t. This is called a message modification technique and can considerably speed up the attack. E.g., the changes we made to W_{15} to generate 2^{32} solutions in the den Boer and Bosselaers attack is a simple example.

9.6.1 Bitwise signed difference

The bitwise signed difference can be described by a *binary signed digit representation*. A 32-bit binary signed digit representation is a sequence $b_{31} \ldots b_0 = (b_i)_{i=0}^{31}$ with $b_i \in \{-1, 0, 1\}$, whereas a binary digit representation limits $b_i \in \{0, 1\}$. We will call such a 32-bit binary signed digit representation a BSDR, and note that a difference $\Delta X = (X'[i] - X[i])_{i=0}^{31}$ is a BSDR.

Let the map $\sigma : \{-1, 0, 1\}^{32} \to \{0, 1\}^{32}$ from a BSDR Y to a word X be defined as:

$$\sigma(Y) = \sigma((y_i)_{i=0}^{31}) = \sum_{i=0}^{31} y_i 2^i \bmod 2^{32}.$$

Also let $hw(Y)$ denote the hamming weight (number of non-zero digits) for words Y as well as for BSDRs Y.

Given BSDR ΔX we can uniquely determine the word $\delta X = \sigma(\Delta X)$. However, for any given non-zero word X there are many possible BSDRs Y such that $X = \sigma(Y)$. Let's take as an example $X = 1$, then the following BSDRs Y have $\sigma(Y) = X$:

- $y_{31} = \cdots = y_1 = 0$, $y_0 = 1$;
- $y_{31} = \cdots = y_2 = 0$, $y_1 = 1$, $y_0 = -1$;
- $y_{31} = \cdots = y_3 = 0$, $y_2 = 1$, $y_1 = y_0 = -1$;
- $y_{31} = \cdots = y_4 = 0$, $y_3 = 1$, $y_2 = \cdots = y_0 = -1$;

  $\vdots$
- $y_{31} = 0$, $y_{30} = 1$, $y_{29} = \cdots = y_0 = -1$;
- $y_{31} = 1$, $y_{30} = \cdots = y_0 = -1$;
- $y_{31} = \cdots = y_0 = -1$;

We call this effect carry propagation. More generally for any BSDR Y and for any two consecutive signed bits (y_i, y_{i+1}), we can make the following exchanges without $\sigma(Y)$:

- $(1, 0) \leftrightarrow (-1, 1)$, since $2^i = 2^{i+1} - 2^i$;
- $(-1, 0) \leftrightarrow (1, -1)$, since $-2^i = -2^{i+1} + 2^i$.

Also, we can exchange between $y_{31} = 1$ and $y_{31} = -1$ without changing $\sigma(Y)$ as $-2^{31} \equiv 2^{31} \bmod 2^{32}$.

9.6.2 Sufficient conditions

To show how one can derive sufficient conditions on just Q_i and T_i such that a differential path holds, let's consider the exact differential path for the den Boer and Bosselaers attack, we have:

- $\Delta Q_i = (\pm 1, 0, \ldots, 0)$, $\delta Q_i = \sigma(\Delta Q_i) = 2^{31}$;
- $\delta W_i = 0$;
- $\delta F_i = 2^{31}$;
- $\delta R_i = \delta T_i = 0$.

As the boolean function operates bitwise and there are no differences in the bit positions 0 up to 30, we see that $\Delta F_i[b] = 0$ for $b = 0, \ldots, 30$ and $i = 0, \ldots, 63$. Let's consider the boolean function of the first 16 steps at bit position 31 where we have three input bits and thus eight possible input cases:

$Q_i[31] \| Q_{i-1}[31] \| Q_{i-2}[31]$	$Q'_i[31] \| Q'_{i-1}[31] \| Q'_{i-2}[31]$	$F_i[31]$	$F'_i[31]$	$\Delta F_i[31]$
000	111	0	1	1
001	110	1	1	0
010	101	0	0	0
011	100	1	0	-1
100	011	0	1	1
101	010	0	0	0
110	001	1	1	0
111	000	1	0	-1

To achieve $\delta F_i = 2^{31}$ there are only 4 allowed values for these three input bits: 000, 011, 100, 111. So one can see that $\delta F_i = 2^{31}$ if and only if $Q_{i-1}[31] = Q_{i-2}[31]$. This condition $Q_{i-1}[31] = Q_{i-2}[31]$ is thus both necessary and sufficient to obtain $\delta F_i = 2^{31}$ and $\delta Q_{i+1} = 2^{31}$ for $i = 0, \ldots, 15$, and only involves variables related to M and not to M'.

For steps $i = 16, \ldots, 31$, we have the following output cases:

$Q_i[31] \| Q_{i-1}[31] \| Q_{i-2}[31]$	$Q'_i[31] \| Q'_{i-1}[31] \| Q'_{i-2}[31]$	$F_i[31]$	$F'_i[31]$	$\Delta F_i[31]$
000	111	0	1	1
001	110	0	1	1
010	101	1	1	0
011	100	0	0	0
100	011	0	0	0
101	010	1	1	0
110	001	1	0	-1
111	000	1	0	-1

To achieve $\delta F_i = 2^{31}$ there are only 4 allowed values for these three input bits: 000, 001, 110, 111. So one can see that $\delta F_i = 2^{31}$ if and only if $Q_i[31] = Q_{i-1}[31]$. This condition $Q_i[31] = Q_{i-1}[31]$ is thus both necessary and sufficient to obtain $\delta F_i = 2^{31}$ and $\delta Q_{i+1} = 2^{31}$ for $i = 16, \ldots, 31$.

For steps $i = 32, \ldots, 47$ there are no sufficient conditions as all output cases have $\delta F_i = 2^{31}$:

$Q_i[31] \| Q_{i-1}[31] \| Q_{i-2}[31]$	$Q'_i[31] \| Q'_{i-1}[31] \| Q'_{i-2}[31]$	$F_i[31]$	$F'_i[31]$	$\Delta F_i[31]$
000	111	0	1	1
001	110	1	0	-1
010	101	1	0	-1
011	100	0	1	1
100	011	1	0	-1
101	010	0	1	1
110	001	0	1	1
111	000	1	0	-1

For steps $i = 48, \ldots, 63$, we have the following output cases:

$Q_i[31] \| Q_{i-1}[31] \| Q_{i-2}[31]$	$Q'_i[31] \| Q'_{i-1}[31] \| Q'_{i-2}[31]$	$F_i[31]$	$F'_i[31]$	$\Delta F_i[31]$
000	111	1	0	-1
001	110	0	0	0
010	101	0	1	1
011	100	1	1	0
100	011	1	1	0
101	010	1	0	-1
110	001	0	0	0
111	000	0	1	1

To achieve $\delta F_i = 2^{31}$ there are only 4 allowed values for these three input bits: 000, 010, 101, 111. So one can see that $\delta F_i = 2^{31}$ if and only if $Q_i[31] = Q_{i-2}[31]$. This condition $Q_i[31] = Q_{i-2}[31]$ is thus both necessary and sufficient to obtain $\delta F_i = 2^{31}$ and $\delta Q_{i+1} = 2^{31}$ for $i = 48, \ldots, 63$.

Combining these conditions we find:

- $Q_{14}[31] = Q_{13}[31] = \cdots = Q_{-2}[31]$;
- $Q_{31}[31] = Q_{30}[31] = \cdots = Q_{15}[31]$;
- $Q_{62}[31] = Q_{60}[31] = Q_{58}[31] = \cdots = Q_{46}[31]$;
- $Q_{63}[31] = Q_{61}[31] = Q_{59}[31] = \cdots = Q_{47}[31]$;

Having found the set of sufficient conditions we have now reduced the problem of finding a pair $(CV, W), (CV', W')$ that follows the differential path to the problem of finding a $(CV, W$ that satisfies the sufficient conditions. This provides significant advantages: Firstly, we don't have to keep track of the computation (CV', W') anymore, providing a factor 2 speed up. Secondly, we notice sooner that a (CV, W) is not a solution, e.g., if step 13 fails with $\delta Q_{14} \neq 2^{31}$ then this will be noticed already in step 11 when $Q_{12}[31] \neq Q_{11}[31]$.

i	$Q_i[31]$ $\cdots$ $Q_i[0]$
-3	....
-2	....
-1	^....... ....
0	^....... ....
1	^....... ....
2	^....... ....
3	^....... ....
4	^....... ....
5	^....... ....
6	^....... ....
7	^....... ....
8	^....... ....
9	^....... ....
10	^....... ....
11	^....... ....
12	^....... ....
13	^....... ....
14	^....... ....
15	....
16	^....... ....
17	^....... ....
18	^....... ....
19	^....... ....
20	^....... ....
21	^....... ....
22	^....... ....
23	^....... ....
24	^....... ....
25	^....... ....
26	^....... ....
27	^....... ....
28	^....... ....
29	^....... ....
30	^....... ....
31	^....... ....
32	....
33	....
34	....
35	....
36	....
37	....
38	....
39	....
40	....
41	....
42	....
43	....
44	....
45	....
46	....
47	....
48	M....... ....
49	M....... ....
50	M....... ....
51	M....... ....
52	M....... ....
53	M....... ....
54	M....... ....
55	M....... ....
56	M....... ....
57	M....... ....
58	M....... ....
59	M....... ....
60	M....... ....
61	M....... ....
62	M....... ....
63	M....... ....
64	....

.	$Q_i[b] \in \{0,1\}$
^	$Q_i[b] = Q_{i-1}[b]$
M	$Q_i[b] = Q_{i-2}[b]$

Table 1: den Boer and Bosselaer conditions

9.6.3 Message modification

As mentioned before there is a bijective mapping between W_i and Q_{i+1}. This implies that given $Q_{i-3}, \ldots, Q_{i+1}$ there is a unique value for W_i determined by:

1. $R_i = Q_{i+1} - Q_i$;
2. $T_i = RL(R_i, 32 - RC_i)$;
3. $F_i = BF_i(Q_i, Q_{i-1}, Q_{i-2})$;
4. $W_i = T_i - Q_{i-3} - F_i - AC_i$.

Hence one can trivially find a message block that satisfies the conditions up to Q_{16}:

1. Choose $Q_{-3}, \ldots, Q_{16}$ that satisfy the sufficient conditions;
2. Compute $W_0, \ldots, W_{15}$ via the above relation.

This only costs 16 steps or 1/4-th compression function.

A more careful consideration of the message word order over the second round (steps $16, \ldots, 31$) allows us to easily fulfill sufficient conditions up to Q_{19}:

- $W_{16} = W_1$ can be changed by choosing different $Q_{-3}, \ldots, Q_2$ satisfying conditions. This will also change W_0 and other words but those are used later on. Thus we can easily fulfill conditions on Q_{17} by considering various solutions for $Q_{-3}, \ldots, Q_2$.
- $W_{17} = W_6$ can be changed by choosing different $Q_3, \ldots, Q_7$ satisfying conditions. This will also change $W_2, \ldots, W_{10}$ that are used later on in round 2, but it does not change $Q_{-3}, \ldots, Q_2$ or $Q_8, \ldots, Q_{17}$. Thus we can easily fulfill conditions on Q_{18} by considering various solutions for $Q_3, \ldots, Q_7$.
- $W_{18} = W_{11}$ can be changed by choosing different $Q_8, \ldots, Q_{12}$ satisfying conditions. This will also change $W_7, \ldots, W_{15}$ that are used later on in round 2, but it does not change $Q_{-3}, \ldots, Q_7$ or $Q_{13}, \ldots, Q_{18}$. Thus we can easily fulfill conditions on Q_{19} by considering various solutions for $Q_8, \ldots, Q_{12}$.

To find a dBB-collision requires on average $16 + 2 \cdot 2 + 2 \cdot 2 = 24$ steps to satisfy conditions up to Q_{18} (e.g., Q_{17} requires two attempts on average, each attempt costs 2 steps: compute step 1 to find W_1 and compute step 16 to find Q_{17}.) There are $13 + 16 = 29$ conditions on $Q_{19}, \ldots, Q_{63}$ remaining and we can expect on average 2^{29} attempts for different $(Q_8, \ldots, Q_{12})$ satisfying conditions to find a solution (CV, W) that leads to a dBB-collision. With an average cost of $2^{29} \cdot ((1/2) \cdot 2 + (1/4) \cdot 4 + (1/8) \cdot 6 + \ldots) \approx 2^{31}$ steps or 2^{25} compressions.

9.6.4 Advanced message modification

Fortunately, we can do even better using advanced message modification that exploits additional conditions to obtain easy modifications beyond Q_{19} without invalidating previously satisfied conditions. As an example, let's look at the best advanced message modification for MD5.

We're going to change Q_9 into $\widehat{Q_9}$ that in turn affects steps 8,9,10,11 and 12: W_8 is used in step 27, W_9 in step 24, W_{10} in step 21, W_{11} in step 17 and W_{12} in step 31.

We'd like to change Q_{25} in step 24 but without invalidating previously satisfied conditions. However, if W_{10} and/or W_{11} is changed, then we change also $Q_{18}, \ldots$ and thereby potentially invalidate their conditions.

Let's take a closer look how W_{10} and W_{11} are changed exactly:

$$W_{10} = RL(Q_{11} - Q_{10}, 32 - RC_{10}) - Q_7 - BF_{10}(Q_{10}, \widehat{Q_9}, Q_8) - AC_{10}$$

$$W_{11} = RL(Q_{12} - Q_{11}, 32 - RC_{11}) - Q_8 - BF_{11}(Q_{11}, Q_{10}, \widehat{Q_9}) - AC_{11}$$

Clearly W_{10} and W_{11} are only changed if the boolean function output changes. As $BF_{10}(b, c, d) = BF_{11}(b, c, d) = (b \wedge c) \vee (\bar{b} \wedge d)$ selects for the i-th output bit the value $c[i]$ if $b[i] = 1$ and $d[i]$ if

$b[i] = 0$. This implies that:

$$BF_{10}(0, \widehat{Q_9}[j], Q_8[j]) = Q_8[j] = BF_{10}(0, Q_9[j], Q_8[j])$$

$$BF_{10}(1, Q_{10}[j], \widehat{Q_9}[j]) = Q_{10}[j] = BF_{11}(1, Q_{10}[j], Q_9[j])$$

For every bit $Q_9[j]$ of Q_9 that is entirely free (not involved in any sufficient conditions) such that conditions $Q_{10}[j] = 0$ and $Q_{11}[j] = 1$ do not form a contradiction with the set of sufficient conditions, we can do the following. We can add the conditions $Q_{10}[j] = 0$ and $Q_{11}[j] = 1$ to the set of sufficient conditions and thereby use the following advanced message modification at step 24 when all sufficient conditions up to are satisfied:

$$\begin{aligned}
\widehat{Q_9} &= Q_9 \oplus (0^{31-j} \| 1 \| 0^j) \\
\widehat{W_9} &= RL(Q_1 0 - \widehat{Q_9}, 32 - RC_9) - Q_6 - BF_9(\widehat{Q_9}, Q_8, Q_7) - AC_9 \\
\widehat{T_{24}} &= Q_{21} + BF_{24}(Q_{24}, Q_{23}, Q_{22}) + \widehat{W_9} + AC_{24} \\
\widehat{Q_{25}} &= Q_{24} + RL(T_{24}, RC_{24})
\end{aligned}$$

That means that if t bits fulfill the above requirements and given one solution for the sufficient conditions up to Q_{24}, we can generate 2^t different solutions that satisfy the sufficient conditions up to Q_{24}.

If from the above requirements only $Q_{10}[j] = 0$ cannot be fulfilled, one can instead use condition $Q_{10}[j] = 1$ which will force a change in W_{10} and therefore one can use such bits j for advanced message modification at step 21 (as $W_{21} = W_{10}$). To reduce time spent on steps before step 21, one can optimize the number of bits to use for advanced message modification at step 21 even if they could also be used on step 24.

This type of advanced message modification is also called a *tunnel*, and there are 6 different tunnels that change a single bit in some Q_i in the first round. Two can be used at step 20, two can be used at step 21, the other two are used at step 23 and step 24. We've already seen one for step 24 and one for step 21 whose auxiliary conditions contradict each other, meaning that for each bit position one cannot use all 6 tunnels. We leave it to the reader to find the other tunnels.

Now we can use message modification at steps 16, 17, 18, 20, 21, 23 and 24. To find a dBB-collision now requires 16 + 2*2 + 2*2 + 4*4 + 2*2 + 4*4 + 2*2 = 64 steps (1 compression function) to find a solution up to Q_{24} (up to step 23). We can then multiply solutions using the above tunnel to find 2^{23} solutions up to Q_{24}, which is enough to expect a dBB-collision. Note that using early-stop, i.e., stop as soon as a sufficient condition is not satisfied, we calculate on average about 4 steps from step 24 over the attempts. In total, the expected complexity is thus on $1 + 2^{23}(4/64) = 2^{19}$, which is significantly better than the straightforward attack with 2^{49} complexity. This really shows the advantage that we have full knowledge over all intermediate computations instead of only knowing the plaintext and ciphertext as with blockcipher cryptanalysis.

9.6.5 A full collision attack against MD5

To complete Ddn Boer and Bosselaers attack against MD5's compression function to an attack on MD5 itself, one needs to find a message block pair (B, B') such that $\delta CV = f(IV, B') - f(IV, B) = (2^{31}, 2^{31}, 2^{31}, 2^{31})$ and such that CV satisfies the sufficient conditions of dBB's attack on $Q_0, \ldots, Q_{-3}$.

With the techniques above we can find such a message block pair if we have an exact differential path that starts with IV and ends with $\delta Q_{61} = \cdots = \delta Q_{64} = 2^{31}$. Naturally, the differential path should try to use the 2^{31}-type differential steps for as many steps as possible.

A good message block difference for this can be found by reasoning from the last step towards the beginning. So we start with $\delta Q_{61} = \cdots = \delta Q_{64} = 2^{31}$ and go backwards.

Note that we know that in round 4: $\delta F_i = 2^{31}$ with probability $1/2$ and $\delta F_i = 0$ otherwise. Since $\delta W_i = 2^{31}$ and $\delta F_i = 0$ also result in $\delta Q_{i-3} = \delta T_i - \delta F_i - \delta W_i = 2^{31}$ (given that $\delta R_i = 0$ and thus $\delta T_i = 0$), in round 4 the success probability for a 2^{31}-type differential step is $1/2$ both for $\delta W_i = 0$ and for $\delta W_i = 2^{31}$. Hence, if we limit ourselves to $\delta W_i \in \{0, 2^{31}\}$, then round 4 is the same as for dBB's attack.

In round 3 we are not so fortunate: any difference $\delta W_i \neq 0$ implies $\delta Q_i \neq 2^{31}$. But we are still free to choose where to put a difference, and at the beginning of round 3 is the best that we can do. We could choose $\delta W_5 = 2^{31}$, but instead $\delta W_8 = 2^{31}$ in the end turns out to lead to a lower complexity.

Working backwards it remains a very sparse path down to step 23. From the chaining value we can work forwards and there are no differences up to step 8. To create a full differential path that completes the partial paths over steps 0,...,8 and steps 23,...,63, we can use the tools from Project HashClash that use a meet in the middle approach (see `https://marc-stevens.nl/p/hashclash/`) (i.e., the tools generate many path from above and below and tries to connect them in the middle over steps, say, 13,14,15,16.) Such a full differential path for $\delta W_8 = 2^{31}$ is given in Table 2.

The complexity of finding a message block pair (B_1, B_1') using the advanced message modifications can be estimated as:

- Steps 0-15: 16 steps;
- Step 16: has 10 conditions (8 conditions on Q_{17} and rotation probability $\approx 1/4$) so we expect 2^{10} trials, with each trail taking 2 steps (one to compute W_{16}, another to compute Q_{17}), so expected cost is 2^{11} steps;
- Step 17: has 8 conditions, so expected cost is about 2^9 steps;
- Step 18-19: has about 16 conditions on Q_{19} and Q_{20}, so expected cost is $2^{16} \cdot 4$ steps;
- Step 20: 4 conditions, thus cost is 2^5 steps;
- Step 21-22: 11 conditions, thus cost is 2^{13} steps;
- Step 23: 3 conditions (2 on Q_{24} and rotation probability $\approx 1/2$, thus cost is 2^4 steps;
- Step 24: 27 conditions left (25 conditions on $Q_{25}, \ldots, Q_{64}$ and rotation probability $\approx 1/4$), on average we compute about 4 steps using early-stop, thus expected cost is $2^{27} \cdot 4$;

Assuming we have enough advanced message modifications for step 24, which we have in this case, the complexity is dominated by $2^{27} \cdot 4$ steps $\approx 2^{25}$ compression function calls.

i	$Q_i[31]$ ⋯ $Q_i[0]$	$\Pr[\delta R_i \mid \delta T_i]$
-3	....	
-2	....	
-1	....	
0	....	1
1	....	1
2	....	1
3	....	1
4	....	1
5	....	1
6	....	1
7	.... .1.....	1
8	....0.. ..... .00.....	1
9	..... 1....0.. ..... 1+1.....	1
10	...0...0 1....-.. .....0 0+.....	1
11	.100..01 +.1..-.. 1000.000 +-.....	1
12	.11-..1+ -00^^-01 0111^10- 0+0..1.^	1
13	.-−+..-0 +1-+++10 +---++++ +-1.10^+	1
14	!0+10.0- ++1011+- ---100-- 0+-.1+-0	1
15	.1100.11 0011-001 1.100101 +010-000	0.885742
16	..0.-..0 1+..0.01 1^0...01 .111.0-.	0.279297
17	..... .-1.+.1. ..... ^.--^...	0.895508
18	....^... .10...+. 0....10. .....^.	1
19	.....0.0 .1+.^... 0....11. ..^^....	0.943359
20	.....1.1 .....^. -....++.	0.993164
21	.....-.+ ..^..... ....0...	0.987305
22	0.0..... ..... ^....^^. ....1...	1
23	1.1..^.^ ....+...	0.506836
24	-.-.....	0.629883
25	....0. ....^...	1
26	^.^..... .....1.	1
27	....+.	0.868164
28	....	1
29	0..... .....^.	1
30	....	0.483398
31	-.....	1
32	X.....	1
33	X.....	1
34	X.....	1
35	X.....	1
36	X.....	1
37	X.....	1
38	X.....	1
39	X.....	1
40	X.....	1
41	X.....	1
42	X.....	1
43	X.....	1
44	X.....	1
45	X.....	1
46	X.....	1
47	X.....	1
48	M.....	1
49	M.....	1
50	M.....	1
51	M.....	1
52	M.....	1
53	M.....	1
54	M.....	1
55	M.....	1
56	#.....	1
57	M.....	1
58	M.....	1
59	M.....	1
60	M.....	1
61	M.....	1
62	M.....	1
63	M.....	1
64	X.....	

.	$Q_i[b] \in \{0,1\}$, $Q'_i[b] = Q_i[b]$
X	$Q_i[b] \in \{0,1\}$, $Q'_i[b] \neq Q_i[b]$
0	$Q_i[b] = 0$, $Q'_i[b] = 0$
1	$Q_i[b] = 1$, $Q'_i[b] = 1$
+	$Q_i[b] = 0$, $Q'_i[b] = 1$
-	$Q_i[b] = 1$, $Q'_i[b] = 0$
^	$Q_i[b] = Q_{i-1}[b]$ ($Q'_i[b] = Q_i[b]$ for $i < 31$)
M	$Q_i[b] = Q_{i-2}[b]$ ($Q'_i[b] = Q_i[b]$ for $i < 31$)

Table 2: Differential path to obtain $\delta CV = \overline{2^{31}}$ for dBB's attack