

A Practical Platform for Cube-Attack-like Cryptanalyses

CS 758: Cryptography/Network Security Course Project

Bo Zhu, Wenye Yu and Tao Wang
{bo.zhu, wenye.yu, t55wang}@uwaterloo.ca

University of Waterloo

Abstract. Recently, various cryptanalysis methods related to Cube Attack have attracted a lot of interest. We designed a practical platform to perform such cryptanalysis attacks. We also developed a web-based application at <http://cube-attack.appspot.com/>, which is open to public for simple testing and verification. In this paper, we focus on linearity testing and try to verify the data provided in several papers. Some interesting results produced in our work indicate certain improper assumptions were made in these papers.

1 Introduction

Cube Attack was announced by Dinur and Shamir in 2008 [1], and published at Eurocrypt'09 [2]. Cube Attack was first designed to search linear expressions of the secret keys in stream ciphers. In Cube Attack, a cipher can be treated as a black box and expressed in Boolean functions. By manipulating the inputs of certain public variables of the cipher, the attacker hopes to get enough information to solve the linear Boolean equations to recover the secret keys. The idea behind Cube Attack can be found in several previous works, such as Algebraic IV Differential Attack [3,4] (AIDA) and Higher Order Differential Attack [5]. Later, the idea of Cube Attack has been extended to other aspects of cryptanalysis, not only limited to the linearity of stream ciphers. For example, the authors of [6] detected certain non-random behaviour of the hash function MD6 [7], and in [8] the authors modified Cube Attack to find quadratic equations.

A stream cipher consists of several shift registers and update rules for the registers. At each step the rules are applied once to update the registers, and the registers are shifted once to give an output. Typically, the encryption process of stream ciphers is simply XORing the string of plaintext with the successive outputs of the ciphers. Trivium is a stream cipher designed by De Cannière and Preneel in 2005 [9]. Due to its compact design, Trivium has served as an experimental target to perform several cryptanalysis methods. For instance, Vielhaber presented AIDA against Trivium with 576 initialization rounds [3]. Cube Attack was also performed to break Trivium with 735 initialization rounds [2]. Aumasson *et al.* augmented Cube Attack with property testers (calling it Cube Tester) which can detect non-randomness of Trivium with 790 rounds [6].

In this paper, we emphasize on the linearity testing for Trivium, which is the most commonly studied situation. We implemented an usable platform to perform such Cube-Attack-like cryptanalysis methods mentioned above. Based on the platform, we wrote an open web-application such that every one interested in this topic can easily perform some analyses. We also tried to verify the data provided in several papers, and this verification produces rather interesting results, for which discussions are given.

The next section briefly explains the mathematical background about Boolean algebra, Trivium, Cube Attack and AIDA. Section 3 includes the detailed descriptions of our cryptanalysis platform and website application. The verification results and discussions are given in Section 4. Section 5 concludes the paper and discusses some further works.

2 Preliminaries

In this section, we will first introduce the definition of Boolean functions. Based on Boolean functions, the details of the stream cipher Trivium and two attacks on Trivium – Cube Attack and AIDA, will be given. Finally, we will briefly discuss linearity testing.

2.1 Boolean Function

Let \mathbb{F}_2^n denote the n -dimension vector space over $\mathbb{F}_2 = \{0, 1\}$. The following definition is from [10].

Definition 1. *A Boolean function of n variables is a map from \mathbb{F}_2^n to \mathbb{F}_2 .*

Boolean functions can always be expressed as multi-variable polynomials over \mathbb{F}_2 in Algebraic Normal Form (ANF), as

$$f(x_0, x_1, \dots, x_{n-1}) = \sum_{\alpha \in \mathbb{F}_2^n} c_\alpha x_0^{a_0} x_1^{a_1} \cdots x_{n-1}^{a_{n-1}},$$

where $\alpha = (a_0, a_1, \dots, a_{n-1})$ and the coefficient c_α is a constant in \mathbb{F}_2 . Boolean functions in ANF consist of only two operations, addition modulo 2 (i.e. XOR) and multiplication modulo 2 (AND). The ANF of a Boolean function is also unique [11]. The Boolean functions in this paper are all written in this way.

2.2 Specification of Trivium

A stream cipher can be viewed as an algorithm which produces a stream of bits based on some public initial values and secret keys. Each of these output bits can be seen as being produced by a Boolean function f_i . Trivium is designed to generate up to 2^{64} bits of key stream from an 80-bit secret key and an 80-bit initial value (IV).

The step function of Trivium uses a 288-bit internal state, denoted as (s_1, \dots, s_{288}) . The key stream generation process is given as the following algorithm.

```

for  $i = 1$  to  $N$  do
   $t_1 \leftarrow s_{66} + s_{93}$ 
   $t_2 \leftarrow s_{162} + s_{177}$ 
   $t_3 \leftarrow s_{243} + s_{288}$ 
   $z_i \leftarrow t_1 + t_2 + t_3$ 
   $t_1 \leftarrow t_1 + s_{91} \cdot s_{92} + s_{171}$ 
   $t_2 \leftarrow t_2 + s_{175} \cdot s_{176} + s_{264}$ 
   $t_3 \leftarrow t_3 + s_{286} \cdot s_{287} + s_{69}$ 
   $(s_1, s_2, \dots, s_{93}) \leftarrow (t_3, s_1, \dots, s_{92})$ 
   $(s_{94}, s_{95}, \dots, s_{177}) \leftarrow (t_1, s_{94}, \dots, s_{176})$ 
   $(s_{178}, s_{279}, \dots, s_{288}) \leftarrow (t_2, s_{178}, \dots, s_{287})$ 
end for

```

For the initialization process, the key and IV are loaded into the 288-bit initial state, and all the remaining bits, except s_{286} , s_{287} , and s_{288} , are set to 0. This is illustrated as below

$$\begin{aligned} (s_1, s_2, \dots, s_{93}) &\leftarrow (K_1, \dots, K_{80}, 0, \dots, 0) \\ (s_{94}, s_{95}, \dots, s_{177}) &\leftarrow (IV_1, \dots, IV_{80}, 0, \dots, 0) \\ (s_{178}, \dots, s_{285}, s_{286}, s_{287}, s_{288}) &\leftarrow (0, \dots, 0, 1, 1, 1) \end{aligned}$$

Before outputting the key stream bit z_i , the internal state is rotated for 4 cycles ($N = 4 \times 288 = 1152$ steps) by the same method described above.

2.3 Cube Attack

In Cube Attack, we consider the problem of predicting the 1-bit output of a function, f , based on its m -bit public input value $V = \{v_1, v_2, \dots, v_m\}$ and the n -bit secret key $K = \{k_1, k_2, \dots, k_n\}$; for convenience let $X = V \cup K$.

Suppose I is certain a subset of V . The function f can be written in the following form.

$$f(X) \equiv t_I \cdot p_{S(I)} + q_I(X),$$

where we have

- I is called the *cube*;
- t_I is the multiple of all variables whose indices are in I ;
- $p_{S(I)}$ is called the *superpoly*;
- q_I contains any and all terms that are not divisible by t_I .

Our objective is to find representations of the above form where $p_{S(I)}$ is a linear polynomial of the variables in K , the key; each linear polynomial gives us 1-bit information about K . When $p_{S(I)}$ is such a linear polynomial, we call t_I a maxterm.

The main observation that allows us to make use of this linear polynomial is as follows.

Theorem 1. $p_{S(I)} \equiv \sum_{v \in C_I} p$ modulo 2, where C_I is the set of all possible value assignments to the values indexed by I .

A proof is presented in [2]. This theorem gives us access to the linear polynomial corresponding to any maxterm; that is to say, if we are able to find such a representation, we will be able to use the linear polynomial in the keys to get 1 bit of information about the key.

2.4 AIDA

Algebraic IV Differential Attack (AIDA) is presented first in [3] and later in [4] as a powerful tool to analyze stream ciphers. In AIDA, the Boolean functions of stream ciphers are written as

$$f(v_1, v_2, \dots, v_n, \kappa) = \bigoplus_{I \subseteq \{1, 2, \dots, n\}} a_I(\kappa) v_I^\wedge,$$

where v_1, v_2, \dots, v_n are the bits of IV, and κ denotes the vector of secret key bits. The notation v_I^\wedge is similar to t_I in Cube Attack, and $a_I(\kappa)$ is the coefficient of the monomial v_I^\wedge .

The fundamental principle of AIDA is the following equation.

$$a_I(\kappa) = \bigoplus_{M \subseteq I} d_M,$$

where d_M denotes the corresponding entries in the truth table when the IV bits whose indices are in the set M are set to one, and the other bits are assigned zero. Thus, $2^{|I|}$ queries of the function f are enough for calculating the value of $a_I(\kappa)$.

We think the main difference between Cube Attack and AIDA is how they treat the indices not in the cubes. In Cube Attack, the variables outside the cubes are left undetermined¹, because their values do not influence the final output, i.e. $p_{S(I)}$. On the other hand, in AIDA, these variables are assigned zero. This small difference will eventually result in big gaps in theoretical and practical analyses. We will give more discussions on this in Section 4.1.

¹ Please see the second line on the sixth page of [2].

2.5 Linearity Testing

By the standard Blum-Luby Rubinfeld (BLR) linearity testing [12], a polynomial f is linear in its inputs if we have

$$f(0) + f(x) + f(y) + f(x + y) = 0 \text{ for all } x, y$$

Suppose that we test the linearity of a function f by uniformly choosing random input values x and y . It is of interest to consider how likely it is that this method demonstrates that a function is not linear. Suppose f is not linear and contains some term $x_1x_2\dots x_i$. Then this term evaluates to 1 with chance 2^{-i} (where all of the variables in the term are taken to be 1) and 0 with chance $1 - 2^{-1}$. However, note that when i increases, the function containing this term also tends to be more linear in the sense that this non-linear term rarely appears. A tight bound on the relationship between the linearity of the function and how often the BLR linearity testing rejects is given in [13]. One interesting result shown in the paper can be presented as follows.

Theorem 2. *Let $\text{Dist}(f) = \min_g \{Pr_u[f(u) \neq g(u)]\}$, where g is a linear function using the same domain and range as f . Then, using the BLR test, the rejection probability of BLR is at least as large as $\text{Dist}(f)$.*

In other words, the rejection probability of BLR is at least the *non-linearity* of the function concerned. This demonstrates that BLR is a decent test.

3 Implementations of the Platform and Web-based Application

In this section, we will show how we implemented the practical platform for Cube-Attack-like cryptanalyses, and discuss different kinds of linearity testing algorithms used in the platform. At last, a brief description of the design of our cryptanalysis website will be given.

3.1 A Practical Cryptanalysis Platform

We provide a platform to perform the Cube-Attack-like cryptanalyses. It is an implementation of the two attacks on Trivium in practice. The program can be divided into two parts: the part for generating Trivium key stream employs the C source code with an interface for external invocation, and the main part of the platform is realized in Python code, which enables the portability of the platform. This platform can be further decomposed into four components:

1. *Higher Order Differential* calculates the higher order derivatives needed to perform the actual attack methods, such as AIDA or Cube. The difference between AIDA and Cube lies in their attack process, which is illustrated in Section 2.4. Both of the methods are tested in our work.
2. *Property Testing* detects the properties, such as balance, low degree, linearity, and presence of a variable, on the original polynomial. The property testing in the project is mainly focused on linearity testing.
3. *Ciphers* invokes external libraries written in C to gain efficiency in the computation of stream ciphers. There are two versions of libraries adopted: one imports `ctypes` and the other imports `Python.h`. The version with `Python.h` is used in the actual implementation, as it allows Python methods to be invoked directly in C programming and thus it is faster.
4. *Main Controller* acts as the attack control that communicates with outsider.

Each of these four components performs different roles independently, and thus are reusable in other applications. The use of script language Python as our platform language is because of its simplicity and portability, which is an obvious advantage in our website design in Section 3.3.

An overview of the platform code organization is illustrated in Fig. 1.

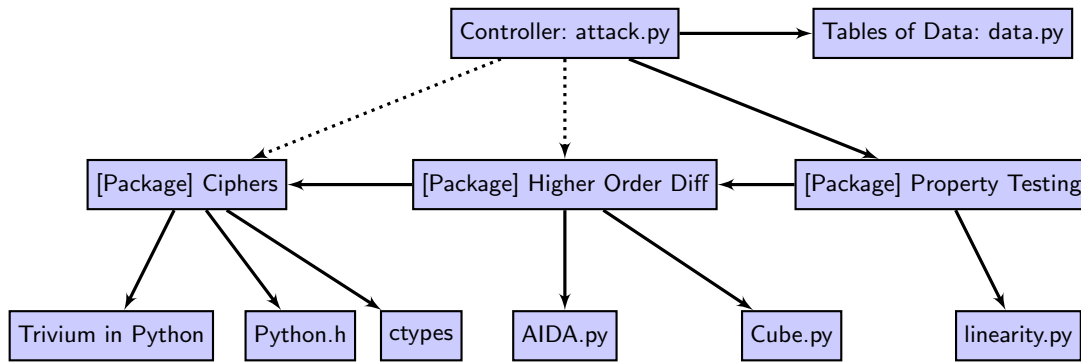


Fig. 1. Organization of the source code for the platform.

3.2 Linearity Testing Algorithms

One of the important goals of this project is to verify the experimental results of attacks on the stream cipher Trivium stated in related papers. Therefore in this section, the objective will be to test whether or not a Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ is linear, and if so, we want to discover the expression of f . The theorem of linearity testing is mentioned in Section 2.5. An algorithm strictly following the definition is stated as Algorithm 1. If the equation $f(0) + f(x) + f(y) + f(x + y) = 0$ holds for (sufficiently large) C times, then we consider f as a linear function. And after passing the linearity testing, the algorithm will continue to output the indices of the linear variables. There is a small trick that $f(z) + f(0) = 1$ implies the i -th input bit serves as a linear variable in f , which is like applying AIDA to a 1-dimension cube. However, this textbook algorithm has some drawbacks. For example,

Algorithm 1 Standard Linearity Testing

```

for  $c = 1$  to  $C$  do
    Randomly choose two input values  $x$  and  $y$ 
    if  $f(0) + f(x) + f(y) + f(x + y) \neq 0$  then
        Reject and halt
    end if
end for
for  $i = 1$  to  $n$  do
     $z \leftarrow (0, 0, \dots, 1, \dots, 0)$  where only the  $i$ -th bit is 1
    if  $f(z) + f(0) = 1$  then
        Output  $i$ 
    end if
end for

```

for each trail of the testing, we should generate two random inputs, and evaluate the function f three times ($f(0)$ can be evaluated only once, and stored for later use).

During the verification process, the linear variables are usually provided and need to be tested carefully. Define a function $h(S)$ to be the LSB of the set S 's Hamming weight, i.e. $h(S) = 0$ when the number of non-zero variables in the set is even; $h(S) = 1$ when odd. Then we design the following Algorithm 2.

It is easy to see that if f is a linear function in terms of the variables in S , then $f(x) + h(S) = f(0)$ will always hold. One disadvantage of this algorithm is that if the size of the set S increases, the number of testings will grow exponentially. However, fortunately the numbers of the linear variables that we have tested are fairly small, typically 1 or 2. Algorithm 2 is suitable for verification and counting the

Algorithm 2 Dense Linearity Testing

```

 $S \leftarrow$  the set of linear variables to be tested
for  $c = 1$  to  $C$  do
  for each possible 0/1 combinations of the values of the variables in  $S$  do
    Randomly assign 0/1 to the other variables of  $x$  not in  $S$ 
    if  $f(x) + h(S) \neq f(0)$  then
      Reject and halt
    end if
  end for
end for

```

number of failures, because each linear variable is tested equally and thoroughly. This algorithm is the one we actually used in our verification programs. As long as one 0/1 combination fails, the counter of failures will increase by 1.

For attacks, we also devise an efficient algorithm, shown as Algorithm 3. Compared with Algorithm 1, this algorithm moves ahead the process of searching linear variables, and then test the linearity of each term. It is easy to prove that as long as these terms are linear, their linear combination is also linear. In Algorithm 2, if the number of linear variables is d , then f will be evaluated $2^d \cdot C$ times for verification. While in Algorithm 3, we only need to perform $2 \cdot d \cdot C$ times.

Algorithm 3 Term-by-Term Linearity Testing

```

 $S \leftarrow$  an empty set
for  $i = 1$  to  $n$  do
   $x \leftarrow (0, 0, \dots, 1, \dots, 0)$  where only the  $i$ -th bit is 1
  if  $f(x) + f(0) = 1$  then
    Add  $i$  into the set  $S$ 
  end if
end for
for each  $j$  in the set  $S$  do
  for  $c = 1$  to  $C$  do
    Randomly choose an input value  $y$ 
     $z \leftarrow y$ 
     $y_j \leftarrow 0$  and  $z_j \leftarrow 1$ 
    if  $f(y) = f(z)$  then
      Reject and halt
    end if
  end for
end for

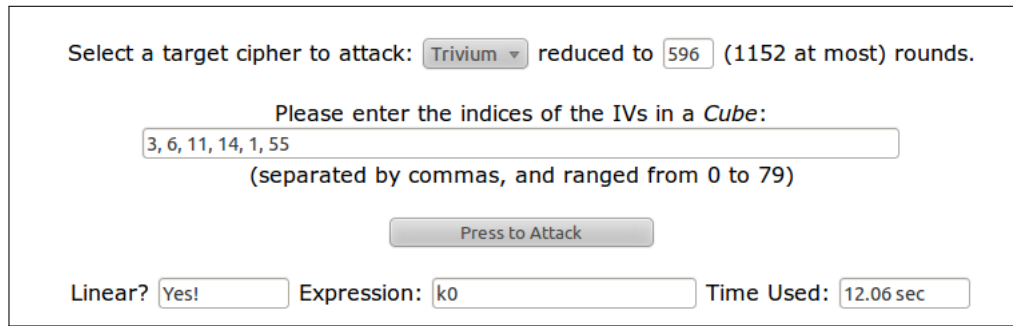
```

3.3 Website Application

Besides the platform for Cube-Attack-like cryptanalyses running on local machines, a web-based application is also developed and is open to public². This website aims to provide a user-friendly interface to any person who is interested in Cube-Attack-like cryptanalyses.

Linearity is tested at the website. The user first can select from a list of target ciphers, such as Trivium (and only Trivium is provided at this moment), to perform an attack. By sending the request to the server, our program is able to verify whether we can obtain a linear function in terms of the secret key bits, and if so, the corresponding key expression will be returned and displayed on the web page, along with the total execution time. Fig. 2 shows part of the web page.

² Please have a try at <http://cube-attack.appspot.com/>.



Select a target cipher to attack: reduced to (1152 at most) rounds.

Please enter the indices of the IVs in a *Cube*:

(separated by commas, and ranged from 0 to 79)

Linear? Expression: Time Used:

Fig. 2. An illustration of the Cube-Attack-like cryptanalysis website (part).

The application is developed by Google App Engine with Python³, which is portable and easy to maintain. The web application interacts with Google's web server using the CGI protocol. Third-party frameworks such as Django is also used, with the AJAX technique to synchronize the dynamic response on the web page. Here is a brief description of how the whole process works: When a user sends the remote procedure call (RPC) requesting for the linearity testing, the RPCHandler at server side captures the AJAX request and the Python methods defined in class RPCMethods will be remote called directly and thus able to invoke other Python methods to perform the linearity testing. The portability of Python allows us to easily integrate our previous offline platform with the web application.

As a web application, security becomes a big concern. The remote calls are limited to only those methods in a single class RPCMethods. Besides, the use of interface libraries is restricted, so we changed our generating code of Trivium key stream from C code to a Python version, which is more conformed, but slower.

4 Testing Results and Discussions

By running our program, we have tested the data provided in several papers [2,3,8,14]. During the testing, we found some interesting problems. For examples, Dinur and Shamir did not follow their theoretical analysis when they were programming, in which case they cannot guarantee the terms derived from the obtained Cubes were maxterms. Nevertheless, this set of data falls perfectly into the theorem of AIDA [3]. And we disprove the argument given in another paper [14], where the authors state the data in [3] is wrong. We will give the detailed programming results and analyses in this section. The following subsections are titled using the names of tested papers.

4.1 Cube Attacks on Tweakable Black Box Polynomials

In Cube Attack, the attacker assigns all the possible combinations of 0/1 values to variables in the cube, and leave all the other variables undetermined. This means that in the testing we should assign random values to the variables outside the cubes. However, in this way, all the data in this paper failed during the verification process. This situation is also mentioned in [14]. Our testing results are shown in Table 1.

The values in the column named *#Failures (Random)* show the failure numbers of the linearity testing in the case when the IV bits outside the cubes are chosen randomly. However if the bit variables outside the cubes are set to zero, then all the data can pass the linearity test (please see the numbers in the last column of Table 1). Dinur and Shamir also mentioned in their paper⁴ that all the public variables outside the cubes were set to 0 in their program. However, in this way, the obtained cubes

³ For more details, please visit <http://code.google.com/appengine/docs/python/overview.html>.

⁴ Please see the last fifth line on the twentieth page of [2].

Table 1. Verification Results for Table 1 in [2] (1000 times for each entry).

| #Init Rounds | IV Bits in a Cube | Linear Key Bits | #Failures (Random) | #Failures (Zero) |
|--------------|--|-----------------|--------------------|------------------|
| 675 | v2, v13, v20, v24, v37, v42, v43, v46, v53, v55, v57, v67 | k0, k9, k50 | 995 | 0 |
| 673 | v2, v12, v17, v25, v37, v39, v46, v48, v54, v56, v65, v78 | k0, k24 | 941 | 0 |
| 674 | v3, v14, v21, v25, v38, v43, v44, v47, v54, v56, v58, v68 | k1, k10, k51 | 995 | 0 |
| 672 | v3, v13, v18, v26, v38, v40, v47, v49, v55, v57, v66, v79 | k1, k25 | 939 | 0 |
| 678 | v0, v5, v7, v18, v21, v32, v38, v43, v59, v67, v73, v78 | k2, k34, k62 | 995 | 0 |
| 677 | v1, v6, v8, v19, v22, v33, v39, v44, v60, v68, v74, v79 | k3, k35, k63 | 1000 | 0 |
| 675 | v11, v18, v20, v33, v45, v47, v53, v60, v61, v63, v69, v78 | k4 | 748 | 0 |
| 677 | v5, v14, v16, v18, v27, v31, v37, v43, v48, v55, v63, v78 | k5 | 731 | 0 |
| 675 | v1, v3, v6, v7, v12, v18, v22, v38, v47, v58, v67, v74 | k7 | 750 | 0 |
| 676 | v1, v12, v19, v23, v36, v41, v42, v45, v52, v54, v56, v66 | k8, k49, k68 | 994 | 0 |
| 684 | v0, v4, v9, v11, v22, v24, v27, v29, v44, v46, v51, v76 | k11 | 750 | 0 |
| 673 | v0, v5, v8, v11, v13, v21, v22, v26, v36, v38, v53, v79 | k12 | 745 | 0 |
| 673 | v0, v5, v8, v11, v13, v22, v26, v36, v37, v38, v53, v79 | k13 | 742 | 0 |
| 672 | v2, v5, v7, v10, v14, v24, v27, v39, v49, v56, v57, v61 | k14 | 775 | 0 |
| 685 | v0, v2, v9, v11, v13, v37, v44, v47, v49, v68, v74, v78 | k15 | 731 | 0 |
| 675 | v1, v6, v7, v12, v18, v21, v29, v33, v34, v45, v49, v70 | k16 | 740 | 0 |
| 677 | v8, v11, v15, v17, v26, v23, v32, v42, v51, v62, v64, v79 | k17 | 735 | 0 |
| 676 | v0, v10, v16, v19, v28, v31, v43, v50, v53, v66, v69, v79 | k18 | 744 | 0 |
| 672 | v4, v9, v10, v15, v21, v24, v32, v36, v37, v48, v52, v73 | k19 | 750 | 0 |
| 675 | v7, v10, v18, v20, v23, v25, v31, v45, v53, v63, v71, v78 | k20 | 748 | 0 |
| 675 | v11, v16, v20, v22, v35, v43, v46, v51, v55, v58, v62, v63 | k20, k50 | 944 | 0 |
| 673 | v10, v13, v15, v17, v30, v37, v39, v42, v47, v57, v73, v79 | k21, k66 | 931 | 0 |
| 673 | v2, v4, v21, v23, v25, v41, v44, v54, v58, v66, v73, v78 | k22 | 733 | 0 |
| 672 | v3, v6, v14, v21, v23, v27, v32, v40, v54, v57, v70, v71 | k23 | 736 | 0 |
| 672 | v3, v5, v14, v16, v18, v20, v33, v56, v57, v65, v73, v75 | k24 | 744 | 0 |
| 676 | v6, v11, v14, v19, v33, v39, v44, v52, v58, v60, v74, v79 | k28 | 768 | 0 |
| 675 | v1, v7, v12, v18, v21, v25, v29, v45, v46, v61, v68, v70 | k29 | 745 | 0 |
| 674 | v2, v8, v13, v19, v22, v26, v30, v46, v47, v62, v69, v71 | k30 | 784 | 0 |
| 673 | v3, v9, v14, v20, v23, v27, v31, v47, v48, v63, v70, v72 | k31 | 752 | 0 |
| 672 | v4, v10, v15, v21, v24, v28, v32, v48, v49, v64, v71, v73 | k32 | 748 | 0 |
| 680 | v2, v4, v6, v12, v23, v29, v32, v37, v46, v49, v52, v76 | k33 | 761 | 0 |
| 678 | v0, v5, v7, v13, v18, v21, v32, v38, v43, v59, v73, v78 | k34, k62 | 937 | 0 |
| 677 | v1, v6, v8, v14, v19, v22, v33, v39, v44, v60, v74, v79 | k35, k63 | 940 | 0 |
| 677 | v2, v4, v5, v8, v15, v19, v27, v32, v35, v57, v71, v78 | k36 | 730 | 0 |
| 678 | v0, v3, v4, v9, v20, v28, v33, v41, v54, v58, v72, v79 | k38, k56 | 946 | 0 |
| 674 | v8, v11, v13, v17, v23, v25, v35, v45, v47, v54, v70, v79 | k39, k57, k66 | 1000 | 0 |
| 676 | v0, v6, v10, v16, v19, v31, v43, v50, v66, v69, v77, v79 | k40, k58, k64 | 993 | 0 |
| 674 | v2, v15, v17, v20, v21, v37, v39, v44, v46, v56, v67, v73 | k41 | 756 | 0 |
| 674 | v1, v16, v20, v22, v34, v37, v38, v53, v58, v69, v71, v78 | k42, k60 | 951 | 0 |
| 673 | v2, v7, v14, v22, v41, v45, v48, v58, v68, v70, v72, v76 | k43 | 751 | 0 |
| 672 | v3, v14, v16, v18, v20, v23, v32, v46, v56, v57, v65, v73 | k44, k62 | 928 | 0 |
| 676 | v0, v6, v10, v16, v18, v28, v31, v43, v53, v69, v77, v79 | k45, k64 | 947 | 0 |
| 684 | v2, v8, v11, v13, v28, v31, v35, v37, v49, v51, v68, v78 | k46, k55 | 931 | 0 |
| 676 | v5, v8, v20, v32, v36, v39, v45, v51, v65, v69, v76, v78 | k47 | 754 | 0 |
| 678 | v2, v4, v10, v14, v16, v22, v25, v44, v49, v51, v57, v78 | k48 | 724 | 0 |
| 676 | v1, v12, v19, v23, v36, v41, v42, v45, v52, v56, v69, v75 | k49, k62 | 939 | 0 |
| 674 | v1, v7, v8, v13, v21, v23, v28, v30, v47, v68, v71, v75 | k51, k62 | 942 | 0 |
| 674 | v5, v8, v9, v12, v16, v18, v23, v40, v44, v63, v66, v70 | k52 | 751 | 0 |
| 675 | v2, v11, v21, v24, v32, v55, v57, v60, v63, v66, v70, v77 | k53 | 739 | 0 |
| 675 | v4, v7, v10, v18, v20, v25, v50, v53, v61, v63, v71, v78 | k54, k60 | 930 | 0 |
| 674 | v5, v12, v16, v19, v22, v36, v47, v55, v63, v71, v77, v79 | k55, k64 | 929 | 0 |
| 677 | v4, v9, v18, v21, v23, v27, v32, v38, v43, v58, v67, v69 | k56 | 751 | 0 |
| 675 | v1, v7, v9, v14, v18, v21, v33, v40, v45, v49, v59, v68 | k57 | 743 | 0 |
| 673 | v2, v6, v12, v13, v19, v23, v30, v48, v55, v59, v69, v79 | k58 | 744 | 0 |
| 681 | v5, v7, v10, v13, v15, v17, v28, v40, v47, v73, v76, v79 | k60 | 756 | 0 |
| 673 | v13, v21, v24, v39, v42, v46, v48, v51, v55, v61, v72, v78 | k61 | 741 | 0 |
| 674 | v2, v4, v10, v11, v19, v34, v47, v55, v56, v58, v69, v77 | k62 | 755 | 0 |
| 674 | v5, v7, v10, v15, v17, v35, v40, v47, v52, v57, v76, v79 | k63 | 739 | 0 |
| 673 | v8, v11, v13, v17, v23, v25, v35, v47, v62, v64, v68, v79 | k64 | 757 | 0 |
| 682 | v2, v3, v13, v15, v19, v29, v32, v37, v39, v51, v76, v79 | k65 | 756 | 0 |
| 678 | v5, v7, v10, v13, v15, v17, v35, v40, v52, v70, v76, v79 | k66 | 755 | 0 |
| 677 | v5, v20, v24, v29, v33, v35, v37, v39, v63, v65, v74, v78 | k67 | 732 | 0 |
| 676 | v1, v12, v19, v23, v36, v41, v52, v54, v56, v66, v69, v75 | k68 | 776 | 0 |

cannot be guaranteed to construct maxterms. To illustrate this situation, please consider the following Boolean function.

$$k_0 \cdot v_0 + k_1 \cdot v_0 v_1$$

Based on the original definition, in this Boolean function, only $v_0 v_1$ is a maxterm. But if we let $\{v_0\}$ to be a *fake* cube and assign 0 to the variable outside the cube, i.e. v_1 , then the linearity testing of

Table 2. Verification Results for the Data in [3] (1000 times for each entry).

| #Init Rounds | IV Bits in a Cube | Linear Key Bits | #Failures (Zero) |
|--------------|-----------------------------|-----------------|------------------|
| 596 | v3, v6, v11, v14, v1, v55 | k0 | 0 |
| 579 | v3, v6, v11, v14, v7, v32 | k1, k64 | 0 |
| 579 | v3, v6, v11, v14, v13, v31 | k2, k65 | 0 |
| 578 | v3, v6, v11, v14, v5, v46 | k3 | 0 |
| 576 | v6, v11, v14, v0, v78 | k4 | 0 |
| 610 | v3, v6, v11, v14, v40, v50 | k5 | 0 |
| 588 | v3, v6, v11, v14, v22, v53 | k7 | 0 |
| 588 | v3, v6, v11, v14, v35, v62 | k8 | 0 |
| 594 | v3, v6, v11, v14, v23, v40 | k10 | 0 |
| 603 | v3, v6, v11, v14, v20, v31 | k13 | 0 |
| 577 | v3, v6, v11, v14, v76, v78 | k15 | 0 |
| 587 | v3, v6, v11, v14, v19, v78 | k16 | 0 |
| 586 | v3, v6, v11, v14, v22, v39 | k18 | 0 |
| 579 | v3, v11, v14, v22, v48 | k24 | 0 |
| 579 | v3, v11, v14, v21, v48 | k25 | 0 |
| 578 | v3, v6, v11, v22, v47 | k26 | 0 |
| 582 | v3, v6, v11, v33, v43 | k35 | 0 |
| 579 | v6, v11, v14, v48, v54 | k37 | 0 |
| 577 | v6, v11, v14, v51, v78 | k38 | 0 |
| 597 | v3, v6, v11, v14, v50, v57 | k54 | 0 |
| 577 | v3, v6, v11, v14, v25, v49 | k55 | 0 |
| 587 | v3, v6, v11, v13, v23 | k56, k62 | 0 |
| 611 | v3, v6, v11, v14, v9, v40 | k58, k64 | 0 |
| 588 | v3, v11, v14, v37, v47 | k59, k65 | 0 |
| 586 | v3, v6, v11, v14, v39, v73 | k60 | 0 |
| 603 | v3, v6, v11, v14, v22, v74 | k61 | 0 |
| 603 | v3, v6, v11, v14, v22, v73 | k62 | 0 |
| 596 | v3, v6, v11, v14, v2, v29 | k63 | 0 |
| 579 | v3, v6, v11, v14, v1, v32 | k64 | 0 |
| 579 | v3, v6, v11, v14, v15, v33 | k65 | 0 |
| 595 | v3, v6, v11, v14, v39, v64 | k66 | 0 |
| 595 | v3, v6, v11, v14, v39, v63 | k67 | 0 |
| 580 | v3, v27, v30, v78, v2, v46 | k14 | 0 |
| 599 | v3, v27, v30, v78, v0, v68 | k17 | 0 |
| 597 | v3, v27, v30, v78, v2, v49 | k19 | 0 |
| 624 | v3, v27, v30, v78, v7, v11 | k22 | 0 |
| 605 | v3, v27, v30, v78, v11, v45 | k29 | 0 |
| 605 | v3, v27, v30, v78, v0, v16 | k31 | 0 |
| 590 | v27, v30, v78, v1, v36 | k32 | 0 |
| 588 | v3, v27, v30, v78, v13, v50 | k34 | 0 |
| 587 | v3, v27, v30, v78, v34, v37 | k57 | 518 |
| 582 | v1, v6, v7, v11, v18, v44 | k20 | 0 |
| 582 | v1, v6, v7, v11, v19, v55 | k21 | 0 |
| 582 | v1, v7, v79, v18, v42 | k9 | 517 |
| 581 | v1, v7, v11, v79, v18, v43 | k11 | 501 |
| 606 | v1, v7, v11, v79, v18, v70 | k57 | 0 |
| 578 | v1, v7, v11, v79, v13, v48 | k68 | 0 |

k_0 will also succeed. We should mention that this does not mean Dinur and Shamir made mistakes in their theoretical analysis or the programming results, and this just shows their programming results do not serve a support to their theorem. Assigning zero to the variables outside the cubes is also a powerful approach to perform attacks, which is exactly what AIDA does. It is easy to see that if Cube Attack can be performed on a certain maxterm (i.e. leaving the other bits undetermined), then AIDA can succeed on the same maxterm (letting the other variables to be zero), but not vice versa.

We will keep the computers running for testing Table 2 in [2], but this may take a very long time, because Table 2 involves three times number of variables than Table 1, requiring 2^{20} more computations for each cube. Maybe we can put the verification results of Table 2 on our website later.

4.2 Breaking One.Fivium by AIDA – an Algebraic IV Differential Attack

Algebraic IV Differential Attack was proposed earlier than Cube Attack, and it also aims to deduce linear functions of Trivium. We also verified the data provided in the AIDA paper. Our results are listed in Table 2.

Table 3. Verification Results for the Data in [14] (1000 times for each entry).

| #Init Rounds | IV Bits in a Cube | Linear Variables | #Failures (Zero) |
|--------------|---|-------------------------|------------------|
| 579 | v3, v20, v28, v36, v42, v55, v77, v78 | k68 | 0 |
| 579 | v18, v26, v36, v45, v61, v73, v78, v79 | v77, v64, k67 | 0 |
| 588 | v11, v18, v34, v37, v45, v51, v70, v79 | v78, k66 | 0 |
| 581 | v1, v3, v28, v34, v51, v61, v67 | k65 | 0 |
| 578 | v3, v12, v19, v29, v37, v62, v77 | k64 | 0 |
| 577 | v8, v13, v21, v39, v53, v73, v74 | k63 | 0 |
| 576 | v6, v7, v12, v13, v15, v16, v36, v73 | k62 | 0 |
| 584 | v0, v10, v35, v45, v55, v58, v72, v77 | k61 | 0 |
| 581 | v6, v7, v10, v27, v35, v36, v67 | v72, v9, v8, k60 | 0 |
| 587 | v1, v20, v29, v36, v48, v55, v73 | k59 | 0 |
| 586 | v8, v16, v19, v28, v52, v62, v69, v72 | k58 | 0 |
| 593 | v0, v10, v11, v23, v25, v26, v29, v57, v68, v71 | k57 | 0 |
| 578 | v5, v6, v11, v27, v44, v55, v60, v67 | k56 | 0 |
| 578 | v0, v3, v7, v20, v21, v31, v66 | k55 | 0 |
| 577 | v5, v6, v11, v44, v60, v65, v67 | k54 | 0 |
| 581 | v17, v25, v27, v35, v54, v62, v63, v79 | v65, v64, v50, k53 | 0 |
| 579 | v1, v2, v8, v39, v61, v62, v69, v70 | v64, v63, v49, v7, k52 | 0 |
| 584 | v15, v23, v32, v47, v49, v58, v76 | k51 | 0 |
| 584 | v0, v5, v14, v23, v38, v48, v67 | k50 | 0 |
| 585 | v14, v22, v30, v45, v48, v50, v59, v75 | k49 | 0 |
| 586 | v4, v29, v38, v43, v46, v47, v57, v66, v73 | k48 | 0 |
| 587 | v18, v28, v38, v39, v42, v45, v46, v65, v79 | k47 | 0 |
| 614 | v1, v17, v19, v21, v24, v27, v59, v60, v71 | v25, k46 | 0 |
| 590 | v9, v18, v25, v28, v43, v45, v55, v69 | k45 | 0 |
| 577 | v2, v21, v29, v40, v57, v66, v73 | v20, k44 | 0 |
| 591 | v1, v7, v8, v32, v39, v42, v67, v74 | v40, k43 | 0 |
| 592 | v7, v15, v29, v38, v41, v42, v50, v75 | v39, k42 | 0 |
| 589 | v3, v9, v12, v22, v30, v49, v52, v53 | v51, v38, k41 | 0 |
| 595 | v19, v30, v36, v38, v43, v46, v58, v63, v79 | k40 | 0 |
| 595 | v4, v5, v21, v22, v37, v38, v39, v72 | v36, k39 | 0 |
| 580 | v3, v7, v11, v23, v44, v49, v50 | v48, v35, k38 | 0 |
| 582 | v1, v7, v9, v15, v46, v47, v59, v68 | v49, v48, v34, k37 | 0 |
| 584 | v7, v21, v23, v45, v46, v58, v74, v76 | v48, v47, v33, k36 | 0 |
| 581 | v22, v25, v41, v44, v45, v51, v55, v58, v67 | v47, v46, v32, k35 | 0 |
| 583 | v1, v15, v45, v46, v50, v57, v68, v69 | v44, v31, k34 | 0 |
| 582 | v5, v22, v28, v31, v42, v43, v51, v75 | v45, v44, v30, v27, k33 | 0 |
| 585 | v0, v3, v32, v39, v41, v42, v47, v48, v61 | v44, v43, v29, k32 | 0 |
| 587 | v4, v20, v37, v42, v43, v54, v64 | v41, v28, k31 | 0 |
| 588 | v10, v11, v25, v26, v39, v40, v47, v56, v70 | v42, v41, v27, k30 | 0 |
| 589 | v0, v2, v11, v30, v40, v41, v53, v54 | v39, v26, k29 | 0 |
| 588 | v18, v28, v37, v38, v42, v45, v46, v65, v79 | v40, v39, v25, k28 | 0 |
| 579 | v5, v9, v10, v11, v12, v42, v68, v77 | k3 | 0 |
| 576 | v5, v8, v12, v28, v31, v67, v74 | v68, v29, k2 | 0 |
| 590 | v9, v10, v19, v33, v41, v68, v77 | v67, k1 | 0 |
| 578 | v3, v12, v37, v63, v65, v71, v74 | v66, k0 | 0 |

We can see most of the entries of the table in [3] passed the linearity testing. However, in another paper [14], which also tried to verify the data of AIDA, shows that almost all the data is wrong, which contradicts our results. We think the authors of [14] made a mistake that they confused the indices used in AIDA and Cube Attack. In [3] the indices are counted from 1 to 80, while the indices in [2] are listed from 0 to 79.

4.3 Cube Attacks on Trivium

The paper [14] mentioned in the last subsection also provides new cubes. In these cubes, the linear equations involve both the key and IV bits. All the data passed our testing, and the results are listed in Table 3.

However there is a small mistake in the original Table 3 in [14]. The title of the second column is

$$p(x_1, \dots, x_{80}, v_1, \dots, v_{80}),$$

but the indices the authors actually used in the table are from 0 to 79.

Table 4. Verification Results for the Data in [8] (*10 times* for each entry).

| #Init | Indices of the IV Bits in a Cube | Linear Key Bits | #Failures (Zero) |
|-------|---|-----------------|------------------|
| 709 | 6, 8, 9, 13, 22, 24, 28, 30, 32, 36, 39, 40, 43, 45, 47, 48, 60, 63, 67, 68, 73, 76, 79 | k14 | 10 |
| 709 | 2, 9, 15, 17, 27, 28, 32, 40, 44, 46, 52, 54, 59, 64, 68, 70, 71, 72, 73, 74, 76, 78, 79 | k15 | 6 |
| 709 | 1, 3, 9, 10, 13, 14, 16, 28, 34, 37, 42, 51, 52, 56, 59, 60, 62, 68, 69, 72, 74, 79 | k16 | 7 |
| 709 | 4, 5, 6, 10, 11, 12, 17, 19, 21, 26, 32, 40, 44, 49, 54, 58, 60, 61, 67, 72, 74, 77, 78 | k17 | 8 |
| 709 | 5, 9, 15, 16, 20, 21, 32, 33, 35, 38, 41, 43, 46, 52, 56, 58, 60, 61, 62, 69, 77, 78, 79 | k18 | 9 |
| 709 | 6, 8, 13, 17, 23, 27, 28, 33, 44, 45, 46, 53, 54, 56, 60, 61, 67, 68, 72, 74, 75, 77, 79 | k19 | 10 |
| 709 | 1, 2, 7, 13, 15, 18, 19, 23, 29, 34, 35, 36, 38, 47, 49, 54, 57, 62, 64, 65, 66, 68, 74 | k20 | 10 |
| 709 | 4, 7, 10, 11, 19, 20, 22, 23, 24, 30, 32, 33, 38, 41, 49, 52, 54, 59, 66, 67, 69, 74, 77 | k21 | 10 |
| 709 | 8, 16, 18, 22, 24, 26, 29, 31, 34, 36, 40, 41, 45, 46, 47, 48, 50, 59, 63, 69, 72, 76, 78 | k22 | 6 |
| 709 | 6, 10, 13, 16, 19, 25, 28, 35, 39, 42, 44, 48, 57, 61, 62, 63, 64, 65, 67, 68, 73, 77, 78 | k23 | 8 |
| 709 | 2, 4, 7, 15, 17, 18, 20, 23, 24, 27, 29, 35, 45, 47, 48, 51, 57, 59, 63, 65, 67, 74, 77 | k24 | 10 |
| 710 | 3, 5, 8, 16, 18, 19, 21, 24, 25, 28, 30, 36, 46, 48, 49, 52, 58, 60, 64, 66, 68, 75, 78 | k25 | 10 |
| 709 | 5, 10, 13, 14, 15, 22, 26, 27, 32, 35, 36, 45, 46, 50, 51, 56, 59, 60, 63, 64, 77, 78, 79 | k33 | 8 |
| 710 | 2, 6, 8, 9, 19, 23, 24, 29, 32, 33, 34, 42, 47, 49, 51, 52, 53, 57, 61, 64, 73, 77 | k35 | 10 |
| 709 | 0, 3, 6, 8, 11, 17, 28, 34, 38, 39, 41, 43, 46, 51, 52, 53, 54, 56, 64, 65, 70, 72, 78 | k39 | 8 |
| 709 | 5, 6, 11, 19, 27, 31, 32, 39, 40, 44, 47, 49, 51, 52, 56, 58, 59, 63, 65, 66, 69, 71, 79 | k40 | 6 |
| 709 | 7, 9, 10, 15, 17, 24, 25, 26, 33, 36, 43, 45, 52, 56, 59, 60, 61, 63, 68, 71, 74, 77 | k41 | 8 |
| 710 | 8, 10, 11, 16, 18, 25, 26, 27, 34, 37, 44, 46, 53, 57, 60, 61, 62, 64, 69, 72, 75, 78 | k42 | 7 |
| 711 | 9, 11, 12, 17, 19, 26, 27, 28, 35, 38, 45, 47, 54, 58, 61, 62, 63, 65, 70, 73, 76, 79 | k43 | 9 |
| 709 | 4, 5, 7, 13, 15, 18, 27, 30, 33, 34, 36, 39, 42, 44, 45, 46, 51, 53, 57, 63, 75, 77, 78 | k47 | 6 |
| 709 | 6, 7, 15, 19, 27, 30, 35, 37, 44, 45, 46, 47, 49, 50, 56, 59, 60, 67, 70, 71, 72, 75, 79 | k48 | 9 |
| 709 | 0, 8, 14, 18, 25, 28, 31, 35, 38, 42, 44, 45, 51, 52, 58, 60, 66, 67, 70, 73, 76, 77 | k49 | 9 |
| 709 | 0, 2, 8, 11, 14, 15, 17, 21, 22, 28, 31, 32, 39, 41, 52, 53, 59, 60, 65, 67, 74, 77, 78 | k50 | 7 |
| 709 | 1, 8, 10, 15, 18, 26, 28, 29, 33, 35, 37, 38, 42, 51, 53, 55, 57, 60, 61, 65, 66, 67, 75 | k51 | 6 |
| 710 | 7, 10, 11, 12, 15, 21, 29, 32, 37, 39, 41, 44, 47, 53, 56, 57, 59, 62, 63, 66, 70, 76 | k21, k52 | 10 |
| 710 | 1, 4, 8, 10, 11, 12, 14, 15, 19, 22, 24, 29, 31, 33, 39, 42, 50, 52, 55, 58, 60, 61, 65 | k53 | 10 |
| 712 | 9, 12, 13, 14, 17, 23, 31, 34, 39, 41, 43, 46, 49, 55, 58, 59, 61, 64, 65, 68, 72, 78 | k23, k54 | 10 |
| 713 | 10, 13, 14, 15, 18, 24, 32, 35, 40, 42, 44, 47, 50, 56, 59, 60, 62, 65, 66, 69, 73, 79 | k24, k55 | 10 |
| 709 | 1, 3, 6, 10, 11, 14, 15, 16, 23, 25, 28, 35, 40, 41, 42, 44, 46, 52, 58, 66, 68, 69, 75 | k57 | 10 |
| 709 | 8, 12, 14, 19, 26, 28, 30, 40, 41, 42, 43, 48, 50, 53, 59, 62, 63, 67, 71, 72, 74, 79 | k21, k49, k58 | 10 |
| 709 | 6, 14, 16, 31, 37, 40, 43, 48, 50, 53, 54, 55, 57, 58, 60, 61, 62, 68, 72, 73, 74, 76 | k59 | 9 |
| 709 | 3, 4, 14, 16, 26, 29, 30, 38, 40, 43, 47, 54, 56, 58, 60, 64, 65, 67, 69, 70, 75, 76, 77 | k60 | 7 |
| 711 | 3, 8, 11, 14, 16, 17, 18, 20, 22, 24, 27, 33, 35, 38, 44, 48, 52, 53, 59, 66, 73, 77 | k61 | 10 |
| 712 | 4, 9, 12, 15, 17, 18, 19, 21, 23, 25, 28, 34, 36, 39, 45, 49, 53, 54, 60, 67, 74, 78 | k62 | 10 |
| 709 | 2, 5, 9, 17, 21, 27, 28, 30, 35, 37, 46, 48, 50, 53, 54, 60, 61, 63, 65, 69, 71, 73, 79 | k19, k63 | 10 |
| 709 | 1, 7, 12, 15, 18, 27, 30, 41, 44, 46, 47, 48, 49, 52, 53, 54, 56, 59, 62, 63, 66, 69, 79 | k67 | 6 |
| 709 | 6, 11, 16, 19, 26, 34, 36, 39, 41, 42, 47, 49, 52, 54, 57, 59, 66, 67, 71, 72, 76, 79 | k72 | 8 |
| 709 | 1, 3, 4, 6, 12, 14, 15, 19, 25, 26, 28, 29, 35, 40, 49, 52, 57, 64, 66, 67, 68, 72, 75 | k73 | 8 |
| 710 | 2, 4, 5, 7, 13, 15, 16, 20, 26, 27, 29, 30, 36, 41, 50, 53, 58, 65, 67, 68, 69, 73, 76 | k74 | 6 |
| 711 | 3, 5, 6, 8, 14, 16, 17, 21, 27, 28, 30, 31, 37, 42, 51, 54, 59, 66, 68, 69, 70, 74, 77 | k75 | 10 |
| 712 | 4, 6, 7, 9, 15, 17, 18, 22, 28, 29, 31, 32, 38, 43, 52, 55, 60, 67, 69, 70, 71, 75, 78 | k76 | 10 |

4.4 The Cube Attack on Stream Cipher Trivium and Quadraticity Tests

Recently, one paper which extends the idea of Cube-Attack-like cryptanalyses to solving quadratic functions was put online [8]. This paper also provides some new cubes to compute linear functions. We also tested this set of data, but unfortunately little data passed. The results are listed in Table 4.

5 Conclusion and Further Works

In this paper, we have designed a platform to perform the Cube-Attack-like cryptanalyses. Efficient algorithms are applied in the program. The experimental data in the papers [2,3,8,14] are tested and interesting results showed up. By analyzing the testing results, we find certain improper assumption made by Dinur and Shamir when they implemented their theorem by setting the variables outside cubes to be 0. And another paper [14] argues that the verifications for AIDA all failed, but our second finding implies that the author may have been confused by the indices used in AIDA and Cube Attack, because our experimental result shows a pretty good survival of the data from AIDA.

Besides the local/offline platform, a web-based application is also launched for public use. Linearity verification of AIDA on Trivium is available to test online by user-specified IV indices and with reduced rounds. Indication of whether the result is linear and the matched keys expression are returned and displayed on the web page, along with the total processing time.

We also found and fixed a bug in the source code of Trivium provided on the official website. The original source code cannot be compiled on many 64-bit platforms, such as Mac and Ubuntu. We also notified the authors of Trivium and provided the patch to the code. Please refer to the appendix for the details.

In the future, we will continue exploring the linearity verification of data in other papers, such as [6] and [15]. Besides, other properties, such as unbalance and presence of a variable, will also be tested based on our current work. In order to do all jobs efficiently, we plan to change the platform language from Python to C, so that cubes with large sizes are also possible to be evaluated. In addition, although the paper [4] is not targeted for the attacks on Trivium, it indeed reveals a number of innovative ideas that can be used in our future works to speed up the linearity testing algorithm, e.g. by utilizing Reed-Muller transform.

References

1. Dinur, I., Shamir, A.: Cube Attacks on Tweakable Black Box Polynomials. Cryptology ePrint Archive, Report 2008/385 (2008) <http://eprint.iacr.org/>.
2. Dinur, I., Shamir, A.: Cube Attacks on Tweakable Black Box Polynomials. EUROCRYPT'09. LNCS 5479 (2009) pp. 278–299
3. Vielhaber, M.: Breaking ONE.FIVIUM by AIDA – an Algebraic IV Differential Attack. Cryptology ePrint Archive, Report 2007/413 (2007) <http://eprint.iacr.org/>.
4. Vielhaber, M.: AIDA Breaks BIVIUM (A&B) in 1 Minute Dual Core CPU Time. Cryptology ePrint Archive, Report 2009/402 (2009) <http://eprint.iacr.org/>.
5. Lai, X.: Higher Order Derivatives and Differential Cryptanalysis. Communications and Cryptography: Two Sides of One Tapestry (1994) pp. 227
6. Aumasson, J., Dinur, I., Meier, W., Shamir, A.: Cube Testers and Key Recovery Attacks on Reduced-Round MD6 and Trivium. FSE'09. LNCS 5665 (2009) pp. 1–22
7. Rivest, R., Agre, B., Bailey, D., Crutchfield, C., Dodis, Y., Fleming, K., Khan, A., Krishnamurthy, J., Lin, Y., Reyzin, L., et al.: The MD6 hash function – A proposal to NIST for SHA-3. (2008)
8. Mroczkowski, P., Szmids, J.: The Cube Attack on Stream Cipher Trivium and Quadraticity Tests. Cryptology ePrint Archive, Report 2010/580 (2010) <http://eprint.iacr.org/>.
9. De Canniere, C., Preneel, B.: Trivium specifications. In: eSTREAM, ECRYPT Stream Cipher Project, Citeseer
10. Cusick, T., Stanica, P.: Cryptographic Boolean Functions and Applications. Academic Press (2009)
11. Carlet, C.: Boolean Function. In: Encyclopedia of Cryptography and Security. Springer (2005)
12. Blum, M., Luby, M., Rubinfeld, R.: Self-testing/correcting with applications to numerical problems. In: Proceedings of the twenty-second annual ACM symposium on Theory of computing. STOC '90, New York, NY, USA, ACM (1990) 73–83
13. Bellare, M., Coppersmith, D., Hastad, J., Kiwi, M., Sudan, M.: Linearity Testing in Characteristic Two. In: 1995 IEEE 36th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press (1995) 432
14. Bedi, S.S., Pillai, N.R.: Cube Attacks on Trivium. Cryptology ePrint Archive, Report 2009/015 (2009) <http://eprint.iacr.org/>.
15. Dinur, I., Shamir, A.: Breaking Grain-128 with Dynamic Cube Attacks. Cryptology ePrint Archive, Report 2010/570 (2010) <http://eprint.iacr.org/>.

Appendix. A Bug in the Source Code of Trivium

There is a bug in the original source code of Trivium provided on the official website. Here we briefly talk about how to target and fix the bug.

Since different C compilers may interpret the code differently, the source code can pass the compilation on 32-bit platform as well as Windows 64-bit Visual Studio building environment, but will fail on 64-bit Linux or Mac OS, since the latter two are more restricted on compiling rules.

The bug appears in `trivium.c`, at the line where $Z(w)$ is pre-defined as the following statement.

```
#define Z(w) (U32TO8_LITTLE(output + 4 * i, U8TO32_LITTLE(input + 4 * i) ^ w))
```

The error may not be so obvious if we only look at the code itself. We found the problem by expanding and checking the assembled code: Although the reason is not clear, the outer pair of brackets in the macro will be unbalanced after compilation. To solve the problem, the original code should be manually changed as follows.

```
#define Z(w) U32TO8_LITTLE(output + 4 * i, U8TO32_LITTLE(input + 4 * i) ^ w)
```

Please notice that, while the code is working well on our computers, we cannot guarantee its liability on other machines.