UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

Master's Degree in Computer Science

Cube Attack against Stream Ciphers: design, engineering and testing

Thesis Advisor:
Prof. Andrea Visconti

Master Thesis by:
Marco Garlet
886469

Academic Year 2020/2021

Abstract

Cube attack is a chosen-plaintext attack introduced by Dinur and Shamir in 2009[10] generalizing Vielhaber's Algebraic IV Differential Attack (AIDA)[16].

The basic idea is that any symmetric cipher can be described using a function or polynomial in n public variable and m private variable.

The public variable is called *Initialation Vector*(IV) while the private variable is the secret key. Such techniques are applied to target weakness of the underlying polynomial of the cipher seeking hidden relations with key's secret bits.

Once found such relations the attacker is able to solve the resulting system of multivariate polynomial equations in terms of secret bit keys aiming to achieve a full (or partial) key recovery.

Solving large multivariate polynomial systems as well as quadratic equations modulo 2, is a NP-complete problem.

The main goal of the cube attack is to reach an easy-to-solve linear equations system exploiting properties of the underlying polynomial.

Cube attack is mainly used against *stream cipher*, *block cipher* and *cryptographic hash function*.

Contents

Chapter 1

Introduction

Stream ciphers are symmetric key ciphers in which every element of the plaintext stream(bit, byte or block) is xored with a particular block coming from a pseudo-random generation function.

The concept of these ciphers comes from OTP in which a random sequence, produced by a pseudo-random generation function, is xored with the plaintext obtaining the ciphertext.

On the other hand the receiver, by submitting the same *nonce* and *key* to the pseudo random generation function, is able to produce internally the same key stream and achieving plaintext by xoring the ciphertext with the previous obtained sequence.

The output of a strong stream cipher is comparable to (and should be indistinguishable from) a contiguous bit stream produced by a Pseudo Random Number Generator (PRNG).

Therefore the output of a generator is said to be *pseudorandom* if it is indistinguishable from true random numbers by any efficient algorithm (polinomyal in time for *Probabilistic Turing Machine*[1]).

For this kind of ciphers, the design scheme for keystream generation play a central role as well as the choosen *modes of operation*[7].

Cube attack requires a black box access to the cipher and it must be possible to record a bit (always in the same position) in output [10]. Polynomial representation different cipher components are known, therefore, ideally it should be possible to proceed with a white box version of the attack. However, since the *master polynomial* was not included in documentation, it was necessary to use the black box approach.

Another important consideration to do is that this work is focused on keystream generation. The attack focus on robustness of the underlying polynomial without consider modes of operation and authentication primitives. However it's possible to target different cipher phase and modules exploiting the same properties introduced by this attack [12].

A whitebox approach is *Dynamic Cube Attack* [4] which replace some random operation issued by *Cube Attack* with deterministic functions obtained by a cryptanalytic phase on some cipher components.

1.1 Objective

The aim of this research is to analyze several stream ciphers for the application of cube attack. After a study of the literature a new parallel implementation using CUDA platform is proposed.

Existing results, carried out by previous researches[18][12], are validated and new relations are reported.

Then the effectiveness of using GPUs are determined by exploring how many rounds our attack version is able to process comparing with current literature.

Another objective of this work is to determine how some best practises in CUDA programming impact on the performance of the attack. In order to find new relations efficiently, some CUDA best practices are applied on ciphers, trying to accelerate the entire exploit. How to deal with *dependencies* problem on older version of NVIDIA graphic card and different versions of maxterm mining procedure are proposed and analyzed.

Finally we validate new relations developing a probabilistic algorithm iterating online phase of cube attack on several random private keys.

Chapter 2

Overview

I report here some basic definitions, moreover I implemented a POC of the attack on simple and known polynomial[6]:

2.1 Definitions

For this attack is given a black box encryption machine which can be viewed as an unknown boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$.

Any boolean function can be expressed, in the Algebraic Normal Form(ANF), as a polynomial $p(x)$ such that evaluating $p(x)$ is equivalent to computing $f(x)$.

This polynomial is in $GF(2)$ with $n + m$ inputs bit $(x_1, \cdots, x_n, v_1, \cdots, v_n)$, where bits $x_1, \cdots, x_n$ represent the secret key, while $v_1, \cdots, v_n$ the public variable (*IV*).

Now let's ignore distinction between public and private information and denote all inputs with $x_1, \cdots, x_n$, obtaining a polynomial $p(x_1, \cdots, x_n)$ over $GF(2)$ in ANF:

$$p(x_1, \cdots, x_n) = \sum_{i=0}^{2^n-1} a_i x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n} \tag{2.1}$$

where the monomial coefficients $a_i \in \{0,1\}$ and $i = \sum_{j=1}^{n} i_j 2^{j-1}$ where $(i_1, \cdots, i_n)$ is the n-bit binary representation of i.

Assume some polynomial $p(x_1, \cdots, x_n)$ and a set $I \subseteq \{1, \cdots, n\}$.

Let t_I be a subterm of p which contains variables indexed by I, factorizing p by t_I will yelds:

$$p\left(x_1, \cdots, x_n\right) \equiv t_I \cdot p_{S(I)} + q\left(x_1, \cdots, x_n\right) \tag{2.2}$$

$p_{S(I)}$ is called *superpoly* of I in p. Note that for any p and I, the superpoly does not contain any common variable with t_I and each term in remainder polynomial $q(x_1, \cdots, x_n)$ misses at least one variable from I.

Definition 2.1.1 (maxterm) *A maxterm of p is a term t_I such that deg $\left(p_{S(I)}\right) \equiv 1$, the superpoly of I in p is a linear non constant polynomial.*

Any k-sized subset of indices to a variables in a polynomial p defines a k-dimensional boolean cube with corners corresponding to all possible assignments of 0/1 to the variables in the k-sized subset of I.

So the cube can be represented by a set C_I of 2^k vectors which are the cube corners. A vector $v \in C_I, p_{|v}$ is defined to be the derivation of p in which the variables in the k-sized subset of I are set to the values in v leaving all other variables undetermined.

Any $v \in C_i$ define a new derived polynomial $p_{|v}$ with $n - k$ variables.

Summing all these derived polynomials, for each assignment in C_I, we end up with a new polynomial which is denoted by: $p_I \triangleq \sum_{v \in C_I} p_{!v}$, and this new polynomial has some interesting property exploited in [10].

Theorem 2.1.1 (main observation) *For any polynomial p and subset of variables I(for simplicity imagine is the k-sized subset of index), $p_I \equiv p_{S(I)}$ mod 2.*

So finding one superpoly $p_{S(I)}$ is equal to find the polynomial obtained by summing all over possible assignment in the k-sized subset of index I.

Obtaining this *superpolys* require to choose arbitrary secret keys, and I provide this method during implementation.

Consider $J \subseteq \{1, \cdots, n\}$ be the index set of the private variables of terms in $p_{S(I)}$, so it can be written as:

$$p_{S(I)} = a_0 + \sum_{i=0}^{n} a_i x_i \tag{2.3}$$

where a_0 is the constant term and $a_i = 1$ iff $i \in J$. Recalling that J is the index set of the private linear variables in the superpoly, the problem of finding $p_{S(I)}$ can be reformulated into searching $\{a_0 \cup J\}$ exploiting the following theorems:

Theorem 2.1.2 (retrieve constant term) *The constant term a_0 can be computed by summing over the cube C_I, setting all remain variables to 0.*

Theorem 2.1.3 (retrieve coefficient) *The coefficient a_i in superpoly $p_{S(I)}$ is computed by summing over cube C_I setting all j-th variable to zero, adding the result with the constant term a_0*

All linear expression found can be converted into a set of linearly independent equations. This phase is focused on compute the right hand for each superpolys.

To do that an attacker has to access to the *real* cipher with the secret encription key that want to recover and compute sums over the subset I sending public keys crafted over all possible value of k-sized index set.

The system can be solved in order to get the unknown key bits.

Example 1 *Let $p(x_1, x_2, v_1, v_2, v_3) = v_1v_2x_1x_2 + v_1v_2v_3x_1x_2 + v_1v_2x_1$ be a polynomial representation of a cryptosystem $E(\bar{x}, \bar{v})$ with 2 private variables and 3 public.*
Let $I = \{1, 2\}$ be a candidate maxterm, we define:

$$\bar{v}_I = \{v_1, v_2\}, \bar{v}_{\bar{I}} = \{v_3\} \tag{2.4}$$

Let $\bar{v}_{\bar{I}} = 1$ we obtain $C_I(1)$, the set of public variable having $\bar{v}_{\bar{I}} = 1$:

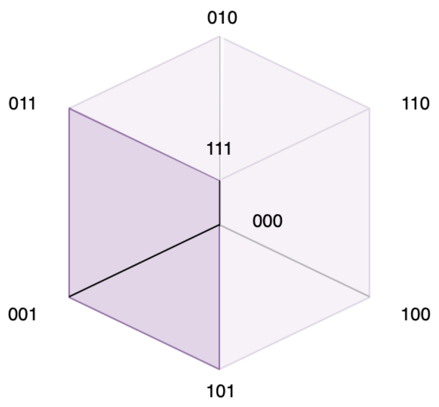


Figure 2.1: highlighting C_I

Let $P_{S(I)}$ be the polynomial obtained by I:

$$P_{S(I)}(\bar{x}, \bar{v}) = P_{S(I)}(\bar{x}, \bar{v}_{\bar{I}}) = P_{S(I)}(x_1, x_2, 1) = \sum_{\bar{v} \in C_I(1)} E(x_1, x_2, \bar{v}) \tag{2.5}$$

The coefficients of $P_{S(I)}(x_1, x_2, 1)$ could be retrieved as following:

- $P_{S(I)}(0,0,1) = \sum_{\bar{v} \in C_I(1)} E(0,0,\bar{v}) = \underbrace{0}_{E(0,0,0,0,1)} + \underbrace{0}_{E(0,0,0,1,1)} + \underbrace{0}_{E(0,0,1,0,1)} + \underbrace{0}_{E(0,0,1,1,1)} = 0$
- $P_{S(I)}(1,0,1) = \sum_{\bar{v} \in C_I(1)} E(1,0,\bar{v}) = \underbrace{0}_{E(1,0,0,0,1)} + \underbrace{0}_{E(1,0,0,1,1)} + \underbrace{0}_{E(1,0,1,0,1)} + \underbrace{1}_{E(1,0,1,1,1)} = 1$
- $P_{S(I)}(0,1,1) = \sum_{\bar{v} \in C_I(1)} E(0,1,\bar{v}) = \underbrace{0}_{E(0,1,0,0,1)} + \underbrace{0}_{E(0,1,0,1,1)} + \underbrace{0}_{E(0,1,1,0,1)} + \underbrace{0}_{E(0,1,1,1,1)} = 0$
- $P_{S(I)}(1,1,1) = \sum_{\bar{v} \in C_I(1)} E(1,1,\bar{v}) = \underbrace{0}_{E(1,1,0,0,1)} + \underbrace{0}_{E(1,1,0,1,1)} + \underbrace{0}_{E(1,1,1,0,1)} + \underbrace{1}_{E(1,1,1,1,1)} = 1$

From above relations we have deduct $P_{S(I)}(x_1, x_2, 1) = x_1$, we can prove it considering the symbolic sum:

$$P_{S(I)}(x_1, x_2, 1) = \sum_{\bar{v} \in C_I(1)} p(x_1, x_2, \bar{v}) = \underbrace{0+0+0}_{E(x_1,x_2,0,0,1)} + \underbrace{0+0+0}_{E(x_1,x_2,0,1,1)} + \underbrace{0+0+0}_{E(x_1,x_2,1,0,1)} + \\ + \underbrace{x_1x_2 + x_1x_2 + x_1}_{E(x_1,x_2,1,1,1)} = x_1 \tag{2.6}$$

Finally, following a choosen plaintext scheme, an attacker could retrieve the value x_1, computing the sum $b = \sum_{v \in C_I(1)} E(x_1, x_2, \bar{v})$ solving the linear equation $x_1 = b$.

2.2 Ciphers

2.2.1 TRIVIUM

Trivium is defined as a hardware oriented synchronous stream cipher and it was proposed as a secure fast stream cipher extremely flexible [3].

The 288-bit internal state of Trivium is made up of three shift registers of varying lengths. A bit is shifted into each of the three shift registers using a non-linear combination of taps from that and one other register at each round, resulting in one bit of output. The key and IV are written into two of the shift registers to initialize the cipher, with the remaining bits beginning in a fixed pattern; the cipher state is then updated $4 \cdot 288 = 1152$ times, so that every bit of the internal state is dependent on every bit of the key and IV in a complex nonlinear way.

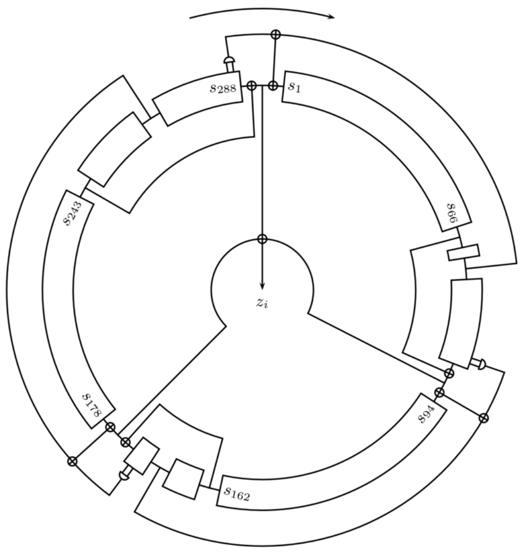


Figure 2.2: TRIVIUM

The first version of cube attack proposed by Shamir [10] was tested against different versions of reduced cipher having respectively 672, 735 and 767 initialization rounds.

More recent works exploit cipher above 784 initialization rounds [5][15] exploiting algebraic properties.

2.2.2 Grain-128AEAD

Grain-128AEAD is an authenticated encryption algorithm and has been designed with 128-bit security in mind (as specified by NIST requirements [11]) .

It takes a variable length plaintext, a fixed-length nonce (*IV*) of 96 bits, and a fixed length key of size 128 bits and the output is a variable length ciphertext.

AEAD stands for *Authenticated Encryption With Associated Data*, is a form of encryption which simultaneosly assure confidentialy and authenticity of data.

Grain-128AEAD consists into two main building blocks:

- **Pre-output generator**: generates a stream of pseudo-random bits, which are used for encryption and the authentication tag.
- **Authenticator generator**: consisting of a shift register and an accumulator.

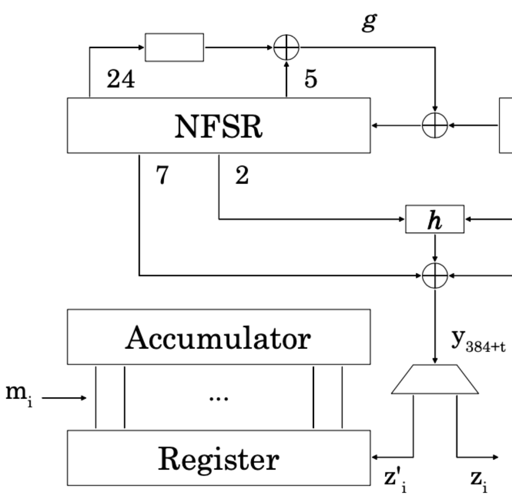


Figure 2.3: Overview of the building blocks in Grain-128AEAD

Before the pre-output can be used as keystream and for authentication, the internal state of the pre-output generator and the authenticator generator registers have to be initialized with a key and nonce.

LFSR is denoted $S_t = \left[s_0^t, s_1^t, ..., s_{127}^t\right]$ and the content of the 128-bit NFSR as $B_t = \left[b_0^t, b_1^t, ..., b_{127}^t\right]$.

The update function of the LSFR is given by:

$$s_{127}^{t+1} = s_0^t + s_7^t + s_{38}^t + s_{70}^t + s_{81}^t + s_{96}^t = \mathcal{L}(S_t) \tag{2.7}$$

While the update function of NFSR is given by:

$$\begin{aligned} b_{127}^{t+1} &= s_0^t + b_0^t + b_{26}^t + b_{56}^t + b_{91}^t + b_3^t b_{67}^t + b_{11}^t b_{13}^t + b_{17}^t b_{18}^t + b_{27}^t b_{59}^t + b_{40}^t b_{48}^t + b_{61}^t b_{65}^t \\ &+ b_{68}^t b_{84}^t + b_{22}^t b_{24}^t b_{25}^t + b_{70}^t b_{78}^t b_{82}^t + b_{88}^t b_{92}^t b_{93}^t b_{95}^t \\ &= s_0^t \mathcal{F}(B_t) \end{aligned}$$

The state is initialized as follow:

- The 128 NFSR bits are loaded with the bits of the key: $b_i^0 = k_i, i \in [0, 127]$
- The 96 LFSR elements are loaded with nonce bits: $s_i^0 = IV_i, i \in [0, 95]$, the remaining 32 bits are filled with 31 ones and a zero: $s_i^0 = 1, i \in [96, 126]\,, s_{127}^0 = 0$

The cipher is then clocked 256 times (rounds number), feeding back the pre-output function and XORing with the input both the LFSR and the NFSR as following:

$$s_{127}^{t+1} = \mathcal{L}(S_t) + y_t, t \in [0, 255] \tag{2.8}$$

$$b_{127}^{t+1} = s_0^t + \mathcal{F}(B_t) + y_t, t \in [0, 255] \tag{2.9}$$

Then the authenticator generator is initialized by loading the accumulator with preoutput keystream.

After initializing, the pre-output is used to generate keystream bits z_i for encryption and authentication bits z_i' to update the register in the accumulator generator. The keystream is generated as:

$$z_i = y_{384+2i} \tag{2.10}$$

every even bit from pre-output generator is taken as a keystream bit. Authentication bit are generated as:

$$z_i' = y_{384+2i+1} \tag{2.11}$$

every odd bit from pre-output generator is taken as an authentication bit. Finally, given a message m of length L, is encrypted as:

$$c_i = m_i \oplus z_i, i \in [0, L) \tag{2.12}$$

2.2.3 MORUS-640-128

MORUS is a family of authenticated ciphers and was presented as a candidate in CAESAR competition. It comes with two different internal state size: 640 and 1280 bits, with two different key size: 128 or 256 bits.

There are three algorithms: MORUS-640-128, MORUS-1280-128, and MORUS-1280-256.

Operations performed in MORUS can be divided in 5 different phases:

1. Initialization

2. Processing associated data
3. Encryption
4. Finalization
5. Decryption and tag verification

In this work the initialization phase is targeted. MORUS has five state elements: $S_{0,0}, \cdots, S_{0,4}$ each of 128 bits in length loaded as follow:

- $S_{0,0}^{-16} = IV_{128}$
- $S_{0,1}^{-16} = K_{128}$
- $S_{0,2}^{-16} = 1^{128}$
- $S_{0,3}^{-16} = const_0$
- $S_{0,4}^{-16} = const_1$

One of the main component is the state update function which update the state 16 times as follow:

$$S^{i+1} = StateUpdate(S^i, 0),\ i \in \{-16, \ldots, -1\} \tag{2.13}$$

After 16 rounds the state element $S_{0,1}^{0}$ is XORed with the key K and will be the initial internal state.

After processing associated data there's an encryption phase where a 16 byte plaintext P_i is encrypted to C_i. The plaintext block is then used to update the state.

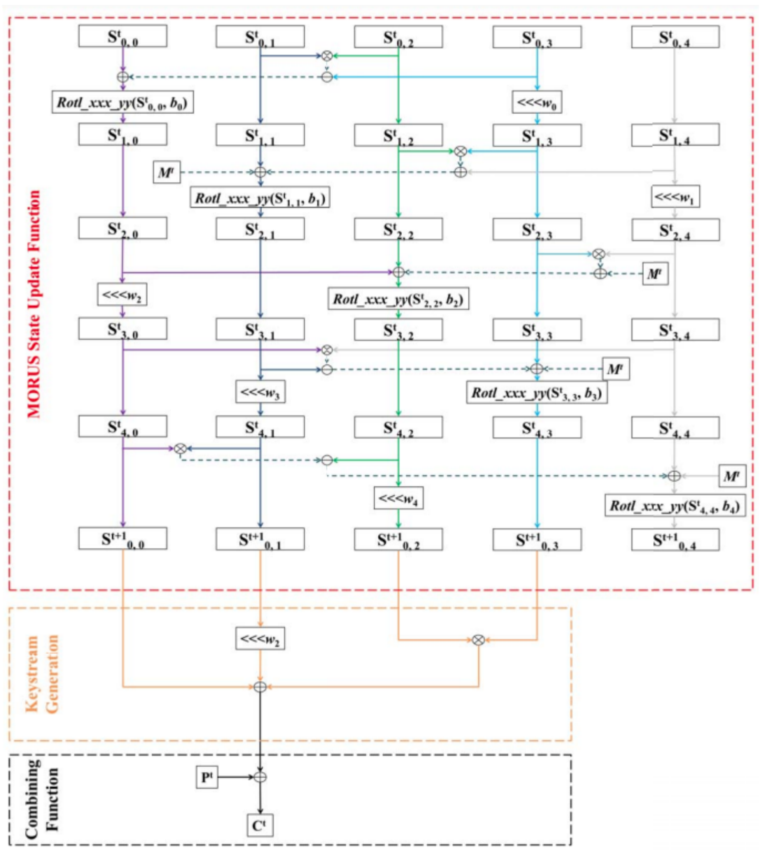


Figure 2.4: MORUS scheme

Chapter 3

Algorithms

The attack is divided into two main phases:

- **Offline**: it consists into finding *maxterm*. This process start with randomly chosen index set, which are used to build the cube assigments for public variable's bits and then store each output bit coming from the cipher, using randomly chosen private keys, and finally use one of linearity tests methods in literature to check if the maxterm we have selected is a factor of linear non-constant polynomial (*superpoly*). However after this, there's only a bunch of bit positions describing the maxterm. They will be used as input for the following phase called *superpoly reconstruction*. The main observation here is that the number of black box cipher's accesses is less than the number of accesses required to interpolate the entire unknown polynomial.
- **Online**: This phase aim to obtain secret key's bits building linear system of equation using superpolies obtained in previous phase. For each superpoly found an attacker must query the target cipher (that have the secret key to recover) and using cubes assignements over the same public value's bit summing the outputs and use it as the right hand of equation.

After these two phases a system of linear equations with secret key's bit as unknown values must be solved and lot of methods, such as Guass elimination, can be adopted[9].

Considering a cryptosystem described by $n + m$ variable polynomial:

$$p(x_1, \cdots, x_n, v_1, \cdots, v_m) \tag{3.1}$$

with n-bit key and m-bit of public variable, the two stage of cube attack are introduced (Fig:3.1, 3.2).

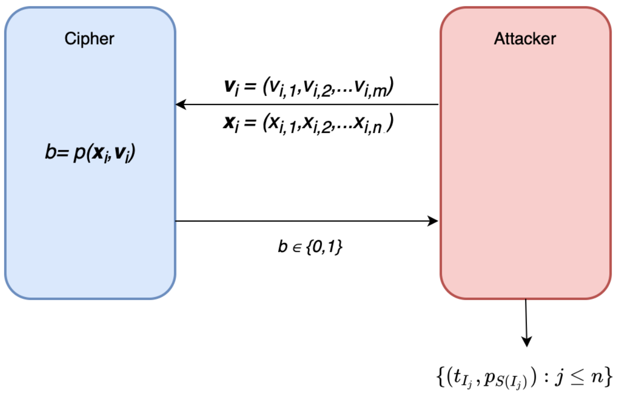

Cipher
$b = p(\mathbf{x}_i, \mathbf{v}_i)$
Attacker
$\mathbf{v}_i = (v_{i,1}, v_{i,2}, \dots v_{i,m})$
$\mathbf{x}_i = (x_{i,1}, x_{i,2}, \dots x_{i,n})$
$b \in \{0,1\}$
$\{(t_{I_j}, p_{S(I_j)}) : j \leq n\}$

Figure 3.1: offline phase scheme

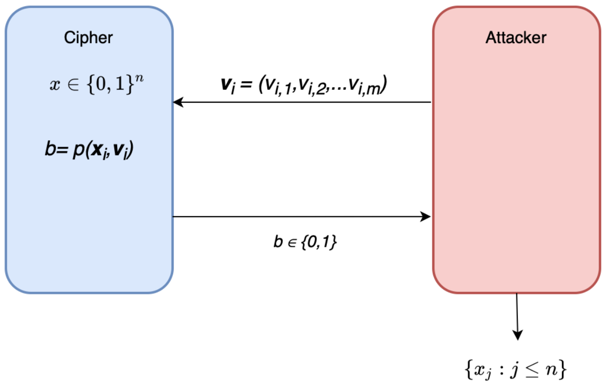

Cipher
$x \in \{0,1\}^n$
$b = p(\mathbf{x}_i, \mathbf{v}_i)$
Attacker
$\mathbf{v}_i = (v_{i,1}, v_{i,2}, \dots v_{i,m})$
$b \in \{0,1\}$
$\{x_j : j \leq n\}$

Figure 3.2: online phase scheme

3.1 Offline phase

3.1.1 Maxterm mining

Dinur and Shamir proposed random-walk approach to finding maxterms[10][13]. I decided to follow these steps:

1. Choose a random size $k \leq m$, and a random index-set $I : |I| = k$
2. For several random private variables compute $p_I = \sum_{v \in C_I} p_{|v}$

- if the cube sums are constant, I is too large: remove a random variable from I and repeat (2)
- if the cube sums are not linear, I is too small: add a new index and repeat (2)
- if the cube sums are linear (non-constant) t_I is a *maxterm*

To check linearity I used Blumed Luby Rubinfeld (BLR) test[10]: given random variables in the domain of $f, i \in [1, 3 \cdot N], K_{i,1}, K_{i,2}$ the BLR test checks:

$$f(0) + f(K_{i,1}) + f(K_{i,2}) = f(K_{i,1} + K_{i,2}) \tag{3.2}$$

if the above condition hold $\forall i$, then f is linear with a probability of $1 - 2^{-N}$.

3.1.2 Superpoly reconstruction

Given a maxterm t_I in a hidden master polynomial $p\,(x_1, \cdots, x_n)$ I wish to reconstruct the superpoly $p_{S(I)}$ in ANF. Given $J \subseteq \{1, \cdots, n\}$ the index set of the private variable in $p_{S(I)}$:

$$p_{S(I)} = a_0 + \sum_{i=1}^{n} a_i x_i \tag{3.3}$$

where a_0 is the constant term and $a_i = 1$ iff $i \in J$. Recalling that J is the index set of the private linear variables in the superpoly. So the problem of finding $p_{S(I)}$ can be reformulated into searching $\{a_0 \cup J\}$.

Theorem 3.1.1 *The coefficient a_i in superpoly $p_{S(I)}$ is computed by the summing over cube C , setting all but i-th variable to zero, and adding the result to the constant term a_0.*

```
Data: Master polynomial $p\left(x_1, \cdots, x_n, v_1, \cdots, v_m\right)$
      Maximum dimension $m_d$, Maximum tries $T$ per term
Result: Maxterm index-set $I$ or reject
Begin
    $k \overset{R}{\leftarrow} \{1, \cdots, \min\left(m, m_d\right)\}$ ;
    for $i \leftarrow 1$ to $k$ do
        $I_i \overset{R}{\leftarrow} \{1, \cdots, m\} \setminus I$
    end
    for $i \leftarrow 1$ to $T$ do
        $n_0, n_1 \leftarrow 0, 0$;
        $p_0 \leftarrow p_{C_I}\left(0\right)$;
        if $p_0 = 1$ then
            $n_1 \leftarrow n_1 + 1$;
        else
            $n_0 \leftarrow n_0 + 1$;
        end
        for $i \leftarrow 1$ to $N$ do
            $k_1 \overset{R}{\leftarrow} \{0, 1\}^n$// random key 1
            $k_2 \overset{R}{\leftarrow} \{0, 1\}^n$// random key 2
            $p_1 \leftarrow p_{C_I}\left(k_1\right)$;
            $p_2 \leftarrow p_{C_I}\left(k_2\right)$;
            $p_{1,2} \leftarrow p_{C_I}\left(k_1 \oplus k_2\right)$;
            // Update counters:
            $n_1 \leftarrow n_1 + p_1 == 1 + p_2 == 1$;
            $n_0 \leftarrow n_0 + p_1 == 0 + p_2 == 0$;
            // Linearity test:
            if $p_0 \oplus p_1 \oplus p_2 \neq p_{1,2}$ then
                // Non-linear, add term:
                $k \leftarrow k + 1$;
                $I_k \overset{R}{\leftarrow} \{1, \cdots, m\} \setminus I$;
                break;
            end
        end
        if $n_0 = 2 \cdot N + 1$ or $n_1 = 2 \cdot N + 1$ then
            // Constant, remove term:
            $j \overset{R}{\leftarrow} I$;
            $I \overset{R}{\leftarrow} I \setminus j$;
            $k \leftarrow k - 1$;
        else
            return $I$;
        end
        return reject;
    end
end
```

Algorithm 1: Finding a maxterm procedure

```
Data: Master polynomial $p(x_1, \cdots, x_n, v_1, \cdots, v_m)$
      Maxterm index-set $I$
Result: Reconstructed superpoly $p_{S(I)}$
begin
    $J \leftarrow \emptyset$ // Index of coifficient in superpoly
    for $i \leftarrow 1$ to $n$ do
        $x_i \leftarrow 0$ // Set key to zero
    end
    for $i \notin I$ do
        $v_i \leftarrow 0$;
    end
    $p_I \leftarrow 0$ for $v \in C_I$ do
        $p_I \leftarrow p_I \oplus p_{|v}$;
    end
    $a_0 \leftarrow p_I$ // Retrieving remaining terms
    for $j \leftarrow 1$ to $n$ do
        for $i \leftarrow 1$ to $n$ do
            if $i = j$ then
                $x_i \leftarrow 1$;
            else
                $x_i \leftarrow 0$;
            end
        end
        $p_I \leftarrow 0$;
        for $v \in C_I$ do
            $p_I \leftarrow p_I \oplus p_{|v}$;
        end
        $a_j \leftarrow a_0 \oplus p_I$;
        if $a_j = 1$ then
            $J \leftarrow J \cup \{j\}$;
        end
    end
    return $p_{S(I)} = a_0 + \sum_{j \in J} x_j$ // Return symbolic superpoly
end
```

Algorithm 2: Superpoly reconstruction procedure

```
Data: Master polynomial $p(x_1, \cdots, x_n, v_1, \cdots, v_m)$
      Cube index-set $I$
Result: True if $I$ is a Maxterm, False otherwise
1  begin
2      $n_0, n_1 \leftarrow 0, 0$;
3      $p_0 \leftarrow p_{C_I}(0)$;
4      for $i \leftarrow 1$ to $N$ do
5          $k_1 \xleftarrow{R} \{0,1\}^n$// random key 1
6          $k_2 \xleftarrow{R} \{0,1\}^n$// random key 2
7          $p_1 \leftarrow p_{C_I}(k_1)$;
8          $p_2 \leftarrow p_{C_I}(k_2)$;
9          $p_{1,2} \leftarrow p_{C_I}(k_1 \oplus k_2)$;
           // Update counters:
10         $n_1 \leftarrow n_1 + p_1 == 1 + p_2 == 1$;
11         $n_0 \leftarrow n_0 + p_1 == 0 + p_2 == 0$;
           // Linearity test:
12         if $p_0 \oplus p_1 \oplus p_2 \neq p_{1,2}$ then
13             return False;
14         end
15     end
16     if $n_0 = 2 \cdot N + 1$ or $n_1 = 2 \cdot N + 1$ then
           // Constant, reject:
17         return False;
18     end
19     return True;
20 end
```

Algorithm 3: Cube validation procedure

3.2 Online Phase

With the online phase we target the encryption algorithm on a secret key K_s trying to exploit relations found in offline phase.

Using the above notation we have to find secret key relations computing:

$$p_{C_I}(K_s) = \sum_{v \in C_I} p_{!v} \tag{3.4}$$

On a target unknown K_s.

Given a Maxterm index-set I, $p_{C_I}(k_s)$ is computed and the following relation is considered:

$$p_{S(I)} = p_{C_I}(K_s) \tag{3.5}$$

Given I compute $p_{C_I}(K_s)$ is straightforward. The validation of the superpolies found in the previous phase was validated as follows: the online computation was iterated if exist a relation by setting different private keys K_s.

```
Data: Master polynomial $p(x_1, \cdots, x_n, v_1, \cdots, v_m)$
      Cube index-set $I$
      Superpoly $p_{S(I)}$ as index set of private variable in $J \subseteq \{1, \cdots, n\}$
      key $K_s$
Result: True if $p_{S(I)}$ is a valid Superpoly on Maxterm index-set $I$, False otherwise
1 begin
2     for $i \leftarrow 1$ to $L$ do
3         $p_k \leftarrow p_{C_I}(K_s)$;
4         if $\sum_{i \in p_{S(I)}} K_{s_i} \neq p_k$ then
5             return False;
6         end
7     end
8     return True;
9 end
```

Algorithm 4: Testing Key cracking

3.3 Time complexity

Deian Stefan [13] estimates Time Complexity in *Maxterm mining* to be:

$$O(T \cdot ((9N + 1) \cdot 2^{md})) \tag{3.6}$$

- T is defined as number of tries.
- $3N$ is the number of *BLR* tests (worst case scenario).
- m_d is a user-defined upper bound for I sets lengths.

The most complex operations are the calculus of $p_1, p_2, p_{1,2}$ each of which requires IV vectors obtained by assigning all possible values of bit positions in I and keeping constant the others.

Given n private variables, complexity of *Superpoly reconstruction* is:

$$O((n + 1) \cdot 2^I) \tag{3.7}$$

Preprocessing is considered a *one-time* effort since the goal is to find linear *superpolies* and are then used by an attacker to find secret key bits, however is computationally intensive and takes the major portion of the attack.

The complexity of Online phase, given d the degree of cryptosystem and n the number of secret bits:

$$O(n^2 + n \cdot 2^{d-1}) \tag{3.8}$$

Moreover, the complexity of preprocessing is for one *Maxterm* and one *Superpoly* respectively. The online phase is quicker for ciphers described by a low degree master polynomial, making this attack very attractive and efficient key-recovery.

Chapter 4

CUDA implementation

4.1 Description

4.1.1 Finding maxterm

To maximize the parallelism, new *Maxterm mining* procedures were implemented. This was achieved by reducing the dependency of data and code.

The most trivial solution consists in launching several instances of the same function. However, each kernel thread require high computational resources. This would led to *register spilling*.

Another approach consists in identify parts of procedure that could be executed in parallel with no constraints inherit from data dependency. *Maxterm mining* procedure was splitted in 4 GPU-based tasks:

- *Random_k*: it selects random set dimensions for I which will contains bit positions for IV public information. The number of selected dimension is equal to the number of *INSTANCES* to be launched and it's user defined.

- *random_I_unique*: for each INSTANCES dimension k, a set of indexes I is built. Every set must contain unique bit positions. Thus, data dependency occurs and every thread is used to build different I ensuring the uniqueness of each element.

- *set_cubes_host* compute all different IV vectors obtained by assigning all possible 1/0 value to the bits having position in I. In this way it is performed a kernel call to *generate_IV*. For each I set and each thread id, information about bit position in I is given, by writing IV as result. According to this method, for each cube of length m a kernel with

2^m threads is called.Threads will write IV vectors as result. To maximize the parallelism I used streams to compute IV vectors from different I sets in parallel. This task not included in all versions of proposed program due to global memory limitations.

- The last phase of this new implementation consists into rewrite the original algorithm by exploiting the results achieved in the previous steps. As previously said, the most time-expensive operations are the p_{C_I}. The solution I implemented use the *cuda_encrypt* kernel. In particular, for each cube, a number of threads equal to the number of different IV values are launched. Moreover, each thread call the cipher with a *fixed* private key which is passed as argument. This procedure is performed for each cube. The main problem of the original algorithm is that the cube vertices sets could change during loops executions. In order to overcome this limit, I designed a new approach which consist in running iteratively all the cubes. Then each cube is tested in parallel using streams. Finally a transformation (expansion or reduction) is performed to all the failed cubes. Because the key passed as argument to compute p_{C_I} will not change for all the threads in the *cuda_encrypt* kernel, the key is stored using constant memory. All cubes which succeeded all the BLR filters, were placed in a $maxterm$ list meanwhile the excluded ones will not be processed. The cubes which failed the BLR tests must be updated adding or removing an element randomly.

BLR filtering is the most important task, calling the cipher $2^{|C|} \cdot N$ for each coefficient p_i. In order to maximize CUDA parallelism I designed a kernel procedure called *cuda_encrypt* which call the cipher $2^{|C|}$ times, and overlap each kernel execution using N CUDA streams. However this could led to some false dependency problem introduced either by the pseudocode and the depth-first approach of algorithm, disabling fully utilization of the device. In order to better understand this behaviour, by implementing the following stream design strategies:

1. Depth first order (calling each kernel in a cuda stream sequentially).

2. Breadth first order (combine the same kernel execution on different streams together).

Fig:4.1 summarize the general *Maxterm finding* algorithm's design. Each cube is computed allocating a number of thread equal to the vertices of the cubes (all possible assignments of IV in I).

After N BLR tests over all I sets, the cubes can be placed in a solution list or could be updated according to BLR result, meaning recomputing the cubes vertices.

Finally, a file with valid *maxterms* is produced.

4.1.2 Superpoly reconstruction

The original algorithm code is reported. Some intensive tasks were achieved exploiting the GPU. Private keys generation and cube computing were implemented using these methods:

- key_gen_superpoly_reconstruction: launching a number of threads equals to the key size in bit. The tid-th thread will write in global memory a 0 key, except $K_{th} = 1$.
- cuda_encrypt: is used to compute the cipher over I sets on previously generated private keys.

Every time a cube is computed over a private key, a check must be performed in order to include a coefficient in *superpoly*. A *superpoly* file, having the same indexing of *maxterm* file is written.

4.1.3 Online validation

The online validation is the final step which goal is to find *secret* bit relations. As explained before, GPU is used to call the cipher function over a cube. In this case, the attacker can not set an arbitrary secret key and it can not be changed during the whole process. The secret key is written in *constant memory* and cuda_encrypt_2_exploit kernel is called. The cube and relative superpoly is read from previously generated files, and finally the whole validation process is orchestrated via CPU.

4.2 Cipher specification

Bit and byte order may differ according to cipher design. This lead to different IV generation, superpoly reconstruction methods and afflict online phase.

IV and K are represented using unsigned byte array.

4.2.1 Building cubes

Each cipher of Grain family has a bit/byte-oriented implementation.

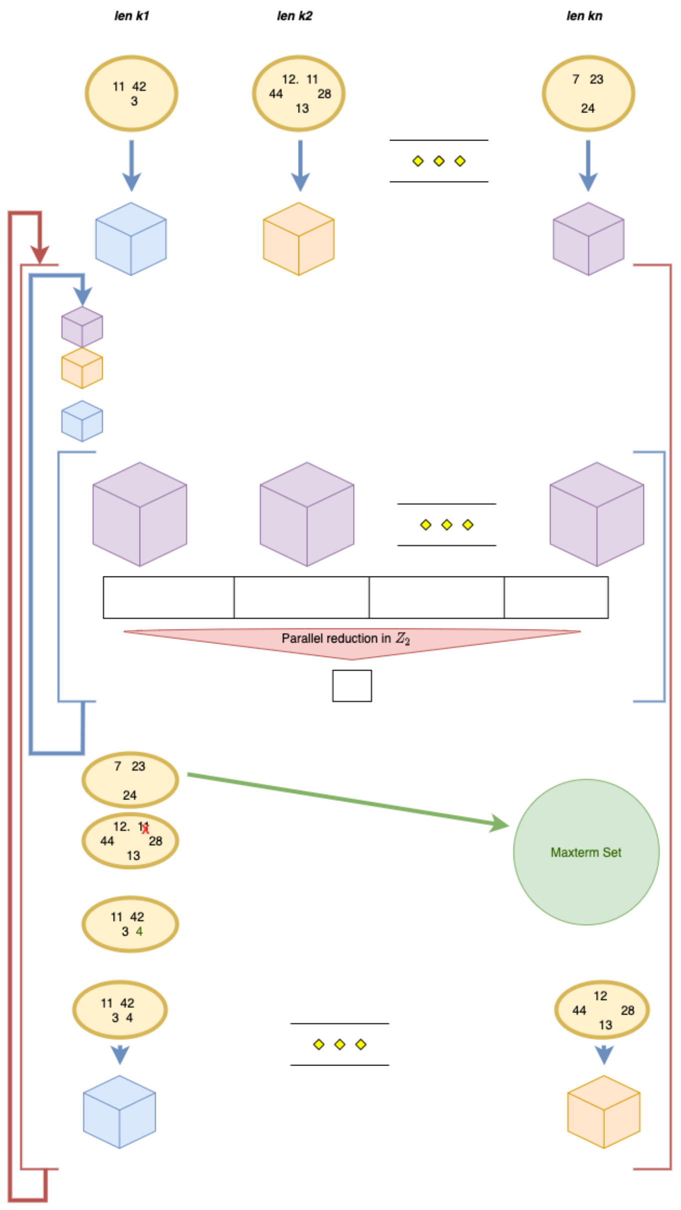


Figure 4.1: Parallel Design of maxterm mining procedure, each cube assignment is replied N times using CUDA streams

If a I set contains these elements: $\{76, 11\}$, according to bit and byte-oriented implementation in [8], the following IV vectors must be generated:

- $[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$
- $[0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0]$
- $[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 16, 0]$
- $[0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 16, 0]$

Trivium [3] has a bit oriented representation the same C will generate these IV:

- $[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$
- $[0, 0, 0, 0, 0, 0, 0, 0, 0, 8]$
- $[0, 16, 0, 0, 0, 0, 0, 0, 0, 0]$
- $[0, 16, 0, 0, 0, 0, 0, 0, 0, 8]$

MORUS([17]) has an opposite order choice than Grain:

- $[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$
- $[0, 0, 0, 0, 0, 0, 0, 0, 0, 16, 0, 0, 0, 0, 0, 0]$
- $[0, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$
- $[0, 8, 0, 0, 0, 0, 0, 0, 0, 16, 0, 0, 0, 0, 0, 0]$

4.2.2 Building superpolies

Recalling superpoly reconstruction equation 3.4 and theorem3.1.1, i indexes for key bits x_i must adhere cipher bit/byte orientation:

1. Grain:

k_0	$0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1$
k_1	$0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2$
	$\dots$
k_{n-2}	$64, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0$
k_{n-1}	$128, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0$

2. Trivium:

k_0	$128, 0, 0, 0, 0, 0, 0, 0, 0, 0$
k_1	$64, 0, 0, 0, 0, 0, 0, 0, 0, 0$
	$\dots$
k_{n-2}	$0, 0, 0, 0, 0, 0, 0, 0, 0, 2$
k_{n-1}	$0, 0, 0, 0, 0, 0, 0, 0, 0, 1$

3. MORUS:

k_0	$1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0$
k_1	$2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0$
	$\dots$
k_{n-2}	$0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 64$
k_{n-1}	$0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 128$

4.2.3 Online assignments

In online phase, equations are formed according to 3.5. In order to check satisfiability on key bits, random K are formed and only the $i - th$ positions are selected according to P_{S_I}

For instance, given $k_3 \oplus k_8 = 1$ as relation and a randomly choosen key vector $K = [7A, B2]$, these assignments are carried out:

- Grain: 01001101 01011110, $k_3 = 0, k_8 = 0$
- Trivium: 01111010 10110010, $k_3 = 1, k_8 = 1$
- MORUS: 01011110 01001101, $k_3 = 0, k_8 = 1$

The equation $k_3 \oplus k_8 = 1$ is valid only on MORUS.

Chapter 5

Code

5.1 Overview

To invoke the ciphers in a kernel CUDA, prototypes of downloaded cipher code had to be changed.

Preparing the function's method for a CUDA kernel give the opportunity to speed up the cipher using *loop unrolling* technique.

The procedure, briefly introduced in the previous section, invoke several time the cipher. Being able to speed it up, means accelerate each kernel thread.

Another improvement was achieved by removing all useless schemes, such as AEAD module for Grain. Hence, we were interested to attack the keystream generation function.

As described in [10] cube attack needs one fixed-position bit of the output ciphertext. One can tweak this value by the following user-defined global variables:

- BIT_POSITION
- BYTE_POSITION

5.1.1 Finding maxterms

My version of maxterm mining procedure could be splitted in 4 phases each of which making use of different strategies provided by the CUDA architecture in order to achieve batter performance:

1. **Select initial lengths**: random set's lengths were selected using *random_k* kernel and *cuRAND* library. Each thread make a *curand_uniform* call writing result in *global memory*.

2. **Build I sets**: using *random_I_unique* kernel, each thread takes a dimension k as input and built its I set. Each element of this set represent a bit positions $\in [0, IV\_dim - 1]$ recalling $|IV| = IV\_dim$. Each element is randomly chosen with *cuRAND* library and unique.

3. **Build Cubes from I sets** *(optional)*: The previous phase return a list of sets I. Each set contains a list of bit position in IV. The goal is to build a set of IV vectors by rounding all bits in bit positions over all possible values while keeping the others constant.

   Each *generate_IV* kernel launch an amount of threads equal to 2^k where $k = |I|$. The set I resides in global memory and each thread, according to thread id, have to build the id-th vector IV_{id} and write it in global memory. For a I set a list of all possible IV vectors over I is called *cube representation of I*. Each cube is processed independently from others enabling the use of cuda streams for increase parallelism. The number of cubes to build is user-defined by INSTANCES. All IV results are written on allocated pinned memory. To overcome memory limitation, this phase could be skipped with a CUDA function *generate_IV*. In *cuda_encrypt* the I set is passed as parameter and with *tid* is possible to build the *tid-th* IV vector.

4. **Searching for maxterms**: Rewriting maxterm mining exploiting previosly created kernels. For each cube generated by an I set, N BLR tests must be performed on the same cube over 3 different random keys plus the empty key. The strategy was to design a parallel execution with a fixed cube over different keys. The execution over one cube with one fixed private key is performed by *cuda_encrypt* kernel (described in Fig:5.1). Each vertices in the cube, represent an assignment over IV vector and each thread in *cuda_encrypt* takes one of this assignment calling the cipher with a key k placed in constant memory. The 3 random keys, for a cube, were executed sequentially. However to maximize the parallelism the N tests were performed exploiting CUDA Streams. Then, the I sets were updated accordingly. Some sets were merged with solutions and no longer considered while others were updated by adding/removing one element. Updating these sets brought the program to recompute *generate_IV* kernel after N tests were performed for each cubes. This phase was implemented in *set_cube* function as I mapped each cube C, to be updated, in one CUDA stream calling *generate_IV* kernel for C.

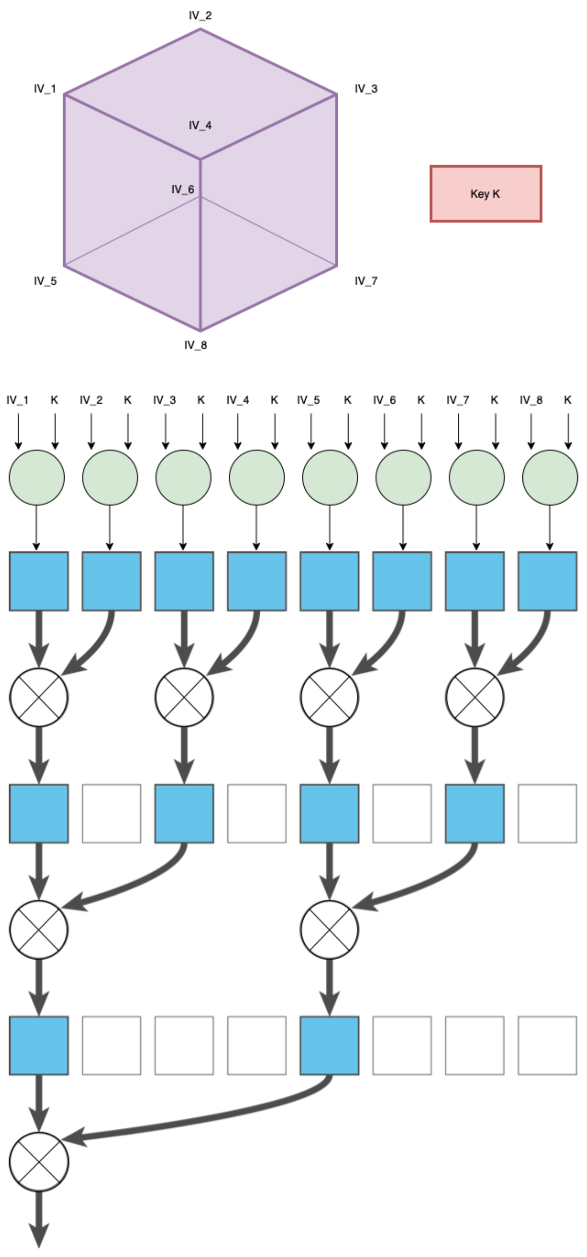


Figure 5.1: *cuda_encrypt* kernel and parallel reduction sum $\mathbb{Z}_2$

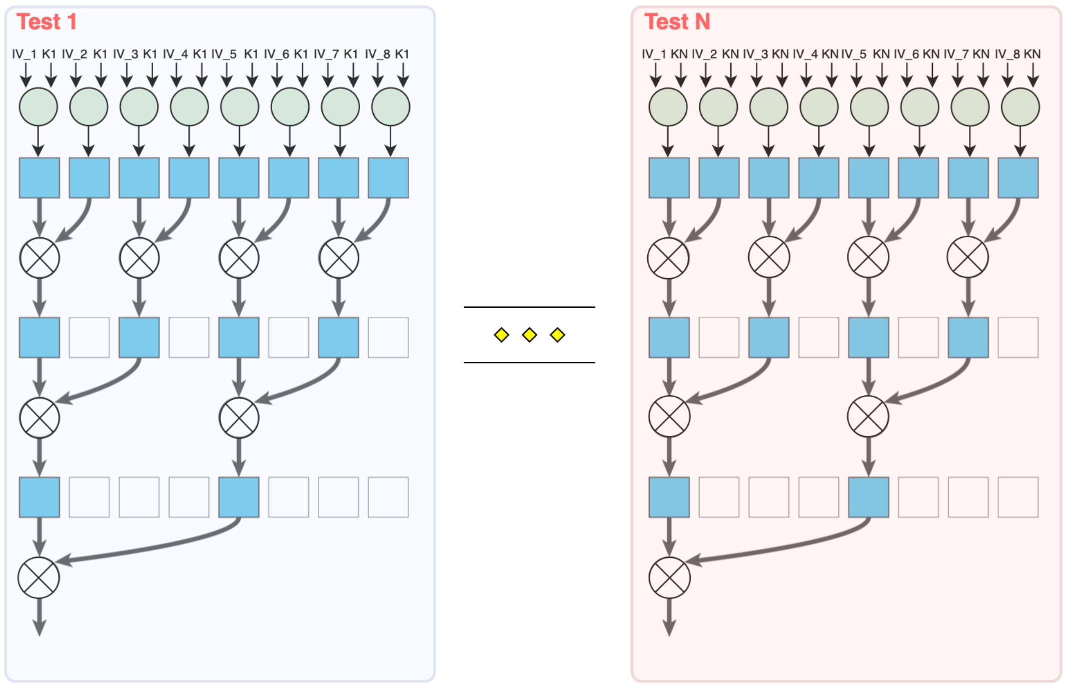

Test 1
IV_1 K1 IV_2 K1 IV_3 K1 IV_4 K1 IV_5 K1 IV_6 K1 IV_7 K1 IV_8 K1
Test N
IV_1 KN IV_2 KN IV_3 KN IV_4 KN IV_5 KN IV_6 KN IV_7 KN IV_8 KN

Figure 5.2: Proposed *maxterm mining* procedure

5.1.2 Superpolies

Superpoly reconstruction approach, begin by reading cubes resulting from the previous phase. Private keys are computed exploiting GPU, using *key_gen_superpoly_reconstruction* where every thread id will generate a specific key K with 1 in K_{tid}. Then the cipher is computed over a cube C and K using *cuda_encrypt_constant_key* saving key in constant memory. The process is iterated for every K and all over the cubes.

5.1.3 Online phase and validation

After *superpolies* are being extracted from the offline phase, the validation process start iterating online step over different keys. Fixing a key K and a maxterm I the process start computing P_{C_I} over K using GPU. Then the assignments in P_{S_I} are carried out according to K and verified on P_{C_I}. All this process is achieved by calling $valid$ procedure computing P_{C_I} exploiting GPU while the assignments are checked via CPU procedure.

5.2 Avoiding dependencies

The last phase, called *Searching for maxterm*, summurize the *maxterm mining* procedure. It employs N computation on each cube retrieving the needed coefficient for BLR testing. Following the step described by the original algorithm, for each filter execution $i \in [1, N]$, will led to dependencies on CUDA streams avoiding device full utilization (Fig:5.3). For each $i \in [1, N]$ *cuda_encrypt*, *sumCubeReduceInterleaved* kernel will be executed sequentially for each coefficient p_0, p_1, p_2, p_{1_2}. Another approach is to group same kernel executions on different streams together. This brings to a new BLR filtering design. The previous approach force host side BLR tests to check the retrieved coefficient using GPU functions: *cuda_encrypt* and *sumCubeReduceInterleaved*. This new approach led to combine different N filter execution in the same stream loop, by pushing breadth first order. Each stream will call the *cuda_encrypt* kernel over all possible assignment in C iterating the process over N different tests. This approach brought to combine the different tests in order to speed up the whole application. Meaning to run a new kernel called *cudaTestBLR* which compute for each $i \in [1, N]$, i.e. all BLR linearity tests. Each test result will be stored into global array *out* where $\forall i \in [0, N-1]$, $out\,[i]$ will be the *i-th* BLR test result, then if *sumCubeReduceInterleaved* over this array is 0 then the cube C pass all N linearity tests. A similar approach is used to check if C is constant, i.e. the previous p_0, p_1, p_2, p_{1_2} return the same value for all tests.

5.3 Key generation

BLR tests requires lot of randomly chosen private keys. This phase could be accomplished following different approaches in terms of random generation and performance. First, I had to consider how many keys must be created in worst case scenario for a given cube C. N parameter is a coefficient for the number of BLR tests a cube must pass in order to be included in a solution. The exact number of BLR tests is N and each test requires 4 different keys:

- k_0 (zero key)
- $k1$ randomly chosen
- $k2$ randomly chosen

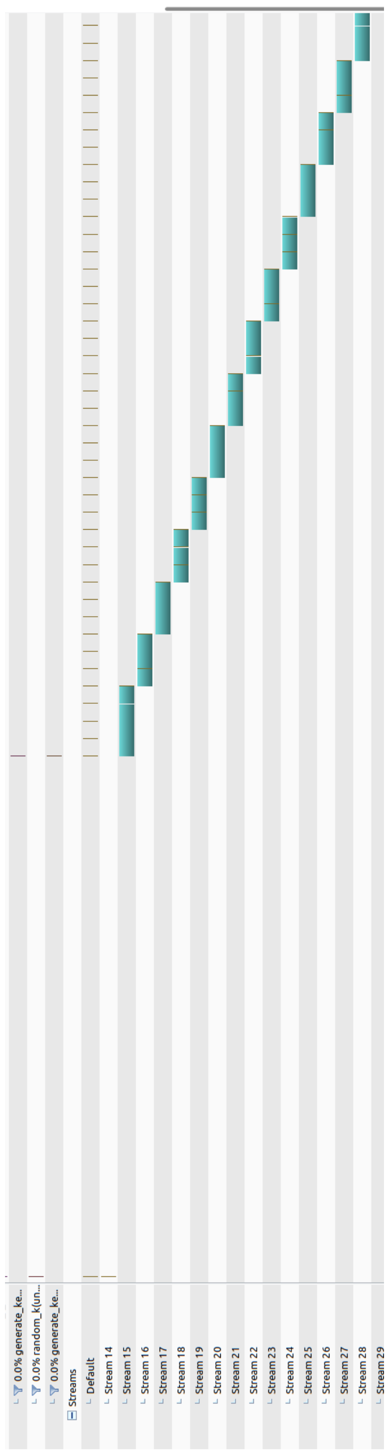


Figure 5.3: dependencies in *searching maxterm* phase

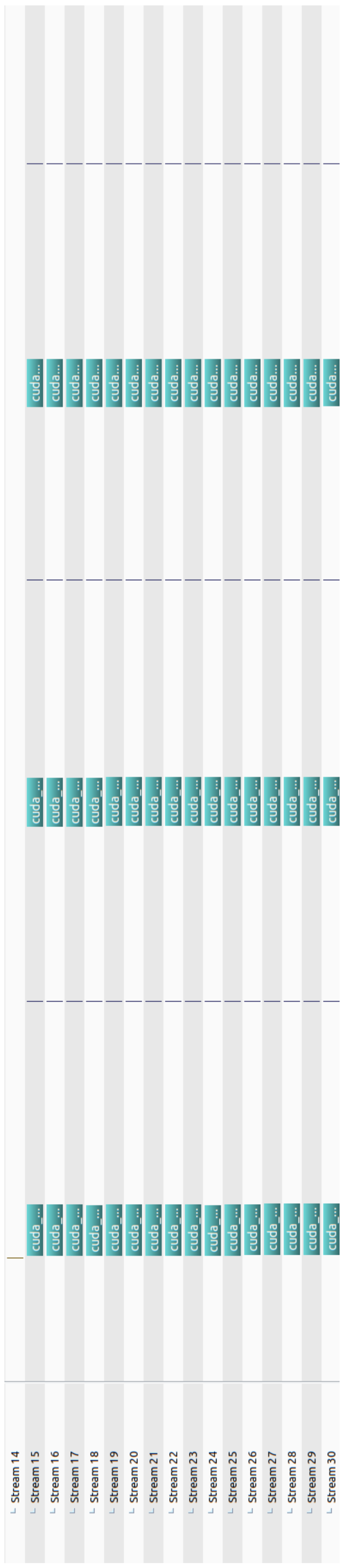


Figure 5.4: *searching maxterm* phase without dependencies

- $k1 \oplus k2$

N streams were allocated along with pinned memory for all possible IV vectors. Given a cube C, each stream perform one BLR test involving the $p_0, p_1, p_2, p_{1,2}$ calculus. Each coefficient require a cuda kernel invoking the *Grain* cipher with a fixed key k over all possible IV assignments in C. For a single coefficient, Grain cipher is invoked N times for a single try T. The number of key to be generated is $2 \cdot T \cdot N$ for a single cube C. The number of private keys to generate is strictly related to the number of BLR tests and number of tries T, however it has nothing to do with the length of C (number of all possible IV assignment). For each different cube, the random private key set could be the same and it will be updated on every different tries for the remainder cubes.

In this work two different approaches on key generation were proposed. Both of these strategies involves the *uniform distribution* over a single byte of randomly chosen key k:

1. Host function *random_key*: every different CUDA stream has to call this function two times for k_1 and k_2 generation. It could have negative impact on the performance since it is iterated over streams.

2. Kernel functions *init_random_key* and *generate_key_set*: exploit device in order to generate random private keys. Because generating one random key is equal to generate 16 bytes randomly, I decided to generate $16 \cdot N$ random bytes exploiting cuRAND. For each $t_1 \in [0, T]$ a set of k_1 is generated for all N cuda streams by a kernel where each thread write random byte in global memory following coalescent and aligned memory access. The same work is performed for the k_2 set. Finally, for each stream, k_1 and k_2 keys are selected following stream id. Additional kernel called *generate_key_set_xor* was employed to generate the xor keys from the previous set by xoring one byte at a time.

Albeith the previous strategies, each *compute_cube* function(which call *cuda_encrypt* under the hood) will compute one cube for a fixed private key k placed in constant memory.

5.4 Deal with global memory limit

The $cuda\_encrypt$ kernel call the cipher $2^{|I|}$ times where I is the candidate set to form a maxterm. Each call then stores the cipher result in an unsigned byte array addressed by tid. Problem on global memory arise when the set I is too large and become impossible to allocate an array of $2^{|I|}$ bytes. For instance on a *Tesla M10* (*Maxwell* microarchitecture) the limit of global memory is 8524136448 bytes (7.9387 Gigabytes), while the max cube length supported by CUDA dynamic allocator is bounded to 32, which means 2^{32} unsigned byte array.

To overcome this, two main strategies could be implemented:

1. make a CUDA thread call the cipher w times where $w = \frac{|I|}{2^{32}}$. Each thread make:

$$global\_arr[tid] = \sum_{i=tid}^{tid+w} enc(K, IV_i)$$

2. call *cuda_encrypt* several times via host code. This means that each iteration will sum partial results before call the kernel for the next window.

In order to avoid device/host memory transfer latency, the first strategy is followed.

Global memory limit set a strict boundary on both key generation and *Build Cubes* strategies. Precomputations of very large arrays will be simply non feasible for very large cubes. Overcome this with window strategy means to generate K via host code and IV dinamically in *cuda_encrypt* kernel (*IV_gen* device function), in order to stay into memory boundary even for very large I sets.

5.5 Cipher code integration

Importing cipher's code is a very challenging task. Different ciphers have different primitives and structures/type management, so finding a uniform representation is required.

The *cube attack* can target different cipher's components and in this work we are focusing on *key-generation* resulting from initialization phase.

However, is still possible to attack other sub-modules. For instance, under the assumption that an attacker can manipulate the internal bit state, it could be taken the *encryption* phase [12].

Make some facilities to import external code is one of the purposes of this work and start from isolating some cipher's specifications.

Here some parameter to set every time a new cipher is imported:

- BIT_POSITION_APP, BIT_POSITION: they define a mask used to select *out bit*.
- IV_dim, K_dim, BIT_K, BIT_IV: dimension in byte and bit respectively of IV and K

- **__device__ uint8_t** d_key_2_guess$[K_DIM]$: device unsigned byte array using to define K in constant memory.
- **uint8_t** h_key_2_guess$[K_DIM]$: unsigned byte array using to define K in host code.

The cipher code is isolated in *cipher.h* file which is imported in each involved step of attack. Then, two functions has to be defined:

1. *cuda_encrypt*: call target function with IV and K as parameter.
2. *cuda_encrypt_exploit*: call target function with IV passed as parameter and K defined in one of the above unsigned byte array.

cuda_encrypt and *cuda_encrypt_exploit* have to return a output bit of target cipher according to BIT/BYTE_POSITION.

A set of cipher's dependant functions are defined:

- IV_gen: given i and set I, it returns the $i - th\ IV$ vector according to I.
- key_gen_superpoly_reconstruction: generate BIT_K private keys with $K_{tid} = 1$.

These functions change according to byte/bit ordering of the cipher. When importing a cipher, one of the four proposed ordering strategy has to be selected.

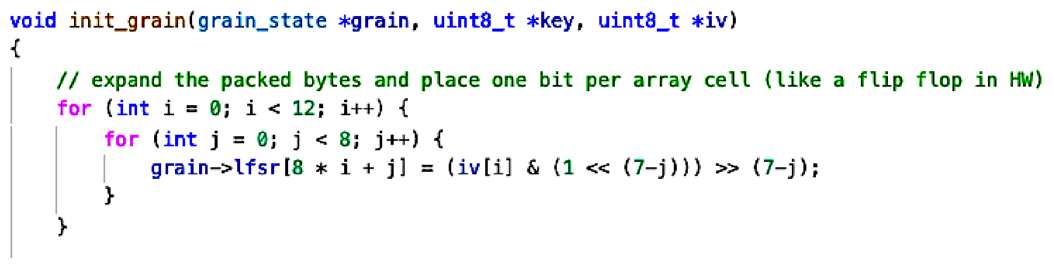
5.6 Cipher code optimization

After the inclusion of the cipher, some CUDA best practises could be applied in order to accelerate the whole process. *Loop unrolling* is a trade-off optimization technique. For a certain factor could improve performance and be beneficial[14].

In addition to *unrolling*, general refactoring of the cipher's code could led to a better memory alignment in read/write accesses. In some cases it is possible to avoid entire useless loops such as in MORUS initialization function(Fig:5.7).

5.7 Profiling

GPUs specifications:



```
void init_grain(grain_state *grain, uint8_t *key, uint8_t *iv)
{
    // expand the packed bytes and place one bit per array cell (like a flip flop in HW)
    for (int i = 0; i < 12; i++) {
        for (int j = 0; j < 8; j++) {
            grain->lfsr[8 * i + j] = (iv[i] & (1 << (7-j))) >> (7-j);
        }
    }
```

Figure 5.5: Grain init *original*

- Nvidia GeForce GT 750M(compute capability 3.0) with CUDA toolkit 10.1(Device does not support HyperQ).
- Tesla M10 (compute capability 5.0) with CUDA 11.4.

For each version of the attack proposed in this work, the correctness was ensured using CPU methods which call the original ciphers proposed by the authors.

In *maxterm mining*, since lot of steps in procedure is random, *Cube validation* is used to analyze performance, fixinig a cube C and N number of BLR tests.

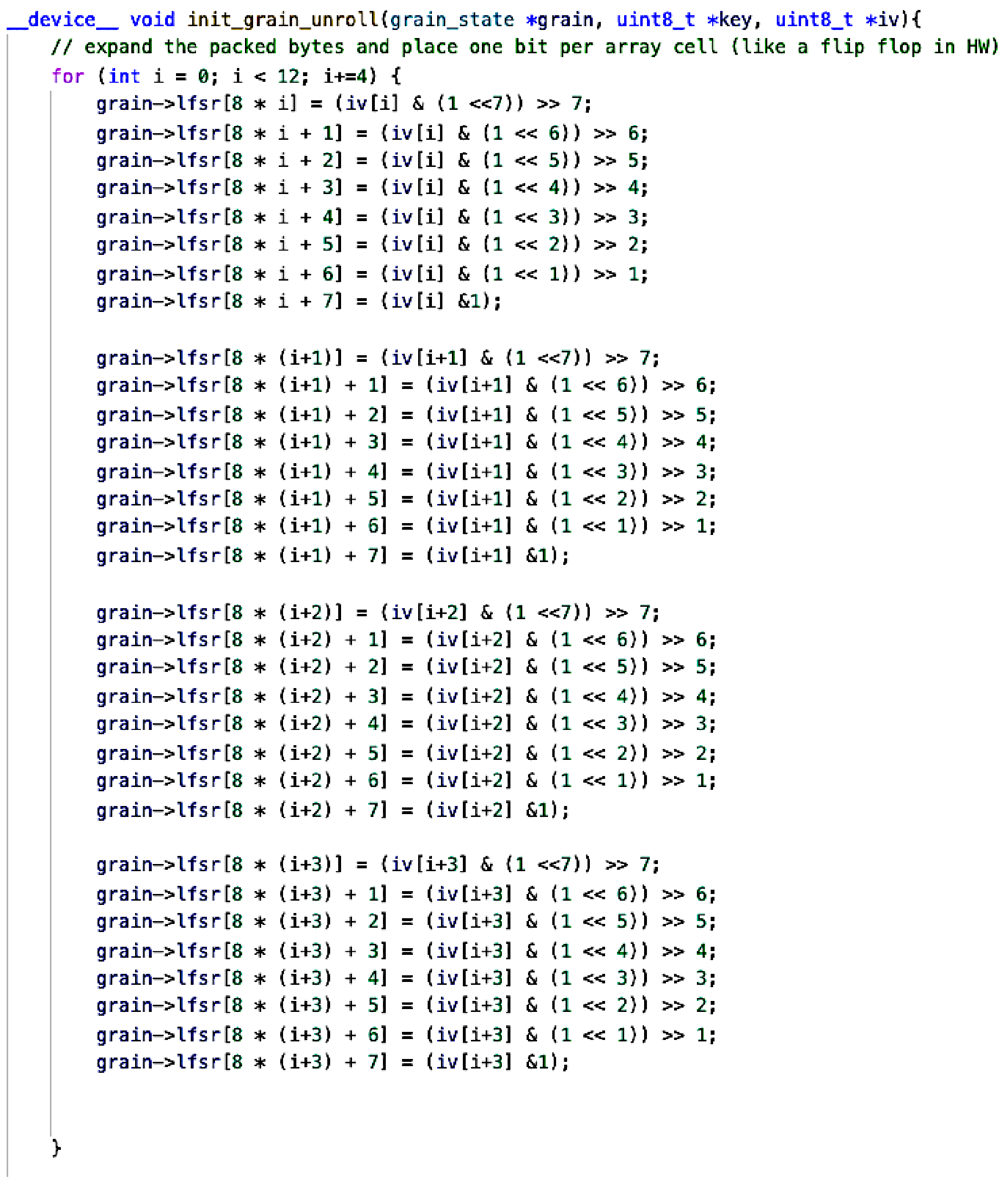
First I compare the author's cipher implementation with its unrolled version launching *cuda_encrypt* kernel function 1024 times with $blocksize = 32$ on the same IV set and private key. Results on *Grain cipher* are reported in the following table:

GPU	KERNEL	TIME	SPEEDUP	GLD THR.	GST THR.
GeForce GT 750M	Classic	59.738ms		2.3587MB/s	66.854KB/s
GeForce GT 750M	Unroll	12.638ms	4.72	11.092MB/s	314.41KB/s
Tesla M10	Classic	325.85μs	38.78	291.04MB/s	2.9104MB/s
Tesla M10	Unroll	267.10μs	1.21	353.88MB/s	3.5388MB/s

In this work, different implementations of maxterm mining procedure were proposed. Retrieve values for a single p_i iterated over all cube vertix and repeated for each BLR filter operation. These coefficient represent the most complex task in whole procedure.

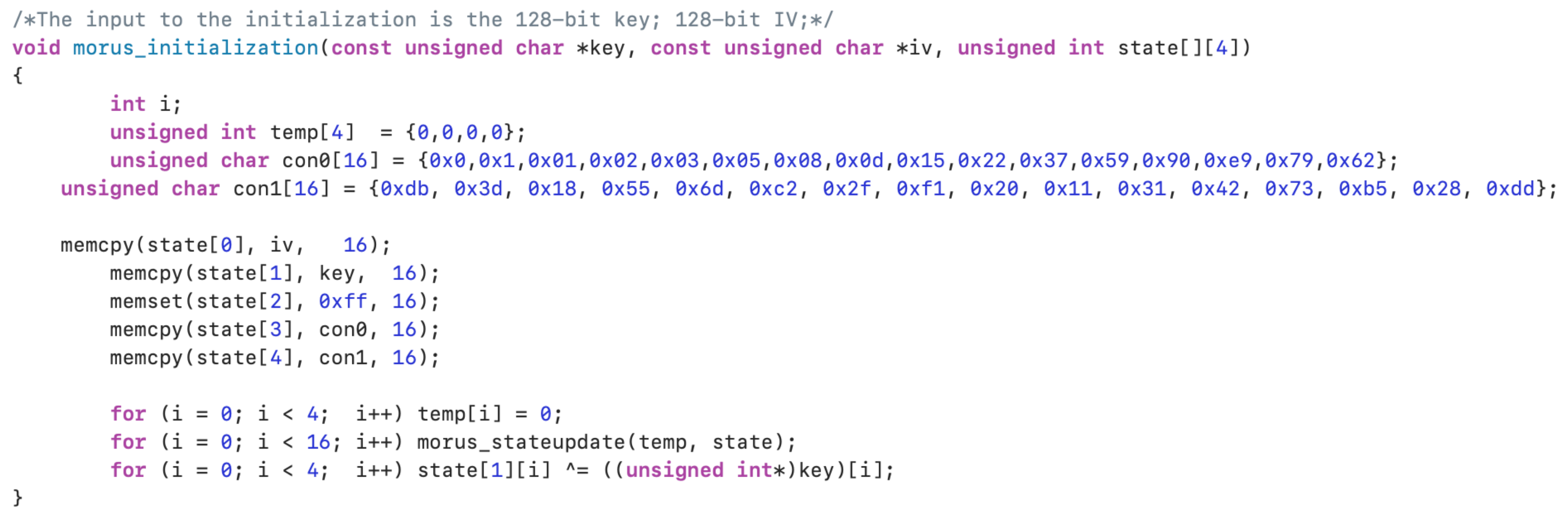
In this section I distinguish different solutions as follow:

- **CPU**: is the CPU implementation of *Cube validation* procedure.
- **GPU version 1**: it generates IV in *cuda_encrypt* kernel, private keys host side, using streams for each BLR test.



```c
__device__ void init_grain_unroll(grain_state *grain, uint8_t *key, uint8_t *iv){
    // expand the packed bytes and place one bit per array cell (like a flip flop in HW)
    for (int i = 0; i < 12; i+=4) {
        grain->lfsr[8 * i] = (iv[i] & (1 <<7)) >> 7;
        grain->lfsr[8 * i + 1] = (iv[i] & (1 << 6)) >> 6;
        grain->lfsr[8 * i + 2] = (iv[i] & (1 << 5)) >> 5;
        grain->lfsr[8 * i + 3] = (iv[i] & (1 << 4)) >> 4;
        grain->lfsr[8 * i + 4] = (iv[i] & (1 << 3)) >> 3;
        grain->lfsr[8 * i + 5] = (iv[i] & (1 << 2)) >> 2;
        grain->lfsr[8 * i + 6] = (iv[i] & (1 << 1)) >> 1;
        grain->lfsr[8 * i + 7] = (iv[i] &1);

        grain->lfsr[8 * (i+1)] = (iv[i+1] & (1 <<7)) >> 7;
        grain->lfsr[8 * (i+1) + 1] = (iv[i+1] & (1 << 6)) >> 6;
        grain->lfsr[8 * (i+1) + 2] = (iv[i+1] & (1 << 5)) >> 5;
        grain->lfsr[8 * (i+1) + 3] = (iv[i+1] & (1 << 4)) >> 4;
        grain->lfsr[8 * (i+1) + 4] = (iv[i+1] & (1 << 3)) >> 3;
        grain->lfsr[8 * (i+1) + 5] = (iv[i+1] & (1 << 2)) >> 2;
        grain->lfsr[8 * (i+1) + 6] = (iv[i+1] & (1 << 1)) >> 1;
        grain->lfsr[8 * (i+1) + 7] = (iv[i+1] &1);

        grain->lfsr[8 * (i+2)] = (iv[i+2] & (1 <<7)) >> 7;
        grain->lfsr[8 * (i+2) + 1] = (iv[i+2] & (1 << 6)) >> 6;
        grain->lfsr[8 * (i+2) + 2] = (iv[i+2] & (1 << 5)) >> 5;
        grain->lfsr[8 * (i+2) + 3] = (iv[i+2] & (1 << 4)) >> 4;
        grain->lfsr[8 * (i+2) + 4] = (iv[i+2] & (1 << 3)) >> 3;
        grain->lfsr[8 * (i+2) + 5] = (iv[i+2] & (1 << 2)) >> 2;
        grain->lfsr[8 * (i+2) + 6] = (iv[i+2] & (1 << 1)) >> 1;
        grain->lfsr[8 * (i+2) + 7] = (iv[i+2] &1);

        grain->lfsr[8 * (i+3)] = (iv[i+3] & (1 <<7)) >> 7;
        grain->lfsr[8 * (i+3) + 1] = (iv[i+3] & (1 << 6)) >> 6;
        grain->lfsr[8 * (i+3) + 2] = (iv[i+3] & (1 << 5)) >> 5;
        grain->lfsr[8 * (i+3) + 3] = (iv[i+3] & (1 << 4)) >> 4;
        grain->lfsr[8 * (i+3) + 4] = (iv[i+3] & (1 << 3)) >> 3;
        grain->lfsr[8 * (i+3) + 5] = (iv[i+3] & (1 << 2)) >> 2;
        grain->lfsr[8 * (i+3) + 6] = (iv[i+3] & (1 << 1)) >> 1;
        grain->lfsr[8 * (i+3) + 7] = (iv[i+3] &1);


    }
```

Figure 5.6: Grain init *unrolled*



```c
/*The input to the initialization is the 128-bit key; 128-bit IV;*/
void morus_initialization(const unsigned char *key, const unsigned char *iv, unsigned int state[][4])
{
        int i;
        unsigned int temp[4]  = {0,0,0,0};
        unsigned char con0[16] = {0x0,0x1,0x01,0x02,0x03,0x05,0x08,0x0d,0x15,0x22,0x37,0x59,0x90,0xe9,0x79,0x62};
    unsigned char con1[16] = {0xdb, 0x3d, 0x18, 0x55, 0x6d, 0xc2, 0x2f, 0xf1, 0x20, 0x11, 0x31, 0x42, 0x73, 0xb5, 0x28, 0xdd};

    memcpy(state[0], iv,   16);
        memcpy(state[1], key,  16);
        memset(state[2], 0xff, 16);
        memcpy(state[3], con0, 16);
        memcpy(state[4], con1, 16);

        for (i = 0; i < 4;  i++) temp[i] = 0;
        for (i = 0; i < 16; i++) morus_stateupdate(temp, state);
        for (i = 0; i < 4;  i++) state[1][i] ^= ((unsigned int*)key)[i];
}
```

Figure 5.7: MORUS *temp* array

- **GPU version 2**: makes each p calculus independent. It combine *cuda_encrypt* invocation over different p in the same breadth.

The following table compare the execution time of *Cube valid* using TRIVIUM cipher, with $2^8, 2^{12}$ vertices, $N\_ROUND = 576, 672$ and $N = 128, N = 1024$ respectively:

HW	**CUBE 2^8, N=128, N_ROUND=576**	**TIME**	**SPEEDUP**
Intel i7-4700HQ	CPU	$11sec.$	
GeForce GT 750M	GPU version 1	$251ms$	43.82
GeForce GT 750M	GPU version 2	$173ms$	1.45
Tesla M10	GPU version 1	$246ms$	-1.42
Tesla M10	GPU version 2	$173ms$	1.42

HW	**CUBE 2^8, N=1024, N_ROUND=576**	**TIME**	**SPEEDUP**
Intel i7-4700HQ	CPU	$1.33min.$	
GeForce GT 750M	GPU version 1	$1.42s$	65.49
GeForce GT 750M	GPU version 2	$905ms$	1.56
Tesla M10	GPU version 1	$913ms$	-1.00
Tesla M10	GPU version 2	$370ms$	2.46

HW	**CUBE 2^{12}, N=128, N_ROUND=672**	**TIME**	**SPEEDUP**
Intel i7-4700HQ	CPU	$4.18min.$	
GeForce GT 750M	GPU version 1	$1.90s$	132
GeForce GT 750M	GPU version 2	$1.86s$	1.02
Tesla M10	GPU version 1	$557ms$	3.33
Tesla M10	GPU version 2	$515ms$	1.08

HW	**CUBE 2^{12}, N=1024, N_ROUND=672**	**TIME**	**SPEEDUP**
Intel i7-4700HQ	CPU	$28.70min.$	
GeForce GT 750M	GPU version 1	$14.57s$	118
GeForce GT 750M	GPU version 2	$14.29s$	1.01
Tesla M10	GPU version 1	$2.93s$	4.87
Tesla M10	GPU version 2	$2.65s$	1.10

Here is reported some statistics about Global memory read/write of GPU_version_2 on *Grain*:

HW	GPU_version 2($\lvert C\rvert = 6$)	gld_eff.	gst_eff.	gld_tr.	gst_tr.
GeForce GT 750M	cudaTestBLR	100%	100%	9	3
GeForce GT 750M	generate_key_set_xor	100%	100%	96	48
GeForce GT 750M	cuda_encrypt	6.73%	14.84%	136	8
Tesla M10	cudaTestBLR	100%	100%	36	3
Tesla M10	generate_key_set_xor	100%	100%	384	48
Tesla M10	cuda_encrypt	6.25%	5.47%	48	4

Here it is reported a brief description of remaining kernel functions used in previous phase of the procedure:

1. *random_k*: make use of *cuRAND* library creating $INSTANCES$ initial random dimensions of cubes. Each length is limited by M_d parameter.

2. *random_I_unique*: make use of *cuRAND* library creating initial I sets for each cube.

3. *generate_IV*: make use of cuda streams, one for each cube C, to allocate IV vectors and computing all possible assignment for each C.

4. *IV_gen*: alternative way to generate IV vector directly in *cuda_encrypt* function. Used to deal with large cubes.

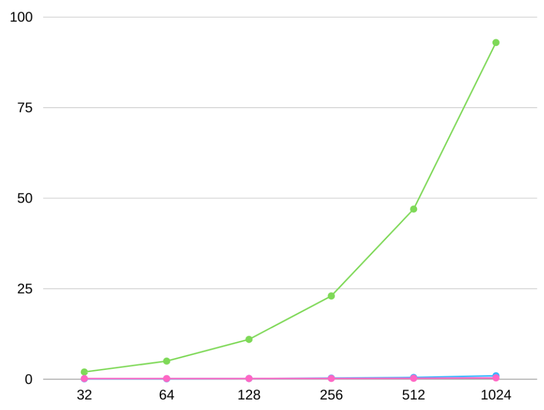


Figure 5.8: Trivium 576 rounds, $|I| = 8$, y : exec. time (s), x : N BLR tests
🟩 Intel i7-4700HQ
🟦 GeForce GT 750M
🟥 Tesla M10

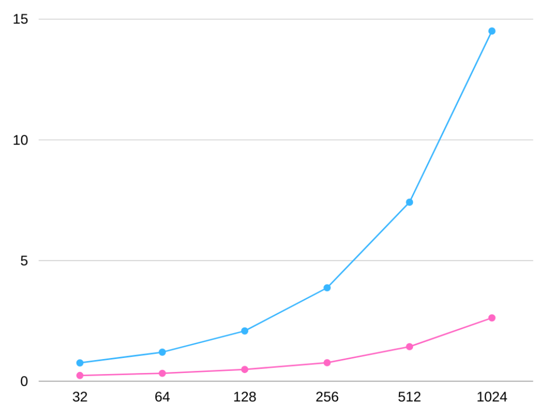


Figure 5.9: Trivium 672 rounds, $|I| = 12$, y : exec. time (s), x : N BLR tests
🟦 GeForce GT 750M
🟥 Tesla M10

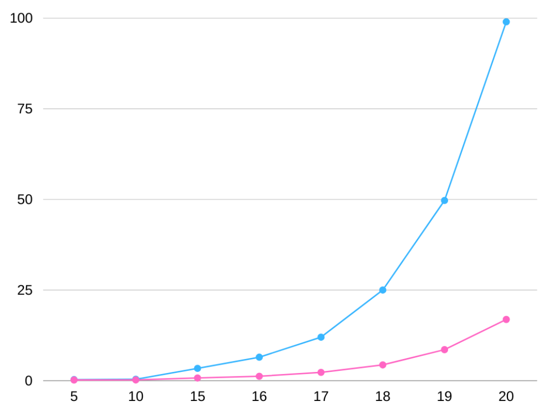


Figure 5.10: Trivium 576 rounds, N BLR tests = 32, y : exec. time (s), x : $|I|$ size
🟦 GeForce GT 750M
🟥 Tesla M10

5.8 Usage

The *offline* phase of the attack is orchestrated using a Bash script.

Then, after a user-defined number of relations are extracted, the proposed iterated *online phase* could start. For this, a Python3 script is used.

```
Result: cube_test.txt and superpolies.txt files with maxterms and superpolies extracted
        from the cipher
1 begin
2     launch MaxtermMining program to find Maxterms
3     write results in cube_test_window.txt
4     append cube_test_window.txt in cube_test.txt
5     append superpolies_window.txt in superpolies.txt
6     if superpolies are not enough then
7         GOTO 2
8     end
9 end
```

Algorithm 5: offline phase script

```
Data: superpolies.txt and cube_test.txt
Result: a list of valid relations
1  begin
2      for $I \in$ cube_test.txt do
3          for $i \leftarrow 1$ to $100$ do
4              generate a random key $K$
5              $p_k \leftarrow$ compute the cipher over $C_I$ and $K$ using CUDA program
6              $p'_k \leftarrow$ according to $K$ give the proper assignments to $P_{S_I}$
7              if $p_k \neq p'_k$ then
8                  break
9              end
10         end
11         if $i = 100$ then
12             $I$ is valid maxterm and associated $P_{S_I}$ is valid
13         end
14     end
15 end
```

Algorithm 6: online phase script

Chapter 6

Results

Some of Maxterms and Superpolies reported from literature [18][2][5][12] and validated by our implementation is reported.

6.0.1 Trivium

round	out_bit	maxterm	superpoly
576	579	3,20,28,36,42,55,77,78	K_{68}
576	579	1,20,23,45,49,56	$1 \oplus K_{12}$
576	579	2,11,14,24,31,34	$1 \oplus K_3 \oplus K_{36}$
576	579	3,9,11,30,33,70	K_{37}
576	579	18,26,36,45,61,73,78,79	K_{67}
576	579	5,9,10,11,12,42,68,77	K_3
576	579	1,2,8,39,61,62,69,70	K_{52}
672	673	0,5,8,11,13,21,22,26,36,38,53,79	K_{12}
672	673	2,12,17,25,37,39,46,48,54,56,65,78	$1 \oplus K_0 \oplus K_{24}$
672	673	2,4,21,23,25,41,44,54,58,66,73,78	K_{22}
672	673	3,9,14,20,23,27,31,47,48,63,70,72	K_{31}
799	799	2,5,6,9,10,13,21,23,25,27,29,32,34,36,38,40,42,44,45 ,48,51,53,55,57,59,63,65,68,73,78	K_1
799	799	2,5,6,9,13,16,19,21,23,25,27,29,30,32,34,36,38,40,42 ,44,45,48,53,57,59,61,65,68,73,75,78	K_3

6.0.2 MORUS-640-128

round	out_bit	maxterm	superpoly
4	20	49,13,110	K_4
4	17	27,107,61	K_{18}
4	83	7,6,33	K_{44}
4	123	102,83,22	$K_{60} \oplus 1$
4	95	107,104,58	$K_{64} \oplus 1$
4	59	69,21,9	$K_{52} \oplus 1$
4	65	85,10,124	$K_{34} \oplus 1$
4	31	58,68,40	K_{49}
4	60	7,104,125	$K_{87} \oplus 1$
4	93	102,119,56	$K_{94} \oplus 1$
4	103	1,66,19	$K_{104} \oplus 1$
4	58	82,106,3	K_{120}

Chapter 7

Conclusion

In this work it is presented the Cube attack implemented on GPUs which has proved to be well-suited for parallel computation. By Highlighting the independent subset of instructions in the procedure, it was possible to partially overcome calculations complexity.

Although the unrolling technique has improved the cipher, the procedure was really inefficient in terms of occupancy and load efficiency.

The validation of the attack on GPUs has shown to be up 132 times faster than CPUs. However, by increasing the number of threads for each kernel, the performance decreased.

The implementation support offline and online stage of the attack managing more than one cubes with update operation exploiting GPU and CUDA streams. A good performance was achieved.

In this work, multiple versions of the *maxterm mining* procedure were proposed. In particular, it is showed that improvements in term of precomputing were obtained, despite the fact of the limited hardware resources.

The global memory limitation was managed using a *window* strategy by increasing latency on each kernel thread.

Importing cipher was challenging because of different coding standards and different cipher schemes. Hence, by isolating specification-dependent functions and by implementing a general interface, made context switching between different encryption functions easier.

After the development of the attack, it was tested on three different ciphers. In detail three ciphers were attacked: Trivium, MORUS and Grain.

New *maxterms* with relative *superpolies* were found and validated through the proposed method.

Observing more than one bit produced by the attacked cipher's function, lead to discover more relations using the same input variables. This approach could maximize the results reached by our work and will be included in future release of this project.

Appendix A

Trivium

round	out_bit	maxterm	superpoly
576	576	13,6,44,38,47,33	K_{40}
576	576	32,21,9,34,45,1	$K_{54} \oplus K_{60}$
576	576	74,11,43,20,61,50	K_{20}
576	576	59,67,21,34,17,51,35,64,18,19	K_{56}
576	576	21,48,37,44,41,32,79,27	$1 \oplus K_{54} \oplus K_{60}$
576	576	66,79,21,58,55,41,12,63	K_{25}
576	576	56,46,55,77,34,17,43	K_{19}
576	576	32,22,61,58,6,17,35	K_{62}
576	576	48,62,12,78,10,35,9	$1 \oplus K_6$
576	576	47,5,20,67,57,10,36,71,9	K_4
576	577	59,24,16,75,44,29,70,30	$1 \oplus K_{18}$
576	577	79,51,32,6,71,25	K_{40}
576	577	72,36,61,65,32,29,42,73,50,28	$1 \oplus K_{52}$
576	577	5,14,78,26,52,30	$1 \oplus K_{20}$
576	577	10,79,44,45,54,53,72,35,43	K_{39}
576	577	71,31,61,2,74,64,3	K_{58}
576	577	69,67,49,34,47,53,70,41,1,72	K_{56}
576	577	17,60,3,7,58,19,41	K_{19}
576	577	47,45,13,77,67,64,29	K_{56}
576	577	47,24,76,14,7,57,48	K_{18}
576	578	49,56,1,26,62,78,67,60	$1 \oplus K_2 \oplus K_{65}$
576	578	7,46,31,27,11	K_3
576	578	61,6,8,76,20,56,34	K_{63}
576	579	77,52,36,60,65,68,59,51,19	K_{54}
576	579	16,40,13,11,8,68,12,2,5	$1 \oplus K_1$
576	579	75,3,29,28,6,13,12,27,49	$K_3 \oplus K_{51}$
576	579	64,67,34,33,72,31,56,71,15,12	K_{54}
672	673	23,7,8,61,14,34,66,60,2,54,73,17,31,3,74,63	$1 \oplus K_{21} \oplus K_{51}$
672	673	8,21,7,69,62,48,19,20,64,66,17,46,45,2,18,47,78	K_{55}

round	**out_bit**	**maxterm**	**superpoly**
672	673	79,17,23,73,58,63,33,1,62,56,4,40,50,25,76,16	K_{65}
672	673	68,72,42,77,34,11,43,20,26,64,21,24,47,22,66	K_{66}
672	673	53,60,17,21,46,69,56,70,51,39,16,37,10,12,38,74 ,15,66	K_{63}
672	673	36,24,47,43,77,26,58,5,1,50,60,12,55,52,4	K_{64}
672	674	51,15,53,10,2,72,21,56,68,60,12,66,9,49,43,59	K_{55}
672	674	17,77,64,26,57,15,78,6,12,62,61,31,22,59,76	K_{63}
672	674	58,23,73,48,46,4,63,64,17,62,36,7,29,27,49,1,13 ,68,75	K_{62}
672	674	74,43,4,6,14,15,36,51,40,76,48,60,67,42,65,19,9	$1 \oplus K_{54} \oplus K_{63}$
672	674	69,14,39,32,19,62,17,75,53,26,58,52,16,10,42,20 ,25,43	$1 \oplus K_{21}$
700	702	58,22,44,69,60,33,10,27,5,36,31,79,13,57,15,26 ,34,61,21,4,23,35	$1 \oplus K_{65}$
700	702	78,38,26,50,52,76,79,13,75,73,72,3,2,22,31,74, 7,29,42,60,44	K_{65}
700	702	48,64,20,23,32,49,38,27,71,67,68,57,24,36,18, 39,2,65,45,5,15,53,79,47	K_{65}
700	702	47,3,34,12,25,26,22,6,16,41,50,79,27,29,57,60, 9,33,52,23	K_{64}

Appendix B

MORUS-640-128

round	out_bit	maxterm	superpoly
4	103	85,2,56	$1 \oplus K_{40}$
4	103	24,58,98	K_{79}
4	103	93,42,104,6	$1 \oplus K_{115}$
4	103	55,2	$1 \oplus K_{40}$
4	103	65,19,72	$1 \oplus K_{95} \oplus K_{121}$
4	103	105,23,26,90,43	$1 \oplus K_{102}$
4	103	123,41	$1 \oplus K_{77} \oplus K_{103}$
4	103	24,2	$1 \oplus K_{40}$
4	103	96,65,1	K_{119}
4	103	65,113,16	$1 \oplus K_{104}$
4	103	82,65,3,14	K_{123}
4	103	13,33,79,1,86	K_{110}
4	103	41, 123	$1 \oplus K_{77} \oplus K_{103}$
4	103	2,48	$1 \oplus K_{40}$
4	103	59,83,3,16	K_{121}
4	103	95,106,33,105,86	K_{44}
4	103	2,24	$1 \oplus K_{40}$
4	103	65,85,58	$1 \oplus K_{64}$
4	103	2,89,77	$1 \oplus K_{40}$
4	103	19,113,65	$1 \oplus K_{104}$
4	103	56,95,2	$1 \oplus K_{40}$

round	**out_bit**	**maxterm**	**superpoly**
4	103	71,82,2,77	$1 \oplus K_{25}$
4	103	74,25,87,4,43	$1 \oplus K_{113}$
4	103	121,105,125,11,98,110,7	K_{45}
4	103	72,33,86,91,25	K_{82}
4	103	33,86,57,6	K_{95}
4	123	110,45,69,117,93	K_{107}
4	123	6,55,86,109,37	$1 \oplus K_{17} \oplus K_{43}$
4	123	22,36	$1 \oplus K_{60}$
4	123	63,43	K_{69}
4	123	94,63,81,79	K_{69}
4	123	36,63	$1 \oplus K_{69}$
4	123	70,60,22	$K_{13} \oplus K_{39}$
4	123	22,97,112	$1 \oplus K_{60}$
4	123	22,12	$1 \oplus K_{60}$
4	123	7,54,44,23	$1 \oplus K_{102}$
4	123	27,77,9,26,107	K_{103}
4	123	84,77,62,113	$1 \oplus K_{75} \oplus K_{101}$
4	123	45,62,92,50	$1 \oplus K_{98}$
4	123	4,126,22	$K_{13} \oplus K_{39}$
4	123	43,63	K_{69}

Appendix C

Grain-128AEAD

round	out_bit	maxterm	superpoly
8	8	39,35,88,52	K_{30}
8	8	15	$1 \oplus K_{46} \oplus K_{78} \oplus K_{93} \oplus K_{104} \oplus K_{110}$
8	8	28,85	K_{110}
12	12	53,9	K_{29}
12	12	21,66	$1 \oplus K_{31} \oplus K_{63} \oplus K_{78} \oplus K_{89} \oplus K_{95} \oplus K_{121} \oplus K_{125} \oplus K_{127}$
16	16	2,83	K_{112}
16	16	20,70	$1 \oplus K_{31} \oplus K_{37} \oplus K_{53} \oplus K_{62} \oplus K_{63} \oplus K_{78} \oplus K_{81} \oplus K_{89} \oplus K_{90} \oplus K_{95} \oplus K_{111} \oplus K_{121} \oplus K_{124} \oplus K_{125} \oplus K_{127}$
16	16	53,2	$1 \oplus K_{29}$
16	16	83,13	K_{112}
24	24	66,12,40	K_{26}
24	24	12,89,54	K_{28}
24	24	57,43,2,62	K_{23}
24	24	1,69	K_{94}
24	24	89,19,39,28,41	K_{25}
24	24	31,36	K_{14}
24	24	1,61,86	K_{109}
24	24	12,74	K_{105}
24	24	4,70	K_{93}

round	**out_bit**	**maxterm**	**superpoly**
24	24	84,1,62	K_{111}
24	24	12,94,73	K_{106}
24	24	39,58,29,45,6	K_{21}
24	24	92,12,86	K_{109}
25	25	87,2,65	K_{108}
25	25	13,87,51	K_{108}
25	25	93,9,30	K_{58}
25	25	22,58,7,36,63	K_{14}
25	25	16,37	K_{13}
26	26	25,14,83	K_{112}
26	26	46,3	K_{20}
27	27	39,18	K_{11}
27	27	4,69,73	K_{106}
28	28	0,32	K_{18}
29	29	13, 93, 6, 20, 40	$1 \oplus K_{26}$
30	30	7,19,63	K_{61}
30	30	63,19,90	K_{61}
30	30	44,2,95	K_{22}
31	31	72,61,3	K_{107}

Bibliography

[1] Lawrence Bassham et al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. en. 2010. URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=906762 (visited on 04/02/2020).

[2] SS Bedi and N Rajesh Pillai. "Cube attacks on Trivium". In: *Cryptology ePrint Archive* (2009). (Visited on 06/13/2020).

[3] Christophe De Canniere and Bart Preneel. *TRIVIUM Specifications*. (Visited on 05/16/2020).

[4] Itai Dinur and Adi Shamir. *Breaking Grain-128 with Dynamic Cube Attacks*. Cryptology ePrint Archive, Report 2010/570. https://ia.cr/2010/570. 2010. (Visited on 04/02/2021).

[5] Pierre-Alain Fouque and Thomas Vannet. *Improving key recovery to 784 and 799 rounds of Trivium using optimized cube attacks*. Springer, 2013. (Visited on 06/02/2021).

[6] GitHub. *Cube attack POC*. https://github.com/MarcoGarlet/crypto/blob/master/attack/cube/att.py. 2020. (Visited on 06/06/2021).

[7] Jovan Dj. Golić. *Modes of Operation of Stream Ciphers*. Ed. by Douglas R. Stinson and Stafford Tavares. Berlin, Heidelberg, 2001. (Visited on 01/01/2021).

[8] Martin Hell et al. "Grain-128AEADv2-A lightweight AEAD stream cipher". In: (). (Visited on 06/16/2021).

[9] Jeff Howe and Shawn Bratcher. *Parallel Gaussian Elimination*. 1996. (Visited on 03/02/2020).

[10] Adi Shamir Itai Dinur. *Cube Attacks on Tweakable Black Box Polynomials*. 2009. DOI: https://doi.org/10.1007/978-3-642-01001-9_16. eprint: 978-3-642-01000-2. (Visited on 01/08/2020).

[11] *NIST Lightweight Cryptography*. URL: `https://csrc.nist.gov/projects/lightweight-cryptography` (visited on 02/01/2022).

[12] Iftekhar Salam et al. "Investigating cube attacks on the authenticated encryption stream cipher MORUS". In: *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE. 2017, pp. 961–966. (Visited on 11/12/2021).

[13] Deian Stefan. "Analysis and Implementation of eSTREAM and SHA-3 Cryptographic Algorithms". In: 2011. (Visited on 08/11/2020).

[14] John E Stone et al. "Accelerating molecular modeling applications with graphics processors". In: *Journal of computational chemistry* 28.16 (2007), pp. 2618–2640. (Visited on 03/11/2020).

[15] Yao Sun. *Cube Attack against 843-Round Trivium*. Cryptology ePrint Archive, Report 2021/547. `https://ia.cr/2021/547`. 2021. (Visited on 01/02/2022).

[16] Michael Vielhaber. *Breaking ONE.FIVIUM by AIDA an Algebraic IV Differential Attack*. vielhaber@gmail.com 13814 received 28 Oct 2007. 2007. URL: `http://eprint.iacr.org/2007/413` (visited on 01/07/2021).

[17] Hongjun Wu and Tao Huang. *The authenticated cipher MORUS*. 2014. (Visited on 10/12/2021).

[18] Bo Zhu, Wenye Yu, and Tao Wang. "A practical platform for cube-attack-like cryptanalyses". In: *Cryptology ePrint Archive* (2010). (Visited on 06/11/2020).

This page intentionally left blank