Abstract

Hash function is one of the most important primitive of the cryptography. Knudsen and preneel provided a method of how to construct the hash function from a sepcial compression function which based on the linear error correcting codes. However, their secure claim is incorrect and several attacks are given in [12,13,14]. This project will give a brief introduction of KP construction and the two attacks: Watanabe's attack and revised Watanabe's attack. A series of results would be collected from the programs based on the two attacks and reasonable analysis will be given.

Contents

1. Introduction

Currently the cryptographic hash function is one of the most important primitives and foundations of the cryptography world. A hash function could map the data of arbitrary length into a shorter string of fixed length (usually 128 or 160 bits). There are a lot of cryptographic applications, such as digital signatures and message authentication, are based on hash functions. It could data back to the work done by Diffie and Hellman [1] that people sign the digest of the message rather than message itself.

Obviously the hash functions are very important and widely used. For those which are used in cryptographic applications we require that it is difficult to find the messages with the hash values. However, it is not easy to build such hash functions. In order to fulfill the demand of security requirements, there are three properties which should be carried out when we construct a hash function: preimage resistance, second-preimage resistance and collision resistance.

The most common method of constructing a hash function is based on two important principles. The first one call Merkle-Damgård iterative construction, which applies the secure compression function iteratively to build a collision-resistant hash function [4][5]. Another one is called the Davies-Meyer construction, which is the principle of using a blockcipher to construst the function. Matyas-Meyer-Oseas construction can be seen as one of a success example of secure hash function if the underlying blockcipher is secure [6]. A lot of hash functions we use follow the two principles of this approach to reach the required security level (such like SHA family or MDx, like MD5).

The security level of the hash function usually depends on the output length of hash functions. For a hash function with n bits length output, which means there are 2^n possible outputs, the computational complexity of finding a collision is about $2^{n/2}$ using birthday attack [3]. If we focus on the blockcipher-based compression functions [19][20] using an n-bit blockcipher, the complexity of collision attack is at most $2^{n/2}$, take AES as an example (n = 128), the computation bounds are almost unacceptable in practice.

So if we want to get a more secure hash function, one possible way is to get a longer output based on the same input and the same (small) block size. For instance, in 1990s there are several constructions which could output $2n$ bits (twice as long as those constructions before that time) and the complexity of collision finding would be 2^n instead of $2^{n/2}$ [7][8]. These constructions usually require a target ouput size (i.e. $2n$) to be fixed and the underlying compression function which should also be collision resistant to the birthday attack bound for that length.

However, Knudsen and Preneel provided a different construction: for a particular fixed security target, allow the output size change to reach the certain target without "imposing optimal security" [9][10][11].

The key of Knudsen-Preneel construction is to create a 'bigger' compression function which outputs rn bits from r chosen independent compression functions, each compressing cn-bit input into n-bit output. After that they could get the blockcipher-based hash function by following the two principles mentioned above (Merkle-Damgård and Davies-Meyer construction). However, because their security claims are mostly come from this 'bigger' compression function, Özen and Stam decided to focus on the security of the compression function itself and try to find out attacks against the KP construction. As a result, they did find several attacks which could break Knudsen and Preneel's claim [13]

More specifically, let $f_1, \ldots, f_r$ be the r compression functions used in the KP construction. The input of each function is the linear combination of the message blocks and chaining variables (each of them has n bits length). The output (rn bits length) of the construction is the concatenation of the outputs of each compression functions. The key part of this construction is how to get the inputs of the underlying compression functions $f_1, \ldots, f_r$. Here they use the generator matrix G of the corresponding $[r, k, d]$ linear code to compute the input blocks by multiplying the chaining variables (as a row vector) to the generator matrix and get the input to each compression function f_i from the product.

As a result of this construction, a change in the input of the 'bigger' compression function will influence the inputs of at least d underlying compression functions in $f_1, \ldots, f_r$. Particularly, when we use a systematic generator matrix, if two inputs to the systematic part are different, then at least d - 1 inputs to the non-systematic part would be different. So it will require time $2^{(d-1)n/2}$ to find a collision and the preimage attack needs time at least $2^{(d-1)n}$. This is exactly where Knudsen and Preneel's security claim come from.

However, Watanabe has provided a collision finding attack in his manuscript [13]. For a lot of parameter groups, this attack could beat the collision attack given by Knudsen and Preneel. More specifically, his attack works for all parameter sets with $r < 2k$ and with the complexity of $k2^n$. And the paper also showed that the security claim by Knudsen and Preneel is wrong when the parameter set has $r < 2k$ and $d > 3$. For codes with $d = 3$ Watanabe's attack performs the same as the Knudsen and Preneel's one and for codes with $r \geq 2k$ his attack does not work. Moreover, this attack could be seen as the first indication that proves the security claim of the compression function given by

Knudsen and Preneel is incorrect.

Later, another attack against the KP compression function came out (by Özen, Shrimpton and Stam [14]). They found a preimage attack works for 9 in 16 parameter sets that takes time $2^{rn/k}$. They also demonstrated an attack that could find collisions using $2^{rn/2k}$ queries with special argument.

The latest attacks to the KP compression functions are done by Özen and Stam [12] . In that paper four attacks are shown and each of them are able to beat the attack given by Knudsen and Preneel. Specifically, the first two attacks are revised version based on the work of [13] or [14] respectively. Then the last two are done by optimizing one to another so the fouth one would be the best attack so far. In fact, for only 1 in 16 sets that their best attack could not refresh the attack records. In other word, for the rest of those parameter sets their best attack performs as the best one in all of attacks against the KP function.

My work here is to analyse the KP function and to implement the collision finding attacks against it. By focusing on two of the attacks mentioned above, Watanabe's attack [13] and revised Watanabe's attack [12], I have implemented each attack, ran them on different parameter sets, recorded the results and did the analysis based on the results. For those valid outputs (collisions), I compared the query times with the theoretical results given in [13] and [12]. Then for the problems which lead to the stuck of the program and the potential bugs I have tried to fix the implemention and get a better attack. The task of this project is to understand and implement the attacks and see which one performs better in practice.

2. Background

Hash function

Any algorithm which maps data of large length (keys) into data of smaller fixed length can be called a hash function. Figure 1 is an example of hash function which maps course names (large variable length) to integers (smaller fixed length). The return values of a hash function called hash values or hashes (In cryptography, the input to hash function is usually called message and the output called digest).

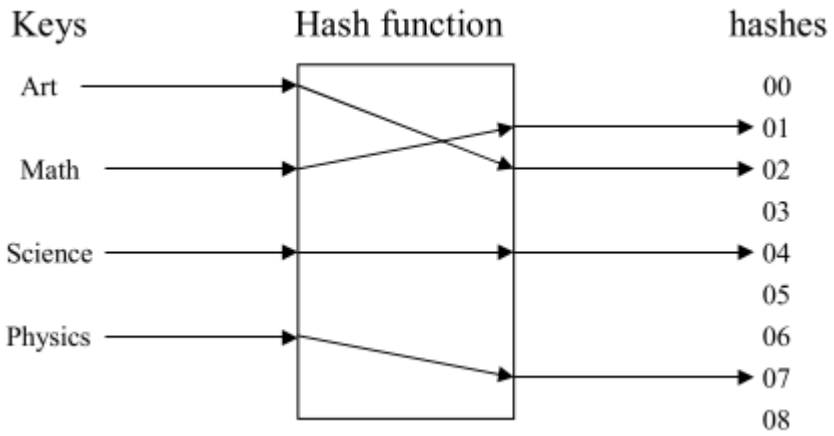

Fig. 1. Hash function which maps course names to integers 00 to 08

A cryptographic hash function should achieve the following three properties as a minimum:

- preimage resistance: for all outputs, it should be "computationally infeasible" to find any input that its hash value is the same as the output. Or for a hash value *h*, it is "computationally infeasible" to find an input value m such that Hash(*m*) = *h*.

- second preimage resistance: for any single input of all inputs, it should be "computationally infeasible" to find another input whose hash value is equal to the previous one. Or for input *m*, it should be "computationally infeasible" to find a *m*' such that Hash(*m*) = Hash(*m*').

- collision resistance: it should be "computationally infeasible" to find two different inputs whose hash value are the same. Or it should be "computationally infeasible" to find *m* and *m*' such that Hash(*m*) = Hash(*m*').

We won't describe the meaning of "computationally infeasible" here because it is not important for what we are going to discuss about and readers can find information in [2] where their definitions are givenn.

General Constructions of Hash Functions

For a compression function $f(\cdot, \cdot)$ which maps two strings of lengths l and l' into fixed length l, the most common method to construction a hash function H is to call $f(\cdot, \cdot)$ iteratively, which called Merkle-Damgård iterative construction, as follows:

$M = (M_1, M_2, \ldots, M_r)$
$H_i = f(H_{i-1}, M_i), i = 1, 2, \ldots, r$

For $H_0 = IV$ as a const value. The message need to be padded so that the length of M is a multiple of l. And the last block of the message, M_r, usually need be padded with the length of the message so that the collision finding attack can not faster than the birthday bound (which called the Merkle-Damgård strengthening).

Clearly, the security of hash function H is closely related to $f(\cdot, \cdot)$. More specifically, Merkle [4] and Damgård [5] have proved the collision resistance of H is dependent on $f(\cdot, \cdot)$. The similar conclusion for preimage resistance has been proved by Lai in [16] (also see [17][18] for one-way hash functions). So the design of trustable underlying compression functions is the key to the construction of hash functions. Usually, using block ciphers is the most common way to construct a compression function.

A block cipher is an algorithm that takes two fixed-length inputs, plaintext and key, and return an output ciphertext with the same length of plaintext. The most famous implementation of blockciphers could be Data Encryption Standard (DES) and Advanced Encryption Standard (AES).

The basic construction using blockcipher is known as the Davies-Meyer scheme which has the following form:

$H_i = E_{mi}(H_{i-1}) \oplus H_{i-1}$

It takes the message m_i as the key and the previous output H_{i-1} as the plaintext.

However, the Davies-Meyer construction have a weakness that one can easily find the output value h for any message m by applying $h = E_m^{-1}(0)$. This attack can be prevented by using the Merkle-Damgård strengthening described before. Most popular hash function including MD5 and SHA-1 take this construction. But MD5 has been broken in 2009 [15]. Table 1 shows the information of some popular hash algorithms:

Algorithm	Output size (bits)	rounds	Known attacks		
			collision	Preimage	Second-prei
MD2	128	18	Yes	Yes	No
MD4	128	48	Yes	Yes	Yes
MD5	128	64	Yes	Yes	No
SHA-0	160	80	Yes	No	No
SHA-1	160	80	Yes	No	No
SHA-256/224	256/224	64	Yes	Yes	No
SHA-512/384	512/384	80	Yes	Yes	No

Table 1: Information of 7 hash algorithms

3. Knudsen-Preneel compression function and Watanabe's attack

3.1 Knudsen-Preneel construction

Recall the main idea of Knudsen-Preneel construction we know that they want to build a hash function based on a 'bigger' compression function whose security is better than the underlying compression functions.

Let $f_i \in$ Func(cn, n) be the underlying compression functions used in the KP construction that $f_i : \{0, 1\}^{cn} \rightarrow \{0, 1\}^n$. But for the consistency of [11] and [13] we will write it as $f_i(\cdot, \cdot)$ which stands for $f_i : \{0, 1\}^{(c-1)n} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ (just observe the cn-bit input as two parts) in this section. Let M_i and H_i represent the first and second input to the function respectively, we will get the definition of multiple construction in [11]:

Definition 3: Multiple Construction

Let $f_i(\cdot, \cdot)$ be underlying compression functions which are independent to each other, then we can get the multiple construction of the target comprssion function with rate $\rho = s/r$ by:

$$H_i^1 = h_1(X_i^1 , Y_i^1)$$

$$H_i^2 = h_2(X_i^2 , Y_i^2)$$

$$\cdots$$

$$H_i^r = h_r(X_i^r , Y_i^r)$$

For X_i^j ,Y_i^j are linear combinations of H_{i-1}^j and $M_i^{j'}$ for $1 \le j \le r$ and $1 \le j' \le s$.

Security claims of [11]

Let b be a divisor of n such that $n = bn'$, $f_i : \{0, 1\}^{(c-1)n} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ be the underlying compression functions, given a $[r ,k ,d]_2^e$ linear code, then there exists a ideal compression function for which finding a collision takes at least $2^{(d-1)n/2}$ operations (notice here $c = e/b$ and $s = ck - r$).

Example 1: example of KP compression function:

Consider $H = \mathrm{KP}^1([5, 3, 3]_4)$, there exists a generator matrix G for the $[5, 3, 3]_4$ code and by defining φ as follows:

$$\varphi(0) = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix},\ \varphi(1) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},\ \varphi(w) = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \text{ and } \varphi(w^2) = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

we get

$$G\begin{bmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & w \\ 0 & 0 & 1 & 1 & w^2 \end{bmatrix} \text{ and } G' = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

And the input W is given as ordered chaining variables by the vector:

$$V = [H_{i-1}^1,\ H_{i-1}^2,\ H_{i-1}^3,\ H_{i-1}^4,\ H_{i-1}^5,\ M_i^1].$$

Then we can compute the product $V \cdot G'$ and get a vector with 10 elements. The first two correspond to the input to f_1, the third and the fourth correspond to f_2 and so on. Then we get the compression function:

$$H_i^1 = f_1(H_{i-1}^1,\ H_{i-1}^2)$$

$$H_i^2 = f_2(H_{i-1}^3, H_{i-1}^4)$$

$$H_i^3 = f_3(H_{i-1}^5,\ M_i^1),$$

$$H_i^4 = f_4(H_{i-1}^1 \oplus H_{i-1}^3 \oplus H_{i-1}^5,\ H_{i-1}^2 \oplus H_{i-1}^4 \oplus M_i^1),$$

$$H_i^5 = f_5(H_{i-1}^1 \oplus H_{i-1}^3 \oplus H_{i-1}^4 \oplus M_i^1,\ H_{i-1}^2 \oplus H_{i-1}^3 \oplus H_{i-1}^5 \oplus M_i^1)$$

The output of the compression is the concatenation of outputs of $f_1, \ldots, f_5$.

3.2 Watanabe's collision-finding attack

In [13] Watanabe has provided a differential attack which could find a collision in time $k2^n$. As the first proof of incorrectness of Knudsen and Preneel's security claim, this attack works for all parameters with $r < 2k$ and $d > 3$.

Let $f_i : \{0, 1\}^{(c-1)n} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ be the underlying compression functions and L be a $[r, k, d]_2^e$ linear code. If given input to L as $X = (X_1, \ldots, X_k)$, the corresponding output is $Y = (Y_1, \ldots, Y_r)$ for $Y_i = L_i(X_1, \ldots, X_k)$, $1 \le i \le r$. And we assume $L_i(X_1, \ldots, X_k) = X_i$, $1 \le i \le k$. Then the attack is shown as follows:

Algorithm 1 Differential attack on a KPCF

Step 1. Consider a system of linear equations

$L_i(X_1, \ldots, X_k) = 0, k + 1 \le i \le r.$ (1)

Because of the assumption $k > n-k$, this system has a non-trivial solution $\Delta = (\Delta_1, \ldots, \Delta_k)$. Fix the solution Δ, then the encoded differential is

$L(\Delta) = (\Delta_1, \ldots, \Delta_k, 0, \ldots, 0)$.

Step 2. For each function f_i $(1 \le i \le k)$, apply differential attack using differential Δ_i. In other words, calculate hash values of X and $X \oplus \Delta_i$ with the fixed Δ_i and variable Xs until $f_i(X)$ and $f_i(X \oplus \Delta_i)$ collide. Denote the collision pair of f_i by $(A_i, A_i \oplus \Delta_i)$.

Step 3. Let A is the concatenated vector consisting of A_i, i.e., $A = (A_1, \ldots, A_k)$. Then A and $A \oplus \Delta$ are colliding pair.

If we take example 1 as an example here, the idea of the attack algorithm is to find the collision input pairs to f_1, f_2, f_3 without changing the inputs to f_4 and f_5

although their inputs are computed from the inputs to f_1 , f_2 , f_3. This will be explained in the next section:

Key of the attack algorithm

Back to the algorithm, the key part of the attack is to find the solution Δ such that
$L(\Delta) = (\Delta_1, \ldots, \Delta_k, 0, \ldots, 0)$.
So we get
$f_i (L_i(A \oplus \Delta)) = f_i (A \oplus \Delta) = f_i (A)$, for $1 \le i \le k$.
And
$f_i (L_i(A \oplus \Delta)) = f_i (L_i(A) \oplus L_i(\Delta)) = f_i (L_i(A) \oplus 0) = f_i (L_i(A))$, for $k + 1 \le i \le r$.
Then A and $A \oplus \Delta$ are colliding input pair for the whole compression function.

The complexity of the attack

Apparently, step 1 and step 3 can be seen as costless, so we can concentrate on the second step. For each f_i , $1 \le i \le k$, we need to compute the output for around 2^n inputs to find a collision. Since we have to find the collision input for k underlying compression functions, the total operations are $k2^n$. So the complexity of the whole attack is about $k2^n$.

Flaw of the attack

As the first attack to the KP function, there are a few flaws can be found in this attack. The first one is that this attack only works for parameters that satisfy $k > n - k$ and $d > 3$. The requirement of $k > n - k$ is to make sure that we are able to find the solution $\Delta = (\Delta_1, \ldots, \Delta_k)$ of $L_i(X_1, \ldots, X_k) = 0$, $k + 1 \le i \le r$ (consider k unknow numbers with $n - k$ equations). So it works for only servel special parameter sets. And sometimes for a particular Δ_i , there is no X_i such that $f_i(X) = f_i(X \oplus \Delta_i)$, so we may need to change the whole Δ. Discussion in details of this situation can be found in later chapters.

4. Preliminaries for revised Watanabe's attack

In this part, I will try to keep the notation in [12] and [14] since their attacks are the latest and the revised Watanabe's attack follows the achievements in [12]. Most formalizations are for the convenience of revised Watanabe's attack in section 5. This also provides us another view of the KP compression function and the corresponding attacks.

Compression funcions.

In cryptography, a compression function $H : \{0, 1\}^{tn} \rightarrow \{0,1\}^{sn}$ is a function that maps tn-bit input to sn-bit output for positive integer parameters $t > s > 0$ and blocksize (would be explained later) $n > 0$. During this project, we usually focus on functions which map $\{0, 1\}^{cn}$ into $\{0, 1\}^{n}$, so we let Func(cn, n) be the set of all functions that compress cn bits into n bits for positive integer c. We call a compression function is a Public Random Funcion (PuRF)-based compression function if it gets its output by accessing to a number of oracles $f_1, \ldots, f_r$, where $f_1, \ldots, f_r$ are randomly picked from Func(cn, n) [14]. Here we won't consider the feedforward, the calls of the oracles are in parallel and the output would be computed based on only the results of the calls.

More specifically, for $i = 1,\ldots r$, let $C_i^{\mathrm{PRE}} : \{0, 1\}^{tn} \rightarrow \{0,1\}^{cn}$ be the preprocessing functions and $C^{\mathrm{POST}} : (\{0, 1\}^{n})^{r} \rightarrow \{0,1\}^{sn}$ be the postprocessing function, for W as the input to the PuRF-based compression function, we compute the $Z = \mathrm{H}^{f1,\ldots fr}(W)$. The process could be described as follows: for $i = 1,\ldots,r$ compute $x_i \leftarrow C_{\mathrm{i}}^{\mathrm{PRE}}(W)$ and $y_i = f_i(x_i)$, then compute $Z \leftarrow C_{\mathrm{i}}^{\mathrm{POST}}(y_1,\ldots, y_r)$. See figure 2 [14]:

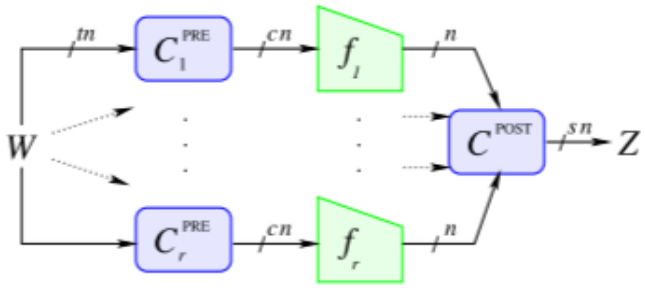

Fig. 2. General form of an *tn*-to-*sn* bit single layer PuRF-based compression function without feedforward based on *r* calls to underlying PuRFs with *cn*-bit inputs and *n*-bit outputs.

Linear error correcting codes.

A $[r, k, d]_2^e$ linear error correcting code C is a linear subspace of $\mathbb{F}_{2^e}^r$ with length r, dimension k ($r \geq k$) and minimum distance d which is the minimum Hamming weight. The dual code of a C is defined by:

$C^{\perp} = \{x \in \mathbb{F}_{2^e}^r \mid \langle x,c \rangle = 0 \ \forall \ c \in C\}$($\langle x, c \rangle = \sum_{i=1}^{n} x_i c_i$). Moreover, the lemma of Singleton bound limits the parameters by $r + 1 \geq d + k$, or focusing on d we have $d \leq r - k + 1$. Codes fit the Singleton bound can be called as maximum distance separable (MDS). An MDS code has a property that its dual code is also an MDS, then we have $d^{\perp} = k + 1$.

For a $[r, k, d]_2^e$ code C, all its codewords can be generated using a matrix G as follows: $C = \{x \cdot G \mid x \in \mathbb{F}_{2^e}^k\}$. Here G is called the generator matrix of C. A generator matrix is systematic if its form is $G = [I_k \mid P]$ where I_k is a $k \times k$ identity matrix and $P \in \mathbb{F}_{2^e}^{k \times (r-k)}$.

Special restriction of *G* by Özen and Stam

Let I be an index set $I \subseteq \{1,\ldots,r\}$, define $G_I \in \mathbb{F}_{2^e}^{k \times |\mathcal{I}|}$ as the restriction of G to the columns whose index number are in I. For a code C and any $I \subseteq \{1,\ldots,r\}$, they aim to define $\tilde{I} \subseteq \{1,\ldots, r\}$ such that $G_{\tilde{I}}$ is invertible whit $I \subseteq \tilde{I}$ or $\tilde{I} \subseteq I$, especially for $|\tilde{I}| = k$. It is easy to find such an $\tilde{I}$ for MDS codes. However, for non-MDS codes, some I does not lead to the $\tilde{I}$ we want. But for a 'target cardinality' it is possible to get an I (what they called admissible [12])

for which such an $\tilde{I}$ does exist.

Blockwise-linear schemes

Most PuRF-based compression functions has a special construction, the Knudsen-Preneel compression function is of no exception. The advantage of using blockwise is that we can get such hash functions whose security can be observed from the blocksize *n*. Here we give the definition of Blockwise-linear scheme in definition 1:

Definition 1: Blockwise-linear scheme

Given positive integer parameters *r*, *t*, *s*, *b*, *c* and matrices $C^{PRE} \in \mathbb{F}_2^{rcb \times tb}$, $C^{POST} \in \mathbb{F}_2^{sb \times rb}$, for all positive integers *n* with *b*|*n*, we set $H = BL^b(C^{PRE}, C^{POST})$ be a family of single-layer PuRF-based compression functions H_n: $\{0, 1\}^{tn} \rightarrow \{0, 1\}^{sn}$. More detailed, for $f_1, \ldots, f_r \in \mathrm{Func}(cn, n)$ and $n = bn'$, given input $W \in \{0, 1\}^{tn}$, we compute $Z = H^{f1,\ldots fr}(W)$ as follows:

1. $X \leftarrow (C^{PRE} \otimes I_{n'}) \cdot W$;
2. Parse $X = (x_i)_{i=1\ldots r}$ and $y_i = f_i(x_i)$ for $i = 1\ldots r$;
3. Parse $(y_i)_{i=1\ldots r}$ and output $Z = (C^{POST} \otimes I_{n'}) \cdot Y$.

Notice here $\otimes$ denotes the calculation of Kronecker product and $I_{n'}$ is a $n' \times n'$ identity matrix

Here the map in first step will sometimes be written as C^{PRE} and its image $\hat{S}(C^{PRE}) \subseteq \{0, 1\}^{rcn}$

Knudsen-Preneel compression function

The details of KP compression function have been shown in section 3. Here we will give another description of KP function used in [14] and [12].

Knudsen and Preneel shown a new construction of hash functions by involving the linear error correcting codes in three papers [7][8][9] and the journal version [9] would be considered as the reference frame. Their work aimed to build a blockcipher-based construction and reached an security-delivered transform. By recalling definition 1 we can see that the KP transform is actually

an instance of blockwise-linear scheme.

Definition 2: Knudsen-preneel transform

Given $G \in \mathbb{F}_{2^e}^{k\times r}$ is the generator matrix of a $[r, k, d]_2^e$ linear code. And let $\varphi : \mathbb{F}_{2^e} \rightarrow \mathbb{F}_2^{e\times e}$ denote an injective ring homomorphism and $\bar{\varphi}$ be the component-wise application. (also there is a group homorphism $\psi : \mathbb{F}_{2^e} \rightarrow \mathbb{F}_2^e$ such that $\psi(gh) = \varphi(g)\cdot\psi(h)$). b is a positive divisor of e so that $ek > rb$. Then Knudsen-Preneel compression function $H = \mathrm{KP}^b([r, k, d]_2^e)$ can be rewritten into $\mathrm{H} = \mathrm{BL}^b(\mathrm{C}^{\mathrm{PRE}}, \mathrm{C}^{\mathrm{POST}})$ for $\mathrm{C}^{\mathrm{PRE}} = \bar{\varphi}(G^T)$ and $\mathrm{C}^{\mathrm{POST}} = I_{rb}$.

From $H = \mathrm{KP}^b([r, k, d]_2^e)$ we would get $H_n : \{0, 1\}^{kcn} \rightarrow \{0, 1\}^{rn}$ and $c = e/b$ is defined for all n with $b|n$. In this project we will focus on the parameter pairs $(b, e) \in \{(1,2), (2,4),(1,3)\}$ and those parameter sets given in Knudsen and Preneel's papers.

Security claim of Knudsen-Preneel construction

Knudsen and Preneel gave their security claim of the hash function based on the corresponding compression they constructed. Under their assumption, if the KP compression function $H = \mathrm{KP}^b([r, k, d]_2^e)$, then it takes time at least $2^{(d-1)n/2}$ to find a collision in H_n (see Theorem 3 and 4 in [11]).

5. Revised Watanabe's Attack

Recall the restrictions of Watanabe's attack, it works for $k > n - k$ and has complexity $k2^n$. Following we will give an attack which reduces the complexity to $d2^n$. This attack requires $k \geq d$ (obviously if $k < d$ the new attack won't run faster than the previous one) and could find many potential collisions.

Corresponding Lemma in [12]

Given the group isomorphism $\rho : \mathbb{F}_{2^e}^{n'} \to \mathbb{F}_2^{en'}$ such that $\rho(g\delta) = (\varphi(g) \otimes I_{n'}) \cdot \rho(\delta)$ for all $g \in \mathbb{F}_{2^e}^{r}$ and $\delta \in \mathbb{F}_{2^e}^{n'}$. And same as the ring isomorphism φ we will defind the component-wise application $\bar{\rho}: \mathbb{F}_{2^e}^{n'r} \to \mathbb{F}_2^{en'r}$.

Then we can have the following lemmas:

Lemma 1 (Lemma 2 in [12])
If $g \in \mathbb{F}_{2^e}^{r}$ and $\delta \in \mathbb{F}_{2^e}^{n'}$ then $\bar{\rho}(g \otimes \delta) \in \hat{S}(C^{\mathrm{PRE}})$ if and only if $g \in C$ or $\delta = 0$.

Lemma 2 (Lemma 3 in [12])
Let n' be an integer and $V_i = \mathbb{F}_2^{en'}$ for $i = 1, \ldots, r$. Let G be the generator matrix for $[r, k, d]_2^e$ code. $\tilde{I} \subseteq \{1, \ldots, r\}$ such that $G_{\tilde{I}}$ is invertible, the inverse is $G_{\tilde{I}}^{-T}$. For all $i \in \tilde{I}$ given $x_i \in V_i$, we will get $\tilde{X} = \sum_{i \in \tilde{I}} x_i$. Then

$$W = (\bar{\varphi}(G_{\tilde{I}}^{-T}) \otimes I_{n'}) \cdot \tilde{X}$$

is the only input that satisfies for $i \in \tilde{I}$, $x_i' = x_i$ for $X' = C^{\mathrm{PRE}}(W)$.

Revised Watanabe's Attack

The key part of Watanabe's attack is to find the solution Δ of $L_i(X_1, \ldots, X_k) = 0$, for
$k + 1 \leq i \leq r$. However, in this new attack, we will calculate a new Δ based on a codeword $g \in C$ and a nonzero multiplier δ by compute $\Delta = \bar{\rho}(g \otimes \delta)$. If we choose the codeword g with a lower weight, the attack would performs much better.

What we noticed here is that Watanabe's algorithm allows to find a codeword g such that $\chi(g) \subseteq \{1, \ldots, k\}$ so we can find a collision in the systematic part of the code and extend it to a full collision of the function. In the new attack, if we pick a random codeword g, with a high probability that $\chi(g)$ may not map to the systematic part. However, the lemma 2 before indicates that $I = \chi(g)$ is admissible to complete a full collision. So we get theorem 1 [12] :

Theorem 1
Given $H = \mathrm{KP}^b([r, k, d]_2{}^e)$ with $d \leq k$. The algorithm 1 will find collisions for H_n in time $d2^n$ by using a codeword g with weight up to k and a nonzero δ.

Algorithm 1 (Revised Watanabe Collision Attack).

Input: $H = \mathsf{KP}^b([r, k, d]_{2^e})$ satisfying $d \leq k$, a nonzero $g \in \mathcal{C} \subseteq \mathbb{F}_{2^e}^r$ with $|\chi(g)| \leq k$, a block size $n = bn'$, and an arbitrary nonzero $\delta \in \mathbb{F}_{2^e}^{n'}$.

Output: A colliding pair $(W, W') \in \left(\{0,1\}^{ekn'}\right)^2$ such that $H_n(W) = H_n(W')$, $W \neq W'$ and $C^{\mathrm{PRE}}(W) \oplus C^{\mathrm{PRE}}(W') = \bar{\rho}(g \otimes \delta)$.

1. Initialization. Compute $\Delta \leftarrow \bar{\rho}(g \otimes \delta)$, set $\mathcal{I} \leftarrow \chi(g)$ and determine $\tilde{\mathcal{I}} \supseteq \mathcal{I}$ for which $G_{\tilde{\mathcal{I}}}$ is invertible.
2. Query Phase. For $i \in \mathcal{I}$ do
   a. Generate a random $x_i \xleftarrow{\$} V_i (= \mathbb{F}_2^{en'})$ and set $x_i' \leftarrow x_i \oplus \Delta_i$;
   b. Query $y_i \leftarrow f_i(x_i)$ and $y_i' \leftarrow f_i(x_i')$;
   c. If $y_i = y_i'$ then keep (x_i, x_i') and proceed to next i, else return to a.
3. Degrees of Freedom. For $i \in \tilde{\mathcal{I}} \backslash \mathcal{I}$ pick $x_i \xleftarrow{\$} V_i$ and set $x_i' \leftarrow x_i$.
4. Finalization. Output (W, W') where

$$W \leftarrow (\bar{\varphi}(G_{\tilde{\mathcal{I}}}^{-T}) \otimes I_{n'}) \cdot (\sum_{i \in \tilde{\mathcal{I}}} x_i) \qquad \text{and} \qquad W' \leftarrow (\bar{\varphi}(G_{\tilde{\mathcal{I}}}^{-T}) \otimes I_{n'}) \cdot (\sum_{i \in \tilde{\mathcal{I}}} x_i') \,.$$

6. The program

The two attack programs attached (see appendix for part of the codes) are implements of the two attack algorithms: Watanabe's attack and revised Watanabe's attack. The programs follow the key idea of the attack with limited and reasonable changes to make them run more effectively and intelligently.

Foundations

The package *linercod*e (in both programs) contains the implements of all operations of the matrix. The basic operations are mostly based on binary. Operations over other vector can be change into binary with the ring and group homomorphism in chapter 4 and 5.

The *functions* package contains all functions that might be used in the attack. For both programs, the CFs and blockcipher file work together as the underlying compression functions. Because the attack is totally unrelated to the construction of the underlying compression functions, so the implements can be very simple. The Phi and PhiTransform file perform as the injective ring homomorphism mentioned above. For revised Watanabe's attack, there are several more files need to be mentioned here. The Rho file contains the group isomorphism used in the attack algorithm. The Codeword file have an algorithm that could generate a codeword based on the given generator matrix. This is for the user's convenience because it can be difficult to compute the codeword for some large code.

For Watanabe's attack program, the *attack* package contains the necessary steps of the attack. First, the Delta file contains the function that return a vaild Δ as described in the algorithm (can be seen as the step 1 of the algorithm). The file named Attack is the whole attack process of the algorithm (step 2 and 3). More specifically, for $i = 0$ to $k - 1$ (which equals 1 to k in the algorithm) the function will take a random x_i and $x_i \oplus \Delta_i$ until $f(x_i) = f(x_i \oplus \Delta_i)$. However, for some particular situation (like for all x_i that $f(x_i)$ and $f(x_i \oplus \Delta_i)$ are not collided) we need to change the Δ and apply the attack again. So the function in AttackOne file is to make sure we can get a vaild output (collision pair).

For revised Watanabe's attack, there are four files in the attack package. The Delta file contains a function that take the codeword (generated from funtion in Codeword file) and returns a vaild Δ based on it. Notice this is the Δ in the revised algorithm (different from the previous one). The QueryPhase file performs as the 2 and 3 (query and degrees of freedom) step in the algorithm which is similar to the step in Watanabe's attack. The difference is there are d loops instead of k loops and the Δ is different. The Finalization file is the 4th

step in algorithm as its name. The AttackOne file has the same function of the previous one with the same name.

Running Time and Query times Counter

Since we want to compare the theory with the partice, it is necessary to record the time and query times during the attack. So each time we call the attack() function in attack.java file, we start a time recoder and set the query counter to 0. The time recoder won't stop until the attack() method ends. The query counter plus one for every call of the underlying compression function. When the program return a pair of vaild collision input, it will output the time and queries used in the attack.

The results of the programs can be found in the next chapter.

Problems

There is a significant problem during the implementing of the program. Recall both attack algorithms, the key step of collision finding is, given a fixed Δ, for all x_i and $x_i' = x_i \oplus \Delta_i$ apply $y_i = f(x_i)$ and $y_i' = f(x_i')$ until y_i and y_i' collide. This step implies that for a fixed Δ_i there exists a collision pair x_i and $x_i' = x_i \oplus \Delta_i$ in the underlying compression function $f()$. However, because both KP construction and the attack algorithms assume that the underlying functions are ideal and the attacks do not contains the construction attack to $f()$, the existence of such a collision for particually Δ_i can not be guaranteed. This may lead to the dead loop of the program because usually we pick x_i as random. To avoid this issue, there are two method can be considered. The first one is to use the exhaustive search method to go through every x_i and if the collision can not be found we step back and change the Δ. Another possible method is to set an upper bound of query times. When the search time reaches this bound we consider it as no collision can be found and change the Δ. If we set the bound as 2^{cn} (the input size of the underlying compression functions) then both methods would perform the same. So in this project the second method would be employed temporally.

Because of both solutions require changing the Δ and the two attack algorithms give their attack complexity based on the assumption that the whole algorithm uses one particular Δ, we need to reset the time recoder and the query counter every time we change the Δ. So it is possible that it takes a long time to output a collision pair but the recorded running time is shorter than the actual running time. But this won't influence the judgement of both algorithms because the query times will also be used for the evaluation.

7. Results of the program

From [13] and [12], the complexities of the Wanatabe's attack and revised Wanatabe's attack are both given. What we actually care about is whether their theoretical and practical attack complexities coincided. So in this chapter we will see some groups of result of the programs based on the two attack algorithms with different parameter sets and message lengths.

Recall the complexities of Wanatabe's attack and revised Wanatabe's attack are $k2^n$ and $d2^n$ respectively. It is obviously that the cost of the attack is related to the parameters k, d and the blocksize n. So the choices of vector space's size (e), codeword length (r) and other less related parameters won't influence the judgement of whether the attack is effective or not.

In this chapter we list the results with the parameter sets in the form $[r, k, d]_2^e$ for $[5,3,3]_2^2$ code, $[8,5,3]_2^2$ code, $[9,5,4]_2^2$ code and $[16,12,4]_2^2$ code with blocksize n = 4, 8, 12 and 16. And we will keep the analysis of the results to the next chapter.

The choice of the parameter sets are based on the following consideration: first, it is necessary to choose parameters with $k > n - k$ (to make sure Watanabe's attack works) or $d \leq k$ (making sure revised Watanabe's attack works); Second, it is easy to extend the program from GF(2) or GF(2^2) to other vector spaces (just change the injection φ for different parameter e); However, it is difficult to collect a large number of data for analysis for large blocksize n (especially for $n > 16$) under the computation capability of PCs or laptops. If we use AES as the underlying blockcipher, it might take hours even days to return one collision of the corresponding compression function. However, what we want here is just to implement and test the attack on 'artificially small n' (as mentioned in project description), so we only choose those blocksizes with n = 4, 8, 12 and 16 for the convience of collecting data and analysising the results.

For each parameter set, we list at least one collision pair of the compression function as the evidence of a successful attack which can be checked and ten groups of attack time. However, the execution time is easily effected by the running environment (like different configurations of the computer or other softerwares running at the same time), we also list the number of how many query operations for the underlying compression function are made during the attack as one of the considerable results. To be clear, the coolision pair below indicates the collided input to the full compression function. For instance, for $[5,3,3]_2^2$ code and n = 4, the input length is ckn = 2*3*4 = 24 bits. Notice we will express Δ and X in hexadecimal string for results which are too long to display in binary.

7.1 Results of Watanabe's Attack

The results for $[5,3,3]_2^2$ code

For $[5,3,3]_2^2$ code, we have

$$G=\begin{bmatrix}1 & 0 & 0 & 1 & 1\\ 0 & 1 & 0 & 1 & w\\ 0 & 0 & 1 & 1 & w^2\end{bmatrix} \text{ and } G'=\begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0\\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1\\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1\\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0\\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1\\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1\end{bmatrix}$$

①. $n = 4$.

Result 1:

Δ = 00000000100010001000100 0

Coolision pair:

X = 010101011100011110100111 and

X'= 010101010100111100101111

Running time:
1325587 ns

Operations
28 queries for underlying comperssion function

Result 2:

Δ = 00000000110011001100110 0

Coolision pair:

X = 010000011011100110011011 and

X'= 010000010111010101010111

Running time

1615568 ns

Operations
34 queries for underlying comperssion function

②. n = 8.

Result 3:

Δ =0000000000011011000110110000000000001101100011011

Coolision pair:

X= 1100011101011110111111011101001100111111111010001

and

X'=1100011101000101111001101101001100100100110010 10

Running time
11941182 ns

Operations
290 queries for underlying comperssion function

Result 4:

Δ =0000000011100011000000001110001100000000000000000

Coolision pair:

X = 0100010101110110000011011011001011101010111011

and

X'= 0100010110010101000011010101000111101010111011

Running time
372674 ns

Operations
51 queries for underlying comperssion function

③. n = 12.

Result 5:

Δ=0x00000313b13b13b13f

Coolision pair:

X =0x87165281695649ecd7 and

X'=0x871652e31f712b9af0

Running time
94103859 ns

Operations

1366 queries for underlying comperssion function

④. n = 16.

Result 6:

Δ=0x00007aab7aab00007aab7aab

Coolision pair:

X =0xb794a23282f973d9a8fef898 and

X'=0xb794d899f85273d9d2558233

Running time
1374257901 ns

Operations

111317 queries for underlying comperssion function

Other results can be seen in Table 2 and Table 3.

Groups	blocksize $n = 4$	$n = 8$	$n = 12$	$n = 16$
	Running time (ns)	(ns)	(ns)	(ns)
1	1325587	11941182	94103859	1374257901
2	220869	1502426	62191424	3731643267
3	2709283	17713703	115616929	812877055
4	1282566	11022071	259017557	3353509454
5	1385092	14268853	75967755	2514280167
6	2549207	13085182	179566118	1455172934
7	795632	12810567	292913485	584576531
8	250869	12627024	67283691	2262891792
9	2726603	12460802	130956588	1158148312
10	1826768	11457982	138752830	2195239898
Average (10 groups)	1507247.6	11888979.2	141637023.6	1944259731
Average (100 groups)	1238208.8	9028125.5	132549586.9	2079938642.0

Table 2. Watanabe's attack running time results with $[5,3,3]_2^2$ code and $n = 4$, 8, 12 and 16

	blocksize $n = 4$	$n = 8$	$n = 12$	$n = 16$
Attempts	Queries	Queries	Queries	Queries
1	28	290	1366	111317
2	34	248	2726	291152
3	64	613	6998	65044
4	22	503	16575	268579
5	30	799	3564	203804
6	56	446	12383	111075
7	13	499	21846	47225
8	35	663	2994	176435
9	63	567	8278	87174
10	43	535	9602	173257
Average (10 groups)	38.8	516.3	8633.2	153506.2
Average (100 groups)	42.7	606.4	10950.6	168782.4

Table 3. Watanabe's attack query times with $[5,3,3]_2^2$ code and $n = 4$, 8, 12 and 16

The results for $[9,5,4]_2^2$ code

For $[9,5,4]_2^2$ code we have

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & w \\ 0 & 1 & 0 & 0 & 0 & w & w^2 & 1 & w \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & w^2 & w \\ 0 & 0 & 0 & 1 & 0 & w & 1 & 0 & w^2 \\ 0 & 0 & 0 & 0 & 1 & 0 & w^2 & 1 & w^2 \end{bmatrix}$$

And G'=

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

①. $n = 4$.

Result 7:

Δ = 111000000000000011101110111000000000000

Coolision pair:

X = 111110110111100101111100111101100000000 and

X'= 000110110111100110010010000101100000000

Running time:

1394870 ns

Operations

54 queries for underlying comperssion function

Result 8:

Δ = 10110000000000001011011011000000000000

Coolision pair:

X = 11111000001100000101111010110010111111000 and

X'= 01001000001100001110010100000010111111000

Running time:

2645867 ns

Operations

100 queries for underlying comperssion function

②. *n* = 8.

Result 9:

Δ=0x8e0000008e8e8e000000

Coolision pair:

X =0xe4b6761ce4316c1ef7cb and

X'=0x6ab6761c6abfe21ef7cb

Running time:

792280 ns

Operations

137 queries for underlying comperssion function

③. *n* = 12.

Result 10:

Δ=0x375000000000375375375000000000

Coolision pair:

X =0x04952a5c095a04996da2e521ed1469 and

X'=0x33c52a5c095a33ca1895b521ed1469

Running time:

169634967 ns

Operations

13929 queries for underlying comperssion function

④. *n* = 16.

Result 10:

Δ=0x8baf0000000000008baf8baf8baf000000000000

Coolision pair:

X =0x03e02f0df3624e1787cc02f40cf5145e23067c55 and

X'=0x884f2f0df3624e170c63895b875a145e23067c55

Running time:

169634967 ns

Operations

13929 queries for underlying comperssion function

Other resuls can be seen in Table 4 and 5.

Groups	blocksize n = 4	n = 8	n = 12	n = 16
	Running time (ns)	(ns)	(ns)	(ns)
1	1394870	15166425	169634967	1454506089
2	1260496	10106592	23091203	4109703049
3	1542375	22946544	379756086	2919155927
4	193042	15262555	327808702	4646274112
5	758756	11762948	93764711	7799585702
6	378539	11845080	321600092	6487190302
7	570743	16247037	90013674	4546975663
8	405080	11288586	454251309	4074450804
9	866311	13062275	230740779	1351583511
10	348648	17644231	169784427	2598543111
Average (10 groups)	771886.0	14533227.3	226044595.0	3998796827
Average (100 groups)	811762.4	12554847.2	215746882.9	3457682141.6

Table 4. Watanabe's attack running time results wiht $[9,5,4]_2^2$ code and n = 4, 8, 12 and 16

	blocksize n = 4	n = 8	n = 12	n = 16
Attempts	Queries	Queries	Queries	Queries
1	54	964	13929	107161
2	122	562	1904	295503
3	128	1602	29317	218333
4	27	1267	26547	332671
5	17	613	6936	575273
6	43	738	24564	476182
7	82	1251	7055	321874
8	56	826	33909	289591
9	74	861	18066	99059
10	33	1352	12523	186723
Average (10 groups)	63.6	1003.6	17475.0	290237
Average (100 groups)	69.2	1097.1	18876.2	309782.0

Table 5. Watanabe's attack query times with $[9,5,4]_2^2$ code and n = 4, 8, 12 and 16

Below some results for other parameter sets will be shown. Specially, the results for $[16,12,4]_2^2$ code is sightly more important than others because it's fit the condition of $d > 3$.

Groups	blocksize n = 4	n = 8	n = 12	n = 16
	Running time (ns)	(ns)	(ns)	(ns)
1	962413	11812395	903379340	6142646063
2	443073	46761530	865324529	4031405389
3	865753	13705373	303487734	15288131033
4	1486502	21292371	334869381	9244700933
5	574374	19939964	284425255	6631596473
6	613765	21761425	684719147	18475332200
7	881676	24596143	976243620	8513602707
8	336914	12433983	206754490	11994216304
9	1287315	16369678	1020807673	6904349017
10	601752	39715383	382022576	7076474524
Average (10 groups)	805353.7	22838824.5	596203374.5	9430245464.3
Average (100 groups)	854423.1	25518567.2	523176234.4	8945326548.4

Table 6. Watanabe's attack running time results with $[16,12,4]_2^2$ code and n = 4, 8, 12 and 16

	blocksize $n = 4$	$n = 8$	$n = 12$	$n = 16$
Attempts	Queries	Queries	Queries	Queries
1	131	718	62607	449740
2	111	5114	67427	287685
3	157	1507	21318	1101924
4	282	2207	23412	681808
5	159	1234	18425	473917
6	179	1662	43876	1271372
7	152	2120	77125	623033
8	71	2540	14598	864184
9	109	1445	71734	474617
10	170	5177	25613	517565
Average (10 groups)	152.1	2372.4	42613.5	674584.5
Average (100 groups)	161.4	2552.5	45786.2	705547.1

Table 7. Watanabe's attack query time with $[16,12,4]_2^2$ code and $n = 4$, 8, 12 and 16

Groups	blocksize $n = 4$	$n = 8$	$n = 12$	$n = 16$
	Running time (ns)	(ns)	(ns)	(ns)
1	543924	10751366	657946471	2196119339
2	180470	7687010	193307250	5243622024
3	312051	24035736	149077606	1037508119
4	4540800	12816992	114619036	1424996194
5	646730	2338007	142860336	2182842791
6	4945042	2192737	161427779	1264907691
7	388597	24338568	167824961	3773910372
8	457600	14437310	121249489	3202156978
9	453968	10401601	622428270	6003224712
10	354793	11982249	134401998	5457425176
Average (10 groups)	1282397.5	12098157.6	246514319.6	3178671340

Table 8. Watanabe's attack running time results with $[8,5,3]_2^2$ code and $n = 4$, 8, 12 and 16

	blocksize $n = 4$	$n = 8$	$n = 12$	$n = 16$
Attempts	Queries	Queries	Queries	Queries
1	94	399	40133	146083
2	26	129	13705	381052
3	49	2236	11003	70649
4	108	489	7660	95819
5	88	383	10096	153967
6	125	261	9861	87845
7	48	2113	11963	279993
8	65	746	8843	228286
9	38	313	38763	378180
10	55	505	9286	408175
Average (10 groups)	69.6	757.4	16131.3	223004.9

Table 9. Watanabe's attack query times with $[8,5,3]_2^2$ code and $n = 4$, 8, 12 and 16

7.2 Results of Revised Watanabe's Attack

The results for $[5,3,3]_2^2$ code

①. $n = 4$.

Result 1:

Δ = 10101000000000001010100000000010001010

Coolision pair:

$\sum_i x_i$ = 000100010100110010010100 and

$\sum_i x_i^{'}$ = 101110011110010000011110

W = 000100010100100101001100 and

W' = 101110011110001111100100

Running time:

2968813 ns

Operations

69 queries for underlying comperssion function

②. n = 8.

Result 1:

Δ=00000000000000001010100010101000000000000000000101

01000101010001010100010101000

Coolision pair:

$\sum_i x_i$=11111111111001000100110011101111111011101000111

and

$\sum_i x_i^{'}$=01010111010011001110010001000111010001111110111 1

W = 11100100111111111111111111100100010101111111010 0

and

W'= 01001100010101110101011101001100111111101011100

Running time:

4267023 ns

Operations

353 queries for underlying comperssion function

③. n = 12.

Result 1:

Δ=0x00088a88a00000000088a000000000

Coolision pair:

$\sum_i x_i$ = 0x7f98fe07497b074e97 and

$\sum_i x_i^{'}$ = 0x7f90748fe97b8fee97

W = 0x7f98fe07497b7f9f12 and

W'= 0x7f90748fe97b7f9798

Running time:

138732995 ns

Operations

9215 queries for underlying comperssion function

④. n = 16.

Result 1:

Δ=0xa8222280a8222280000000000000008aa22280

Coolision pair:

$\sum_i x_i$ = 0xebb22135969039378c625dad and

$\sum_i x_i^{'}$ = 0x439003b53eb21bb706c07f2d

W = 0xebb2213596903937227ff140 and

W'= 0x439003b53eb21bb7227f7be2

Running time:

1834765871 ns

Operations

91835 queries for underlying comperssion function

More detailed results can be seen in Table 10 and 11.

Groups	blocksize $n = 4$	$n = 8$	$n = 12$	$n = 16$
	Running time (ns)	(ns)	(ns)	(ns)
1	2968813	1380064	82696824	7253029645
2	4041575	4267023	138732995	784347452
3	784457	12689881	265714258	6730154810
4	402565	15668404	205678920	1834765871
5	910553	6644700	128469122	3350578510
6	695619	11372554	195562247	1828295217
7	1433701	16308257	95383632	1710787544
8	616559	5034159	186077840	3269671704
9	4210591	35025680	185652147	4165727208
10	571860	13934731	372904657	3862045852
Average (10 groups)	1663629.3	12232545.3	185687264.2	3478940381.3
Average (100 groups)	11476253.2	10451176.1	197387621.9	2829768230.2

Table 10. Revised Watanabe's attack running time results with $[5,3,3]_2^2$ code and $n = 4$, 8, 12 and 16

	blocksize $n = 4$	$n = 8$	$n = 12$	$n = 16$
Attempts	Queries	Queries	Queries	Queries
1	69	115	5731	342524
2	33	353	9215	40934
3	61	745	12825	322790
4	40	977	9753	91835
5	23	658	7005	172612
6	24	707	11709	96708
7	31	892	5966	85817
8	71	596	8465	164607
9	37	680	9535	224959
10	23	852	16890	242782
Average (10 groups)	41.2	657.5	9709.4	178556.8
Average (100 groups)	44.7	682.4	10846.4	185237.5

Table 11. Revised Watanabe's attack query times with $[5,3,3]_2^2$ code and $n = 4$, 8, 12 and 16

The results for $[9,5,4]_2^2$ code

①. $n = 4$.

Result 1:

Δ=1000100000000000100010000000000000000000010001000100010001000100000000000

Coolision pair:

$\sum_i x_i$ = 10001111010000101100011111000111100111111 and

$\sum_i x_i^{'}$ = 00000111110010100100111101001111000101111

W = 10001111010000001000010001011011001011 0 and

W'= 00000111101010001100101000100101000111110

Running time:

597841 ns

Operations

32 queries for underlying comperssion function

②. $n = 8$.

Result 1:

Δ=0x000022a8000000008aa8a88a22a8a88a0000

Coolision pair:

$\sum_i x_i$ = 0x3943524040d80c964c6b and

$\sum_i x_i^{'}$ = 0x1bebd8e8e8522e3ee4e1

W = 0x27393943510028c75240 and

W'= 0x271b1bebf9000ae5d8e8

Running time:

3892673 ns

Operations

461 queries for underlying comperssion function

③. n = 12.

Result 1:

Δ=0x00000000000088800000000000000888000000000008888
88000

Coolision pair:

$\sum_i x_i$ = 0xa1e617c7a928c7ad29d5c4f24f2653

and

$\sum_i x_i^{'}$ = 0xa1e6174f29284f2d29d5cc7ac7a653

W = 0xa1e617000928c7a928000000000c7a and

W'= 0xa1e6170009284f29280000000004f2

Running time:

104892382 ns

Operations

7612 queries for underlying comperssion function

④. n = 16.

Result 1:

Δ=0x0000000022aaa888000000000000000000000022aaa888a
88822aa22aaa88822aaa888

Coolision pair:

$\sum_i x_i$ = 0x77a98eca037ce8d4e4c703031be22a2f22d6ff07

and

$\sum_i x_i^{'}$ = 0x5503264221d6405c4c4f21a9394882a7007c578f

W = 0x88ae189e77a98eca0000000000000000000088ae

and

W'= 0x028c189e55032642000000000000000000028c

Running time:

7348061276 ns

Operations

444830 queries for underlying comperssion function

More results can be seen in Table 12 and 13

Groups	blocksize n = 4	n = 8	n = 12	n = 16
	Running time (ns)	(ns)	(ns)	(ns)
1	1028064	8005487	104892382	1531522861
2	597841	28401654	195992927	3100538755
3	1084775	3892673	97398412	2723043273
4	519061	6729905	165942053	8951065569
5	801778	14427811	144603875	7348061276
6	333283	3225549	306679453	3538869884
7	266515	11847036	120637610	2452553327
8	824406	7171023	153844701	7069063805
9	2288279	13393881	254087068	3690583300
10	196394	11418490	272103578	5546564996
Average (10 groups)	794039.6	10851350.9	181618205.9	4595186705.6
Average (100 groups)	802547.1	12764980.4		4465870198.2

Table 12. Revised Watanabe's attack running time results with $[9,5,4]_2^2$ code and n = 4, 8 and 16

	blocksize $n = 4$	$n = 8$	$n = 12$	$n = 16$
Attempts	Queries	Queries	Queries	Queries
1	128	694	7612	89459
2	32	610	17612	171868
3	89	461	4889	153430
4	52	776	12612	500988
5	25	1436	15204	444830
6	35	359	22058	183247
7	36	1482	8225	133781
8	86	754	14203	364256
9	51	1563	19957	200767
10	15	1401	13574	338369
Average (10 groups)	54.9	953.6	13594.6	258099.5
Average (100 groups)	57.7	976.9		260764.8

Table 13. Revised Watanabe's attack query times with $[9,5,3]_2^2$ code and $n = 4$, 8 and 16

8. Analysis and Evaluation

In this chapter we will give reasonable analysis to the results collected in the previous chapter and try to fit the theoretical result given in the attack algorithms. Because the program is tested with parameter sets $[5,3,3]_2^2$ code, $[8,5,3]_2^2$ code, $[9,5,4]_2^2$ code and $[16,12,4]_2^2$ code and blocksize n = 4, 8, 12 and 16, the analysis will first focus on these data groups. After that we will try to extend the analysis to a more general situation.

8.1 Results Analysis

For Watanabe's attack, we have the theoretical attack bound that the algorithm can find a collision using about $k2^n$ query operations. Then, for theoretical results compared with the average practical results we have the following table:

	$n = 4$			$n = 8$		
k	Theory $k2^n$	Average	difference	Theory $k2^n$	Average	difference
3	48	42.7	11.04%	768	606.4	21.04%
5	80	69.2	13.50%	1280	1097.1	14.29%
12	192	161.4	15.94%	3072	2552.5	16.91%
average			13.49%			17.41%

Table 14. Comparison of theoretical and partical results of Watanabe's attack

	$N = 12$			N = 16		
k	Theory $k2^n$	Average	difference	Theory $k2^n$	Average	difference
3	12288	10950.6	10.88%	196608	168782.4	14.15%
5	20480	18876.2	7.83%	327680	309782.0	5.46%
12	49152	42613.5	13.30%	786432	705547.1	10.29%
average			10.67%			9.97%

Table 14 Cont.

For revised Watanabe's attack with attack bound $d2^n$ we have

	$n = 4$			$n = 8$		
d	Theory $k2^n$	Average	difference	Theory $k2^n$	Average	difference
3	48	44.7	6.88%	768	682.4	11.15%
4	64	57.7	9.84%	1024	976.9	4.60%
average			8.36%			7.87%

Table 15. Comparison of theoretical and partical results of Revised Watanabe's attack

	$n = 12$			$n = 16$		
d	Theory $k2^n$	Average	difference	Theory $k2^n$	Average	difference
3	12288	10846.4	11.73%	196608	185237.5	5.78%
4	16384		.	262144	260764.8	0.53%
average			11.73%			3.16%

Table 15. Cont.

As can be seen here, for both attacks, the practical results are sightly different form the theoretical ones. However, these differences are acceptable in practice due to the following possible reasons.

First, the average difference between the theory bounds and the practical results is about 11%; this is not very high for the sample size of practical results is only 100. As can be seen above, when the average results of 100 groups are more close to the theoretical values than that of 10 groups. It is predictable that if the sample size goes larger the results would be more close to the theory.

Second, for Watanabe's attack (and revised attack), the theoretical bound is linear dependent to the parameter k (d) and exponential dependent to the blocksize n. So we will try to analyse the dependence between the results and the parameters.

Linear dependence

For Watanabe's attack algorithm, given a fixed blocksize n, the theoretical complexity is $k2^n$ which can be seen as constant value $c * k$. For instance, $n = 4$,

the complexity equels to 16*k. Some linear analysis of the result sets will be shown below:

When n = 4, for k = 3, 5, 12, the practical values are 42.7, 62.9 and 161.4 respectively. So we get the following linear function between k and the results: y = 13.184*x+ 3.2045.

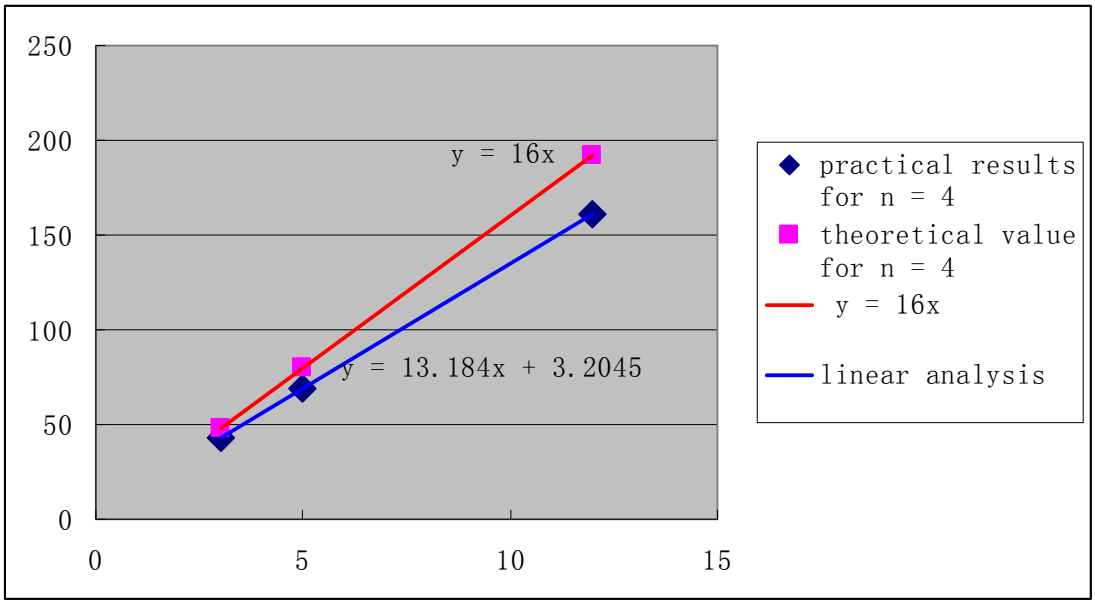

Figure 3. Scatter diagram for results with n = 4, 2^4 =16, and k = 3, 5, 12.

Also apply the analysis for other sets:

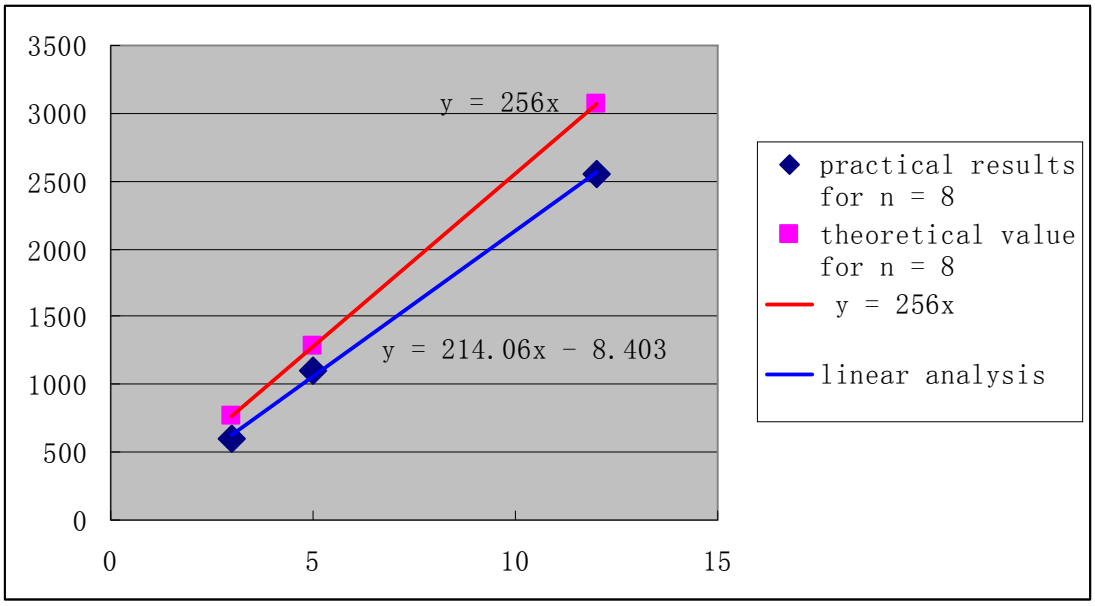

Figure 4. Scatter diagram for results with n = 8, 2^8 =256 and k = 3, 5, 12.

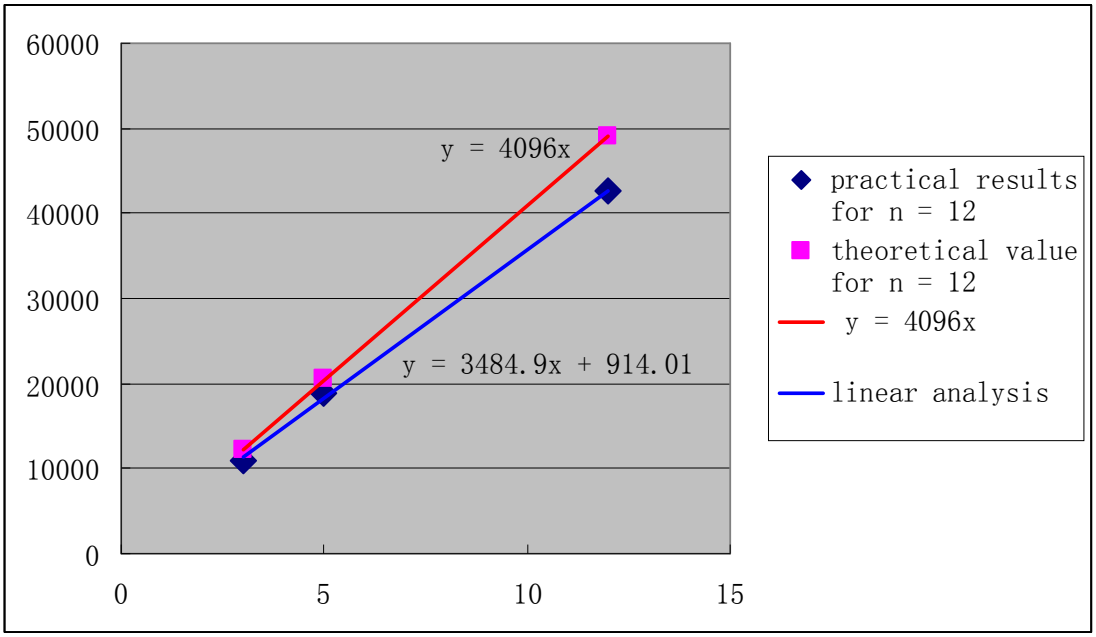

Figure 5. Scatter diagram for results with $n = 12$, $2^{12} = 4096$ and $k = 3, 5, 12$.

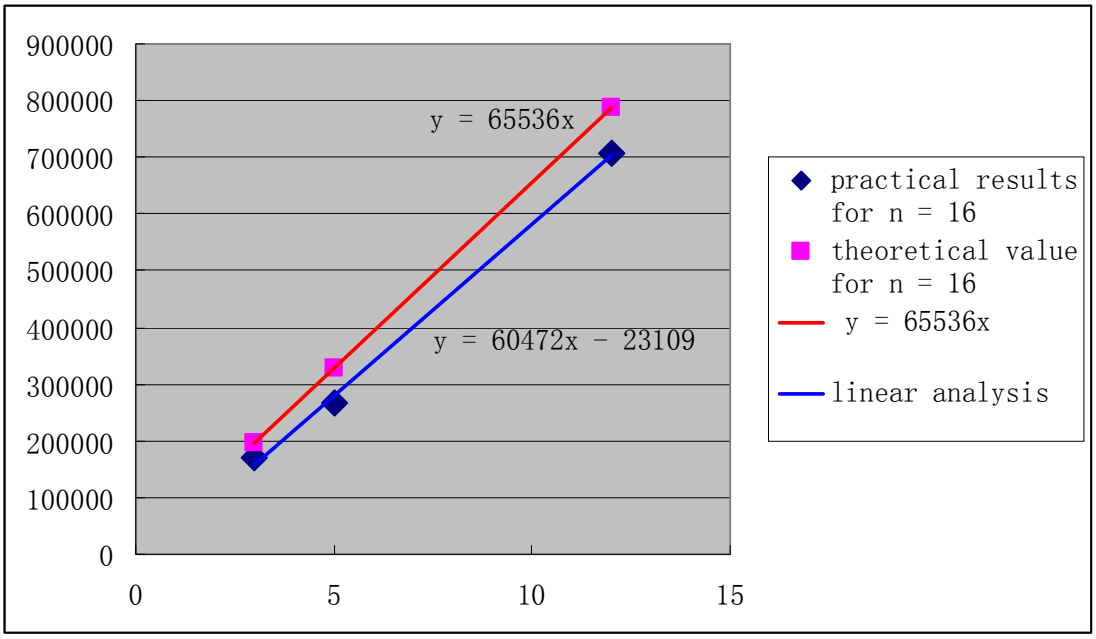

Figure 6. Scatter diagram for results with $n = 16$, $2^{16} = 65536$ and $k = 3, 5, 12$.

As can be seen from the previous figures, the test values of each set do linearly increse with k. The slope of the line is sightly different from the slope of the line draw by the corresponding theoretical result. So for fixed blocksize n, the attack algorithm can be seen as linear dependent on the factor k (replace k by d would lead to the same conclusion for revised Watanabe's attack).

Next we will focus on the exponential dependence between the test results and the blocksize n. For instance, in revised Watanabe's attack, let $d = 3$ be fixed, the query complexity equals to $3*2^n$. For $n = 4$, 8, 12 and 16 the practical results are 44.7, 682.4, 10846.4 and 185237.5 respectively.

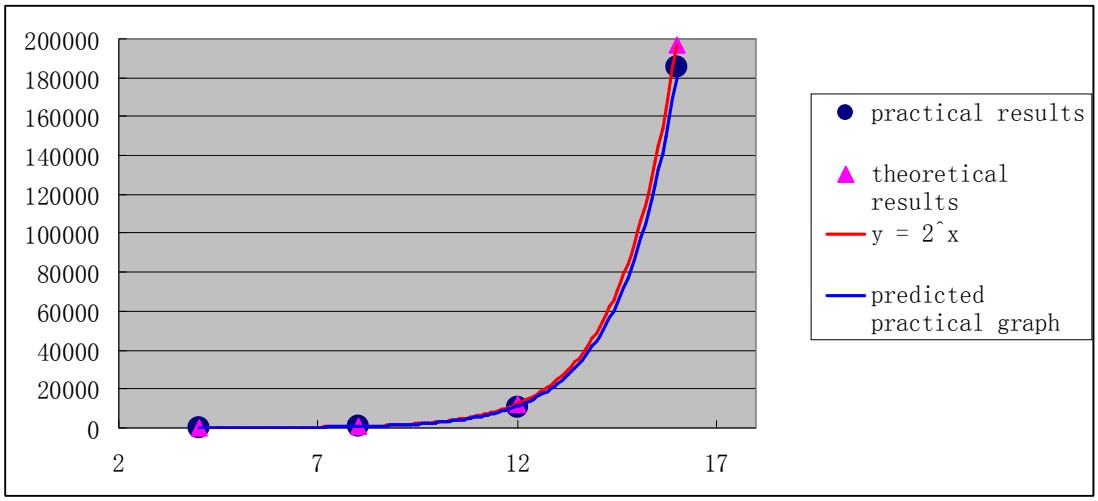

Figure 7. Scatter diagram for results with and $d = 3$ and $n = 4$, 8, 12 and 16

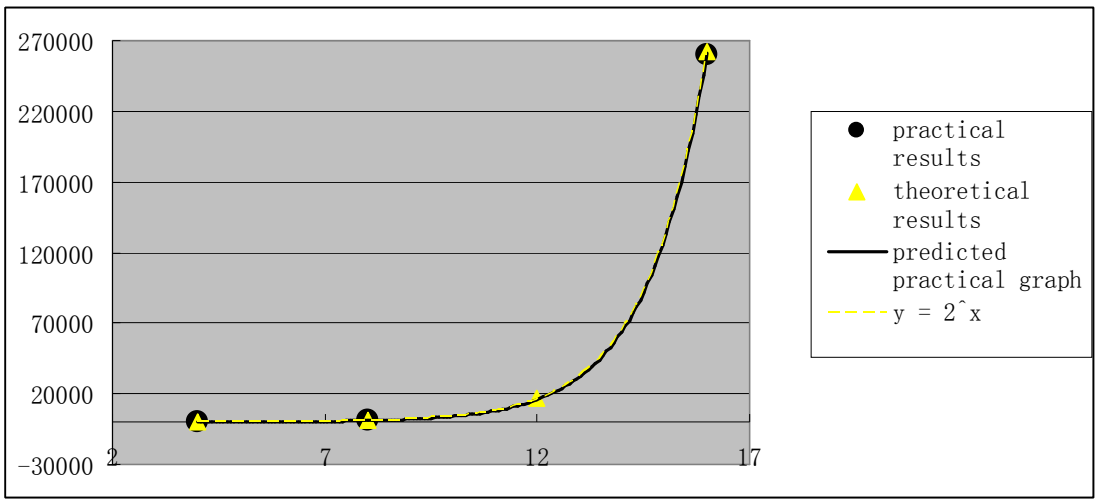

Figure 8. Scatter diagram for results with and $d = 4$ and $n = 4$, 8, 12 and 16

Clearly the predicated practical graph and the graph for function $y = 2^n$, which indecates the theoretical results, coincide very well. So we might be able to come to the conclusion that the practical and the block length n are exponential dependent. Further conclusion may require more mathematical analysis which is not the purpose of this project, so we will just stop here.

Besids the analysis of the dependence, another task here is to try to find out which attack performs better in practice. Here the result sets would be divided into two groups: $d = k$ and $d < k$.

For $d = k = 3$($[5,3,3]_2^2$ code), the test reults are as follows and we compute W/R equals to the average queries of Watanabe's attack divided by that of Revised attack.

	$n = 4$			$n = 8$		
	Watanabe's attack	Revised attack	W/R	Watanabe's attack	Revised attack	W/R
k=d=3	42.7	44.7	95.53%	606.4	682.4	88.86%
k=5,d=4	69.2	57.7	119.93%	1097.1	976.9	112.30%

Table 16. Comparison of Watanabe's attack and Revised Watanabe's attack

	$n = 12$			$n = 16$		
	Watanabe's attack	Revised attack	W/R	Watanabe's attack	Revised attack	W/R
k=d=3	10950.6	10846.4	100.96%	196608	185237.5	106.14%
k=5,d=4	18876.2		.	309782.0	260764.8	118.80%

Table 16. Cont.

The average of W/R for *k=d=3* is 97.87%, this indecates that when $k = d$ the query times of both attack are very close. The average for *k=5,d=4* is 117.16% which implies that when $k > d$ the Watanabe's attack requires more queries. So we can come to the conclusion that when $k = d$ the Watanabe's attack algorithm performs the same as the revised attack; however, when $k > d$ the revised Watanabe's attack algorithm performs better than Watanabe's attack.

8.2 Evaluation

The first task of this project is to understand and implement the attack algorithms of Knudsen-Preneel compression function. The content in chapter 3, 4 and 5 contains the description and analysis of the KP construction and the attack algorithm and can been seen as the proof of understanding the attack. The programs attached which implement the Watanabe and revised Watanabe's attack are the proof of implements. Because both attack programs can find the collision of the KP-constructed compression function in or about the given complexity bound, this task can be considered as achieved successfully. Moreover, based on the analysis above, the revised Watanabe's attack can be

seen as the better attack algorithm. This is an achievement of task two. For the potential risk that given a fixed Δ there might exist no x_i such that $f(x_i)$ and $f(x_i \oplus \Delta_i)$ collide, two possible solutions are given and both can solve the problem with reasonable cost.

So basically most of the objectives of the project have been finished. Althrough for some part (like the data analysis) the work is not good enough because of the lack of experience, the project is still accomplished as a whole.

9. Further work

The use of linear code to build a compression function is really a creative construction of compared with most common cunstructions. Although Knudsen and Preneel compression function has been proved not as secure as it was claimed to be, it still provides us a new thought to create new hash functions. So the following possible further work could be considered.

Strengthen the Knudsen-Preneel construction

The KP compression function can be attacked because the linear code leak the information of the input to the underlying compression function so that people could find the full collision based on the collisions of part of the compression function. It might be possible to hide this leak by using additional operations like permutation.

Implement other attack to the KP function

Besides the attack algorithms mentioned in this project, there are also some other attack to the KP compression function such like the preimage finding attack in [14]. It is worth to try to implement other attacks and test them on different parameter sets to find out which is the best attack to the KP function. Moreover, there might more attack algorithms be found during the implementing process.

10. bibliography

1. W. Diffie and M. E. Hellman, “New directions in cryptography,” IEEE Trans. Inform. Theory, vol. IT-22, pp. 644–654, Nov. 1976.

2. J. Black, P. Rogaway, and T. Shrimpton, “Black-box analysis of the block-cipherbased hash-function constructions from PGV,” Advances in Cryptology, CRYPTO 2002, Springer-Verlag, LNCS 2442, pp. 320–335, 2002.

3. A. Menezes, P. van Oorschot, and S. Vanstone, Handbook of Applied Cryptography, CRC-Press, 1996.

4. R. Merkle, “One way hash functions and DES,” Advances in Cryptology, CRYPTO ’89, Springer-Verlag, LNCS 435, pp. 428–446, 1990.

5. I. Damg˚ard, “A design principle for hash functions,” Advances in Cryptology, CRYPTO ’89, Springer-Verlag, LNCS 435, pp. 416–427, 1990.

6. S. M. Matyas, C. H. Meyer, and J. Oseas, “Generating strong one-way functions with cryptographic algorithm,” IBM Tech. Discl. Bull., vol. 27, no. 10A, pp. 5658–5659, 1985.

7. Knudsen, L., Muller, F.: Some attacks against a double length hash proposal. In: Roy, B.K. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 462–473. Springer, Heidelberg, 2005.

8. Knudsen, L., Muller, F.: Some attacks against a double length hash proposal. In: Roy, B.K. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 462-473. Springer, Heidelberg, 2005.

9. Knudsen, L.R., Preneel, B.: Hash functions based on block ciphers and quaternary codes. In: Kim, K., Matsumoto, T. (eds.) ASIACRYPT 1996. LNCS, vol. 1163, pp. 77–90. Springer, Heidelberg, 1996.

10. Knudsen, L.R., Preneel, B.: Fast and secure hashing based on codes. In: Burt Kaliski, J., Burton, S. (eds.) CRYPTO 1997. LNCS, vol. 1294, pp. 485–498. Springer, Heidelberg, 1997.

11. Knudsen, L.R., Preneel, B.: Construction of secure and fast hash functions using nonbinary error-correcting codes. IEEE Transactions on Information Theory 48(9), 2524–2539, 2002.

12. Onur Özen and Martijn Stam. Collision Attacks against the

Knudsen-Preneel Compression Functions. In Masayuki Abe, editor, Advances in Cryptography—Asiacrypt 2010, 2010.

13. Watanabe, D.: A note on the security proof of Knudsen-Preneel construction of a hash function (2006) (unpublished manuscript), http://csrc.nist.gov/groups/ST/hash/documents/WATANABE_kp_attack.pdf

14. Özen, O., Shrimpton, T., Stam, M.: Attacking the Knudsen-Preneel compression functions. In: Hong, S., Iwata, T. (eds.) FSE 2010. LNCS, vol. 6147, pp. 94–115. Springer, Heidelberg, 2010.

15. Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, Benne de Weger: MD5 considered harmful today: Creating a rogue CA certificate, 2009.

16. X. Lai, "On the design and security of block ciphers," in ETH Series in Information Processing, J. L. Massey, Ed. Konstanz, Germany: Hartung-Gorre Verlag, 1992.

17. M. Bellare and P. Rogaway, "Toward making UOWHF's practical," in Advances in Cryptology, Proc. Crypto'97 (Lecture Notes in Computer Science), B. Kaliski, Ed. Berlin, Germany: Springer-Verlag, 1997, vol. 1294, pp. 470–484, 1997.

18. M. Naor and M. Yung, "Universal one-way hash functions and their cryptographic applications," in Proc. 21st ACM Symp. Theory of Computing, 1989, pp. 387–394, 1989.

19. Preneel, B., Govaerts, R., Vandewalle, J.: Hash functions based on block ciphers: A synthetic approach. In: Stinson, D. (ed.) CRYPTO 1993. LNCS, vol. 773, pp. 368–378. Springer, Heidelberg, 1994.

20. Stam, M.: Blockcipher-based hashing revisited. In: Dunkelman, O. (ed.) Fast Software Encryption. LNCS, vol. 5665, pp. 67–83. Springer, Heidelberg, 2009.

11. Appendix

Finding Δ in Watanabe's attack

```
public String findSingleDelta()
    {
        String str = "";
        String temp = "";
        String result = "";
        Random ran = new Random();


        if(k*n <=16)
        {
            r = r*n;
            k = k*n;
            Matrix id = new
Matrix(n);
            p = p.KroProduct(id);
            Matrix delt = new
Matrix(1,k);
            Matrix zero = new
Matrix(1,r-k);
            int test =
ran.nextInt((int)Math.pow(2,k)) ;
            if(test == 0)
            {
                test ++;
            }
            str =
Integer.toBinaryString(test);
            while(str.length()<k)
            {
                str = "0"+str;
            }
            for(int i = 0; i <
str.length(); i++)
            {
                temp =
str.substring(i,i+1);
                delt.setElement(0,
i, Integer.parseInt(temp));
            }
            Matrix product =
delt.multiply(p);


    while(!product.compare(zero))
            {
                test =
ran.nextInt((int)Math.pow(2,k));
                if(test == 0)
                {
                    test ++;
                }
                if(test >=
Math.pow(2,k))
                {

    System.out.println("delta not
found");
                    return null;
                }
                str =
Integer.toBinaryString(test);

    while(str.length()<k)
                {
                    str = "0"+str;
                }
                for(int i = 0; i <
str.length(); i++)
                {
                    temp =
str.substring(i,i+1);

    delt.setElement(0, i,
Integer.parseInt(temp));
                }
                product =
delt.multiply(p);

            }

            result = "";
```

```
    if(product.compare(zero))
            {
                for(int l = 0; l < k;
l ++)
                {
                    result +=
Integer.toBinaryString(delt.getE
lement(0, l));
                }
                return result;
            }else{
                return null;
            }
        }else{
            Matrix delt = new
Matrix(1,k);
            Matrix zero = new
Matrix(1,r-k);
            int test =
ran.nextInt((int)Math.pow(2,k));
            if(test == 0)
            {
                test ++;
            }
            str =
Integer.toBinaryString(test);

    //System.out.println(str);
            while(str.length()<k)
            {
                str = "0"+str;
            }
            for(int i = 0; i < k; i++)
            {
                temp =
str.substring(i,i+1);
                delt.setElement(0,
i, Integer.parseInt(temp));
            }

            Matrix product =
delt.multiply(p);
    while(!product.compare(zero))
            {
                test =
ran.nextInt((int)Math.pow(2,k));
                if(test == 0)
                {
                    test ++;
                }
                if(test >=
Math.pow(2,k))
                {

    System.out.println("delta not
found");
                    return null;
                }
                str =
Integer.toBinaryString(test);

    while(str.length()<k)
                {
                    str = "0"+str;
                }
                for(int i = 0; i <
str.length(); i++)
                {
                    temp =
str.substring(i,i+1);

    delt.setElement(0, i,
Integer.parseInt(temp));
                }
                product =
delt.multiply(p);

            }

            result = "";


    if(product.compare(zero))
            {
                int in =
```

```
ran.nextInt((int)Math.pow(2, n));
                String str1 =
Integer.toBinaryString(in);

    while(str1.length()<n)
                {
                    str1 = "0"+str1;
                }
                String str0 =
Integer.toBinaryString(0);

    while(str0.length()<n)
                {
                    str0 = "0"+str0;
                }
                for(int l = 0; l < k;
l ++)
                {

    if(delt.getElement(0, l) == 0)
                    {
                        result +=
str0;
                    }else
if(delt.getElement(0, l) == 1)
                    {
                        result +=
str1;
                    }
                }

    //System.out.println(delt.toS
tring());
                return result;
            }else{
                return null;
            }
        }
    }
```

The collision finding process of Watanabe's attack:

```
while(flag)
{
    x=ran.nextInt((int)Math.pow(2
    ,n));
    y = x^delta.getElement(0, i);
    String xstr =
    Integer.toBinaryString(x);
    String ystr =
    Integer.toBinaryString(y);

    while(xstr.length()<n)
    {
            xstr = "0"+xstr;
    }
    while(ystr.length()<n)
    {
            ystr = "0"+ystr;
    }

    if(p.encode8(xstr).equals(p.e
    ncode8(ystr)))
    {
            flag = false;
            str1 += xstr;
            str2 += ystr;
    }
}
counter ++;
if (counter >= Math.pow(2, n))
{
    System.out.println("no
    collisions found in f"+i);
    String[] a = null;
    return a;
}
```

Function that used for finding codeword in revised attack

```
public Matrixs
findCodeword(Matrixs p, int e)
    {
        int row = p.getRow();
        int column = p.getColumn();
        if(row%2!=0||column%2!=0)
        {

    System.out.println("generator
matrix G size error");
            return null;
        }

        Matrixs G = new
Matrixs(row,row+column);
        Matrixs I = new
Matrixs(row);
        G.setMatrixElement(I, 0,
0);
        G.setMatrixElement(p, 0,
row);

    //System.out.println(G.toStri
ng());


        int k = row/2;

        Matrixs rv = new
Matrixs(1,k);
        long seed =
System.nanoTime();

    //System.out.println(""+seed)
;
        Random ran = new
Random(seed);
        boolean flag = true;
        while(flag)
        {
            for(int i = 0; i < k; i++)
            {
                rv.setElement(0, i,
ran.nextInt((int)Math.pow(2,e)))
;
                if(rv.getElement(0,
i)!=0)
                {
                    flag = false;
                }
            }
        }

    //System.out.println(rv.toStr
ing());

        Rho rho = new Rho(e);
        Matrixs rhoRv =
rho.change(rv);


    //System.out.println(rhoRv.to
String());

        Matrixs rhoResult =
rhoRv.multiply(G);


    //System.out.println(rhoResul
t.toString());

        Matrixs result =
rho.changeBack(rhoResult);

        return result;
    }
```

Find Δ in revised Watanabe's attack

```
public Matrixs findDelta()
    {
        long seed =
System.nanoTime();
        Random ran = new
Random(seed);
        int mul =
ran.nextInt((int)Math.pow(2, n));
        if(mul==0)
        {
            mul++;
        }
        String mulStr =
Integer.toBinaryString(mul);
        while(mulStr.length()<n)
        {
            mulStr = "0" + mulStr;
        }
        Matrixs mulMatrix = new
Matrixs(1,n);
        for(int i = 0; i < n; i++)
        {
            String s =
mulStr.substring(i, i+1);
           mulMatrix.setElement(0,
i, Integer.parseInt(s, 2));
        }

    //System.out.println(mulMatri
x.toString());

        Rho rho = new Rho(2);
        Matrixs result =
rho.changeR(g, mulMatrix);
        return result;
    }
```