Cryptanalysis on GPUs with the Cube Attack: Design, Optimization and Performances Gains

Marco Cianfriglia*
"Roma Tre" University
Rome, Italy
cianfriglia@mat.uniroma3.it

Stefano Guarino
Institute for Applied Computing (IAC - CNR)
Rome, Italy
s.guarino@iac.cnr.it

***Abstract*—The cube attack is a flexible cryptanalysis technique, with a simple and fascinating theoretical implant. It combines offline exhaustive searches over selected tweakable public/IV bits (the sides of the "cube"), with an online key-recovery phase. Although virtually applicable to any cipher, and generally praised by the research community, the real potential of the attack is still in question, and no implementation so far succeeded in breaking a real-world strong cipher. In this paper, we present, validate and analyze the first thorough implementation of the cube attack on a GPU cluster. The framework is conceived so as to be usable out-of-the-box for any cipher featuring up to 128-bit key and IV, and easily adaptable to larger key/IV, at just the cost of some fine (performance) tuning, mostly related to memory allocation. As a test case, we consider previous state-of-the-art results against a reduced-round version of a well-known cipher (Trivium). We evaluate the computational speedup with respect to a CPU-parallel benchmark, the performance dependence on system parameters and GPU architectures (Nvidia Kepler vs Nvidia Pascal), and the scalability of our solution on multi-GPU systems. All design choices are carefully described, and their respective advantages and drawbacks are discussed. By exhibiting the benefits of a complete GPU-tailored implementation of the cube attack, we provide novel and strong elements in support of the general feasibility of the attack, thus paving the way for future work in the area.**

***Keywords*—Cube attack; GPU; framework; performance**

I. Introduction

Dinur and Shamir's cube attack [1] is a chosen-plaintext attack in which linear equations binding key bits are extracted through exhaustive searches over selected public/IV bits – the edges of the *cubes* the attack is named after. The success of the attack depends on its ability to detect imbalances into the distribution of monomials in the polynomial representation of the target cipher. Since these statistical flaws are generally unknown beforehand, the attack is often run without a clear prior insight into a convenient strategy for selecting the cubes – an approach made possible by the fact that the attack only requires black-box access to the attacked cipher. Possible practical strategies include exploring cubes of different (possibly large) size, trying many different sets of indices, and varying the binary assignment of the public bits not belonging to the tested cube, all solutions that come at an exponential cost. As in *Time-Memory-Data Trade-Off* (TMDTO) attacks, the extensiveness of the pre-computation stage, which ultimately determines the attack's success rate, must be carefully tuned on the available computing power, memory, and data.

While previous CPU-based approaches seem to pursuit a balanced use of time and memory, in this paper we present an implementations of the cube attack that fully leverages the potential of Graphics Processing Units (GPUs) to boost the parallel search for suitable cubes. As a test case, we replicate and extend previous results [2] against reduced-round versions of the well-known cipher Trivium [3]. Without neglecting the relevance of our findings, that we briefly report and comment, this paper is tailored towards a careful evaluation of the performance of our implementation; a detailed analysis of our results is reported in a more crypto-oriented article [4]. The contributions hereby provided can be summarized as follows:

- We show how to tune the design and implementation of the cube attack to the characteristics of GPUs, in order to fully exploit parallelization while coping with limited memory.
- We present a flexible framework to mount cube attacks against any cipher, under the sole condition that the cipher is as well implemented in GPU. The tool is independent of the GPU architecture, and it supports extension to multi-GPU systems.
- We carefully analyse the performance of our implementation, in terms of: (i) speedup with respect to a CPU implementation, (ii) dependence on system parameters, (iii) comparison among different architectures (including latest generation GPU cards), and (iv) impact of a multi-GPU distributed approach.
- We underline the advantages and drawbacks of a GPU-based cube attack from a cryptographic perspective. We especially focus on the possibility offered by our implementation – with negligible additional costs – to generalize the attack to a dimension never explored in previous works, with several benefits that we identify and briefly discuss.

This article is organized as follows: we start by positioning our paper in the literature, reviewing basic notions and previous work related to the cube attack and to distributed/parallel cryptanalysis (Section II); we then describe the design of our

*Also Research Associate at IAC CNR

978-1-5386-3250-5/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCS.2017.114

GPU-based cube attack (Section III), and report the results of a preliminary test-bed attack (Section IV); finally, we present a detailed performance analysis (Section V), followed by a recap of attained goals and directions for future work (Section VI).

II. Background

Let us start with a few basic notions and a brief overview of related work, needed to contextualize, motivate, and fully understand our contributions.

A. The Cube Attack

The cube attack is a widely applicable method of cryptanalysis introduced by Dinur and Shamir [1], based on a construction similar to Vielhaber's AIDA [5]. The underlying idea, object of extensions (*dynamic* cube attacks [6], [7], *cube testers* [8], [9]) and generalizations [10], [11] in the literature, is to extract from the unknown polynomial representation of the target cipher a set of linear (or low-degree) equations binding key variables, replacing a symbolic factorization with an exhaustive evaluation over selected public variables.

Let $E(\mathbf{x}, \mathbf{y})$ denote the target cipher, as a function of two vectors: the n public variables $\mathbf{x}$ (the IV) and the k private variables $\mathbf{y}$ (the key K). A generic keystream bit z can be expressed as $z = p(\mathbf{x}, \mathbf{y})$, where p is the polynomial representation of E, and all variables appear in p with degree 1, at most. The idea of the attack is to choose a subset of m public variables $\mathbf{x}_I \subset \mathbf{x}$ indexed by $I \subset \{1, \ldots, n\}$ and focus on the quotient $p_{S(I)}(\mathbf{x}_{\overline{I}}, \mathbf{y})$ of the division of $p(\mathbf{x}, \mathbf{y})$ by the monomial $t_I = \prod_{x \in \mathbf{x}_I} x$. By definition, $p_{S(I)}(\mathbf{x}_{\overline{I}}, \mathbf{y})$, called the *superpoly* of I in p, only depends on public variables $\mathbf{x}_{\overline{I}}$ indexed by I's complement $\overline{I}$ (other than $\mathbf{y}$). If we find $p_{S(I)}(\mathbf{x}_{\overline{I}}, \mathbf{y})$ and assign any value $\mathbf{v}_{\overline{I}}$ to $\mathbf{x}_{\overline{I}}$, we obtain a polynomial $p_{S(I)}(\mathbf{v}_{\overline{I}}, \mathbf{y}) = p_{S(I)}(\mathbf{y})$ only binding key variables[1]. If I is such that $p_{S(I)}(\mathbf{y})$ is *linear*, the monomial t_I is called a *maxterm for p with the assignment* $\mathbf{v}_{\overline{I}}$. If we can identify maxterms and find the symbolic expression of their superpolys, we obtain a system of linear equations that can be used to recover the secret key.

Unfortunately, we cannot find $p_{S(I)}(\mathbf{x}_{\overline{I}}, \mathbf{y})$ symbolically because we do not know $p(\mathbf{x}, \mathbf{y})$ in the first place. To make up for it, we observe that all monomials in $p_{S(I)}(\mathbf{x}_{\overline{I}}, \mathbf{y})$ do not contain any of the variables $\mathbf{x}_I$, whereas all monomials in the remainder $q(\mathbf{x}, \mathbf{y})$ of the division of $p(\mathbf{x}, \mathbf{y})$ by t_I do not contain *at least* one of the variables in $\mathbf{x}_I$. For this reason, if $C_I(\mathbf{v}_{\overline{I}})$ denotes the *cube* composed by all 2^m possible binary assignments to $\mathbf{x}$ conditioned to $\mathbf{x}_{\overline{I}} = \mathbf{v}_{\overline{I}}$, the sum of $p(\mathbf{x}, \mathbf{y})$ over $C_I(\mathbf{v}_{\overline{I}})$ yields [1]

$$\sum_{\mathbf{v} \in C_I(\mathbf{v}_{\overline{I}})} p(\mathbf{v}, \mathbf{y}) = p_{S(I)}(\mathbf{v}_{\overline{I}}, \mathbf{y}) = p_{S(I)}(\mathbf{y}) \qquad (1)$$

regardless of the values assigned to $\mathbf{y}$. In other words, we can find $p_{S(I)}(\mathbf{y})$ through an exhaustive sum over $\mathbf{x}_I$.

[1]The standard assumption is $\mathbf{v}_{\overline{I}} = \mathbf{0}$, but this is not actually required.

The following example will help clarifying the notation.

Example 1. Let $n = 3$, $k = 1$, and

$$p(x_1, x_2, x_3, y_1) = x_1x_2y_1 + x_1x_3 + x_1x_2 + x_2y_1 + x_3y_1 + 1$$

If $I = \{1, 2\}$ (*i.e.*, $\mathbf{x}_I = \{x_1, x_2\}$, $\mathbf{x}_{\overline{I}} = \{x_3\}$), we have

$$p(x_1, x_2, x_3, y_1) = \underbrace{x_1x_2}_{t_I}\underbrace{(y_1 + 1)}_{p_{S(I)}} + \underbrace{(x_1x_3 + x_2y_1 + x_3y_1 + 1)}_{q(x_1, x_2, x_3, v_1)}$$

If $x_3 = 1$ (*i.e.*, $\mathbf{v}_{\overline{I}} = \{1\}$), summing p over the cube $C_I(\mathbf{v}_{\overline{I}}) = \{001, 011, 101, 111\}$, we have:

$$\sum_{\mathbf{v} \in C_I(\mathbf{v}_{\overline{I}})} p(\mathbf{v}, \mathbf{y}) = \underbrace{y_1 + 1}_{p(001, y_1)} + \underbrace{y_1 + y_1 + 1}_{p(011, y_1)} + \underbrace{1 + y_1 + 1}_{p(101, y_1)} + \underbrace{y_1 + 1 + 1 + y_1 + y_1 + 1}_{p(111, y_1)} = y_1 + 1$$

that is exactly $p_{S(I)}(y_1)$.

Despite the cube attack and its variants have shown promising results against several ciphers (*e.g.*, Trivium [1], Grain [6], Hummingbird-2 [12], Katan and Simon [7], Quavium [13]), the attack also attracted harsh criticism [14]. Indeed, all implementations of the cube attack so far targeted reduced-round variants of a cipher. With no evidence that the full version can be equally attacked, the feasibility and convenience of cube attacks is still disputed. However, in contrast with a diffused belief that the cube attack only works if the polynomial p has low degree, Fouque and Vannet [2] argued (and, to some extent, experimentally showed) that effective cube attacks can be run not aiming at p's maximum degree, but rather exploiting a nonrandom p by searching for maxterms of significantly lower degree.

B. Distributed/Parallel Cryptanalysis

In cryptanalysis, distributed computing and/or parallel processing can be used to render attacks computationally or storage-wise feasible/practical. Outsourcing onerous computations to third parties is an increasing trend, with no specific restrainers to applications in this context: Smart et al. [15], for instance, developed a new methodology to assess cryptographic key strength on the cloud. The interest of the research community, however, is mostly towards assessing the impact of latest generation hardware on the performance of cryptanalytic tools. Along this line, Marks et al. [16] provided numerical evidence of the potential of mixed GPU (AMD, Nvidia) & CPU technology to data encryption and decryption algorithms. Focusing on GPU, Milo et al [17] leveraged GPUs to quickly test passphrases used to protect private keyrings of OpenPGP cryptosystems, showing that the time complexity of the attack can be reduced up to three-orders of magnitude with respect to a standard procedure, and up to ten times with respect to a highly tuned CPU implementation. Dictionary Attacks can also significantly benefit from GPUs, as shown by Agostini [18] attacking the BitLocker technology commonly used in Windows OSes to encrypt disks.

For what concerns the cube attack, some early papers boosted cube attacks using programmable parallel hardware. Aumasson et al. [19] used an FPGA implementation of cube testers on Grain-128, that allowed them running 256 instances of Grain-128 in parallel, each instance being itself parallelized by a factor 32. Dinur et al. [20] instead relied on RIVYERA, a massively parallel reconfigurable hardware, in order to experimentally verify the correctness and expected complexity of their dynamic cube attack against Grain-128. Finally, Fan and Gong [12] made use of GPUs to perform a side channel cube attack on Hummingbird-2, based on an efficient term-by-term quadraticity test. However, differently from our complete framework for running cube attacks on GPUs, Fan and Gong [12] only used GPUs to parallelize (in a somewhat naive way, as motivated at the end of Section III-C) the evaluation of the cipher over all subcubes of a maximal cube of size 16. Although they showed that GPUs are a viable option to boost the performance of specific tasks, we do provide the first all-round GPU implementation of the cube attack, thoroughly designed for maximal performance, and reusable out-of-the-box for other ciphers.

III. The Proposed GPU Implementation of the Attack

In this section, we present, detail, and discuss our attack, designed to run on a cluster equipped with Graphics Processing Units (GPUs). As previously mentioned, the success of a cube attack is highly dependent on suitable implementation choices. In order to better explain our own approach, we start with an analysis of the cube attack from a more implementative perspective.

A. Practical Cube Attack

At a high level, any practical implementation of the cube attack requires performing the following steps:

Choosing a candidate maxterm t_I*:* Since the complexity of evaluating cube C_I scales exponentially with $|I|$, finding a convenient strategy to select the index set I (or, equivalently, the candidate maxterm t_I), is of primary importance. A common assumption is the existence of a threshold degree, cutting apart monomials that yield a nonlinear superpoly from monomials that yield a constant one. Even assuming that the degree of most maxterms lies around some threshold, the lack of information about the distribution of monomials in p prevents any prior educated guess at both the value of the threshold and the actual selection of variables in t_I. In the literature, a few approaches have been proposed to try to address both issues at once. Dinur and Shamir [1] proposed a *random walk* over index sets, starting from a random set I and iteratively updating I adding or subtracting random elements according to the experimentally tested degree of the superpoly $p_{S(I)}$. Alternatively, Fouque and Vannet [2] used the *Moebius transform* to concurrently compute the sums over all subcubes of a maximal cube $C_{I_{\max}}$ characterized by having the variables $\mathbf{x}_{\overline{I}}$ set to $\mathbf{0}$. Additionally, they suggested a heuristic to identify cubes expected to behave better than others, and used it to select the most promising maximal set $I_{\max}$. However, none of these two strategies is suitable for GPUs, as we will better describe in Section III-C.

Testing $p_{S(I)}$ ***for linearity****:* In principle, assessing if t_I is a maxterm requires finding the symbolic expression of $p_{S(I)}(\mathbf{v}_{\overline{I}}, \mathbf{y})$, for all possible assignments $\mathbf{v}_{\overline{I}}$ to variables $\mathbf{x}_{\overline{I}}$. However, efficient *probabilistic* linearity tests [21], [22] can be safely used in practice. Additionally, aiming at minimize the degree of $p_{S(I)}(\mathbf{v}_{\overline{I}}, \mathbf{y})$, only $p_{S(I)}(\mathbf{0}, \mathbf{y})$ is usually considered in the literature. We argue that this is not necessarily the best choice, as motivated by our results presented in Section IV. In any case, at this stage we can omit the dependence on $\mathbf{v}_{\overline{I}}$ and assume that $p_{S(I)}(\mathbf{y})$ only depends on $\mathbf{y}$. Probabilistic tests involve verifying if

$$p_{S(I)}(\mathbf{u}_1 + \mathbf{u}_2) = p_{S(I)}(\mathbf{u}_1) + p_{S(I)}(\mathbf{u}_2) + p_{S(I)}(\mathbf{0}) \quad (2)$$

holds for random pairs of vectors $\mathbf{u}_1, \mathbf{u}_2$. In fact, (2) must be true for all $\mathbf{u}_1, \mathbf{u}_2$ if $p_{S(I)}$ is linear, whereas, in general, it holds with probability $\frac{1}{2}$. Practically, (2) means

$$\sum_{\mathbf{v} \in C_I} E(\mathbf{v}, \mathbf{u}_1 + \mathbf{u}_2) = \sum_{\mathbf{v} \in C_I} E(\mathbf{v}, \mathbf{u}_1) + \sum_{\mathbf{v} \in C_I} E(\mathbf{v}, \mathbf{u}_2) + \sum_{\mathbf{v} \in C_I} E(\mathbf{v}, \mathbf{0})$$

thus requiring four *numerical* sums.

Finding linear equations*:* Each maxterm t_I yields a linear equation

$$p_{S(I)}(\mathbf{y}) = \sum_{\mathbf{v} \in C_I} E(\mathbf{v}, K) \quad (3)$$

whose left side is a linear combination of the key variables $\mathbf{y}$ with coefficients found *offline*, whereas the right side is a number found *online*, and whose solution is the sought unknown assignment of the key K to $\mathbf{y}$. Finding the right side of (3) involves a single sum, assuming the availability of the 2^m keystream bits produced with K assigned to $\mathbf{y}$, as $\mathbf{x}$ takes all possible assignments $\mathbf{v} \in C_I$. Finding the *symbolic* expression on the left side of (3) instead requires $k+1$ sums: the free term of $p_{S(I)}(\mathbf{y})$ is

$$p_{S(I)}(\mathbf{0}) = \sum_{\mathbf{v} \in C_I} E(\mathbf{v}, \mathbf{0})$$

whereas the coefficient of each variable y_i is

$$p_{S(I)}(\mathbf{e}_i) + p_{S(I)}(\mathbf{0}) = \sum_{\mathbf{v} \in C_I} E(\mathbf{v}, \mathbf{e}_i) + \sum_{\mathbf{v} \in C_I} E(\mathbf{v}, \mathbf{0})$$

where $\mathbf{e}_i$ is the unit vector with all null coordinates except $y_i = 1$.

Solving the linear system*:* Finally, once a set of linear superpolys have been found, we need to solve the obtained linear system. This can be achieved with any suitable technique described in the literature.

B. The Setting

Generally speaking, GPUs are processing units characterized by the following advantages and limitations:

Computing: Each unit features a large number (*i.e, thousands*) of simple cores, that make possible running a much higher number of parallel threads compared to a standard CPU. More precisely, the GPU's basic processing unit is the *warp* consisting of 32 threads each. Threads are designed to work on 32-bit words, and the performance is maximized if all threads belonging to the same warp execute exactly the same operations at the same time on different but contiguous data.

Memory: The so-called *global* memory available on a GPU is limited, typically between 4 and 12 GB. Each thread can independently access data (random access is fully supported, but costly performance-wise). However, when threads in a warp access consecutive 32-bit words, the cost is equivalent to a single memory operation. Concurrent readings and writings by different threads to the same resources, which require some level of synchronization, should be avoided to prevent serialization that defeats parallelism.

The basic step of the attack is the sum of $E(\mathbf{v}, \mathbf{y})$ over all elements $\mathbf{v}$ of a cube C_I. Each time we sum over a cube, the key variables $\mathbf{y}$ are fixed, either to a random $\mathbf{u}_j$ for the linearity tests, or to $\mathbf{0}$ and to versors $\mathbf{e}_i$ for determining the superpoly. In both cases exactly the same sum $\sum_{\mathbf{v} \in C_I} E(\mathbf{v}, \mathbf{u}_j)$ must be performed for all elements of a set of keys $\{\mathbf{u}_1, \ldots, \mathbf{u}_M\}$.

We define the following strategy for carrying out the sums over a cube with the goal of maximizing the parallelization and fully exploiting at its best the computational power offered by GPUs:

- Assigning to all the threads within a warp the computation of the same cube C_I but with a different key $\mathbf{u}_j$. This choice guarantees that all threads perform the same operation at the same time for the entire computation.
- Leveraging the GPU computational power to calculate all the elements of a cube C_I, providing to the threads just a bit-mask representing the set I. With this approach we can exploit all available GPU memory to store the cubes evaluations and minimize, at the same time, the number of memory access operations.
- Defining a keystream generator function $E(\mathbf{x}, \mathbf{y})$ which outputs a 32-bit word, and letting each thread work on the whole word, fully leveraging the GPU computing model. This approach offers two remarkable benefits: (i) considering 32 keystream bits altogether is equivalent to concurrently attacking 32 different polynomials, and (ii) working on 32-bit integers fits much better with the GPUs features, whereas forcing the threads to work on single bits would critically affect the performance of the attack. As a drawback, attacking 32 keystream bits altogether increases (of a factor 32) the memory needed for storing the cubes' evaluation, thus imposing some limitations on the size of the cubes to be tested, as we will clarify later.
- Choosing the number M of keys to be a multiple of the warp size in order to perform the probabilistic linearity test on 32 keystream bits at the same time and for all M keys.

C. The Attack

The practical review of the cube attack presented in Section III-A brought to light that the attack requires, for each cube, exponentially many calls to the target cipher E, followed by as many sums of the resulting outputs. In Section III-B we identified a few tips to tailor the computation of a cube so as to unleash the potential of GPUs. However, the proposed strategy prescribes considering altogether all M different keys per cube needed to run the linearity test. This means that, if T denoted the available memory and $|T|$ its size, the amount of usable memory is *de facto* reduced to $|T|/M$, thus making even more strict the already severe memory constraints characterizing GPUs.

In CPU-based cube attacks, main memory is mostly used to store single evaluations of the cipher over the vertices of some maximal cube $C_{I_{\max}}$. In [1], these evaluations are used to perform a *random walk* that, starting from a random subset $I \subset I_{\max}$, iteratively tests the superpoly $p_{S(I)}$ to decide whether the degree of t_I should be increased or decreased. In [2], the table storing these $2^{|I_{\max}|}$ values is *Moebius-transformed* to compute at once the sums over $\binom{|I_{\max}|}{d}$ subcubes of $C_{I_{\max}}$ of degree d, for $d = 0, \ldots, |I_{\max}|$. These cubes are all possible subcubes C_I of $C_{I_{\max}}$ in which the variables $\mathbf{x}_{\overline{I}}$ have been set to $\mathbf{0}$. None of these two strategies is suitable for GPUs: the stochastic nature of the random walk prevents the sequence of steps from being determined *a priori*, since the computation is performed only when (and if) needed; the Moebius transform requires a rigid schema of calculations and a large number of alternating read and write operations in memory that must be synchronized. Both approaches are conceived for implementations in which computational power is a constraint (while memory is not), and all advantages of using the Moebius Transform are lost in case of parallel processing. Additionally, storing single evaluations of the cipher in T means testing only subcubes of a maximal cube of size $|I_{\max}| = \log_2(|T|/M)$, but, with the memory available in current GPUs, $\log_2(|T|/M)$ is not large enough for any reasonably strong cipher.

The proposed design of the attack relies on the following rationale: exploring only a portion of the maximal cube $C_{I_{\max}}$, considering only subsets $I \subseteq I_{\max}$ characterized by a non-empty *minimal* intersection $I_{\min}$. Quite naturally, a similar design leads to two distinct CUDA[2] kernels, respectively responsible for: (1) computing many variants of the cube $C_{I_{\min}}$, one for each of the possible combinations of the indices in $I_{\max} \setminus I_{\min}$, and writing the results in memory; (2) combining the stored results to test all cubes C_I such that $I_{\min} \subseteq I \subseteq I_{\max}$. Following this approach, the size of the explored $I_{\max}$ can be raised to $|I_{\max}| = |I_{\min}| + \log_2(|T|/M)$, with read and write memory operations carried out by different kernels.

With respect to the notation introduced in Section II-A, let

[2]CUDA is the software framework used for programming Nvidia GPUs.

us distinguish the public variables $\mathbf{x}$ into three sets $\mathbf{x}_{\text{fix}}$, $\mathbf{x}_{\text{free}}$, and $\mathbf{x}^*$, of size d_{fix}, d_{free}, and $n-d$, respectively, where $d = d_{\text{fix}} - d_{\text{free}}$. The variables $\mathbf{x}_{\text{fix}}$ correspond to the *fixed* components of $C_{I_{\max}}$ identified by $I_{\min}$, *i.e.*, $I_{\min} = \{i_1, \ldots, i_{d_{\text{fix}}}\}$, whereas the variables $\mathbf{x}_{\text{free}}$ correspond to the remaining *free* components of $C_{I_{\max}}$, *i.e.*, $I_{\max} \setminus I_{\min} = \{j_1, \ldots, j_{d_{\text{free}}}\}$ and $|I_{\max}| = d$. The variables $\mathbf{x}^*$ are the remaining public variables that fall outside $I_{\max}$.

The two kernels of our attack can be described as follows:

Kernel 1: It uses $2^{d_{\text{free}}}$ warps. Since, as described before, the 32 threads belonging to the same warp perform exactly the same operations but for different keys, in the following we simply consider a representative thread per warp and ignore the private variables $\mathbf{y}$.[3] For $t = 0, \ldots, 2^{d_{\text{free}}} - 1$, thread (*i.e.*, warp) s sums $E(\mathbf{v}, \mathbf{y})$ over each vertex of the cube $C^s_{I_{\min}}$ of size d_{fix} determined by the assignment of the d_{free}-bit representation $\mathbf{v}_{\text{free}}$ of integer s to the variables $\mathbf{x}_{\text{free}}$ and of $\mathbf{0}$ to the variable $\mathbf{x}^*$. Finally, thread s writes the sum in the s^{th} entry of table T, so that, at the end of the execution of the kernel, each entry of T contains the sum over a cube of size d_{fix}. These evaluations allow for testing the monomial $t_{I_{\min}}$ with all the aforementioned assignments to the other $n - d_{\text{fix}}$ variables.

Kernel 2: By simply combining the values stored in T at the end of Kernel 1, it is now possible to explore cubes of potentially any size $d_{\text{fix}} + \delta$, with $0 \leq \delta \leq d_{\text{free}}$. Although the exploration can potentially follow many other approaches (*e.g.*, a random walk as in [1]), the large computing power of our platform suggests to test cubes exhaustively. Moreover, we extend the exhaustive search to an area never reached, to the best of our knowledge, in the literature. For all I such that $I_{\min} \subseteq I \subseteq I_{\max}$, this kernel considers all variants of cube C_I obtained assigning all possible combinations of values to the variables in $I_{\max} \setminus I$. More precisely, for each possible choice of $\delta \in [0, d_{\text{free}}]$, there are exactly $\binom{d_{\text{free}}}{\delta} 2^{d_{\text{free}}-\delta}$ distinct cubes of size $d_{\text{fix}} + \delta$ available. In fact, we can choose δ free variables (the additional dimensions of the cube) in $\binom{d_{\text{free}}}{\delta}$ different ways, and we can choose the fixed assignment to the remaining $d_{\text{free}} - \delta$ variables in any of the $2^{d_{\text{free}}-\delta}$ possible combinations.

We would like to highlight that Kernel 2 is computationally dominated by Kernel 1, so the cost of our exhaustive search is negligible. This way, our design allows considering any possible assignment to variables outside the cube, to finally address the common conjecture (never proved in the literature), that assigning $\mathbf{0}$ is the best possible solution. As a matter of fact, this means a significant gain in terms of number of cubes tested with respect to previous works: with $2^{d_{\text{free}}}$ bits of memory, the approach used in [2] allows considering $2^{d_{\text{free}}}$ cubes, whereas we are able to test $3^{d_{\text{free}}}$ different cubes.

Finally, let us underline the significant divergence between our work and [12], the first paper to ever use GPUs for a cube attack. In [12], Fan and Gong use GPUs to accelerate the summation of the polynomial p over all subcubes of a maximal cube of size 16. However, they use a rather trivial approach: for a cube of size k, they launch 2^k threads in charge of evaluating the cipher E over each of the 2^k vertices of the cube, followed by a parallel reduction process where these 2^k values are summed using so-called *shared* memory. Basically, they just use GPUs to call E in parallel over multiple inputs, but sums over different cubes are processed sequentially, and the impact of the warp structure is completely overlooked. Additionally, they only consider a very small maximal cube, not discussing whether their scheme scales to larger dimensions. Conversely, the design and implementation of our GPU kernels thoroughly covers all steps of the attack, maximizing the performance subject to all architectural constraints, such as warp size and limited memory. Besides the high-level engineering described before, we implement a few low-level optimizations: all computations internal to our kernels only rely on registers, that are the fastest type of memory available on GPUs, avoiding any use of shared memory, and we leverage some built-in CUDA functions, such as *warp shuffle*, to efficiently exchange values between threads belonging to the same warp, when needed (*e.g.*, when we perform the linearity tests).

[3]The work that here is assigned to a single thread can be actually split among any number of threads, reassembling the results at the end. We will not consider this possibility here for the sake of clarity.

IV. Test case: Results Against Trivium

To test the viability of our implementation, we mounted an attack against a reduced-round variant of Trivium, a stream cipher conceived by De Cannière and Preneel [3], part of the eSTREAM portfolio. Trivium generates up to 2^{64} bits of output from an 80-bit key K and an 80-bit Initial Vector IV, and it shows remarkable resistance to cryptanalysis despite its simplicity and its excellent performance. Trivium is composed by a 288-bit internal state consisting of three shift registers of length 93, 84 and 111, respectively. The feedback to each of these registers and the output bit of the cipher are obtained through non-linear combinations involving in total 15 out of the 288 internal state bits. To initialize the cipher, K and IV are written into two of the shift registers, with a fixed pattern filling the remaining bits. 1152 initialization rounds guarantee that the output begins to be produced only after all key-bits and IV-bits have been sufficiently mixed together to define the internal state of the registers.

In recent years, several implementations of the cube attack attempted at breaking Trivium. Quedenfeld et al. [23] found cubes for Trivium up to round 446. Srinivasan [24] obtained 69 extremely sparse linearly independent superpolys for Trivium reduced to 576 rounds. In their seminal paper [1], Dinur and Shamir found 63, 53, and 35 linearly independent superpolys after, respectively, 672, 735, and 767 rounds. Fouque and Vannet [2] even improved over Dinur and Shamir, by obtaining 42 linearly independent superpolys after 784 rounds, and 12 linearly independent superpolys (plus 6 quadratic superpolys) after 799 rounds. To the best of our knowledge, these are the best results against Trivium to date. However, O'Neil [25]

suggests that even the full version of Trivium exhibits limited randomness, thus making of Trivium a perfect candidate for further attempts to evaluate the effectiveness of the cube attack.

First of all, in order to validate our implementation of the cube attack, we symbolically evaluated the polynomial p of Trivium up to 400 initialization rounds, and used p to identify all possible maxterms and their superpoly. We then ran the attack to find all maxterms whose variables belonged to selected sets I. Our experimental findings matched the symbolical findings. Next, we considered Trivium reduced to 768 initialization rounds, and we ran additional experiments specifically designed to reproduce (and possibly extend) results from [2]. The rationale was to provide, at once: (i) a further validation of the correctness of our code; (ii) a direct comparison of our results with the state-of-the-art; and (iii) an immediate means to assess the advantages of our approach.

The attack ran on a cluster composed by 3 nodes, each equipped with 2 Tesla K80 with 24 GB of *global* memory and 4 Intel Xeon CPU E5-2640 with 128 GB of RAM. We recall that each K80 is in turn composed by two K40 with 12GB of global memory each. We launched 12 runs based on 12 different pairs $I_{\min}, I_{\max}$, chosen so as to guarantee that each of the 12 linearly independent superpolys found in [2] after 799 initialization rounds was to be found by one of our runs. Differently from the tests presented in Section V, designed to permit an analysis of the dependence of the performance of our tool on system parameters, for the *real* attack the size of $I_{\min}$ and $I_{\max} \setminus I_{\min}$ is $d_{\text{fix}} = 25$ and $d_{\text{free}} = 16$, respectively, for all runs, so that all maximal cubes have size $d = d_{\text{fix}} + d_{\text{free}} = 41$. In all the reported experiments, we used a *complete-graph* linearity test [22] based on combining 10 randomly sampled keys, following the common practice for cube attacks [1], [2]. As explained and motivated before, in our scheme, each call to Trivium produces 32 key-stream bits, which we concurrently attack in search of superpolys. The most significant practical consequence of a similar construction is the ability to devise attacks to Trivium reduced to any number of initialization rounds ranging from 768 to 799, at the cost of a single attack: in short, an attack to Trivium reduced to $768 + i$ initialization rounds can count upon all superpolys found in correspondence of the j^{th} output bit after 768 rounds, for all $j \geq i$.

Figure 1 shows the number of linearly independent superpolys found by our attack, as a function of the number of initialization rounds, comparing our findings with those of [2]. Overall, our results extend the state-of-the-art in a remarkable way, especially if we consider that our quest for maxterms was circumscribed to multiples of 12 *base* monomials of degree 25. In particular, let us highlight a few aspects that emerge from Figure 1:

- Our attack allows a full key recovery up to 781 initialization rounds.
- For 784 rounds, we find 59 linearly independent superpolys, compared with the 42 found in [2].
- Since our runs were designed to include all 12 maxterms found in [2] after 799 initialization rounds, it is not surprising that our results are no worse than theirs. Yet, we found 3 more linearly independent superpolys, reaching rank 15.

Finally, let us recall that when we test the monomial composed of all variables in some set $I_{\min} \subseteq I \subseteq I_{\max}$, we exhaustively assign values to all public variables in $I_{\max} \setminus I$, thus concurrently testing the linearity of $2^{41-|I|}$ possibly different superpolys. To stress the primary benefits of this feature of our attack, we compare our full results with analogous results in which this additional feature has been disabled, showing that in the latter case the effectiveness of the attack is significantly reduced. It is important to underline that this possibility was overlooked in the literature due to computational constraints, but it comes almost free-of-charge in our framework.

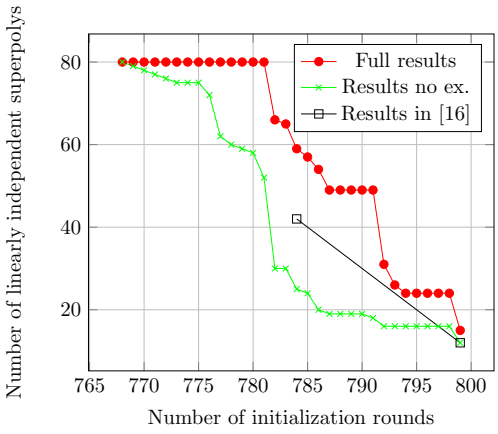

Figure 1: The results of our attack for different reduced-round variants of Trivium: full results vs. results without exhaustive search (no ex.) vs. previous work

V. Performance Analysis

To evaluate the performance of our GPU based solution, we carried out an extensive experimental campaign on the previously described cluster. In addition, we had limited access to an additional node equipped with a Tesla P100, a latest generation GPU card based on the Pascal architecture. For comparison, we also developed a parallel CPU version of the cube attack based on OpenMP. This implementation exploits the 32 cores of the four Intel(R) Xeon(R) CPU E5-2640 and scales linearly as the size of the problem. Each performance test was executed 5 times and the average time is reported.

In Figure 2, we report the speedup gained by the GPU version with respect to the parallel CPU version. Anchoring the size of $I_{\min}$ to $d_{\text{fix}} = 16$, we evaluated the two solutions over a growing maximal cube $C_{I_{\max}}$, whose size is exponential in the cardinality d_{free} of the index set $I_{\max} \setminus I_{\min}$. The experiments show that the benefit of using the GPU version grows with the number of free variables d_{free} considered, outreaching a $80\times$ speedup when $d_{\text{free}} = 16$. It is worth

noticing that all versions rely on the same base functions to implement Trivium.

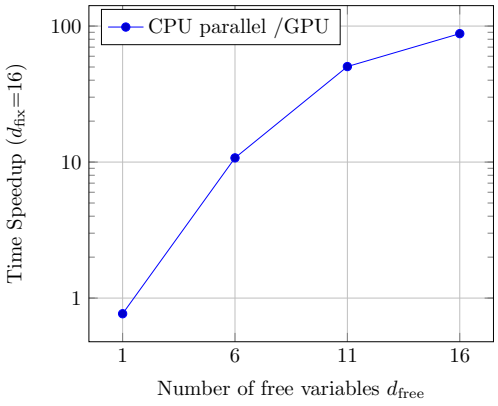

Figure 2: Speedup of GPU w.r.t. parallel CPU

A second set of experiments was aimed at evaluating how the GPU solution scales when d_{free} increases. To this end, we measured the execution time of the attack on a Kepler K40, as a function of d_{free}, for different values of d_{fix}. These measurements, reported in Figure 3, show that: (i) varying d_{fix} only yields an additive shift; (ii) when the size of the problem is large enough to guarantee the full utilization of computational resources ($d_{\text{free}} \geq 9$), our solution scales roughly linearly with the size of the problem, *i.e.*, exponentially with the number of dimensions of the explored cube. To explain why $d_{\text{free}} = 9$ behaves as a threshold, we need to delve into some of the configuration aspects discussed in Section III.

According to our design, each warp is assigned the computation over a single subcube, so that all the information a thread needs are computed locally, and only the final results are stored in memory. We thus know how many warps are needed for a specific computation, and, based on our code optimization, we can predict the exact number of GPU blocks used for that computation. Specifically, the number of blocks N_{blocks} is determined as:

$$N_{\text{blocks}} = \left\lceil 2^{d_{\text{free}}} \cdot \frac{S_{\text{warp}}}{S_{\text{block}}^{\max}} \right\rceil$$

where S_{warp} is the warp size (*i.e.*, number of threads per warp), while $S_{\text{block}}^{\max}$ is the maximum block size (*i.e.*, maximum number of threads per block). In our case, $S_{\text{warp}} = 32$ and $S_{\text{block}}^{\max} = 1024$, thus yielding $N_{\text{blocks}} = \lceil 2^{d_{\text{free}}-5} \rceil$. This means that $d_{\text{free}} = 8$ and $d_{\text{free}} = 9$ correspond to, respectively, $N_{\text{blocks}} = 8$ and $N_{\text{blocks}} = 16$. These numbers should be compared with the number of Streaming Multiprocessors (SMs) of the GPU card used, that is 13 on the K40. $d_{\text{free}} = 9$ is the first value for which the number of allocated blocks is larger than the number of SMs, producing at least one active block per SM, and thus an use of the GPU at operating speed. Let us underline that we observed an analogous behaviour on the Pascal P100, but with the threshold shifted to 11: when $d_{\text{fix}} = 25$, the execution time stays roughly constant (approximately 1010 sec) when $6 \leq d_{\text{free}} \leq 10$, before suddenly doubling when $d_{\text{free}} = 11$. This is perfectly coherent with the characteristics of the P100, as it has 60 SMs, and $2^{11-5} = 64$.

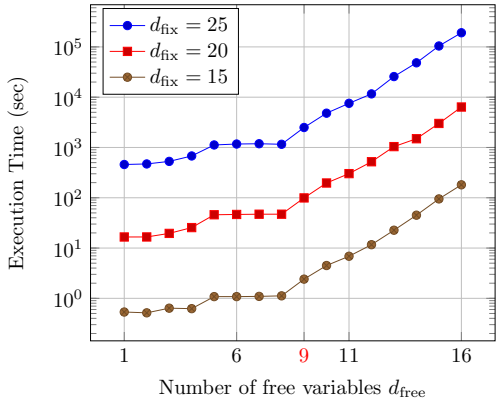

Figure 3: GPU performance analysis

Moreover, in order to assess the impact of the GPU architecture on the performance of our framework, we compared the execution time of the attack on the Pascal P100 and on the Kepler K40. Interestingly, running the experiments on the P100 did not require any code adjustment, we just had to recompile our code for that architecture. Although we could not replicate on the P100 all the experiments run on the K40 due to limited availability, Figure 4 clearly shows that using the Pascal architecture yields a significant speedup. We set $d_{\text{fix}} = 25$ and $d_{\text{free}} \in \{6, 11, 16\}$, measuring an increasing speedup as the size of the problem grows. When $d_{\text{free}} = 16$, in particular, the execution time on the P100 becomes more than $5\times$ smaller than on the K40. These very promising preliminary tests of the performance of the Pascal architecture support the viability of our approach, and suggest that the search for maxterms of a cube attack can significantly benefit from the progress of GPU technologies.

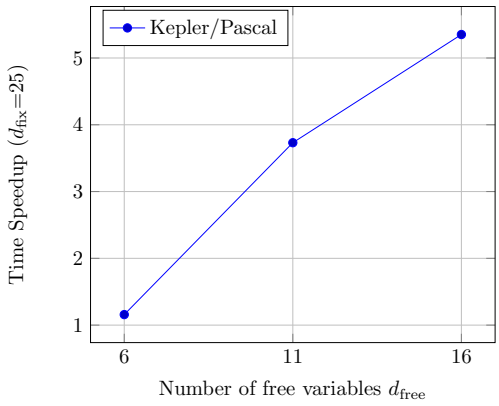

Figure 4: Speedup of Pascal P100 w.r.t. Kepler K40

Finally, we compared the execution time of the attack on cubes of size $d_{\text{fix}} = 25$ and $d_{\text{free}} \in \{6, 11, 16\}$, when the

workload is equally distributed among 1, 2, 4, and 8 K40s, in order to evaluate the scalability of our framework on multi-GPU systems. The distribution process splits a cube of size $|I|$ among 2^l GPUs by assigning to each GPU a subcube of size $|I - l|$; it then merges all 2^l partial computations altogether obtaining the original cube. As mentioned before for the mono-GPU experiments, in order to use the GPUs at operating speed, for each of them the number of allocated blocks should be larger than the number of SMs. For this reason, as reported in Figure 5, the speedup is linear with respect to the number of GPUs only for $d_{\text{free}} = 16$.

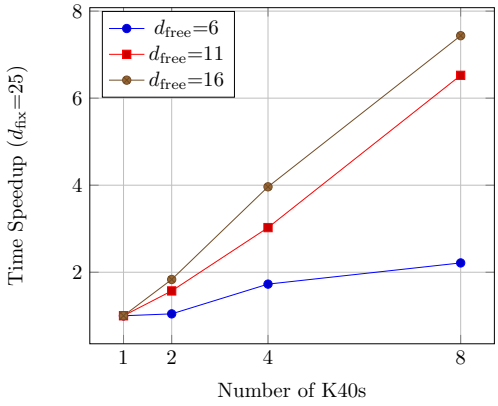

Figure 5: Speedup of multi-GPU w.r.t. mono-GPU

VI. Conclusions and Future Work

This work has presented and discussed the first all-round GPU-tailored implementation of the cube attack, resulting in a flexible and powerful framework, validated against known results in the literature, and soon to be released into the public domain. Our tool can be used against any cipher (with minimal effort), and it supports both latest generation GPU architectures and workload distribution over multi-GPU systems.

We provided a careful performance analysis that shows the feasibility and the scalability of our approach. This opens new prospects related to the possibility of expanding the quest for superpolys to a dimension never explored in previous works. We plan to extend our experimental campaign to test our framework on other well-known ciphers; in particular, experiments against Grain-128 are already in progress. More generally, we expect our framework to be the starting point of future attacks, thus paving the way for further research able to assess the real potential of the still disputed yet praised cube attack.

References

[1] I. Dinur and A. Shamir, "Cube attacks on tweakable black box polynomials," in *Advances in Cryptology-EUROCRYPT 2009*. Springer, 2009, pp. 278–299.

[2] P.-A. Fouque and T. Vannet, *Improving Key Recovery to 784 and 799 Rounds of Trivium Using Optimized Cube Attacks*. Springer Berlin Heidelberg, 2014, pp. 502–517.

[3] C. De Cannière, *Trivium: A Stream Cipher Construction Inspired by Block Cipher Design Principles*. Springer Berlin Heidelberg, 2006, pp. 171–186.

[4] M. Cianfriglia, S. Guarino, M. Bernaschi, F. Lombardi, and M. Pedicini, "A Novel GPU-Based Implementation of the Cube Attack - Preliminary Results Against Trivium," *15th International Conference on Applied Cryptography and Network Security (ACNS2017)*, 2017, (To appear).

[5] M. Vielhaber, "Breaking one.fivium by aida an algebraic iv differential attack," 2007.

[6] I. Dinur and A. Shamir, *Breaking Grain-128 with Dynamic Cube Attacks*. Springer Berlin Heidelberg, 2011, pp. 167–187.

[7] Z. Ahmadian, S. Rasoolzadeh, M. Salmasizadeh, and M. R. Aref, "Automated dynamic cube attack on block ciphers: Cryptanalysis of simon and katan," *IACR Cryptology ePrint Archive*, vol. 2015, p. 40, 2015.

[8] J.-P. Aumasson, I. Dinur, W. Meier, and A. Shamir, "Cube testers and key recovery attacks on reduced-round md6 and trivium," in *Fast Software Encryption*. Springer, 2009, pp. 1–22.

[9] A. Baksi, S. Maitra, and S. Sarkar, "New distinguishers for reduced round trivium and trivia-sc using cube testers," in *WCC2015-9th International Workshop on Coding and Cryptography 2015*, 2015.

[10] R. Winter, A. Salagean, and R. C.-W. Phan, "Comparison of cube attacks over different vector spaces," in *Proceedings of the 15th IMA International Conference on Cryptography and Coding - Volume 9496*. Springer, 2015, pp. 225–238.

[11] A. Agnesse and M. Pedicini, "Cube attack in finite fields of higher order," in *Proceedings of the Ninth Australasian Information Security Conference - Volume 116*, ser. AISC '11. Australian Computer Society, Inc., 2011, pp. 9–14.

[12] X. Fan and G. Gong, *On the Security of Hummingbird-2 against Side Channel Cube Attacks*. Springer Berlin Heidelberg, 2012, pp. 18–29.

[13] S. Zhang, G. Chen, and LiJianhua, "Cube attack on reduced-round Quavium," ICMII-15 Advances in Computer Science Research, 2015.

[14] D.-J. Bernstein, "Why haven't cube attacks broken anything?" https://cr.yp.to/cubeattacks.html last accessed 2016-11-11.

[15] T. Kleinjung, A. Lenstra, D. Page, and N.-P. Smart, "Using the Cloud to determine key strengths," in *Progress in Cryptology - INDOCRYPT 2012*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7668, pp. 17–39.

[16] M. Marks, J. Jantura, E. Niewiadomska-Szynkiewicz, P. Strzelczyk, and K. Góźdź, "Heterogeneous GPU&CPU cluster for high performance computing in cryptography," *Computer Science*, vol. 13, no. 2, pp. 63–79, 2012.

[17] F. Milo, M. Bernaschi, and M. Bisson, "A fast, GPU based, dictionary attack to OpenPGP secret keyrings," *J. Syst. Softw.*, vol. 84, no. 12, pp. 2088–2096, dec 2011.

[18] E. Agostini, "Bitlocker dictionary attack using GPUs," Univ. of Cambridge Passwords 2015 Conference, 2015.

[19] J.-P. Aumasson, I. Dinur, L. Henzen, W. Meier, and A. Shamir, "Efficient fpga implementations of high-dimensional cube testers on the stream cipher grain-128," *SHARCS'09 Special-purpose Hardware for Attacking Cryptographic Systems*, p. 147, 2009.

[20] I. Dinur, T. Güneysu, C. Paar, A. Shamir, and R. Zimmermann, "An experimentally verified attack on full grain-128 using dedicated reconfigurable hardware," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2011, pp. 327–343.

[21] M. Blum, M. Luby, and R. Rubinfeld, "Self-testing/correcting with applications to numerical problems," in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. ACM, 1990, pp. 73–83.

[22] A. Samorodnitsky and L. Trevisan, "A PCP characterization of NP with optimal amortized query complexity," in *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. ACM, 2000, pp. 191–199.

[23] F.-M. Quedenfeld and C. Wolf, "Algebraic properties of the cube attack," *IACR Cryptology ePrint Archive*, vol. 2013, p. 800, 2013.

[24] C. Srinivasan, U. U. Pillai, K. Lakshmy, and M. Sethumadhavan, "Cube attack on stream ciphers using a modified linearity test," *Journal of Discrete Mathematical Sciences and Cryptography*, vol. 18, no. 3, pp. 301–311, 2015.

[25] S. O'Neil, "Algebraic structure defectoscopy," 2007, tools for Cryptanalysis 2007 Workshop sean@cryptolib.com 13859 received 23 Sep 2007, last revised 12 Dec 2007.