Massive Superpoly Recovery with Nested Monomial Predictions

Kai Hu[1,5], Siwei Sun[2], Yosuke Todo[3], Meiqin Wang[1,5(✉)], and Qingju Wang[4]

[1] School of Cyber Science and Technology, Shandong University, Qingdao, Shandong, China. hukai@mail.sdu.edu.cn,mqwang@sdu.edu.cn
[2] School of Cryptology, University of Chinese Academy of Sciences, Beijing, China. siweisun.isaac@gmail.com
[3] NTT Social Informatics Laboratories. yosuke.todo.xt@hco.ntt.co.jp
[4] SnT, University of Luxembourg. qjuwang@gmail.com
[5] Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Qingdao, Shandong, China.

Abstract. Determining the exact algebraic structure or some partial information of the superpoly for a given cube is a necessary step in the cube attack – a generic cryptanalytic technique for symmetric-key primitives with some secret and public tweakable inputs. Currently, the division property based approach is the most powerful tool for exact superpoly recovery. However, as the algebraic normal form (ANF) of the targeted output bit gets increasingly complicated as the number of rounds grows, existing methods for superpoly recovery quickly hit their bottlenecks. For example, previous method stuck at round 842, 190, and 892 for Trivium, Grain-128AEAD, and Kreyvium, respectively. In this paper, we propose a new framework for recovering the exact ANFs of massive superpolies based on the monomial prediction technique (ASIACRYPT 2020, an alternative language for the division property). In this framework, the targeted output bit is first expressed as a polynomial of the bits of some intermediate states. For each term appearing in the polynomial, the monomial prediction technique is applied to determine its superpoly if the corresponding MILP model can be solved within a preset time limit. Terms unresolved within the time limit are further expanded as polynomials of the bits of some deeper intermediate states with symbolic computation, whose terms are again processed with monomial predictions. The above procedure is iterated until all terms are resolved. Finally, all the sub-superpolies are collected and assembled into the superpoly of the targeted bit. We apply the new framework to Trivium, Grain-128AEAD, and Kreyvium. As a result, the exact ANFs of the superpolies for 843-, 844- and 845-round Trivium, 191-round Grain-128AEAD and 894-round Kreyvium are recovered. Moreover, with help of the Möbius transform, we present a novel key-recovery technique based on superpolies involving *all* key bits by exploiting the sparse structures, which leads to the best key-recovery attacks on the targets considered.

Keywords: Cube Attack, Superpoly, Trivium, Grain-128AEAD, Kreyvium, Division Property, Monomial Prediction

1 Introduction

The cube attack was proposed by Dinur and Shamir at EUROCRYPT 2009 against symmetric-key primitives with a secret key and a public input [16]. For a cipher with a secret key $\boldsymbol{k} \in \mathbb{F}_2^m$ and a public input $\boldsymbol{x} \in \mathbb{F}_2^n$, any output bit of the cipher can be regarded as a Boolean function in $\boldsymbol{k}$ and $\boldsymbol{x}$, denoted as $f(\boldsymbol{x}, \boldsymbol{k})$. For a constant $\boldsymbol{u} \in \mathbb{F}_2^n$, let $\boldsymbol{x}^{\boldsymbol{u}} = \prod_{u_i=1} x_i$ where u_i and x_i are the ith coordinate of $\boldsymbol{u}$ and $\boldsymbol{x}$, respectively. Then $f(\boldsymbol{x}, \boldsymbol{k})$ can be written uniquely as

$$f(\boldsymbol{x}, \boldsymbol{k}) = p \cdot \boldsymbol{x}^{\boldsymbol{u}} + q(\boldsymbol{x}, \boldsymbol{k}),$$

where each term of $q(\boldsymbol{x}, \boldsymbol{k})$ misses at least one variable in $\{x_i : u_i = 1\}$. Let $\mathbb{C}_{\boldsymbol{u}} = \{\boldsymbol{x} \in \mathbb{F}_2^n : \boldsymbol{x} \preceq \boldsymbol{u}\}$, where $\boldsymbol{x} \preceq \boldsymbol{u}$ means $x_i \leq u_i$ for all $0 \leq i \leq n-1$. Then, we have

$$\bigoplus_{\boldsymbol{x} \in \mathbb{C}_{\boldsymbol{u}}} f(\boldsymbol{x}, \boldsymbol{k}) = \bigoplus_{\boldsymbol{x} \in \mathbb{C}_{\boldsymbol{u}}} (p \cdot \boldsymbol{x}^{\boldsymbol{u}} + q(\boldsymbol{x}, \boldsymbol{k})) = p. \tag{1}$$

We call p the superpoly of the cube term $\boldsymbol{x}^{\boldsymbol{u}}$ or the cube $\mathbb{C}_{\boldsymbol{u}}$. Note that p is a Boolean function in $\boldsymbol{k}$ and $\boldsymbol{x}[\bar{\boldsymbol{u}}] = \{x_i : u_i = 0\}$, thus sometimes this fact is signaled by the notation $p(\boldsymbol{x}[\bar{\boldsymbol{u}}], \boldsymbol{k})$.

Typically, in the cube attack, the attacker first recovers the superpoly in the offline phase, and then queries the cipher oracle over the cube to compute the summation given by Equation (1), i.e., the value of the superpoly. Information of the secret keys can be obtained from the equation of the superpoly and its value. Hence recovering superpolies is a crucial step in the cube attack.

In early applications of cube attacks [16,32,17,46], the target ciphers are regarded as black boxes and the superpoly recovery is achieved by experimental test. Hence, superpolies recovered in this way have to be extremely simple (typically linear or quadratic functions). In [38], the conventional bit-based division property [40] was first introduced to probe the structure of the superpoly, which allows us to identify some key bits that do not appear in the superpoly. This is the first time that the targeted cipher is regarded as a non-black box object in performing the cube attacks. By setting the key bits that are not involved in the superpoly to arbitrary constants and varying the remaining l key bits, one can obtain the truth table of the superpoly for a subsequent key-recovery attack with complexity $2^{|I|+l}$, where $I = \{i : u_i = 1\}$ is the so-called cube indices. The complexity of recovering the superpoly could be further improved by computing the upper bound on the algebraic degree of the superpoly [41].

At ASIACRYPT 2019, Wang et al. took the three-subset bit-based division property model with the pruning technique to recover the exact superpoly for the first time [42]. However, as the method needs to test every possible monomial in the superpoly, its usage is practically limited when the superpoly is dense. In [43], Ye and Tian introduced a division property-aided algebraic method to recover the exact superpolies by recursively expressing the output of a cipher as the bits of intermediate states and discarding those terms that have no contribution to the superpoly. They found out that several superpolies recovered

in [41] were actually constants, based on which we can only perform distinguishing attacks rather than key-recovery attacks. In [19,20], Hao et al. proposed the three-subset division property without unknown subsets (3SDPwoU) and utilized the Gurobi `PoolSearchMode` to enumerate all possible three-subset trails. By counting the number of trails, they could recover the exact superpolies. In [23], Hu et al. proposed the *monomial prediction* technique aided by the divide-and-conquer strategy to speed up the enumeration of the monomial trials, and more superpolies have been recovered. Besides, Ye and Tian also introduced a pure algebraic method in [47]. By representing the output bit in a polynomial of the intermediate states, the superpoly can be recovered for some so-called useful cubes directly. Recently, Sun claimed that a superpoly of a 78-dimensional cube for 843-round Trivium can be recovered [35] without describing details of the method employed.

Contribution. As the number of rounds grows, the superpolies for certain cubes become increasingly complex. Existing methods for superpoly recovery quickly hit their bottlenecks [42,49,19,20,23,47]. Motivated by this fact, we propose a new framework with nested monomial predictions which scales well for massive superpoly recovery. In this framework, the targeted output bit is first expressed as a polynomial of the bits of some intermediate state. For each term appearing in the polynomial, the monomial prediction technique is applied to determine its superpoly if the corresponding MILP model can be solved within a given time limit. Terms unresolved within the time limit are further expanded as polynomials of the bits of some deeper intermediate states with symbolic computation, whose terms are again processed with monomial predictions. The above procedure is iterated until all terms are resolved. Finally, all the sub-superpolies are collected and assembled into the superpoly of the targeted bit. All the source codes of our framework is available in the public domain `https://github.com/hukaisdu/massive_superpoly_recovery.git`.

We apply the framework to some important symmetric-key ciphers, including Trivium (ISO/IEC standard), Grain-128AEAD (one of the ten Finalists of the NIST lightweight cryptography standardization process), and Kreyvium (designed for fully Homomorphic encryption). For Trivium, we are the first to obtain superpolies for up to 845-round Trivium. For Grain-128AEAD, we recover two 191-round superpolies, while the previous best results reach only 190 rounds. For Kreyvium, we recover a 894-round superpoly, penetrating two more rounds than the best previous results. The details of the superpolies recovered by the new framework and the previous ones are shown in Table 1.

To perform key-recovery attacks based on these superpolies, we face a difficulty that makes existing key-recovery techniques inferior to exhaustive key search: the superpolies are too complicated whose ANFs involve all secret key bits. With help of the Möbius transformation, we present a novel key-recovery technique based on superpolies involving all key bits exploiting the disjoint properties. Applying this technique with the recovered superpolies leads to the best key-recovery attacks on the targets considered (see Table 2).

Table 1: Summary of the exact superpolies recovered practically for round-reduced TRIVIUM, Grain-128AEAD, and Kreyvium.

Cipher	Rounds	Dim	#Term	Degree	Balancedness¶	Method	Ref.
TRIVIUM	818	35	189,540	22	$2^{-11.8}$	Algberaic§	[47]
	835	37	471,120	23	$2^{-10.0}$	Algberaic§	[47]
	837	37	5,011,664	26	$2^{-8.0}$	Algberaic§	[47]
	832	72	3	3	0.375	Pruning & GE†	[38,42,49]
	838	37	2,877,096	25	$2^{-8.3}$	Algberaic§	[47]
	840	78	67	4	0.5	3SDP/u	[19,20]
	840	75	41	4	0.5	Mon. Pred.	[23]
	840	76	6	3	0.5	Mon. Pred.	[23]
	840	76	4	2	0.5	Mon. Pred.	[23]
	840	47	390,899	20	0.02	Nested	App. C.1
	840	49	357,989	20	0.08	Nested	App. C.1
	840	42	31,647	17	0.14	Nested	App. C.1
	840	53	116,145	17	0.26	Nested	App. C.1
	840	56	7,549	14	0.30	Nested	App. C.1
	840	62	1,253	12	0.44	Nested	App. C.1
	841	78	53	5	0.5	3SDP/u	[19,20]
	841	76	3,632	9	0.5	Mon. Pred.	[23]
	841	77	11,161	8	0.5	Mon. Pred.	[23]
	841	56	20,485	16	0.48	Nested	App. C.2
	842	78	975	6	0.5	3SDP/u	[20]
	842	76	5,147	8	0.5	Mon. Pred.	[23]
	842	77	4,174	8	0.5	Mon. Pred.	[23]
	842	56	343,000	17	0.50	Nested	App. C.3
	843	78	16,561	8	0.5	–‡	[35]
	843	56	1,671,492	17	0.50	Nested	Section 5.1
	843	57	7,985,786	19	0.50	Nested	Section 5.1
	843	55	359,466	17	0.49	Nested	Section 5.1
	843	54	628,607	18	0.50	Nested	Section 5.1
	843	76	38,021	18	0.50	Nested	Section 5.1
	844	55	1,770,734	19	0.50	Nested	Section 5.1
	844	54	917,468	17	0.49	Nested	Section 5.1
	845	55	19,967,968	22	0.50	Nested	Section 5.1
	845	54	12,040,654	21	0.50	Nested	Section 5.1
Grain-128AEAD	190∗	95	$178 \sim 18,958$	$19 \sim 24$	$0.012 \sim 0.196$	3SDP/u	[19,20]
	190	96	1,097	21	0.032	3SDP/u	[19,20]
	191	95	3,053,028	27	0.312	Nested	Section 5.2
	191	96	2,398,450	27	0.293	Nested	Section 5.2
Kreyvium	892	115	6	1	0.5	3SDP/u	[20]
	893	118	5⋆	1	0.5	3SDP/u	[20]
	894	119	191	4	0.5	Nested	Section 5.3

¶: The balancedness is measured by the probability that the superpoly is 1.
§: In [47], the complete ANFs are not given. We take our framework to recover them.
†: In [39], Todo et al. showed this superpoly could be recovered in 2^{77} by the conventional bit-based division property. In [42,49], the superpoly was recovered practically by the method of three subset division property with a pruning technique and the recursively-expressing method.
‡: In [35], Sun claimed they recovered a superpoly for 843-round TRIVIUM but no details of their technique was present.
∗: In [19], the authors recovered superpolies for 15 different 95-dimensional cubes.
⋆: In [20], there is an extra term pre-computed offline with 2^{118} time complexity.

Table 2: A summary of the key-recovery attacks on Trivium, Grain-128AEAD, and Kreyvium. Here we do not consider the key recovery attacks under the weak-key seeting such as the works in [47,29].

Cipher	Rounds	Type	Data	Time	Reference
Trivium	672	Cube	$2^{18.6}$	2^{17}	[16]
	709	Cube	2^{23}	$2^{29.14}$	[32]
	767	Cube	2^{31}	2^{45}	[16]
	784	Cube	2^{33}	2^{39}	[17]
	799	Cube	2^{38}	2^{62}	[17]
	802	Cube	2^{37}	2^{72}	[46]
	805	Corr. Cube	2^{28}	2^{73}	[30]
	805	Cube	2^{38}	$2^{41.4}$	[48]
	806	Cube	2^{16}	2^{64}	[48]
	832	Cube	2^{72}	2^{79}	[39,49,42]
	835	Corr. Cube	2^{35}	2^{75}	[30]
	840	Cube	2^{78}	$2^{79.6}$	[19,20]
	840	Cube	$2^{76.6}$	$2^{77.8}$	[23]
	840	**Cube**	$\mathbf{2^{62}}$	$\mathbf{2^{76.32}}$	**App. C.1**
	841	Cube	2^{78}	$2^{79.6}$	[23]
	841	Cube	2^{77}	$2^{78.6}$	[23]
	841	**Cube**	$\mathbf{2^{56}}$	$\mathbf{2^{78}}$	**App. C.2**
	842	Cube	2^{78}	$2^{79.6}$	[23]
	842	Cube	2^{77}	$2^{78.6}$	[23]
	842	**Cube**	$\mathbf{2^{56}}$	$\mathbf{2^{78}}$	**App. C.3**
	843	Cube	2^{78}	$2^{79.6}$	[35]
	843	**Cube**	$\mathbf{2^{56}}$	$\mathbf{2^{77}}$	**Section 6.2**
	844	**Cube**	$\mathbf{2^{56}}$	$\mathbf{2^{78}}$	**Section 6.2**
	845	**Cube**	$\mathbf{2^{56}}$	$\mathbf{2^{78}}$	**Section 6.2**
Grain-128a†	169	Condit. Diff.	2^{47}	small	[29]
	182	Cube	2^{88}	2^{129}	[38,39]
	182	Cube	2^{88}	2^{127}	[38,39,41]
	183	Cube	2^{92}	2^{127}	[41]
	183	Cube	2^{95}	2^{127}	[41]
	190	Cube	2^{96}	2^{123}	[19,20]
Grain-128AEAD	–	State Recovery	–	Practical∗	[12]
	191	**Cube**	$\mathbf{2^{96}}$	$\mathbf{2^{126.26}}$	**Section 6.2**
Kreyvium	849	Cube	2^{61}	2^{127}	[39,41]
	872	Cube	2^{85}	2^{127}	[39,41]
	891	Cube	2^{113}	2^{127}	[19,20]
	892	Cube	2^{115}	2^{127}	[18,19,20]
	893	Cube	2^{118}	2^{119}	[20]
	894	**Cube**	$\mathbf{2^{119}}$	$\mathbf{2^{127}}$	**Section 6.2**

†: Since in our assumption, the Grain-128AEAD is the same as Grain-128a, we provided the results for Grain-128a for a better comparison.
∗: In [12], the authors showed that if the state after the initialization ($t = 384$) is known, then the secret key can be recovered in practical time.

2 Division Property and Monomial Prediction

The division property [37] was proposed by Todo initially as generalized integral attacks [27] (a.k.a. Square attacks [13] or higher-order differential attacks [28,26]). The division property was successfully applied to many primitives. In particular, it was employed to break the full MISTY1 block cipher [31], which undoubtedly demonstrates its powerfulness [36,6].

At the early stage, the division property works in a word-oriented approach, and the propagation of the division properties only considers the algebraic degrees of the local components. Subsequently, by considering the division property at the bit level, Todo and Morii [40] introduced the bit-based division property [7]. With a deeper understanding of the propagation of the bit-based division properties for local components [8], Xiang et al. introduced a MILP-based method to search for the conventional (a.k.a. two-subset) bit-based division properties automatically [45].

From then on, a series of researches on extending the application scope or increasing the accuracy of the algorithms for detecting division properties were conducted [15,14,25,24]. To capture not only balanced but also constant output bits as well as some cancellation characteristics ignored by the conventional bit-based division property, the so-called three-subset bit-based division property was proposed [40]. In [42,44], Wang et al. presented the automated methods for detecting the three-subset bit-based division properties. In [19,20], Hao et al. proposed the three-subset bit-based division property without unknown subsets (3SDPwoU). Eventually, we arrive at methods for detecting division properties with perfect accuracy.

The monomial prediction is another language for describing the division properties from a pure polynomial viewpoint [23]. They are equivalent although they start from different perspectives. In this paper, we mainly take the conceptions of the monomial prediction to interpret our new framework, so in the remaining of this section, we introduce some basic language of the monomial prediction.

2.1 Notations and Definitions

We use bold italic lowercase letters to represent bit vectors. For an n-bit vector $\boldsymbol{u} = (u_0, \cdots, u_{n-1}) \in \mathbb{F}_2^n$, its complementary vector is denoted by $\bar{\boldsymbol{u}}$, where $u_i \oplus \bar{u}_i = 1$ for $0 \leq i < n$. The Hamming weight of $\boldsymbol{u}$ is $wt(\boldsymbol{u}) = \sum_{i=0}^{n-1} u_i$. For $\boldsymbol{u}, \boldsymbol{x} \in \mathbb{F}_2^n$, $\boldsymbol{x}[\boldsymbol{u}]$ denotes a sub-vector of $\boldsymbol{x}$ with respect to $\boldsymbol{u}$ as $\boldsymbol{x}[\boldsymbol{u}] = (x_{i_0}, x_{i_1}, \ldots, x_{i_{wt(\boldsymbol{u})-1}}) \in \mathbb{F}_2^{wt(\boldsymbol{u})}$, where $i_j \in \{0 \leq i \leq n-1 : u_i = 1\}$. For any n-bit vectors $\boldsymbol{u}$ and $\boldsymbol{u'}$, we define $\boldsymbol{u} \succeq \boldsymbol{u'}$ if $u_i \geq u'_i$ for all i. Similarly, we define $\boldsymbol{u} \preceq \boldsymbol{u'}$ if $u_i \leq u'_i$ for all i.

Boolean Function. Let $f : \mathbb{F}_2^n \to \mathbb{F}_2$ be a Boolean function whose *algebraic normal form* (ANF) is

$$f(\boldsymbol{x}) = f(x_0, x_1, \ldots, x_{n-1}) = \bigoplus_{\boldsymbol{u} \in \mathbb{F}_2^n} a_{\boldsymbol{u}} \prod_{i=0}^{n-1} x_i^{u_i},$$

where $a_{\boldsymbol{u}} \in \mathbb{F}_2$, and

$$\boldsymbol{x}^{\boldsymbol{u}} = \pi_{\boldsymbol{u}}(\boldsymbol{x}) = \prod_{i=0}^{n-1} x_i^{u_i} \text{ with } x_i^{u_i} = \begin{cases} x_i, \text{ if } u_i = 1, \\ 1, \text{ if } u_i = 0, \end{cases}$$

is called a monomial. We use the notation $\boldsymbol{x}^{\boldsymbol{u}} \to f$ to indicate that the coefficient of $\boldsymbol{x}^{\boldsymbol{u}}$ in f is 1, i.e., $\boldsymbol{x}^{\boldsymbol{u}}$ appears in f. Otherwise, $\boldsymbol{x}^{\boldsymbol{u}} \nrightarrow f$. In this work, we will

use $\boldsymbol{x}^{\boldsymbol{u}}$ and $\pi_{\boldsymbol{u}}(\boldsymbol{x})$ interchangeably to avoid using the awkward notation ${\boldsymbol{x}^{(i)}}^{\boldsymbol{u}^{(j)}}$ when both $\boldsymbol{x}$ and $\boldsymbol{u}$ have superscripts.

Vectorial Boolean Function. Let $\boldsymbol{f} : \mathbb{F}_2^m \to \mathbb{F}_2^n$ be a vectorial Boolean function with $\boldsymbol{y} = (y_0, y_1, \ldots, y_{m-1}) = \boldsymbol{f}(\boldsymbol{x}) = (f_0(\boldsymbol{x}), f_1(\boldsymbol{x}), \ldots, f_{n-1}(\boldsymbol{x}))$. For $\boldsymbol{v} \in \mathbb{F}_2^n$, we use $\boldsymbol{y}^{\boldsymbol{v}}$ to denote the product of some coordinates of $\boldsymbol{y}$:

$$\boldsymbol{y}^v = \prod_{i=0}^{m-1} y_i^{v_i} = \prod_{i=0}^{m-1} (f_i(\boldsymbol{x}))^{v_i},$$

which is a Boolean function in $\boldsymbol{x}$.

2.2 Monomial Prediction

Let $\boldsymbol{f} : \mathbb{F}_2^{n_0} \to \mathbb{F}_2^{n_r}$ be a composite vectorial Boolean function of a sequence of r smaller function $\boldsymbol{f}^{(i)} : \mathbb{F}_2^{n_i} \to \mathbb{F}_2^{n_{i+1}}, 0 \le i \le r-1$ as

$$\boldsymbol{f} = \boldsymbol{f}^{(r-1)} \circ \boldsymbol{f}^{(r-1)} \circ \cdots \circ \boldsymbol{f}^{(0)}. \tag{2}$$

For $0 \le i \le r-1$, suppose $\boldsymbol{x}^{(i)} \in \mathbb{F}_2^{n_i}$ and $\boldsymbol{x}^{(i+1)} \in \mathbb{F}_2^{n_{i+1}}$ are the input and output of the ith component function $\boldsymbol{f}^{(i)}$. Considering a monomial of $\boldsymbol{x}^{(0)}$, say $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)})$, it is easy to find all the monomials of $\pi_{\boldsymbol{u}^{(1)}}(\boldsymbol{x}^{(1)})$ that contain $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)})$, i.e., $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \to \pi_{\boldsymbol{u}^{(1)}}(\boldsymbol{x}^{(1)})$; for every such $\pi_{\boldsymbol{u}^{(1)}}(\boldsymbol{x}^{(1)})$, we then find all the $\pi_{\boldsymbol{u}^{(2)}}(\boldsymbol{x}^{(2)})$ satisfying $\pi_{\boldsymbol{u}^{(1)}}(\boldsymbol{x}^{(1)}) \to \pi_{\boldsymbol{u}^{(2)}}(\boldsymbol{x}^{(2)})$; finally, if we are interested in whether $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \to \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$, we may collect some transitions from $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)})$ to $\pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$ as

$$\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \to \pi_{\boldsymbol{u}^{(1)}}(\boldsymbol{x}^{(1)}) \to \cdots \to \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)}).$$

Every such transition is called a monomial trail from $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)})$ to $\pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$, denoted by $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \rightsquigarrow \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$. All the trails from $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)})$ to $\pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$ are denoted by $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \bowtie \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$, which is the set of all trails. Then whether $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \to \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$ is determined by the size of $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \bowtie \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$, represented as $|\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \bowtie \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})|$. If there is no trail from $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)})$ to $\pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$, we say $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \not\rightsquigarrow \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$ and accordingly $|\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \bowtie \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})| = 0$.

Theorem 1 (Integrated from [19,20,21,23]). *Let $\boldsymbol{f} = \boldsymbol{f}^{(r-1)} \circ \boldsymbol{f}^{(r-1)} \circ \cdots \circ \boldsymbol{f}^{(0)}$ defined as above. $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \to \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$ if and only if*

$$|\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \bowtie \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})| \equiv 1 \pmod 2.$$

Propagation Rules for the Monomial Trail and the MILP Model. Any component of a symmetric cipher can be regarded as a vectorial Boolean function as $\boldsymbol{f} : \mathbb{F}_2^n \to \mathbb{F}_2^m, \boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$. According to the definition of the monomial prediction [23], the propagation rule for $\boldsymbol{f}$ can be described by a set of tuples

generated with Algorithm 5, which in turn can be described with a set linear inequalities [34,33,9] and thus modeled with MILP. Since any symmetric primitive can be represented as a sequence of basic operations such as XOR, AND and COPY, it suffices to give the propagation rules for these basic functions. We provide their concrete propagation rules and MILP models in App. A.

Gurobi Solver and PoolSearchMode. In this paper, we choose the Gurobi solver [2] as our MILP tool to trace the propagation trails. Gurobi supports a special mode called `PoolSearchMode`, which is useful to extract all possible solutions of a model. In [19,20,23], this mode has been successfully used to enumerate all the trails. In this paper, we use the notation

$$\mathcal{M}.\text{PoolSearchMode} \leftarrow 1$$

to signal that the `PoolSearchMode` is turned on. For more on Gurobi and the `PoolSearchMode`, readers are requested to refer to the Gurobi manual [3].

3 Cube Attack and Superpoly Recovery

In the context of the symmetric-key cryptanalysis, we typically regard each output bit of a primitive as a parameterized Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ whose algebraic normal form is

$$f_{\boldsymbol{k}}(\boldsymbol{x}) = \bigoplus_{\boldsymbol{u}\in\mathbb{F}_2^n} a_{\boldsymbol{u}}(\boldsymbol{k})\boldsymbol{x}^{\boldsymbol{u}}, \boldsymbol{x} \in \mathbb{F}_2^n, \boldsymbol{k} \in \mathbb{F}_2^m,$$

where the coefficient $a_{\boldsymbol{u}}(\boldsymbol{k})$ of the monomial $\boldsymbol{x}^{\boldsymbol{u}}$ can be regarded as a Boolean function of $\boldsymbol{k}$. In this paper, we denote the coefficient of $\boldsymbol{x}^{\boldsymbol{u}}$ in f by $a_{\boldsymbol{u}}(\boldsymbol{k}) = \mathsf{Coe}(f, \boldsymbol{x}^{\boldsymbol{u}})$. Since the function mapping $(\boldsymbol{x}, \boldsymbol{k})$ to $f_{\boldsymbol{k}}(\boldsymbol{x})$ can be expressed as a Boolean function from $\mathbb{F}_2^{n+m}$ to $\mathbb{F}_2$, we may use $f(\boldsymbol{x}, \boldsymbol{k})$ to denote the parameterized Boolean function $f_{\boldsymbol{k}}(\boldsymbol{x})$ when there is no confusion.

3.1 Cube Attack

Let $f(\boldsymbol{x}, \boldsymbol{k})$ be a parameterized Boolean function from $\mathbb{F}_2^{n+m}$ to $\mathbb{F}_2$, and $\boldsymbol{u}$ be a constant vector. $f(\boldsymbol{x}, \boldsymbol{k})$ can be represented uniquely as

$$f(\boldsymbol{x}, \boldsymbol{k}) = p(\boldsymbol{x}[\bar{\boldsymbol{u}}], \boldsymbol{k}) \cdot \boldsymbol{x}^{\boldsymbol{u}} + q(\boldsymbol{x}, \boldsymbol{k}),$$

where each term of $q(\boldsymbol{x}, \boldsymbol{k})$ is not divisible by $\boldsymbol{x}^{\boldsymbol{u}}$. $\boldsymbol{x}^{\boldsymbol{u}}$ is called a *cube term*, and $\mathbb{C}_{\boldsymbol{u}} = \{\boldsymbol{x} \in \mathbb{F}_2^n : \boldsymbol{x} \preceq \boldsymbol{u}\}$ is called a *cube*. The cube we use is sometimes represented by its *cube indices* $I = \{0 \le i \le n-1 : u_i = 1\} \subseteq \{0, 1, \ldots, n-1\}$, and the cube is also denoted by $\mathbb{C}_I$. If we compute the sum of f over $\mathbb{C}_{\boldsymbol{u}}$, we have

$$\bigoplus_{\boldsymbol{x}\in\mathbb{C}_{\boldsymbol{u}}} f(\boldsymbol{x}, \boldsymbol{k}) = \bigoplus_{\boldsymbol{x}\in\mathbb{C}_{\boldsymbol{u}}} (p(\boldsymbol{x}[\bar{\boldsymbol{u}}], \boldsymbol{k}) \cdot \boldsymbol{x}^{\boldsymbol{u}} \oplus q(\boldsymbol{x}, \boldsymbol{k})) = p(\boldsymbol{x}[\bar{\boldsymbol{u}}], \boldsymbol{k}),$$

where $p(\boldsymbol{x}[\bar{\boldsymbol{u}}], \boldsymbol{k})$ is called the *superpoly* of $\mathbb{C}_{\boldsymbol{u}}$. It is easy to check that the superpoly of $\mathbb{C}_{\boldsymbol{u}}$ is just the coefficient of $\boldsymbol{x}^{\boldsymbol{u}}$ in the parameterized Boolean function $f(\boldsymbol{x}, \boldsymbol{k})$, i.e.,

$$p(\boldsymbol{x}[\bar{\boldsymbol{u}}], \boldsymbol{k}) = \mathsf{Coe}\left(f(\boldsymbol{x}, \boldsymbol{k}), \boldsymbol{x}^{\boldsymbol{u}}\right).$$

If we set the variables in $\boldsymbol{x}[\bar{\boldsymbol{u}}]$ to some fixed constants, the superpoly $p(\boldsymbol{x}[\bar{\boldsymbol{u}}], \boldsymbol{k}) = \mathsf{Coe}\left(f, \boldsymbol{x}^{\boldsymbol{u}}\right)$ is a Boolean function of $\boldsymbol{k}$. In this paper, $\boldsymbol{x}[\bar{\boldsymbol{u}}]$ will be always fixed as $\mathbf{0}$.

As mentioned, in the cube attack the superpoly recovery plays a critical role. If the attacker manages to recover the superpoly in the offline phase, then in the online phase, he queries the encryption oracle with the cube and gets the value of the superpoly (0 or 1). Then the attacker obtains an equation of some key bits. By solving this equation, some key information can be extracted. The remaining key bits can be recovered by exhaustive search.

3.2 Superpoly Recovery Based on the 3SDPwoU/Monomial Prediction

To our best knowledge, currently there are four kinds of methods of recovering the exact superpolies for a non-blackbox cipher. A brief introduction to the four methods is provided in App. B. In this subsection, we recall some details about the MILP model for recovering the exact superpoly based on the 3SDPwoU [19,20] or the monomial prediction [23].

As we mentioned, any cipher output bit can be decomposed into a sequence of small vectorial Boolean functions. Then by constructing the MILP models for the propagation rules of these small functions in the way shown in Algorithm 5, we can construct the whole MILP model whose solutions are all valid monomial trails. If we want to recover the superpoly of a cube term $\boldsymbol{x}^{\boldsymbol{u}}$, then we use $\boldsymbol{u}$ to assign the public input variables (plaintext, IV or tweak) in the MILP model. For the secret input (secret key), we just leave them as free variables. And for those constant values of the input, if they are zero, the MILP variable corresponding to the variables are also assigned by zero, while for those constant one input, we let them be free variables.

After the model is constructed, every solution will be a valid monomial trail like the form $\boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}} \rightsquigarrow f$. By calling the Gurobi solver with the `PoolSearchMode` on, we can obtain all solutions of the MILP model. Once we collect all the monomials from $\boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}}$ for f for any $\boldsymbol{v} \in \mathbb{F}_2^m$, we can compute the superpoly of $\boldsymbol{x}^{\boldsymbol{u}}$ as

$$\mathsf{Coe}\left(f, \boldsymbol{x}^{\boldsymbol{u}}\right) = \mathsf{Coe}\left(\bigoplus_{|\boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}} \bowtie f| \equiv 1 \pmod 2} \boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}}, \boldsymbol{x}^{\boldsymbol{u}}\right) = \bigoplus_{|\boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}} \bowtie f| \equiv 1 \pmod 2} \mathsf{Coe}\left(\boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}}, \boldsymbol{x}^{\boldsymbol{u}}\right).$$

In [23], Hu et al. observed that for the composite function f, where

$$f = f^{(r-1)} \circ \boldsymbol{f}^{(r-2)} \circ \cdots \circ \boldsymbol{f}^{(0)},$$

if $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \rightsquigarrow f$, then for $0 < i < r$,

$$|\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \bowtie f| \equiv \sum_{\pi_{\boldsymbol{u}^{(r-i)}}(\boldsymbol{x}^{(r-i)}) \to f} \left|\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \bowtie \pi_{\boldsymbol{u}^{(r-i)}}(\boldsymbol{x}^{(r-i)})\right| \pmod 2.$$

Since computing $|\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \bowtie \pi_{\boldsymbol{u}^{(r-i)}}(\boldsymbol{x}^{(r-i)})|$ one by one is much easier than computing $|\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \bowtie f|$ when i is significantly smaller than r, such a divide-and-conquer strategy helps to speed up the search significantly.

4 Superpoly Recovery with Nested Monomial Predictions

In this section, we introduce a new framework for superpoly recovery that scales well for massive superpolies. In some sense, the new framework is a hybrid of the four previous methods described in App. B. First, we describe the new framework in detail, and then a comprehensive comparison will be made with existing methods.

4.1 The Nested Framework

Given a parameterized Boolean function which consists of a sequence of simple vectorial Boolean functions as

$$f(\boldsymbol{x}, \boldsymbol{k}) = f^{(r-1)} \circ \boldsymbol{f}^{(r-2)} \circ \cdots \circ \boldsymbol{f}^{(0)}(\boldsymbol{x}, \boldsymbol{k}),$$

let the output of $\boldsymbol{f}^{(i)}$ is $\boldsymbol{s}^{(i+1)}$. For simplicity, we always let the dimension of $\boldsymbol{s}^{(i+1)}$ be n. Then we choose a proper positive number (we will elaborate on how to choose it later) r_0 and express f in a polynomial of $\boldsymbol{s}^{(r-r_0)} \in \mathbb{F}_2^n$, i.e.,

$$f(\boldsymbol{x}, \boldsymbol{k}) = \bigoplus_{\substack{\boldsymbol{t}^{(r-r_0)} \in \mathbb{F}_2^n \\ \pi_{\boldsymbol{t}^{(r-r_0)}}(\boldsymbol{s}^{(r-r_0)}) \in \mathbb{S}^{(r-r_0)}}} \pi_{\boldsymbol{t}^{(r-r_0)}}(\boldsymbol{s}^{(r-r_0)}),$$

where $\mathbb{S}^{(r-r_0)} = \{\pi_{\boldsymbol{t}^{(r-r_0)}}(\boldsymbol{s}^{(r-r_0)}) : \pi_{\boldsymbol{t}^{(r-r_0)}}(\boldsymbol{s}^{(r-r_0)}) \to f\}$. Suppose the cube term is $\boldsymbol{x}^{\boldsymbol{u}}$, as Equation (7) shows, we need to compute $\mathsf{Coe}\left(\pi_{\boldsymbol{t}^{(r-r_0)}}(\boldsymbol{s}^{(r-r_0)}), \boldsymbol{x}^{\boldsymbol{u}}\right)$ for each element in $\mathbb{S}^{(r-r_0)}$.

Compute $\mathsf{Coe}\left(\pi_{\boldsymbol{t}^{(r-r_0)}}(\boldsymbol{s}^{(r-r_0)}), \boldsymbol{x}^{\boldsymbol{u}}\right)$. According to the definition, $\boldsymbol{s}^{(r-r_0)}$ is the output vector of a new composite vectorial Boolean function as

$$\boldsymbol{s}^{(r-r_0)} = \boldsymbol{f}^{(r-r_0-1)} \circ \boldsymbol{f}^{(r-r_0-2)} \circ \cdots \circ \boldsymbol{f}^{(0)},$$

then $\pi_{\boldsymbol{t}^{(r-r_0)}}(\boldsymbol{s}^{(r-r_0)})$ is a polynomial of $(\boldsymbol{x}, \boldsymbol{k})$. Hence we can construct the MILP model to enumerate all feasible trails representing $\boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}} \rightsquigarrow \pi_{\boldsymbol{t}^{(r-r_0)}}(\boldsymbol{s}^{(r-r_0)})$ to compute $\mathsf{Coe}\left(\pi_{\boldsymbol{t}^{(r-r_0)}}(\boldsymbol{s}^{(r-r_0)}), \boldsymbol{x}^{\boldsymbol{u}}\right)$ just like [19,20,23]. Different from the

previous methods, we set a time limit $\tau^{(r-r_0)}$ for the MILP model. For a MILP model $\mathcal{M}$, we use

$$\mathcal{M}.\text{TimeLimit} \leftarrow \tau^{(r-r_0)}$$

to denote it. We refer the readers to, e.g., the Gurobi manual [3, Page 591] for more details about the TimeLimit. If the solver hasn't stopped when the time is up, the procedure will be forcibly terminated. For each element in $\mathbb{S}^{(r-r_0)}$, the model of enumerating the trails will end up with three different kinds of status,

1. The model is solved and infeasible, then $\mathsf{Coe}\left(\pi_{\boldsymbol{t}^{(r-r_0)}}(\boldsymbol{s}^{(r-r_0)}), \boldsymbol{x}^{\boldsymbol{u}}\right) = 0$;
2. The model is solved and feasible, and all the solutions has been enumerated, then $\mathsf{Coe}\left(\pi_{\boldsymbol{t}^{(r-r_0)}}(\boldsymbol{s}^{(r-r_0)}), \boldsymbol{x}^{\boldsymbol{u}}\right)$ are obtained [19,20,23];
3. The model is not solved in the time limit $\tau^{(r-r_0)}$.

According to the three different results, we partition $\mathbb{S}^{(r-r_0)}$ into three parts in sequence, say

$$\mathbb{S}^{(r-r_0)} = \mathbb{S}_0^{(r-r_0)} \bigcup \mathbb{S}_p^{(r-r_0)} \bigcup \mathbb{S}_u^{(r-r_0)},$$

where $\mathbb{S}_0^{(r-r_0)}$ is called a *solved-0 set* that contains the elements of case 1, $\mathbb{S}_p^{(r-r_0)}$ is called a a *solved-p set* containing the elements of case 2, and $\mathbb{S}_u^{(r-r_0)}$ is called an *undecided set* containing the elements of case 3. The intersection of any two sets among $\mathbb{S}_0^{(r-r_0)}, \mathbb{S}_p^{(r-r_0)}$ and $\mathbb{S}_u^{(r-r_0)}$ is empty.

The solved-0 set is discarded naturally since the elements in it have no contribution to $\mathsf{Coe}\left(f, \boldsymbol{x}^{\boldsymbol{u}}\right)$. For the solved-p set,

$$p^{(r-r_0)} = \bigoplus_{\pi_{\boldsymbol{t}^{(r-r_0)}}(\boldsymbol{s}^{(r-r_0)}) \in \mathbb{S}_p^{(r-r_0)}} \mathsf{Coe}\left(\pi_{\boldsymbol{t}^{(r-r_0)}}(\boldsymbol{s}^{(r-r_0)}), \boldsymbol{x}^{\boldsymbol{u}}\right)$$

is collected as a part of the whole superpoly $\mathsf{Coe}\left(f, \boldsymbol{x}^{\boldsymbol{u}}\right)$. The undecided set is the only one we proceed with.

To deal with the monomials in the undecided set $\mathbb{S}_u^{(r-r_0)}$, we choose another positive r_1 and expand each monomial in $\mathbb{S}_u^{(r-r_0)}$ in a polynomial of $\boldsymbol{s}^{(r-r_0-r_1)}$. All the monomials from the expression are inserted into the $\mathbb{S}^{(r-r_0-r_1)}$, i.e.,

$$\mathbb{S}^{(r-r_0-r_1)} = \{\pi_{\boldsymbol{t}^{(r-r_0-r_1)}}(\boldsymbol{s}^{(r-r_0-r_1)}) : \pi_{\boldsymbol{t}^{(r-r_0-r_1)}}(\boldsymbol{s}^{(r-r_0-r_1)}) \rightarrow \pi_{\boldsymbol{t}^{(r-r_0)}}(\boldsymbol{s}^{(r-r_0)}), \\ \pi_{\boldsymbol{t}^{(r-r_0)}}(\boldsymbol{s}^{(r-r_0)}) \in \mathbb{S}_u^{(r-r_0)}\}$$

Note that if even-number monomials $\pi_{\boldsymbol{t}^{(r-r_0-r_1)}}(\boldsymbol{s}^{(r-r_0-r_1)})$ are inserted into $\mathbb{S}^{(r-r_0-r_1)}$, they should cancel each other by combining the similar terms. Only those occurring odd-number times should be held. Then we repeat the process of dealing with $\mathbb{S}^{(r-r_0)}$, and keep going to reduce r.

As r reduces, there are two possible results of the whole procedure, the first is for some $r' = r - r_0 - r_1 - \cdots - r_i, i > 0$, $\mathbb{S}_u^{(r')}$ is an empty set. Then we obtain

$$\mathsf{Coe}\left(f, \boldsymbol{x}^{\boldsymbol{u}}\right) = p^{(r-r_0)} \oplus p^{(r-r_0-r_1)} \oplus \cdots \oplus p^{(r')},$$

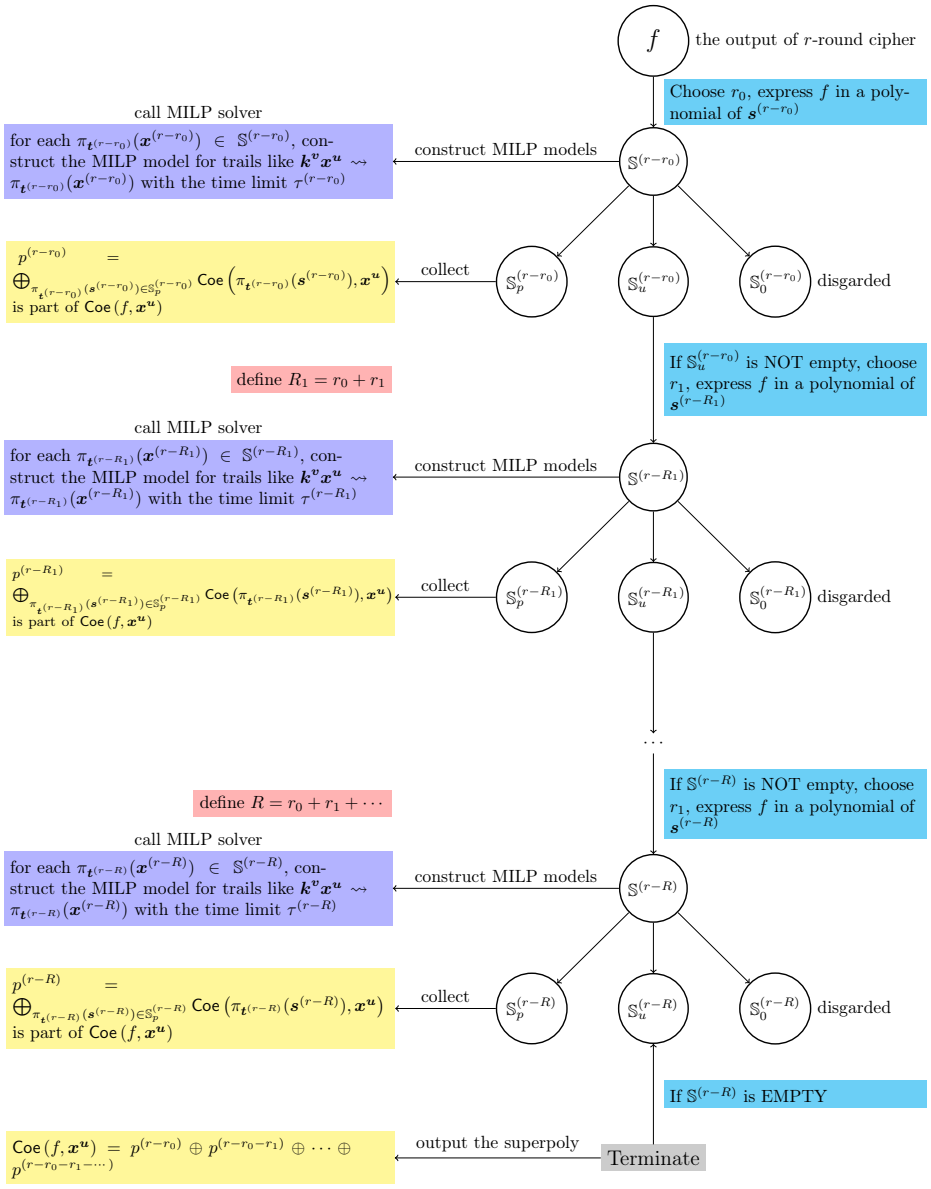

Fig. 1: The nested framework of the superpoly recovery for the cube term $\boldsymbol{x}^{\boldsymbol{u}}$ for r-round cipher f, i.e., $\mathsf{Coe}(f, \boldsymbol{x}^{\boldsymbol{u}})$.

the superpoly is recovered. The second result is we finally get $\mathbb{S}^{(0)}$, it is natural to get the partial superpoly from monomials in $\mathbb{S}^{(0)}$. In this case, we also say $\mathbb{S}_u^{(0)}$ is empty. Hence the superpoly is also recovered.

Algorithm 1: A framework for the superpoly recovery

Procedure `SuperpolyRecFramework`($f(\boldsymbol{x},\boldsymbol{k})$, r, I, *ModelX*):
  Prepare a polynomial $p = 0$
  Initialize $\mathbb{S}_u^{(r)} = \{f\}$
  Prepare a hash table J whose key is the key monomial and the values is an integer
  while $\mathbb{S}_u^{(r)} \neq \emptyset$ **do**
    $r' = r - \texttt{ChooseRiX}(\mathbb{S}_u^{(r)}, r)$
    for $\pi_{\boldsymbol{t}^{(r)}}(\boldsymbol{s}^{(r)}) \in \mathbb{S}_u^{(r)}$ **do**
      `/* Express` $\pi_{\boldsymbol{t}^{(r)}}(\boldsymbol{s}^{(r)})$ `in a polynomial of` $\boldsymbol{s}^{(r')}$ `*/`
      $\mathbb{S}^{(r')} \leftarrow \texttt{Express}(\pi_{\boldsymbol{t}^{(r)}}(\boldsymbol{s}^{(r)}), r, r')$
    Remove the elements occurring even-number times in $\mathbb{S}^{(r')}$
    for $\pi_{\boldsymbol{t}^{(r')}}(\boldsymbol{s}^{(r')}) \in \mathbb{S}^{(r')}$ **do**
      $\mathcal{M} \leftarrow \texttt{ModelX}(r', \pi_{\boldsymbol{t}^{(r')}}(\boldsymbol{s}^{(r')}), I)$
      $\tau^{(r')} = \texttt{ChooseTiX}(r')$
      $\mathcal{M}$.PoolSearchMode $\leftarrow 1$
      $\mathcal{M}$.TimeLimit $\leftarrow \tau^{(r')}$
      Solve $\mathcal{M}$
      if $\mathcal{M}$ *is solved and all the solutions are extracted* **then**
        Extract $\boldsymbol{k}^{\boldsymbol{v}}$ in every found solution
        Increase $J[\boldsymbol{k}^{\boldsymbol{v}}]$ by 1
        Prepare $p^{(r')} = 0$
        for $\boldsymbol{k}^{\boldsymbol{v}}$ *whose* $J[\boldsymbol{k}^{\boldsymbol{v}}]$ *is an odd number* **do**
          $p^{(r')} = p^{(r')} \oplus \boldsymbol{k}^{\boldsymbol{v}}$
        $p = p \oplus p^{(r')}$
      else if $\mathcal{M}$ *is not solved within* $\tau^{(r')}$ **then**
        $\mathbb{S}_u^{(r')} \leftarrow \pi_{\boldsymbol{t}^{(r')}}(\boldsymbol{s}^{(r')})$
  return p

The nested framework can be illustrated by Figure 1 and the procedure `superpolyRecFramework` in Algorithm 1. The procedure `superpolyRecFramework` accepts four inputs: the first stands for the function of the output bit of our target; the second is the round number we are interested in; the third is the cube indices related to the cube term $\boldsymbol{x}^{\boldsymbol{u}}$; and the fourth is a MILP model constructor for computing $\mathsf{Coe}\left(\pi_{\boldsymbol{u}^{(r')}}(\boldsymbol{x}^{(r')}), \boldsymbol{x}^{\boldsymbol{u}}\right)$ based on works in [19,20,23], which is given when we introduce the concrete application. For example, when we target TRIVIUM, the fourth parameter should be `ModelTrivium` in Algorithm 2.

The choices of r_i and $\tau^{(r_i)}$. The choices of r_i and $\tau^{(r_i)}$ play important roles in the whole algorithm since they affect the efficiency directly. When r_i is big, it is sometimes difficult to express $\pi_{\boldsymbol{t}^{(r-r_0-\cdots-r_{i-1})}}(\boldsymbol{s}^{(r-r_0-\cdots-r_{i-1})})$ in $\boldsymbol{s}^{(r-r_0-\cdots-r_i)}$

especially when $r - r_0 - \ldots - r_{i-1}$ has been close to 0. On the contrary, if r_i is too small, the size of $\mathbb{S}^{(r-r_0-\cdots-r_i)}$ will be small, too, then the program is also not efficient, because we have to repeat more times of the expression. Generally speaking, the choice of r_i is heavily related to the position in the life cycle of the nested framework. So we take a dynamic way to decide it. Given $\mathbb{S}^{(r-r_0-\cdots-r_{i-1})}$, we choose that r_i which makes the size of $\mathbb{S}^{(r-r_0-\cdots-r_i)}$ become larger than a given number N for the first time. In our application, we usually choose $N = 10,000$ or $100,000$. In Algorithm 1, the choice of r_i is represented by `ChooseRiX` function, `X` stands for the concrete instance.

The choice of $\tau^{(r_i)}$ affects the efficiency, too, as well as the memory consumption. For a monomial $\pi_{\boldsymbol{t}^{(r-r_0-\cdots-r_i)}}(\boldsymbol{s}^{(r-r_0-\cdots-r_i)})$ that is hard or even impossible to compute out $\mathsf{Coe}\left(\pi_{\boldsymbol{t}^{(r-r_0-\cdots-r_i)}}(\boldsymbol{s}^{(r-r_0-\cdots-r_i)}), \boldsymbol{x}^{\boldsymbol{u}}\right)$, a large $\tau^{(r_i)}$ is pure waste. However, if $\mathsf{Coe}\left(\pi_{\boldsymbol{t}^{(r-r_0-\cdots-r_i)}}(\boldsymbol{s}^{(r-r_0-\cdots-r_i)}), \boldsymbol{x}^{\boldsymbol{u}}\right)$ can be obtained in, e.g., 100 seconds, while we set $\tau^{(r_i)} = 50$ seconds, then $\pi_{\boldsymbol{t}^{(r-r_0-\cdots-r_i)}}(\boldsymbol{s}^{(r-r_0-\cdots-r_i)})$ will be pushed into the undecided set $\mathbb{S}_u^{(r-r_0-\cdots-r_i)}$ and wait to be expressed. Then the 50 seconds is also waste. It is indeed a tough task to choose a proper $\tau^{(r_i)}$. We can only provide some principles and the $\tau^{(r_i)}$ should be obtained according to the concrete instance.

When $r - r_0 - \cdots - r_i$ is closer to r, $\tau^{(i)}$ should be smaller since it is more likely that the model for computing $\mathsf{Coe}\left(\pi_{\boldsymbol{t}^{(r-r_0-\cdots-r_i)}}(\boldsymbol{s}^{(r-r_0-\cdots-r_i)}), \boldsymbol{x}^{\boldsymbol{u}}\right)$ needs an unbearable amount of the time to solve or even impossible to solve. While $r-r_0-\cdots-r_i$ is closer to 0, the model is more likely to be solved in a limited time and expressing $\pi_{\boldsymbol{t}^{(r-r_0-\cdots-r_i)}}(\boldsymbol{s}^{(r-r_0-\cdots-r_i)})$ in $\pi_{\boldsymbol{t}^{(r-r_0-\cdots-r_{i+1})}}(\boldsymbol{s}^{(r-r_0-\cdots-r_{i+1})})$ is more difficult and will spawn thousands of new monomials. Therefore, we prefer to choose a larger $\tau^{(i)}$. The concrete $\tau^{(i)}$ we use for our applications will be given on the spot, i.e., we will give `ChooseRiX` function when discussing the concrete cipher.

4.2 A Comparison with Existing Methods

At first glance, the nested framework is similar to Ye and Tian's recursively-expressing method [49], as we need to express the polynomials in intermediate states, too. However, there is one critical difference between the new framework and the recursively-expressing method. In each step, we partition $\mathbb{S}^{(r')}$ into three parts, say solved-0, solved-p and undecided sets while the recursively-expressing method partitions it into two parts, in the same language with ours, solved-0 and undecided sets. Some parts of the superpoly could be computed out by MILP model when we process the solved-1 set, whereas the recursively-expressing method simply pushes all monomials that should have been in solved-1 set into the undecided set. As a result, the size of the undecided set may become larger and larger. Every such monomial is potential to spawn thousands of new monomials in the next expression. Especially when the superpoly is massive, the size may explode in an exponential way. This is the main reason why their method is not suitable to a large superpoly recovery and longer rounds of Trivium.

The 3SDPwoU and the monomial prediction are embedded in our nested framework as a sub-procedure. However, we use the MILP model in an restrained way rather than totally relying on the MILP solver as done in [19,20,23]. This is important because the internal mechanisms of the MILP solver are unknown. The time consumption is hard to predict beforehand. In some extreme cases, the MILP model is even impossible to be solved but we have no measures to deal with it at all. While in our framework, each MILP model is small and under control by setting the time limit. Besides, since the superpoly is computed in the offline phase, we only need to calculate it once. It is natural for us to resort more computation resources to compute it. Although some solvers like Gurobi support the multithreading property, however, the improvement of the efficiency is not always proportional to the number of threads we use in the experiments. Whereas in the new framework, the program is naturally parallel when processing the monomials in the undecided set, then the efficiency will be proportional to the number of the threads we use. Hence, it is smooth for us to take a multithreading strategy to speed up the search.

As discussed in App. B, Ye and Tian's algebraic methods is potential for massive superpolies but it only works when we find the useful cubes so it has many restrictions when dealing with a casual cube. Most importantly, such requirements for useful cubes are hard to meet when the number of rounds increases. Our method is more general and has no such limitations.

Since Wang's et al. pruning method needs to test every possible monomial of the polynomial one by one, it is meaningful more in theory rather than practice. Our new framework focuses more on the practical recovery of the massive superpolies.

5 Massive Superpoly Recovery

In this section, we apply the new framework to Trivium, Grain-128AEAD, and Kreyvium. As a result, the exact ANFs of the superpolies for 843-, 844- and 845-round Trivium, 191-round Grain-128AEAD and 894-round Kreyvium are recovered, though they are extraordinarily massive. All the experiments are conducted by Gurobi Solver (version 9.1.1) on a work station with 2×AMD EPYC 7302 16-core (32 siblings) Processor 3.3 GHz, (totally 64 threads), 256G RAM, and Ubuntu 20.10. In our platform, the superpolies for 843- and 844-round Trivium are obtained less than two weeks, while the results for 845-round Trivium consume less than three weeks. It costs 31 days to recovery the superpoly for 894-round Kreyvium (who looks quite simple though). The two results for Grain-128AEAD cost 3 and 5 days, respectively. The source codes (as well as the superpolies we recovered) are available in our git repository.

5.1 Superpoly Recovery for Trivium up to 845 Rounds

Trivium is a hardware oriented stream cipher designed by De Cannière and Preneel [10]. It has been selected as part of the eSTREAM portfolio [1] and specified

as an International Standard under ISO/IEC 29192-3 [4]. At the initialization phase, an 80-bit key and an 80-bit IV are loaded into the 288-bit initial state $(s_0, s_1, \ldots, s_{287})$. Then the state is updated through 1152 rounds. This process is summarized by the following pseudo-code:

$$
\begin{aligned}
&(s_0, s_1, \ldots, s_{92}) \leftarrow (K_0, K_1, \ldots, K_{79}, 0, \ldots, 0)\\
&(s_{93}, s_{95}, \ldots, s_{177}) \leftarrow (IV_0, IV_1, \ldots, IV_{79}, 0, \ldots, 0)\\
&(s_{177}, s_{179}, \ldots, s_{287}) \leftarrow (0, \ldots, 0, 1, 1, 1)\\
&\textbf{for}\ \ i = 0 \text{ to } 1151\ \ \textbf{do}\\
&\quad t_1 \leftarrow s_{65} \oplus s_{90} \cdot s_{91} \oplus s_{92} \oplus s_{170}\\
&\quad t_2 \leftarrow s_{161} \oplus s_{174} \cdot s_{175} \oplus s_{176} \oplus s_{263}\\
&\quad t_3 \leftarrow s_{242} \oplus s_{285} \cdot s_{286} \oplus s_{287} \oplus s_{68}\\
&\quad (s_0, s_1, \ldots, s_{92}) \leftarrow (t_3, s_0, s_1, \ldots, s_{91})\\
&\quad (s_{93}, s_{95}, \ldots, s_{177} \leftarrow (t_1, s_{93}, s_{94}, \ldots, s_{175})\\
&\quad (s_{177}, s_{179}, \ldots, s_{287}) \leftarrow (t_2, s_{177}, s_{178} \ldots, s_{286})\\
&\textbf{end for}
\end{aligned}
$$

After the initialization phase, one key stream bit is generated by $z = s_{65} \oplus s_{92} \oplus s_{161} \oplus s_{176} \oplus s_{242} \oplus s_{287}$. When we say r-round Trivium, we mean after r times of updates in the initialization phase, one key bit denoted by z_r is generated. We assume that an attacker has the right to access z_r.

In [23,19,20], the MILP model of Trivium for tracing the three-subset division/monomial trails are proposed. In this paper, we slightly adjust their model to make them suitable to the nested framework. The `TriviumCore` in Algorithm 2 generates the MILP constraints for all the monomial trails of the update function, which is directly borrowed from [19,20]. The procedure `ModelTrivium` generates a model $\mathcal{M}$ as the input of Algorithm 1. All feasible solutions of $\mathcal{M}$ cover all $\boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}} \rightsquigarrow \pi_{\boldsymbol{t}^{(R)}}(\boldsymbol{s}^{(R)})$ where $\boldsymbol{v} \in \mathbb{F}_2^{80}$ and $\boldsymbol{x}^{\boldsymbol{u}}$ is the cube term. The functions that produce the sequences of $r_0, r_1, \ldots, r_i$ and $\tau^{(r-r_0)}, \tau^{(r-r_0-r_1)}, \ldots, \tau^{(r-r_0-\cdots-r_i)}$ for Trivium used in Algorithm 1, i.e., `ChooseRiTrivium` and `ChooseTiTrivium` are given in Algorithm 3.

Superpoly Recovery for 843-Round Trivium. Currently, there is no optimal method of choosing a good cube, so we construct new cubes heuristically as shown in Table 3. It is worth noting that we took the method in [23] to recover the superpoly for I_4, the program had not ended for more than one month and we had to give up. Taking our nested framework, the superpoly for I_4 could be recovered in less than 12 days. Since the superpolies for $I_0, I_1, \ldots, I_4$ are too complicated to present here, we provide them in the git repository. We here only give some information of the five superpolies in Table 4. Since the superpolies are too complicated, the balancedness of each superpoly is tested by 2^{15} random keys.

Superpoly Recovery for 844- and 845-Round Trivium. From Table 3, we know the number of monomial trails and the terms in the superpoly for I_2 is the minimum. We heuristically choose I_2 for 844- and 845-round Trivium and recover the superpolies. The information of the two superpolies are listed

Algorithm 2: Model for the propagation trails of R-round TRIVIUM

Procedure `TriviumCore(` $\mathcal{M}, \mathtt{x}_0, \mathtt{x}_1, \ldots, \mathtt{x}_{287}, i_1, i_2, i_3, i_4, i_5$`):`
- $\mathcal{M}.var \leftarrow \mathtt{y}_{\mathtt{i}_1}, \mathtt{y}_{\mathtt{i}_2}, \mathtt{y}_{\mathtt{i}_3}, \mathtt{y}_{\mathtt{i}_4}, \mathtt{y}_{\mathtt{i}_5}, \mathtt{z}_1, \mathtt{z}_2, \mathtt{z}_3, \mathtt{z}_4, \mathtt{a}$ as binary
- $\mathcal{M}.con \leftarrow \mathtt{x}_{\mathtt{i}_\mathtt{j}} = \mathtt{y}_{\mathtt{i}_\mathtt{j}} \vee \mathtt{z}_\mathtt{j}$ for all $\mathtt{j} \in \{1, 2, 3, 4\}$
- $\mathcal{M}.con \leftarrow \mathtt{a} = \mathtt{z}_3$
- $\mathcal{M}.con \leftarrow \mathtt{a} = \mathtt{z}_4$
- $\mathcal{M}.con \leftarrow \mathtt{y}_{\mathtt{i}_5} = \mathtt{x}_{\mathtt{i}_5} + \mathtt{a} + \mathtt{z}_1 + \mathtt{z}_2$
- **for** $\mathtt{i} \in \{0, 1, \ldots, 287\}$ *w/o* $\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3, \mathtt{i}_4, \mathtt{i}_5$ **do** $\mathtt{y}_\mathtt{i} = \mathtt{x}_\mathtt{i}$
- **return** $(\mathcal{M}, \mathtt{y}_0, \mathtt{y}_1, \ldots, \mathtt{y}_{287})$

Procedure `ModelTrivium(` *round* R, $\pi_{\boldsymbol{t}^{(R)}}(\boldsymbol{s}^{(R)})$, I`):`
- Prepare empty MILP Model $\mathcal{M}$
- $\mathcal{M}.var \leftarrow \mathtt{s}_\mathtt{i}^0$ for $\mathtt{i} \in \{0, 1, \ldots, 287\}$
- **for** $\mathtt{i} = 80$ *to* 92 *and* $\mathtt{i} = 93 + 80$ *to* 284 **do** $\mathcal{M}.con \leftarrow \mathtt{s}_\mathtt{i}^0 = 0$
- **for** $\mathtt{i} = 93$ *to* 172 **do**
  - $\mathcal{M}.con \leftarrow \mathtt{s}_\mathtt{i}^0 = 1 \ \forall\ i - 93 \in I$
  - $\mathcal{M}.con \leftarrow \mathtt{s}_\mathtt{i}^0 = 0 \ \forall\ i - 93 \notin I$
- **for** $r = 0$ *to* $R - 1$ **do**
  - $(\mathcal{M}, \mathtt{x}_0, \ldots, \mathtt{x}_{287}) = \mathtt{TriviumCore}(\mathcal{M}, \mathtt{s}_1^\mathtt{r}, \ldots, \mathtt{s}_{288}^\mathtt{r}, 65, 170, 90, 91, 92)$
  - $(\mathcal{M}, \mathtt{y}_0, \ldots, \mathtt{y}_{287}) = \mathtt{TriviumCore}(\mathcal{M}, \mathtt{x}_1, \ldots, \mathtt{x}_{288}, 161, 263, 174, 175, 176)$
  - $(\mathcal{M}, \mathtt{z}_0, \ldots, \mathtt{z}_{287}) = \mathtt{TriviumCore}(\mathcal{M}, \mathtt{y}_1, \ldots, \mathtt{y}_{288}, 242, 68, 285, 286, 287)$
  - $(\mathtt{s}_0^{\mathtt{r}+1}, \ldots, \mathtt{s}_{287}^{\mathtt{r}+1}) = (\mathtt{z}_{287}, \mathtt{z}_0, \ldots, \mathtt{z}_{286})$
- **for** $i = 0$ *to* 287 **do**
  - $\mathcal{M}.con \leftarrow \mathtt{s}_\mathtt{i}^\mathtt{r} = t_i^{(R)}$ `//` $\boldsymbol{t}^{(R)} = (t_0, t_1, \ldots, t_{287})$
- **return** $\mathcal{M}$

Algorithm 3: `ChooseRiTrivium` and `ChooseTiTrivium`

Procedure `ChooseRiTrivium(`$\mathbb{S}$, r`):`
- $r' = 0$
- **while** $|\mathbb{S}'| < 100,000$ *and* $r - r' > 0$ **do**
  - $r' = r' + 1$
  - $\mathbb{S}' = \emptyset$
  - **for** $s \in \mathbb{S}$ **do** $\mathbb{S}' = \mathbb{S}' \cup \mathtt{Express}(s, r, r')$
- **return** r'

Procedure `ChooseTiTrivium(`r`):`
- **if** $r \geq 600$ **then** $\tau = 60$ seconds
- **else if** $r \geq 500$ **then** $\tau = 120$ seconds
- **else if** $r \geq 400$ **then** $\tau = 180$ seconds
- **else if** $r \geq 300$ **then** $\tau = 360$ seconds
- **else if** $r \geq 200$ **then** $\tau = 720$ seconds
- **else if** $r \geq 100$ **then** $\tau = 1200$ seconds
- **else if** $r \geq 20$ **then** $\tau = 3600$ seconds
- **else if** $r \geq 0$ **then** $\tau = \infty$
- **return** τ

Table 3: Cube indices we use for the superpoly recovery of 843-round TRIVIUM

I	$\lvert I\rvert$	Indices
I_0	56	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 36, 38, 40, 42, 45, 47, 49, 51, 53, 55, 57, 60, 62, 64, 66, 68, 70, 72, 77, 75, 79
I_1	57	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 38, 40, 42, 45, 47, 49, 51, 53, 55, 57, 60, 62, 64, 66, 68, 70, 72, 77, 75, 79
I_2	55	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 34, 36, 38, 40, 42, 45, 47, 49, 51, 53, 55, 57, 60, 62, 64, 66, 68, 70, 72, 77, 75, 79
I_3	54	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 32, 34, 36, 38, 40, 42, 45, 47, 49, 51, 53, 55, 57, 60, 62, 64, 66, 68, 70, 72, 77, 75, 79
I_4	76	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 73, 75, 77, 79

Table 4: Details related to the Superpoly of $\mathbb{C}_I$ for 843-round TRIVIUM. The concrete ANFs for them are provided in the git repository.

I	# Trails	#Monomials	#Involved Key Bits	Degree	Balancedness
I_0	44,586,510	1,671,492	80	17	0.50
I_1	217,694,326	7,985,786	80	19	0.50
I_2	6,124,212	359,466	80	17	0.49
I_3	15,587,645	628,607	80	18	0.50
I_4	1,977,228,919	38,021	80	10	0.50

in Table 5. Since the superpolies are too complicated, the balancedness of each superpoly is tested by 2^{15} random keys.

5.2 Superpoly Recovery for 191-Round Grain-128AEAD

Grain-128AEAD [22] is an authenticated encryption algorithm with support for associated data, which has recently been selected as the one of the ten finalist candidates of the NIST lightweight cryptography standardization process. The design of Grain-128AEAD is closely based on the Grain-128a [5] which was introduced in 2011. Before the pre-output bits are used for encryption, a 64-bit shift register and a 64-bit accumulator are also initialized to generate the authentication tag later. In [19,20], Hao et al. assumed that the first pre-output bit could be observed, then the Grain-128AEAD is actually the same as Grain-128a. In this work, we also analyze Grain-128AEAD under this setting.

The internal state of Grain-128AEAD is represented by two 128-bit states as $\boldsymbol{b} = (b_0, b_1, ..., b_{127})$ and $\boldsymbol{s} = (s_0, s_1, \ldots, s_{127})$. The 128-bit key is loaded to the first register $\boldsymbol{b}$, and the 96-bit nonce (the initialization vector for Grain128a) is loaded to the second register $\boldsymbol{s}$. The other state bits are set to 1 except the least

Table 5: Details related to the Superpoly for I_2 for 844- and 845-round TRIVIUM. The concrete ANFs of them are available in the git repository.

I	Round	# Trails	#Monomials	#Involved Key Bits	Degree	Balancedness
I_2	844	186,128,078	1,770,734	80	19	0.50
I_3	844	55,152,796	917,468	80	17	0.49
I_2	845	4,731,073,108	19,967,968	80	22	0.50
I_3	845	1,362,323,454	12,040,654	80	21	0.50

one bit in the second register. Namely, the initial state bits are represented as

$$(b_0, b_1, \ldots, b_{127}) = (K_0, K_1, \ldots, K_{127}),$$
$$(s_0, s_1, \ldots, s_{127}) = (N_0, N_1, \ldots, N_{95}, 1, \ldots, 1, 0).$$

The pseudo code of the update function in the initialization is given as follows.

$$\begin{aligned}
g &\leftarrow b_0 \oplus b_{26} \oplus b_{56} \oplus b_{91} \oplus b_{96} \oplus b_3 b_{67} \oplus b_{11} b_{13} \oplus b_{17} b_{18} \oplus b_{27} b_{59} \oplus b_{40} b_{48} \\
&\quad \oplus b_{61} b_{65} \oplus b_{68} b_{84} \oplus b_{88} b_{92} b_{93} b_{95} \oplus b_{22} b_{24} b_{25} \oplus b_{70} b_{78} b_{82}, \\
f &\leftarrow s_0 \oplus s_7 \oplus s_{38} \oplus s_{70} \oplus s_{81} \oplus s_{96}, \\
h &\leftarrow b_{12} s_8 \oplus s_{13} s_{20} \oplus b_{95} s_{42} \oplus s_{60} s_{79} \oplus b_{12} b_{95} s_{94}, \\
z &\leftarrow h \oplus s_{93} \oplus b_2 \oplus b_{15} \oplus b_{36} \oplus b_{45} \oplus b_{64} \oplus b_{73} \oplus b_{89}, \\
(b_0, b_1, \ldots, b_{127}) &\leftarrow (b_1, \ldots, b_{127}, g \oplus s_0 \oplus z), \\
(s_0, s_1, \ldots, s_{127}) &\leftarrow (s_1, \ldots, s_{127}, f \oplus z).
\end{aligned}$$

In the initialization, the state is updated 256 times without producing an output. After the initialization, the update function is tweaked such that z is not fed to the state, and z is used as a pre-output key stream.

MILP Model. `ModelGrain-128AEAD` in Algorithm 6 produces the MILP model as the fourth input of Algorithm 1. The MILP model is used to enumerate all trails like $\boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}} \rightsquigarrow \pi_{\boldsymbol{t}^{(R)}}(\boldsymbol{s}^{(R)})$ where $\boldsymbol{v} \in \mathbb{F}_2^{128}$, and $\boldsymbol{x}^{\boldsymbol{u}}$ is the cube term we are interested in. Algorithm 6 is slightly adapted from [19,20], the supporting functions such as `funcZ`, `funcG` and `funcF` are directly borrowed (Algorithm 8). The functions that produce the sequences of $r_0, r_1, \ldots, r_i$ and $\tau^{(r-r_0)}, \tau^{(r-r_0-r_1)}, \ldots, \tau^{(r-r_0-\cdots-r_i)}$ for Grain-128AEAD used in Algorithm 1, i.e., `ChooseRiGrain-128AEAD` and `ChooseTiGrain-128AEAD` are given in Algorithm 7. Due to the page limits, all the algorithms are presented in App. D.

Superpoly Recovery for 191-Round Grain-128AEAD. For 191-round Grain-128AEAD, we apply the nested framework to two cubes. The first is $I_0 = \{0, 1, 2, \ldots, 95\}$, where all nonce bits are active. The second is $I_1 = \{0, 1, 2, \ldots, 95\} \backslash \{30\}$, where all IV bits except the 30th are active. The information of the two superpolies are shown in Table 6.

5.3 Superpoly Recovery for 894-Round Kreyvium

Kreyvium is a stream cipher which was designed for the use of the fully Homomorphic encryption [11]. As a variant of TRIVIUM, Kreyvium shares the same

Table 6: Details related to the Superpoly of I_0 and I_1 for 191-round Grain-128AEAD. The concrete ANFs of them are available in the git repository.

I	#Trails	#Monomials	#Involved Key Bits	Degree	Balancedness
I_0	58,442,962	2,398,450	80	27	0.31
I_1	123,946,062	3,053,028	80	27	0.30

internal structure but allows for bigger keys of 128 bits. The main advantage of Kreyvium over Trivium is that it provides 128-bit security (instead of 80-bit) with the same multiplicative depth, and inherits the same security arguments. Kreyvium supports 128-bit IV and consists of five registers, two of them are LFSRs denoted by K^* and IV^*, respectively. Each one of these two registers is rotated independently from the rest of the cipher when updated. The remaining three registers are NFSRs which are identical to those of Trivium. The five registers are initialized as

$$
\begin{aligned}
&(s_0, s_1, \ldots, s_{92}) \leftarrow (K_0, K_1, \ldots, K_{92}) \\
&(s_{93}, s_{95}, \ldots, s_{176} \leftarrow (IV_0, IV_1, \ldots, IV_{83}) \\
&(s_{177}, s_{179}, \ldots, s_{287}) \leftarrow (IV_{85}, \ldots, IV_{127}, 1, \ldots, 1, 0) \\
&(IV^*_{127}, \ldots, IV^*_0) \leftarrow (IV_{127}, \ldots, IV_0) \\
&(K^*_{127}, \ldots, K^*_0) \leftarrow (K_{127}, \ldots, K_0)
\end{aligned}
$$

Then, the state is updated over 1152 rounds, which is also identical with Trivium. The update function is as follows,

for $i = 0$ to 1151 **do**

$$
\begin{aligned}
&t_1 \leftarrow s_{65} \oplus s_{92}, \quad t_2 \leftarrow s_{161} \oplus s_{176}, \quad t_3 \leftarrow s_{242} \oplus s_{287} \oplus K^*_0 \\
&z_i \leftarrow t_1 \oplus t_2 \oplus t_3 \\
&t_1 \leftarrow t_1 \oplus s_{90}s_{91} \oplus s_{170} \oplus IV^*_0 \\
&t_2 \leftarrow t_2 \oplus s_{174}s_{175} \oplus s_{263} \\
&t_3 \leftarrow t_3 \oplus s_{285}s_{286} \oplus s_{68} \\
&t_4 \leftarrow K^*_0, \ t_5 \leftarrow IV^*_0 \\
&(s_0, s_1, \ldots, s_{92}) \leftarrow (t_3, s_0, s_1, \ldots, s_{91}) \\
&(s_{92}, s_{93}, \ldots, s_{176}) \leftarrow (t_1, s_{93}, s_{94}, \ldots, s_{175}) \\
&(s_{177}, s_{178}, \ldots, s_{287}) \leftarrow (t_2, s_{177}, s_{178}, \ldots, s_{286}) \\
&(K^*_{127}, K^*_{126}, \ldots, K^*_0) \leftarrow (t_4, K^*_{127}, K^*_{126}, \ldots, K^*_1) \\
&(IV^*_{127}, IV^*_{126}, \ldots, IV^*_0) \leftarrow (t_5, IV^*_{127}, IV^*_{126}, \ldots, IV^*_1)
\end{aligned}
$$

end for

Only after the initialization finishes, the key stream bit $z_i, i \geq 1152$ is produced. In this paper, we focus on the variant of Kreyvium whose initialization is reduced to R rounds, where the key stream bit is denoted by z_R.

MILP Model. `ModelKreyvium` in Algorithm 10 produces the MILP model as the fourth input of Algorithm 1. The MILP model is used to enumerate all

trails like $\boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}} \rightsquigarrow \pi_{\boldsymbol{t}^{(R)}}(\boldsymbol{s}^{(R)})$ where $\boldsymbol{v} \in \mathbb{F}_2^{128}$, and $\boldsymbol{x}^{\boldsymbol{u}}$ is the cube term we are interested in. Algorithm 10 is slightly adapted from [19,20] and the `TriviumCore` subroutine is identical to that in Algorithm 2. The functions that produce the sequences of $r_0, r_1, \ldots, r_i$ and $\tau^{(r-r_0)}, \tau^{(r-r_0-r_1)}, \ldots, \tau^{(r-r_0-\cdots-r_i)}$ for Kreyvium in Algorithm 1, i.e., `ChooseRiKreyvium` and `ChooseTiKreyvium` are given in Algorithm 11. These algoritms are provided in App. E.

Superpoly Recovery for 893- and 894-Round Kreyvium For 893- and 894-round Kreyvium, we let the 119-dimensional cube indices be

$$I = \{0, 1, \ldots, 127\} \backslash \{6, 66, 72, 73, 78, 101, 106, 109, 110\}.$$

We apply the nested framework to recover the superpolies. For the 893-round Kreyvium, there are 53 trails are obtained. However, only the trails representing the monomial 1 appear odd-number times, i.e., the superpoly of z_{893} is $p_I = 1$.

For 894-round Kreyvium, we get 24,107 trails, and 191 terms are involved in the superpoly in z_{894}. The superpoly is a 4-degree polynomial and involves 77 key bits. Since k_{119} is an independent term, the superpoly is a balance Boolean function. The superpoly is as follows,

6 Key-Recovery Attacks Exploiting Massive Superpolies

Suppose we have recovered the exact ANF of a superpoly $p(\boldsymbol{k})$ for the cube term $\boldsymbol{x}^{\boldsymbol{u}}$ (the corresponding cube is denoted by $\mathbb{C}_{\boldsymbol{u}}$). In the online phase, we first call the cipher oracle to encrypt all elements in the cube and get the value of the superpoly with time complexity $2^{wt(\boldsymbol{u})}$. In this paper, we always use small-dimensional cubes such that the complexity of this step can be ignored. Next, we try to obtain some information of the secret key from the equation:

$$p(\boldsymbol{k}) = \bigoplus_{\boldsymbol{x} \in \mathbb{C}_{\boldsymbol{u}}} f_{\boldsymbol{k}}(\boldsymbol{x}). \tag{3}$$

Suppose that $p(\boldsymbol{k})$ involves n' bits of the n-bit secret key. In the simplest case where $n' \ll n$, i.e., $p(\boldsymbol{k})$ involves only a small part of the secret key, as the situation in [20,23], we can evaluate $p(\boldsymbol{k})$ for every combination of the involved n' key bits and filter out those incorrect keys that violates Equation (3).

However, for the case $n' = n$, i.e., $p(\boldsymbol{k})$ involves all the key bits, the method presented above does not work any more. Indeed, the complexity of evaluating Equation (3) with all possible key values is larger than 2^n, especially for massive superpolies. To tackle this problem, we present a new key-recovery technique with the binary Möbius transforms shown in Algorithm 4 as its fundamental algorithm.

We first introduce a trivial method for the key recovery based on the Möbius transform. It is well known that Möbius transformation is available for the conversion between the ANF and the truth table of any Boolean function. It requires $n \times 2^{n-1}$ 1-bit XORs and 2^n-bit memory complexity. Of course, the complexity is higher than 2^n in $n \geq 2$, but the unit of the complexity is significantly lower. One

Algorithm 4: Möbius transformation

```
Procedure MöbiusTransformation(a[i], 0 ≤ i ≤ 2^n):
    for k = 1 to n do
        for i = 0 to 2^{n-k} do
            for j = 0 to 2^{k-1} − 1 do
                a[2^k i + 2^{k-1} + j] = a[2^k i + j] ⊕ a[2^k i + 2^{k-1} + j]
    return a
```

recovered superpoly can recover at most 1 bit of information, and the exhaustive search is necessary to determine the whole of secret key bits. Considering the difference between each unit of the complexity, the use of the Möbius transformation could be useful already. Although the superpolies we recovered are massive, they are still very sparse when compared with the random polynomials (a random polynomial may contains about 2^{n-1} monomials). Considering the sparse property, in App. G we give a more efficient algorithm to compute the truth table from the ANF. With the efficient algorithm, the Möbius transformation costs only $n \times 2^{n-2}$ XORs for the superpolies we consider in this paper.

6.1 Divide-and-Conquer Method Using the Disjoint Set

Then, we exploit more detailed structural property of the recovered superpolies to give a delicate key recovery attack on ciphers whose superpolies are massive.

Definition 1 (Disjoint set). *Given a superpoly $p(\boldsymbol{k})$ with n variables, if for $0 \leq i \neq j < n$, k_i and k_j are never multiplied mutually in all monomials of $p(\boldsymbol{k})$, then we say k_i and k_j are disjoint. If for a subset of variables $D \subseteq \{k_0, k_1, \ldots, k_{n-1}\}$, every pair of variables like $k_i, k_j \in D$ are all disjoint, we call D a disjoint set.*

Search for a disjoint set of $p(\boldsymbol{k})$. Obviously, there can be many different disjoint sets for $p(\boldsymbol{k})$, while usually we are only interested in the one with the maximum size. To better study the disjoint sets of $p(\boldsymbol{k})$, we introduce the *disjoint matrix*. A matrix $M \in \mathbb{F}_2^n$ is called the disjoint matrix of $p(\boldsymbol{k})$, if $M[i][j] = 0$ when k_i and k_j are disjoint, $M[i][j] = 1$ otherwise, where $M[i][j]$ stands for the value located at the intersection of the ith row and the jth column. Obviously, all the pairs of the disjoint variables can be reflected by the disjoint matrix. Given the disjoint matrix, a locally-optimized disjoint set can be obtained by a greedy algorithm as follows,

1. sort the variables in $\{k_0, k_1, \ldots, k_{n-1}\}$ in certain order, e.g., an increasing order according to the value $\sum_{0 \leq j < n} M[i][j]$ for k_i. The sorted variables are denoted as $\{k'_0, k'_1, \ldots, k'_{n-1}\}$;
2. initialize a set $D = \{k'_0\}$;

3. for $1 \leq i < n$, if k'_i is disjoint with all variables in D, put k'_i into D; otherwise, process the next variable.
4. after all the variables are processed, D is one of the disjoint sets.

Besides the greedy algorithm, noting that every disjoint set is one-to-one mapped to a zero square sub-matrix of M that takes the diagonal of M as the axis of symmetry, then an SAT/SMT model also works for finding a disjoint set with a certain number of variables and sometimes it may find the optimal disjoint set.

We first consider the case where the targeted superpoly is balanced. And later we consider the case where the superpolies are with a significant bias.

Key recovery attacks with single balanced superpoly. If the balanced superpoly $p(\boldsymbol{k})$ has a disjoint set D with m variables and $J = \{k_0, k_1, \ldots, k_{n-1}\}/D$, then $p(\boldsymbol{k})$ can be written as the form

$$p(k_0, k_1, \ldots, k_{n-1}) = \left(\bigoplus_{0 \leq i < m} k_i \cdot p_i(J) \right) \oplus p_m(J) \tag{4}$$

where $p_i(J)$ is a polynomial of the variables in J.

Every $p_i(J)$ involves at most $n - m$ variables, then we can use the Möbius transform to compute the truth tables of $p_0, p_1, \ldots, p_m$ over all possible values of variables in J. Once we get the $m + 1$ truth tables, we can access them and get the values for every key combination in J, then Equation (4) will become a linear expression of variables in D. Considering Equation (3), we get a linear equation of variables in D. For the linear equation, we can remove 1-bit key guessing efficiently after guessing $m - 1$ key bits additionally.

As is pointed out, the complexity of computing the truth table from the ANF of a Boolean function with κ variables by the Möbius transform is $\kappa \times 2^{\kappa-2}$ XORs (see App. G for more details about the complexity). Hence, if a superpoly has a disjoint set with m variables, the above process costs $(m + 1) \times (n - m) \times 2^{n-m-2}$ XORs to construct the truth tables. For each of the 2^{n-m} combinations of variables in J, we access the $m+1$ truth tables to get the values of $p_i, 1 \leq i \leq m$ and construct a linear equation for the variables in D. Thereafter, with 2^{m-1} guesses for the values of any $m - 1$ variables in the linear equations, the value of the remaining one variable can be determined. Finally we call the cipher oracle to test whether the key candidate is correct.

Key recovery attacks with multiple balanced superpolies. Suppose we have recovered N balanced superpolies $p^{(0)}, p^{(1)}, \ldots, p^{(N-1)}$, if D is the disjoint set for all $p^{(i)}, 0 \leq i < N$, we call D their *common disjoint set*. With N superpolies, we may get more linear equations to gain more information of the secret keys. The complexity of the case then consists of

1. constructing the truth tables, which costs $N \times (m + 1) \times (n - m) \times 2^{n-m-2}$ XORs;
2. constructing the linear equations, which is $N \times 2^{n-m} \times (m + 1)$ truth table lookups;

3. guessing the value of $m - N$ (we always let $m > N$) variables, then the remaining N variables can be determined by solving a set of simple linear equations. This step costs $2^{n-m} \times 2^{m-N}$ guesses. For each guess in the third step, call the cipher oracle to verify the key candidate.

The analysis of the complexity actually contains many redundant computations. For example, each sub-polynomial of a superpoly in Equation (4) at most involves $n - m$ variables, while in practice, some sub-polynomials may involves less key bits. In this case, the complexity of constructing the truth tables and the linear equations can be reduced. What's more, for a superpoly, all linear equations are limited within 2^{m+1} different types. So with a precomputed table containing all the linear equations (and their solutions), the complexity of constructing the linear equations can be improved further. Finally, the dominant part of the complexity is 2^{n-N} cipher calls.

Compared with the previous cube attacks, our method requires considerable memory complexities to store the $N \times (m+1)$ truth tables. We will provide the memory cost for each concrete case later.

Key recovery attacks with significantly biased superpolies. When the superpolies we consider are not balanced, then there are some problems with the above process. For example, when a superpoly p is highly biased towards zero, then its component sub-polynomials are very likely to be zero, too. We may get many identities like $0 = 0$ rather than the useful linear equations about the variables in the disjoint set. The information we gain from the superpolies are also reduced. Fortunately, the information of the secret keys contained in the superpolies can be measured by their entropy. In this line of works, Hao et al. also took the entropy to measure the information we can gain from the superpolies of the 190-round Grain-128AEAD in [19,20].

For N superpolies $p^{(0)}, p^{(1)}, \ldots, p^{(N-1)}$, we are interested in the joint probability distribution of

$$P(p^{(0)} = \nu_0, p^{(1)} = \nu_1, \ldots, p^{(N-1)} = \nu_{N-1}) = P_{(\nu_0,\nu_1,\ldots,\nu_{N-1})},\ (\nu_0, \nu_1, \ldots, \nu_{N-1}) \in \mathbb{F}_2^N. \tag{5}$$

The distribution can be determined by experiments, e.g., in this paper, we test 2^{15} random keys to observe this distribution. The entropy of this distribution is

$$E = -\sum_{(\nu_0,\nu_1,\ldots,\nu_{N-1}) \in \mathbb{F}_2^N} P_{(\nu_0,\nu_1,\ldots,\nu_{N-1})} \log P_{(\nu_0,\nu_1,\ldots,\nu_{N-1})}, \tag{6}$$

When we know the entropy of the targeted superpolies, the information we gain from the key recovery process are also known. If we have gained E bit of the key information, then the final complexity is approximately 2^{n-E} cipher calls.

6.2 Applications to Trivium, Grain-128AEAD and Kreyvium

Key recovery attack on 843-round Trivium. Consider the five superpolies for cubes listed in Table 3, if we choose the superpolies for I_0, I_2 and I_3, denoted by $p^{(0)}, p^{(2)}$ and $p^{(3)}$, one of their common disjoint sets is

$$D = \{k_1, k_{39}, k_{43}, k_{12}, k_{37}\}.$$

Then we can decompose $p^{(0)}, p^{(2)}$ and $p^{(3)}$ as follows,

$$\begin{cases} p^{(0)} = k_{37} \cdot p_0^{(0)} \oplus k_{12} \cdot p_1^{(0)} \oplus k_{43} \cdot p_2^{(0)} \oplus k_{39} \cdot p_3^{(0)} \oplus k_1 \cdot p_4^{(0)} \oplus p_5^{(0)} \\ p^{(2)} = k_{37} \cdot p_0^{(2)} \oplus k_{12} \cdot p_1^{(2)} \oplus k_{43} \cdot p_2^{(2)} \oplus k_{39} \cdot p_3^{(2)} \oplus k_1 \cdot p_4^{(2)} \oplus p_5^{(2)} \\ p^{(3)} = k_{37} \cdot p_0^{(3)} \oplus k_{12} \cdot p_1^{(3)} \oplus k_{43} \cdot p_2^{(3)} \oplus k_{39} \cdot p_3^{(3)} \oplus k_1 \cdot p_4^{(3)} \oplus p_5^{(3)} \end{cases}$$

The sub-polynomials of $p^{(0)}$, i.e., $p_i^{(0)}, 0 \leq i \leq 5$ involve respectively 58, 46, 67, 60, 69 and 75 key bits; the sub-polynomials of $p^{(2)}$, i.e., $p_i^{(2)}, 0 \leq i \leq 5$ involve respectively 54, 18, 51, 33, 32 and 74 key bits; and the sub-polynomials of $p^{(3)}$, i.e., $p_i^{(3)}, 0 \leq i \leq 5$ involve respectively 65, 40, 65, 47, 45 and 75 key bits. Then it can be seen that comparing with $p_5^{(0)}, p_5^{(2)}$ and $p_5^{(3)}$, other sub-polynomials involves much less key bits, then the complexity of constructing the truth tables and linear equations for them can be neglected. According to Table 4, these three superpolies are almost balanced. Then the complexity consists of (where $n = 80, m = 5, N = 3$):

1. $3 \times 75 \times 2^{73}$ XORs for constructing the truth tables;
2. 3×2^{75} table lookups for constructing the linear equations;
3. $2^2 \times 2^{75}$ guesses to determine the remaining three bits of information of the keys. For each guess, call the 843-round Trivium to verify the key candidate.

Therefore, the final time complexity is slightly more than 2^{77} 843-round Trivium calls to recover all the secret key bits. To store all the truth tables, we need approximately $2^{76.6}$ bits of memery, which is equivalent to 2^{70} 80-bit blocks.

Key recovery attack on 844-round Trivium. Consider the two superpolies for 844-round Trivium of the cube I_2 and I_3, denoted by $p^{(2)}$ and $p^{(3)}$, respectively, one of the common disjoint sets is

$$D = \{k_1, k_{10}, k_{20}, k_{43}, k_7, k_{22}\}.$$

Then we can decompose $p^{(2)}$ and $p^{(3)}$ as

$$\begin{cases} p^{(2)} = k_{22} \cdot p_0^{(2)} \oplus k_7 \cdot p_1^{(2)} \oplus k_{43} \cdot p_2^{(2)} \oplus k_{20} \cdot p_3^{(2)} \oplus k_{10} \cdot p_4^{(2)} \oplus k_1 \cdot p_5^{(2)} \oplus p_6^{(2)} \\ p^{(3)} = k_{22} \cdot p_0^{(3)} \oplus k_7 \cdot p_1^{(3)} \oplus k_{43} \cdot p_2^{(3)} \oplus k_{20} \cdot p_3^{(3)} \oplus k_{10} \cdot p_4^{(3)} \oplus k_1 \cdot p_5^{(3)} \oplus p_6^{(3)} \end{cases}$$

The numbers of involved key bits in $p_0^{(2)}, p_1^{(2)}, \ldots, p_6^{(2)}$ are respectively 69, 68, 67, 69, 64, 61 and 74, while the numbers for subpolies of $p^{(3)}$ are respectively 57, 62, 63, 62, 50, 63, 74. Furthermore, the superpoly is experimentally balanced. Thereafter, the complexity consists of (where $N = 2, n = 80, m = 6$):

1. $2 \times 74 \times 2^{72}$ XORs for constructing the truth tables;
2. 2×2^{74} truth table lookups for constructing the linear equations;
3. $2^4 \times 2^{74}$ guesses to determine two key variables in the linear equation and for each guess, call 844-round Trivium to check the candidate.

Therefore, the final complexity is slightly more than 2^{78} 844-round TRIVIUM calls to recover all the secret key bits. The memory cost is about 2^{75} bits, equivalent to 2^{69} 80-bit blocks.

Key recovery attack on 845-round Trivium. Consider the superpoly $p^{(2)}$ and $p^{(3)}$ for 845-round TRIVIUM of the cubes I_2 and I_3, respectively, the only common disjoint set is

$$D = \{k_1, k_{10}\}.$$

Then we can decompose $p^{(2)}$ and $p^{(3)}$ as

$$\begin{cases} p^{(2)} = k_1 \cdot p_0^{(2)} \oplus k_{10} \cdot p_1^{(2)} \oplus p_2^{(2)} \\ p^{(3)} = k_1 \cdot p_0^{(3)} \oplus k_{10} \cdot p_1^{(3)} \oplus p_2^{(3)} \end{cases}$$

$p_0^{(2)}, p_2^{(2)}, p_1^{(3)}$ and $p_2^{(3)}$ involve 78 key bits while $p_1^{(2)}$ and $p_0^{(3)}$ involves only 77 key bits. Therefore, the complexity consists of (where $N = 2, n = 80, m = 2$):

1. $4 \times 78 \times 2^{76} + 2 \times 77 \times 2^{75}$ XORs for constructing the truth tables;
2. $4 \times 2^{78} + 2 \times 2^{77}$ truth table lookups for constructing the linear equations;
3. Solver the linear equations of k_1 and k_{10} to determine one key variables. For each candidate, call the 845-round TRIVIUM to verify the candidate.

Note the number of kinds of all linear equations of k_1 and k_{10} is 8, so the complexity of constructing the linear equations and solving them is very small. Table lookups to the big tables may cost a lot. However, considering that the values contained in the truth tables are all single bits. So we can construct these tables parallelly. Then once lookup can obtain all bits that are used to construct the linear equations. Fairly speaking, the final complexity is slightly more than 2^{78} 845-round TRIVIUM calls to recover all the secret key bits. The memory complexity is about 2^{80} bits, which is equivalent to about 2^{74} 80-bit blocks.

Key recovery attack on 191-round Grain-128AEAD. Consider the superpolies $p^{(0)}$ and $p^{(1)}$ for 191-round Grain-128AEAD, one of their common disjoint sets is

$$D = \{k_9, k_6, k_0, k_2, k_7, k_8, k_5, k_4, k_{14}, k_3, k_{11}, k_1\}.$$

Then we can decompose $p^{(0)}$ and $p^{(1)}$ as

$$\begin{cases} p^{(0)} = k_1 \cdot p_0^{(0)} \oplus k_{11} \cdot p_1^{(0)} \oplus \cdots \oplus k_9 \cdot p_{11}^{(0)} \oplus p_{12}^{(0)} \\ p^{(1)} = k_1 \cdot p_0^{(1)} \oplus k_{11} \cdot p_1^{(1)} \oplus \cdots \oplus k_9 \cdot p_{11}^{(1)} \oplus p_{12}^{(1)} \end{cases}$$

The sub-polynomials of $p^{(0)}$, i.e., $p_0^{(0)}, p_1^{(0)}, \ldots, p_{12}^{(0)}$ involves respectively 89, 115, 112, 116, 93, 83, 109, 110, 29, 93, 112, 100, 116 key bits; while the sub-polynomials of $p^{(1)}$, i.e., $p_0^{(1)}, p_1^{(1)}, \ldots, p_{12}^{(1)}$ involves respectively 86, 115, 115, 116, 92, 96, 110, 115, 39, 99, 115, 107, 115 key bits. So for the complexity, it is enough to consider only those superpolies involving at least 115 key bits. Further, since $p^{(0)}$ and $p^{(1)}$ are highly biased, we compute the entropy of them according to Equation (5) and (6). By taking 2^{15} keys, the entropy contained in the two superpolies is about 1.74. Then the complexity approximately consists of (where $n = 128, m = 12, N = 2$):

1. $3 \times 116 \times 2^{114} + 6 \times 115 \times 2^{113}$ XORs for constructing the truth tables;
2. $3 \times 2^{116} + 6 \times 2^{115}$ table lookups for constructing the linear equations;
3. $2^{10} \times 2^{116}$ guesses to determine two bits of key information.
4. For about $2^{116.26}$ guesses from the previous step, we call 191-round Grain-128AEAD for the verification for the key candidate.

The final complexity is then approximately $2^{116.26}$ 191-round Grain-128AEAD calls to recover all the secret key bits. The memory complexity is about $2^{118.6}$ bits which is equivalent to $2^{117.6}$ 128-bit blocks.

Key recovery attack on 894-round Kreyvium. The superpoly for 894-round TRIVIUM is simple involving only 77 key variables, so we can recover all the secret keys in 2^{127} Kreyvium calls by a normal way as done in [19,20,23].

7 Conclusion

In this paper, we propose a nested framework based on the monomial prediction technique for efficiently recovering the massive superpolies. The nested framework iteratively expands the cipher output in the polynomial of intermediate states. For every term in the polynomial, we try to call the MILP solver to recover a part of the superpoly from a smaller MILP model in a limited time. For those terms which cannot be solved in the limited time, we proceed to expand them in deeper intermediate states. Finally, the targeted superpoly can be fully recovered. We apply this new framework to TRIVIUM, Grain-128AEAD and Kreyvium, superpolies for up to 845, 191 and 894 rounds of the three ciphers are recovered. With the disjoint set method taking the sparse property of the variables involved in the superpoly, the key recovery attacks on the corresponding rounds of the three ciphers are improved. However, the disjoint set will take huge memory cost which is a significant weakness. As the number of rounds increases, the superpolies are expected to be more and more massive. Therefore, we put up an open question: how to efficiently recover the secret keys in cube attacks based on massive suoperpolies involving all secret key bits?

Acknowledgment. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions. Kai Hu and Meiqin Wang are supported by the National Natural Science Foundation of China (Grant No. 62002201, Grant No. 62032014), the National Key Research and Development Program of China (Grant No. 2018YFA0704702, 2018YFA0704704), the Major Scientific and Technological Innovation Project of Shandong Province, China (Grant No. 2019JZZY010133), the Major Basic Research Project of Natural Science Foundation of Shandong Province, China (Grant No. ZR202010220025). Siwei Sun is supported by the National Natural Science Foundation of China (61772519) and the Chinese Major Program of National Cryptography Development Foundation (MMJJ20180102). Qingju Wang is funded by Huawei Technologies Co., Ltd (Agreement No.: YBN2020035184). The scientific calculations in this paper have been done on the HPC Cloud Platform of Shandong University.

References

1. eSTREAM: the ECRYPT stream cipher project (2018). https://www.ecrypt.eu.org/stream/. Accessed: 2021-03-23.
2. Gorubi Optimization. https://www.gurobi.com.
3. Gorubi Optimization Reference Manual. https://www.gurobi.com/wp-content/plugins/hd_documentations/documentation/9.1/refman.pdf.
4. ISO/IEC 29192-3:2012: Information technology Security techniques Lightweight cryptography part 3: Stream ciphers. https://www.iso.org/standard/56426.html.
5. Martin Ågren, Martin Hell, Thomas Johansson, and Willi Meier. Grain-128a: a new version of Grain-128 with optional authentication. *Int. J. Wirel. Mob. Comput.*, 5(1):48–59, 2011.
6. Achiya Bar-On and Nathan Keller. A 2ˆ70 attack on the full MISTY1. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 435–456. Springer, 2016.
7. Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK lightweight block ciphers. In *DAC 2015*, pages 175:1–175:6. ACM, 2015.
8. Christina Boura and Anne Canteaut. Another view of the division property. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016*, volume 9814 of *LNCS*, pages 654–682. Springer, 2016.
9. Christina Boura and Daniel Coggia. Efficient MILP modelings for sboxes and linear layers of SPN ciphers. *IACR Trans. Symmetric Cryptol.*, 2020(3):327–361, 2020.
10. Christophe De Cannière and Bart Preneel. Trivium. In Matthew J. B. Robshaw and Olivier Billet, editors, *New Stream Cipher Designs - The eSTREAM Finalists*, volume 4986 of *LNCS*, pages 244–266. Springer, 2008.
11. Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. *J. Cryptol.*, 31(3):885–916, 2018.
12. Donghoon Chang and Meltem Sönmez Turan. Recovering the key from the internal state of Grain-128AEAD. *IACR Cryptol. ePrint Arch.*, 2021:439, 2021.
13. Joan Daemen, Lars R. Knudsen, and Vincent Rijmen. The block cipher Square. In Eli Biham, editor, *FSE '97*, volume 1267 of *LNCS*, pages 149–165. Springer, 1997.
14. Patrick Derbez and Pierre-Alain Fouque. Increasing precision of division property. *IACR Trans. Symmetric Cryptol.*, 2020(4):173–194, 2020.
15. Patrick Derbez, Pierre-Alain Fouque, and Baptiste Lambin. Linearly equivalent s-boxes and the division property. *IACR Cryptol. ePrint Arch.*, 2019:97, 2019.
16. Itai Dinur and Adi Shamir. Cube attacks on tweakable black box polynomials. In Antoine Joux, editor, *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 278–299. Springer, 2009.
17. Pierre-Alain Fouque and Thomas Vannet. Improving key recovery to 784 and 799 rounds of trivium using optimized cube attacks. In Shiho Moriai, editor, *FSE 2013*, volume 8424 of *LNCS*, pages 502–517. Springer, 2013.
18. Yonglin Hao, Lin Jiao, Chaoyun Li, Willi Meier, Yosuke Todo, and Qingju Wang. Links between division property and other cube attack variants. *IACR Trans. Symmetric Cryptol.*, 2020(1):363–395, 2020.

19. Yonglin Hao, Gregor Leander, Willi Meier, Yosuke Todo, and Qingju Wang. Modeling for three-subset division property without unknown subset - improved cube attacks against Trivium and Grain-128AEAD. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020*, volume 12105 of *LNCS*, pages 466–495. Springer, 2020.
20. Yonglin Hao, Gregor Leander, Willi Meier, Yosuke Todo, and Qingju Wang. Modeling for three-subset division property without unknown subset. *J. Cryptol.*, 34(3):22, 2021.
21. Phil Hebborn, Baptiste Lambin, Gregor Leander, and Yosuke Todo. Lower bounds on the degree of block ciphers. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part I*, volume 12491 of *LNCS*, pages 537–566. Springer, 2020.
22. Martin Hell, Thomas Johansson, Willi Meier, Jonathan Sönnerup, and Hirotaka Yoshida. Grain-128AEAD - A lightweight AEAD stream cipher. *NIST Lightweight Cryptography, Round*, 3, 2019.
23. Kai Hu, Siwei Sun, Meiqin Wang, and Qingju Wang. An algebraic formulation of the division property: Revisiting degree evaluations, cube attacks, and key-independent sums. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part I*, volume 12491 of *LNCS*, pages 446–476. Springer, 2020.
24. Kai Hu and Meiqin Wang. Automatic search for a variant of division property using three subsets. In Mitsuru Matsui, editor, *CT-RSA 2019*, volume 11405 of *LNCS*, pages 412–432. Springer, 2019.
25. Kai Hu, Qingju Wang, and Meiqin Wang. Finding bit-based division property for ciphers with complex linear layers. *IACR Trans. Symmetric Cryptol.*, 2020(1):236–263, 2020.
26. Lars R. Knudsen. Truncated and higher order differentials. In Bart Preneel, editor, *FSE'94*, volume 1008 of *LNCS*, pages 196–211. Springer, 1994.
27. Lars R. Knudsen and David A. Wagner. Integral cryptanalysis. In Joan Daemen and Vincent Rijmen, editors, *FSE 2002*, volume 2365 of *LNCS*, pages 112–127. Springer, 2002.
28. Xuejia Lai. Higher order derivatives and differential cryptanalysis. In Richard E. Blahut, Daniel J. Costello, Ueli Maurer, and Thomas Mittelholzer, editors, *Communications and Cryptography, vol 276*, pages 227–233. Springer, 1994.
29. Michael Lehmann and Willi Meier. Conditional differential cryptanalysis of Grain-128a. In Josef Pieprzyk, Ahmad-Reza Sadeghi, and Mark Manulis, editors, *CANS 2012*, volume 7712, pages 1–11. Springer, 2012.
30. Meicheng Liu. Degree evaluation of NFSR-based cryptosystems. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017*, volume 10403 of *LNCS*, pages 227–249. Springer, 2017.
31. Mitsuru Matsui. New block encryption algorithm MISTY. In Eli Biham, editor, *FSE '97*, volume 1267 of *LNCS*, pages 54–68. Springer, 1997.
32. Piotr Mroczkowski and Janusz Szmidt. The cube attack on stream cipher Trivium and quadraticity tests. *Fundam. Informaticae*, 114(3-4):309–318, 2012.
33. Yu Sasaki and Yosuke Todo. New algorithm for modeling s-box in MILP based differential and division trail search. In Pooya Farshim and Emil Simion, editors, *SecITC 2017*, volume 10543 of *LNCS*, pages 150–165. Springer, 2017.
34. Siwei Sun, Lei Hu, Peng Wang, Kexin Qiao, Xiaoshuang Ma, and Ling Song. Automatic security evaluation and (related-key) differential characteristic search: Application to SIMON, PRESENT, LBlock, DES(L) and other bit-oriented block ciphers. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 158–178. Springer, 2014.

35. Yao Sun. Cube attack against 843-round Trivium. *IACR Cryptol. ePrint Arch.*, 2021:547, 2021.
36. Yosuke Todo. Integral cryptanalysis on full MISTY1. In Rosario Gennaro and Matthew Robshaw, editors, *CRYPTO 2015*, volume 9215 of *LNCS*, pages 413–432, 2015.
37. Yosuke Todo. Structural evaluation by generalized integral property. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015*, volume 9056 of *LNCS*, pages 287–314. Springer, 2015.
38. Yosuke Todo, Takanori Isobe, Yonglin Hao, and Willi Meier. Cube attacks on non-blackbox polynomials based on division property. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017*, volume 10403 of *LNCS*, pages 250–279. Springer, 2017.
39. Yosuke Todo, Takanori Isobe, Yonglin Hao, and Willi Meier. Cube attacks on non-blackbox polynomials based on division property. *IACR Cryptol. ePrint Arch.*, 2017:306, 2017.
40. Yosuke Todo and Masakatu Morii. Bit-based division property and application to Simon family. In Thomas Peyrin, editor, *FSE 2016*, volume 9783 of *LNCS*, pages 357–377. Springer, 2016.
41. Qingju Wang, Yonglin Hao, Yosuke Todo, Chaoyun Li, Takanori Isobe, and Willi Meier. Improved division property based cube attacks exploiting algebraic properties of superpoly. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018*, volume 10991 of *LNCS*, pages 275–305. Springer, 2018.
42. SenPeng Wang, Bin Hu, Jie Guan, Kai Zhang, and Tairong Shi. MILP-aided method of searching division property using three subsets and applications. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019*, volume 11923 of *LNCS*, pages 398–427. Springer, 2019.
43. SenPeng Wang, Bin Hu, Jie Guan, Kai Zhang, and Tairong Shi. A practical method to recover exact superpoly in cube attack. *IACR Cryptology ePrint Archive*, 2019:259, 2019.
44. Senpeng Wang, Bin Hu, Jie Guan, Kai Zhang, and Tairong Shi. Exploring secret keys in searching integral distinguishers based on division property. *IACR Trans. Symmetric Cryptol.*, 2020(3):288–304, 2020.
45. Zejun Xiang, Wentao Zhang, Zhenzhen Bao, and Dongdai Lin. Applying MILP method to searching integral distinguishers based on division property for 6 lightweight block ciphers. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016*, volume 10031 of *LNCS*, pages 648–678. Springer, 2016.
46. Chen-Dong Ye and Tian Tian. A new framework for finding nonlinear superpolies in cube attacks against trivium-like ciphers. In Willy Susilo and Guomin Yang, editors, *ACISP 2018*, volume 10946 of *LNCS*, pages 172–187. Springer, 2018.
47. Chen-Dong Ye and Tian Tian. Algebraic method to recover superpolies in cube attacks. *IET Inf. Secur.*, 14(4):430–441, 2020.
48. Chen-Dong Ye and Tian Tian. A practical key-recovery attack on 805-round trivium. *IACR Cryptol. ePrint Arch.*, 2020:1404, 2020.
49. Chendong Ye and Tian Tian. Revisit division property based cube attacks: Key-recovery or distinguishing attacks? *IACR Trans. Symmetric Cryptol.*, 2019(3):81–102, 2019.

Appendix

A Propagation Rules and MILP Models for XOR, AND and COPY

Algorithm 5. The propagation rules for $\boldsymbol{f}: \mathbb{F}_2^n \to \mathbb{F}_2^m, \boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$ can be computed by Algorithm 5, interested readers are refered to [23,19] for more details.

Algorithm 5: Compute the propagation rules for $\boldsymbol{f}: \mathbb{F}_2^n \to \mathbb{F}_2^m, \boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$

```
Procedure ComputeRule(n, m, x, y):
    Initialize an empty set S
    for u ∈ F_2^n do
        for v ∈ F_2^m do
            /* symbolic computation                    */
            if x^u → y^v then put (u, v) into S
    return S
```

Propagation Rules of XOR, AND and COPY. According to [19], the rules for XOR, AND and COPY are shown as follows,

Rule 1 (XOR [19,20]) *Let* $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$ *and* $\boldsymbol{y} = (x_0 \oplus x_1, x_2, \ldots, x_{n-1})$ *be the input and output vector of a* XOR *function. Considering a monomial of* $\boldsymbol{x}$ *as* $\boldsymbol{x}^{\boldsymbol{u}}$*, the monomials* $\boldsymbol{y}^{\boldsymbol{v}}$ *of* $\boldsymbol{y}$ *meet the condition that* $\boldsymbol{x}^{\boldsymbol{u}} \to \boldsymbol{y}^{\boldsymbol{v}}$ *only when* $\boldsymbol{v}$ *satisfy*

$$\boldsymbol{v} = (u_0 + u_1, u_2, \ldots, u_{n-1}), \quad (u_0, u_1) \in \{(0,0), (0,1), (1,0)\}.$$

Rule 2 (AND [19,20]) *Let* $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$ *and* $\boldsymbol{y} = (x_0 \vee x_1, x_2, \ldots, x_{n-1})$ *be the input and output vector of a* AND *function. Considering a monomial of* $\boldsymbol{x}$ *as* $\boldsymbol{x}^{\boldsymbol{u}}$*, the monomials* $\boldsymbol{y}^{\boldsymbol{v}}$ *of* $\boldsymbol{y}$ *meet the condition that* $\boldsymbol{x}^{\boldsymbol{u}} \to \boldsymbol{y}^{\boldsymbol{v}}$ *only when* $\boldsymbol{v}$ *satisfy*

$$\boldsymbol{v} = (u_0, u_2, \ldots, u_{n-1}), \quad (u_0, u_1) \in \{(0,0), (1,1)\}.$$

Rule 3 (COPY [19,20]) *Let* $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$ *and* $\boldsymbol{y} = (x_0, x_0, x_1, x_2, \ldots, x_{n-1})$ *be the input and output vector of a* COPY *function. Considering a monomial of* $\boldsymbol{x}$ *as* $\boldsymbol{x}^{\boldsymbol{u}}$*, the monomials* $\boldsymbol{y}^{\boldsymbol{v}}$ *of* $\boldsymbol{y}$ *meet the condition that* $\boldsymbol{x}^{\boldsymbol{u}} \to \boldsymbol{y}^{\boldsymbol{v}}$ *only when* $\boldsymbol{v}$ *satisfy*

$$\boldsymbol{v} = \begin{cases} (0, 0, u_2, \ldots, u_{n-1}), & \text{if } u_0 = 0 \\ (0, 1, u_2, \ldots, u_{n-1}), (1, 0, u_2, \ldots, u_{n-1}), (1, 1, u_2, \ldots, u_{n-1}), & \text{if } u_0 = 1 \end{cases}$$

MILP Model of the Propagation Trails. When constructing the MILP model, we usually initialize an empty model, denoted by $\mathcal{M}$, then generate some MILP variables such as $\mathcal{M}.var \leftarrow a, b, c$; where a, b, c are some variables we need in the following inequalities. Then some inequalities are added to the model such as $\mathcal{M}.con \leftarrow a + b \geq c$; Finally, according to the language of the MILP tool in the hand, we can call the tool to solve the model and obtain the results including feasible or infeasible. If the model is feasible, we could derive a solution of all the inequalities. Under the special settings, some tools can return all possible solutions.

A MILP model for tracing the propagation trails of a whole cipher can be established by collecting all the inequalities for each component functions. Here we introduce the MILP models for the three mentioned basic functions XOR, AND and COPY. Note that the following models describe the generalized forms of Rule 1, 2 and 3, which are easily derived from their original forms.

Model 1 (XOR [19,20]) *Let* $(a_0, a_1, \ldots, a_{n-1}) \xrightarrow{\text{XOR}} b$ *be a propagation trail of* XOR. *The following inequalities suffice to describe all the valid trails for* XOR,

$$\begin{cases} \mathcal{M}.var \leftarrow a_0, a_1, \ldots, a_{n-1}, b \text{ as binary}; \\ \mathcal{M}.con \leftarrow b = a_0 + a_1 + \cdots + a_{n-1}; \end{cases}$$

Model 2 (AND [19,20]) *Let* $(a_0, a_1, \ldots, a_{n-1}) \xrightarrow{\text{AND}} b$ *be a propagation trail of* AND. *The following inequalities suffice to describe all the valid trails for* AND,

$$\begin{cases} \mathcal{M}.var \leftarrow a_0, a_1, \ldots, a_{n-1}, b \text{ as binary}; \\ \mathcal{M}.con \leftarrow b = a_i, \ \forall i \in \{0, 1, \ldots n-1\}. \end{cases}$$

Model 3 (COPY [19,20]) *Let a* $\xrightarrow{\text{COPY}} (b_0, b_1, \ldots, b_{n-1})$ *be a propagation trail of* AND. *The following inequalities suffice to describe all the valid trails for* COPY,

$$\begin{cases} \mathcal{M}.var \leftarrow a, b_0, b_1, \ldots, b_{n-1} \text{ as binary}; \\ \mathcal{M}.con \leftarrow b_0 + b_1 + \cdots + b_{n-1} \geq a; \\ \mathcal{M}.con \leftarrow a \geq b_i, \ \forall i \in \{0, 1, \ldots, n-1\}. \end{cases}$$

If the MILP solver supports the OR $(\vee)$ *operation, then the model can also be represented by*

$$\begin{cases} \mathcal{M}.var \leftarrow a, b_0, b_1, \ldots, b_{n-1} \text{ as binary}; \\ \mathcal{M}.con \leftarrow a = b_0 \vee b_1 \vee \cdots \vee b_{n-1}; \end{cases}$$

B Previous Methods for Exact Superpoly Recovery Revisit

We now recall existing methods for superpoly recovery in the language of monomial prediction.

Wang et al.'s Pruning Method. In [42], Wang et al. introduced the first systematical method to recover the exact superpoly based on the three-subset division property [40]. For a parameterized composite vectorial Boolean function shown in Equation (2) with $\boldsymbol{x}^{(i+1)} = \boldsymbol{f}^{(i)}(\boldsymbol{x}^{(i)}), 0 \leq i \leq r-1, \boldsymbol{f}^{(i)} : \mathbb{F}_2^{n_i} \to \mathbb{F}_2^{n_{i+1}}$, whether $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \to \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$ is determined by the following procedures [42]:

1. compute $\mathbb{T}_1 = \{\pi_{\boldsymbol{u}^{(1)}}(\boldsymbol{x}^{(1)}) : \pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \to \pi_{\boldsymbol{u}^{(1)}}(\boldsymbol{x}^{(1)}), \boldsymbol{u}^{(1)} \in \mathbb{F}_2^{n_1}\}$.
2. for each $\pi_{\boldsymbol{u}^{(1)}}(\boldsymbol{x}^{(1)}) \in \mathbb{T}_1$, check whether $\pi_{\boldsymbol{u}^{(1)}}(\boldsymbol{x}^{(1)}) \rightsquigarrow \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$ by an MILP model based on the two-subset bit-based division property. If there is no trail, it is discarded.
3. for each surviving $\pi_{\boldsymbol{u}^{(1)}}(\boldsymbol{x}^{(1)}) \in \mathbb{T}_1$, compute $\pi_{\boldsymbol{u}^{(2)}}(\boldsymbol{x}^{(2)})$ satisfying $\pi_{\boldsymbol{u}^{(1)}}(\boldsymbol{x}^{(1)}) \to \pi_{\boldsymbol{u}^{(2)}}(\boldsymbol{x}^{(2)})$ and put all $\pi_{\boldsymbol{u}^{(2)}}(\boldsymbol{x}^{(2)})$ in $\mathbb{T}_2$, i.e., $\mathbb{T}_2 = \{\pi_{\boldsymbol{u}^{(2)}}(\boldsymbol{x}^{(2)}) : \pi_{\boldsymbol{u}^{(1)}}(\boldsymbol{x}^{(1)}) \to \pi_{\boldsymbol{u}^{(2)}}(\boldsymbol{x}^{(2)}), \pi_{\boldsymbol{u}^{(1)}}(\boldsymbol{x}^{(1)}) \in \mathbb{T}_1\}$.
4. discard those monomials in $\mathbb{T}_2$ occurring an even-number of times, and then discard those $\pi_{\boldsymbol{u}^{(2)}}(\boldsymbol{x}^{(2)})$ that have no trails to $\pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$ with the MILP approach. Repeat the above step for $\mathbb{T}_2$.

There are two possible results of the above pruning method: *(1st)* we successfully obtain $\mathbb{T}_r$ that contains odd-number $\pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$, then we determine $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \to \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$. *(2nd)* $\mathbb{T}_i, 1 \leq i \leq r$ will be an empty set after discarding the bad elements, then we determine $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \nrightarrow \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$. In theory, this method is useful in recovering the exact superpoly because it can determine whether $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \to \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$ or not. However, it is impractical for many reasons. Firstly, the size of $\mathbb{T}_i$ may explode as the round grows so that it costs unbearable memory and time to proceed. Secondly, we never know which monomial of the key will appear in the superpoly, which means we have to test all possible monomials of the key one by one, making it impossible for complex superpolies. Indeed, they only gave a theoretical result for 841-round Trivium.

Ye and Tian's Algebraic Method. In [47], Ye and Tian introduced a pure algebraic method to recover the exact superpoly. For a parameterized Boolean function which can be divided into two parts as $f(\boldsymbol{x}, \boldsymbol{k}) = f^{(1)} \circ \boldsymbol{f}^{(0)}(\boldsymbol{x}, \boldsymbol{k})$, denote the output of $\boldsymbol{f}^{(0)}$ by $\boldsymbol{s} = (s_0, s_1, \ldots, s_{n-1})$. We first express $f(\boldsymbol{x}, \boldsymbol{k})$ in a polynomial of $s_0, s_1, \ldots, s_{n-1}$, all the monomials of $\boldsymbol{s}$ contained by f is denoted as $\mathbb{T}$. Then for a cube term $\boldsymbol{x}^{\boldsymbol{u}}$, we have

$$\mathsf{Coe}\left(f, \boldsymbol{x}^{\boldsymbol{u}}\right) = \mathsf{Coe}\left(\bigoplus_{\boldsymbol{t} \in \mathbb{T}} \boldsymbol{s}^{\boldsymbol{t}}, \boldsymbol{x}^{\boldsymbol{u}}\right) = \bigoplus_{\boldsymbol{t} \in \mathbb{T}} \mathsf{Coe}\left(\boldsymbol{s}^{\boldsymbol{t}}, \boldsymbol{x}^{\boldsymbol{u}}\right). \tag{7}$$

Note that each coordinate of $\boldsymbol{s}$ is also a Boolean function of $\boldsymbol{x}$ and $\boldsymbol{k}$. Then by combining all possible terms in the polynomial of s_i satisfying $t_i = 1$, it is possible to compute out $\mathsf{Coe}\left(\boldsymbol{s}^{\boldsymbol{t}}, \boldsymbol{x}^{\boldsymbol{u}}\right)$. Finally, we can recover the whole superpoly according to Equation (7). However, in most cases, $\mathsf{Coe}\left(\boldsymbol{s}^{\boldsymbol{t}}, \boldsymbol{x}^{\boldsymbol{u}}\right)$ is too complicated to compute. To handle this problem, Ye and Tian introduced the *useful cube* and its criterion. If we consider a useful cube, $\mathsf{Coe}\left(\boldsymbol{s}^{\boldsymbol{t}}, \boldsymbol{x}^{\boldsymbol{u}}\right)$ will be easier to calculate. This method is potential for massive superpolies. But an Achilles'

heel of this method is that it is very possible that we cannot find any useful cube at all as the rounds of the target cipher become larger. Actually, they at most recovered some polynomials of 838-round TRIVIUM.

Ye and Tian's Recursively-Expressing Method. In [49], Ye and Tian proposed another method to recover the exact superpoly. This method has some dual property to Wang et al's pruning method [42]. For a parameterized Boolean function which can be divided into a sequence of r vectorial Boolean function, $f(\boldsymbol{x},\boldsymbol{k}) = f^{(r-1)} \circ \boldsymbol{f}^{(r-2)} \circ \cdots \circ \boldsymbol{f}^{(0)}(\boldsymbol{x},\boldsymbol{k})$, and the output of $\boldsymbol{f}^{(i)}$ is denoted by $\boldsymbol{s}^{(i+1)}$. According to [49], the superpoly of a cube term $\boldsymbol{x}^{\boldsymbol{u}}$ can be recovered by the following procedures:

1. express f in a polynomial of $\boldsymbol{s}^{(r-1)}$, and push all monomials of $\boldsymbol{s}^{(r-1)}$ contained by f in $\mathbb{S}^{(r-1)}$, i.e., $\mathbb{S}^{(r-1)} = \{\pi_{\boldsymbol{t}^{(r-1)}}(\boldsymbol{s}^{(r-1)}) : \pi_{\boldsymbol{t}^{(r-1)}}(\boldsymbol{s}^{(r-1)}) \to f\}$.
2. for each monomial in $\mathbb{S}^{(r-1)}$, we take MILP model of the two-subset division property [40,45] to discard those satisfying $\mathsf{Coe}\left(\pi_{\boldsymbol{t}^{(r-1)}}(\boldsymbol{s}^{(r-1)}), \boldsymbol{x}^{\boldsymbol{u}}\right) = 0$.
3. express every surviving element in $\mathbb{S}^{(r-1)}$ by $\boldsymbol{s}^{(r-2)}$, and push all related monomials to $\boldsymbol{s}^{(r-2)}$ in $\mathbb{S}^{(r-2)}$. Repeat the above step for $\mathbb{S}^{(r-2)}$.

If the program finishes properly, then f could be represented by $\boldsymbol{x}[\boldsymbol{u}]$ and $\boldsymbol{k}$, the superpoly of $\boldsymbol{x}^{\boldsymbol{u}}$ is recovered naturally. Of course, it is possible that a certain $\mathbb{S}^{(i)}$ is empty, then the program returns $\mathsf{Coe}(f, \boldsymbol{x}^{\boldsymbol{u}}) = 0$. However, it is likely that there are too many monomials in $\mathbb{S}^{(i)}$ which cannot be discarded, especially for a massive superpoly. In [49], they verified several proposed superpolies in [38,41], but no new superpoly of TRIVIUM for more than 380 rounds is recovered.

Hao's et al.'s PoolSearchMode Method. Based on the `PoolSearchMode` of Gurobi, Hao et al. introduced a method to enumerate all possible propagation trails [19,20]. For a parameterized Boolean function $f(\boldsymbol{x},\boldsymbol{k})$, considering the cube term $\boldsymbol{x}^{\boldsymbol{u}}$, we construct a MILP model over all possible $\boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}} \rightsquigarrow f(\boldsymbol{x},\boldsymbol{k})$ [6]. We turn on the `PoolSearchMode` of Gurobi solver to find all solutions. When the model is solved, we store all the $\boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}}$ int a hash table, and count $|\boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}} \bowtie f|$ for each entry. Only those $\boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}}$'s appearing odd times are presented in f. Finally, the superpoly of $\boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}}$ can be computed as

$$\mathsf{Coe}(f, \boldsymbol{x}^{\boldsymbol{u}}) = \mathsf{Coe}\left(\bigoplus_{|\boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}} \bowtie f| \equiv 1 \pmod 2} \boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}}, \boldsymbol{x}^{\boldsymbol{u}}\right) = \bigoplus_{|\boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}} \bowtie f| \equiv 1 \pmod 2} \mathsf{Coe}(\boldsymbol{k}^{\boldsymbol{v}}\boldsymbol{x}^{\boldsymbol{u}}, \boldsymbol{x}^{\boldsymbol{u}}).$$

Later in [23], Hu et al. observed that for the composite function f, where

$$f = f^{(r-1)} \circ \boldsymbol{f}^{(r-2)} \circ \cdots \circ \boldsymbol{f}^{(0)},$$

if $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \rightsquigarrow f$, then for $0 < i < r$,

$$|\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \bowtie f| \equiv \sum_{\pi_{\boldsymbol{u}^{(r-i)}}(\boldsymbol{x}^{(r-i)}) \to f} \left|\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \bowtie \pi_{\boldsymbol{u}^{(r-i)}}(\boldsymbol{x}^{(r-i)})\right| \pmod 2.$$

[6] In the MILP implementation of the model, $\boldsymbol{v}$ is set as a free variable. We refer the readers to the codes in [19,20,23] for more details about the MILP implementation.

Since computing $|\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \bowtie \pi_{\boldsymbol{u}^{(r-i)}}(\boldsymbol{x}^{(r-i)})|$ one by one is much easier than computing $|\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \bowtie f|$ when i is significantly smaller than r, such a divide-and-conquer strategy helps to speed up the search significantly.

This method enable to recover the superpoly up to 842-round Trivium, 190-round Grain-128AEAD and 892-round Kreyvium. However, there are two serious restrictions on the method. Firstly, the efficiency is still a bottleneck. For example, we took the method in [23] to recover the superpoly for 843-round Trivium with the cube indices I_4 in Table 3, the program did not end after a month. In addition, the Gurobi `PoolSearchMode` supports at most 2,000,000,000 solutions. As the superpoly becomes increasingly complicated, the actual number of trails is likely to surpass this bound. Then, enumerating all solutions with the method of [19,20] to recover the superpoly will fail. In fact, in our result shown later for recovering the superpoly for 845-round Trivium, the number of trails will be at least 4,731,073,108.

C Results for 840-, 841- and 842-Round Trivium

C.1 Improved Key Recovery Attack for 840-Round Trivium

In [23], Hu et al. recovered three balanced superpolies of 75-dimension cubes. Then they could recover 3 bits of key information, the remaining 77 bits information could be searched by force. Then the complexity is about $2^{77.8}$, which is the best key recovery attack for 840-round Trivium. In this subsection, we proposed some new superpolies based on the nested monomial prediction technique. Although none of them are balanced, we can still extract some information of the key if considering them together. The key recovery attacks for 840-round Trivium is improved slightly.

The cube indices we use for attacking 840-round Trivium is listed in Table 7. For each cube indices in Table 7, we take our new framework to recover the superpoly. The number of propagation trails, the number of monomials in the final superpoly, the number of key bits involved in the superpoly, the degree, and the probability that the superpoly is 1 under 2^{15} randomly-chosen keys are given in Table 8. The concrete ANF of these superpolies are given in our public repository.

Considering all 6 cubes in Table 7, from the viewpoint of the entropy, as shown in Equation (5) and (6), we take 2^{15} random keys to test the distribution and then the entropy is about 3.68, i.e., we can extract averagely 3.68 bits of the key information. Since the dimensions of the 6 cube indices are at most 62, and at most 68 key bits are involved in the superpoly, the overhead of filtering the wrong keys in the online phase can be ignored. The dominant part the complexity for the key recovery attack is the exhaustive search, which is $2^{76.32}$ Trivium encryptions.

Table 7: Cube indices we use for the superpoly recovery of 840-round Trivium

I	$\lvert I\rvert$	Indices
I_0	47	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 21, 23, 25, 27, 30, 32, 34, 36, 38, 40, 42, 45, 47, 49, 51, 53, 55, 57, 60, 62, 64, 66, 68, 70, 72, 77, 75, 79
I_1	49	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 23, 25, 27, 30, 32, 34, 36, 38, 40, 42, 45, 47, 49, 51, 53, 55, 57, 60, 62, 64, 66, 68, 70, 72, 77, 75, 79
I_2	52	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 30, 32, 34, 36, 38, 40, 42, 45, 47, 49, 51, 53, 55, 57, 60, 62, 64, 66, 68, 70, 72, 77, 75, 79
I_3	53	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 30, 32, 34, 36, 38, 40, 42, 45, 47, 49, 51, 53, 55, 57, 60, 62, 64, 66, 68, 70, 72, 77, 75, 79
I_4	56	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 36, 38, 40, 42, 45, 47, 49, 51, 53, 55, 57, 60, 62, 64, 66, 68, 70, 72, 77, 75, 79
I_5	62	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 40, 41, 42, 43, 44, 45, 46, 47, 49, 51, 53, 55, 57, 60, 62, 64, 66, 68, 70, 72, 77, 75, 79

Table 8: Some information related to the superpoly of $\mathbb{C}_I$ for 840-round Trivium

I	# Trails	#Monomials	#Involved Key Bits	Degree	Balancedness
I_0	2,380,185	390,899	57	20	0.02
I_1	2,421,495	357,989	70	20	0.08
I_2	320,805	31,647	61	17	0.14
I_3	1,988,685	116,145	60	17	0.26
I_4	80,197	7,549	68	14	0.30
I_5	40,737	1,253	58	12	0.44

C.2 Improved Key Recovery for 841-Round Trivium

In [23], Hu et al. recovered two balanced superpolies of 76-dimension cubes. Then they could recover 2 bits of key information, the remaining 78 bits information could be searched by force. Then the complexity is about $2^{78.6}$, which is the best key recovery attack for 841-round Trivium. In this paper, we consider the following 56-dimension cube indices

$$I = \{\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 36, 38, 40, 42, 45, 47, 49, 51, 53, 55, 57, 60, 62, 64, 66, 68, 70, 72, 77, 75, 79\ \}.$$

By the nested monomial prediction technique, the superpoly of this cube can be recovered. Among totally 390,657 trails, 20,485 terms appear in the superpoly. The superpoly is a polynomial of degree 16, related to 61 key bits.

Since the superpoly is complicated, we evaluate its balancedness by experiments. With 2^{15} different keys, 15,654 keys make the output be 1, i.e., the probability that the output is 1 is approximately 0.48. From the perspective of the entropy, approximately 1 bit information of keys can be extracted on average. Since the complexity of the key filtering by this superpoly in the online phase can be ignored, together with the two superpolies recovered in [23], 3 key bits can be obtained. The following 77 bits are gotten by exhaustive search. The total complexity is 2^{78}.

C.3 Improved Key Recovery for 842-Round Trivium

In [23], Hu et al. recovered two balanced superpolies of 76-dimension cubes. Then they could recover 2 bits of key information, the remaining 78 bits information could be searched by force. Then the complexity is about $2^{78.6}$, which is the best key recovery attack for 842-round Trivium. In this paper, we consider the following 56-dimension cube indices

$$I = \{\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 36, 38, 40, 42, 45, 47, 49, 51, 53, 55, 57, 60, 62, 64, 66, 68, 70, 72, 77, 75, 79\ \}.$$

By the hourglass technique, the superpoly of this cube can be recovered. Among totally 9,395,018 trails, 343,000 terms appear in the superpoly. The superpoly is a polynomial of degree 17, related to 75 key bits. Since the superpoly is complicated, we evaluate its balancedness by experiments. With 2^{15} different keys, 16,392 keys make the output be 1, i.e., the probability that the output is 1 is approximately 0.50. Then this is (almost) a balance polynomial. From the perspective of the entropy, 1 bit information of keys can be extracted on average. Since the complexity of the key filtering by this superpoly in the online phase can be ignored (at most 2^{75}), together with the two superpolies recovered in [23], 3 key bits can be obtained. and the following 77 bits are gotten by exhaustive search. The total complexity is 2^{78}.

D Models and Functions Used for Grain-128AEAD

Algorithm 6: Model for the propagation trails of R-round Grain-128AEAD

Procedure `ModelGrain(`*round* R, $\pi_{\boldsymbol{t}^{(R)}}(\boldsymbol{s}^{(R)})$, I`):`
- Prepare an empty MILP Model $\mathcal{M}$
- $\mathcal{M}.var \leftarrow \mathtt{b}_\mathtt{i}^0$ for $\mathtt{i} \in \{0, 1, \ldots, 127\}$ as binary
- $\mathcal{M}.var \leftarrow \mathtt{s}_\mathtt{i}^0$ for $\mathtt{i} \in \{0, 1, \ldots, 127\}$ as binary
- $\mathcal{M}.con \leftarrow \mathtt{s}_{127}^0 = 0$
- **for** $\mathtt{i} = 0$ *to* 95 **do**
  - $\mathcal{M}.con \leftarrow \mathtt{s}_\mathtt{i}^0 = 1 \ \forall\ i \in I$
  - $\mathcal{M}.con \leftarrow \mathtt{s}_\mathtt{i}^0 = 0 \ \forall\ i \notin I$
- **for** $r = 0$ *to* $R - 1$ **do**
  - $(\mathcal{M}, \mathtt{b}_0', \ldots, \mathtt{b}_{127}', \mathtt{z}^\mathtt{r}) = \mathtt{funcZ}(\mathcal{M}, \mathtt{b}_0^{\mathtt{r}-1}, \ldots, \mathtt{b}_{127}^{\mathtt{r}-1}, \mathtt{s}_0^{\mathtt{r}-1}, \ldots, \mathtt{s}_{127}^{\mathtt{r}-1})$
  - $\mathcal{M}.var \leftarrow \mathtt{zg}, \mathtt{zf}$ as binary
  - $\mathcal{M}.con \leftarrow \mathtt{z}^\mathtt{r} = \mathtt{zg} \vee \mathtt{zf}$
  - $(\mathcal{M}, \mathtt{b}_0'', \ldots, \mathtt{b}_{127}'', \mathtt{g}) = \mathtt{funcG}(\mathcal{M}, \mathtt{b}_0', \ldots, \mathtt{b}_{127}')$
  - $(\mathcal{M}, \mathtt{s}_0'', \ldots, \mathtt{s}_{127}'', \mathtt{f}) = \mathtt{funcF}(\mathcal{M}, \mathtt{s}_0', \ldots, \mathtt{s}_{127}')$
  - **for** $i = 0$ *to* 126 **do**
    - $b_i^r = b_{i+1}''$
    - $s_i^r = s_{i+1}''$
  - $\mathcal{M}.var \leftarrow \mathtt{b}_{127}^\mathtt{r}, \mathtt{s}_{127}^\mathtt{r}$ as binary
  - $\mathcal{M}.con \leftarrow \mathtt{b}_0'' = 0$
  - $\mathcal{M}.con \leftarrow \mathtt{b}_{127}^\mathtt{r} = \mathtt{g} + \mathtt{s}_0'' + \mathtt{zg}$
  - $\mathcal{M}.con \leftarrow \mathtt{s}_{127}^\mathtt{r} = \mathtt{f} + \mathtt{zf}$
  - `/* Additional constraint in [19] */`
  - $\mathcal{M}.con \leftarrow \mathtt{s}_0^\mathtt{r} + \mathtt{z}^\mathtt{r} \leq 1$
- **for** $i = 0$ *to* 127 **do**
  - `/*` $\boldsymbol{t}_i^R = (t_0^R, t_1^R, \ldots, t_{127}^R, t_{128}^R, \ldots t_{255}^R)$ `*/`
  - $\mathcal{M}.con \leftarrow \mathtt{b}_\mathtt{i}^\mathtt{R} = t_i^R$
  - $\mathcal{M}.con \leftarrow \mathtt{s}_\mathtt{i}^\mathtt{R} = t_{i+128}^R$
- **return** $\mathcal{M}$

Algorithm 7: `ChooseRiGrain-128AEAD` and `ChooseTiGrain-128AEAD`

```
Procedure ChooseRiGrain-128AEAD(𝕊, r):
    r' = 1
    while |𝕊'| < 10,000 and r − r' > 0 do
        𝕊' = ∅
        for s ∈ 𝕊 do 𝕊' = 𝕊' ∪ Express(s, r, r')
    return r'
Procedure ChooseTiGrain-128AEAD(r):
    if r ≥ 120 then τ = 60 seconds
    else if r ≥ 110 then τ = 120 seconds
    else if r ≥ 100 then τ = 180 seconds
    else if r ≥ 10 then τ = 360 seconds
    else if r ≥ 0 then τ = ∞
    return τ
```

Algorithm 8: MILP model for NFSR and LFSR in Grain-128AEAD

Procedure $\texttt{funcZ}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \texttt{s}_0, \ldots, \texttt{s}_{127})$**:**
- $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \texttt{s}_0, \ldots, \texttt{s}_{127}, \texttt{a}_1) = \texttt{AND}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \texttt{s}_0, \ldots, \texttt{s}_{127}, \{12\}, \{8\})$
- $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \texttt{s}_0, \ldots, \texttt{s}_{127}, \texttt{a}_2) = \texttt{AND}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \texttt{s}_0, \ldots, \texttt{s}_{127}, \varnothing, \{13, 20\})$
- $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \texttt{s}_0, \ldots, \texttt{s}_{127}, \texttt{a}_3) = \texttt{AND}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \texttt{s}_0, \ldots, \texttt{s}_{127}, \{95\}, \{42\})$
- $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \texttt{s}_0, \ldots, \texttt{s}_{127}, \texttt{a}_4) = \texttt{AND}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \texttt{s}_0, \ldots, \texttt{s}_{127}, \varnothing, \{60, 79\})$
- $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \texttt{s}_0, \ldots, \texttt{s}_{127}, \texttt{a}_5) = \texttt{AND}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \texttt{s}_0, \ldots, \texttt{s}_{127}, \{12, 95\}, \{94\})$
- $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \texttt{x}_1) = \texttt{XOR}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \{2, 15, 36, 45, 64, 73, 89\}, \varnothing)$
- $(\mathcal{M}, \varnothing, \texttt{s}_0, \ldots, \texttt{s}_{127}, \texttt{x}_2) = \texttt{XOR}(\mathcal{M}, \varnothing, \texttt{s}_0, \ldots, \texttt{s}_{127}, \varnothing, \{93\})$
- $\mathcal{M}.var \leftarrow \texttt{z}$ as binary
- $\mathcal{M}.con \leftarrow \texttt{z} = \texttt{x}_1 + \texttt{x}_2 + \sum_{\texttt{i}=1}^{5} \texttt{a}_\texttt{i}$
- **return** $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \texttt{s}_0, \ldots, \texttt{s}_{127}, \texttt{z})$

Procedure $\texttt{funcF}(\mathcal{M}, \texttt{s}_0, \ldots, \texttt{s}_{127})$**:**
- $(\mathcal{M}, \varnothing, \texttt{s}_0, \ldots, \texttt{s}_{127}, \texttt{f}) = \texttt{XOR}(\mathcal{M}, \varnothing, \texttt{s}_0, \ldots, \texttt{s}_{127}, \varnothing, \{0, 7, 38, 70, 81, 96\})$ **return** $(\mathcal{M}, \texttt{s}_0, \ldots, \texttt{s}_{127}, \texttt{f})$

Procedure $\texttt{funcG}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127})$**:**
- $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \texttt{a}_1) = \texttt{AND}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \{3, 67\}, \varnothing)$
- $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \texttt{a}_2) = \texttt{AND}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \{11, 13\}, \varnothing)$
- $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \texttt{a}_3) = \texttt{AND}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \{17, 18\}, \varnothing)$
- $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \texttt{a}_4) = \texttt{AND}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \{27, 59\}, \varnothing)$
- $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \texttt{a}_5) = \texttt{AND}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \{40, 48\}, \varnothing)$
- $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \texttt{a}_6) = \texttt{AND}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \{61, 65\}, \varnothing)$
- $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \texttt{a}_7) = \texttt{AND}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \{68, 84\}, \varnothing)$
- $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \texttt{a}_8) = \texttt{AND}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \{88, 92, 93, 95\}, \varnothing)$
- $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \texttt{a}_9) = \texttt{AND}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \{22, 24, 25\}, \varnothing)$
- $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \texttt{a}_{10}) = \texttt{AND}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \{70, 78, 82\}, \varnothing)$
- $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \texttt{x}) = \texttt{AND}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \varnothing, \{0, 26, 56, 91, 96\}, \varnothing)$
- $\mathcal{M}.var \leftarrow \texttt{g}$ as binary
- $\mathcal{M}.con \leftarrow \texttt{g} = \texttt{x} + \sum_{\texttt{i}=1}^{10} \texttt{a}_\texttt{i}$
- **return** $(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \texttt{g})$

Algorithm 9: MILP model for XOR and AND in Grain-128AEAD

Procedure $\texttt{AND}(\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \texttt{s}_0, \ldots, \texttt{s}_{127}, I, J)$**:**
- $\mathcal{M}.var \leftarrow \texttt{b}'_\texttt{i}, \texttt{x}_\texttt{i}\ \forall\ i \in I$ as binary
- $\mathcal{M}.var \leftarrow \texttt{s}'_\texttt{j}, \texttt{y}_\texttt{j}\ \forall\ j \in J$ as binary
- $\mathcal{M}.var \leftarrow \texttt{z}$ as binary
- $\mathcal{M}.con \leftarrow \texttt{b}_\texttt{i} = \texttt{b}'_\texttt{i} \vee \texttt{x}_\texttt{i}\ \forall\ i \in I$
- $\mathcal{M}.con \leftarrow \texttt{s}_\texttt{i} = \texttt{s}'_\texttt{j} \vee \texttt{y}_\texttt{j}\ \forall\ j \in J$
- $\mathcal{M}.con \leftarrow \texttt{z} = \texttt{x}_\texttt{i}\ \forall\ i \in I$
- $\mathcal{M}.con \leftarrow \texttt{z} = \texttt{y}_\texttt{j}\ \forall\ j \in J$
- **for** $i \in \{0, 1, \ldots, 127\} \backslash I$ **do** $\texttt{s}'_\texttt{i} = \texttt{s}_\texttt{i}$
- **for** $j \in \{0, 1, \ldots, 127\} \backslash J$ **do** $\texttt{b}'_\texttt{j} = \texttt{b}_\texttt{j}$
- **return** $(\mathcal{M}, \texttt{b}'_0, \ldots, \texttt{b}'_{127}, \texttt{s}'_0, \ldots, \texttt{s}'_{127}, \texttt{z})$

Procedure $\texttt{XOR}((\mathcal{M}, \texttt{b}_0, \ldots, \texttt{b}_{127}, \texttt{s}_0, \ldots, \texttt{s}_{127}, I, J)$**:**
- $\mathcal{M}.var \leftarrow \texttt{b}'_\texttt{i}, \texttt{x}_\texttt{i}\ \forall\ i \in I$ as binary
- $\mathcal{M}.var \leftarrow \texttt{s}'_\texttt{j}, \texttt{y}_\texttt{j}\ \forall\ j \in J$ as binary
- $\mathcal{M}.con \leftarrow \texttt{z}$ as binary
- $\mathcal{M}.con \leftarrow \texttt{b}_\texttt{i} = \texttt{b}'_\texttt{i} \vee \texttt{x}_\texttt{i}\ \forall i \in I$
- $\mathcal{M}.con \leftarrow \texttt{s}_\texttt{j} = \texttt{s}'_\texttt{j} \vee \texttt{y}_\texttt{j}\ \forall j \in J$
- $\mathcal{M}.con \leftarrow \texttt{z} = \sum_{\texttt{i} \in \texttt{I}} \texttt{x}_\texttt{i} + \sum_{\texttt{j} \in \texttt{J}} \texttt{y}_\texttt{j}$
- **for** $i \in \{0, 1, \ldots, 127\} \backslash I$ **do** $b'_i = b_i$
- **for** $j \in \{0, 1, \ldots, 127\} \backslash J$ **do** $s'_j = s_j$
- **return** $(\mathcal{M}, \texttt{b}'_0, \ldots, \texttt{b}'_{127}, \texttt{s}'_0, \ldots, \texttt{s}'_{127}, \texttt{z})$

Algorithm 10: Model for the propagation trails of R-round Kreyvium

Procedure $\texttt{LFSR}(\ \mathcal{M}, \mathtt{x}_0, \ldots, \mathtt{x}_{127})$**:**
  $\mathcal{M}.var \leftarrow \mathtt{a}, \mathtt{b}$ as binary
  $\mathcal{M}.con \leftarrow \mathtt{x}_0 = \mathtt{a} \vee \mathtt{b}$
  $(\mathtt{y}_0, \ldots, \mathtt{y}_{127}) = (\mathtt{x}_1, \ldots, \mathtt{x}_{126}, \mathtt{a})$
  return $(\mathcal{M}, y_0, y_1, \ldots, y_{127}, b)$

Procedure $\texttt{ModelKreyvium}$*(round R, $\pi_{\boldsymbol{t}^{(R)}}(\boldsymbol{s}^{(R)})$, I $\boldsymbol{c}$)*:
  Prepare an empty MILP model $\mathcal{M}$
  $\mathcal{M} \leftarrow \mathtt{s}_\mathtt{i}^0\ \forall\ i \in \{0, 1, \ldots, 287\}$ as binary
  $\mathcal{M} \leftarrow \mathtt{x}_\mathtt{i}\ \forall\ i \in \{0, 1, \ldots, 287\}$ as binary
  $\mathcal{M} \leftarrow \mathtt{k}_\mathtt{i}\ \forall\ i \in \{0, 1, \ldots, 287\}$ as binary
  for $\mathtt{i} = 0$ *to* 127 **do**
    $\mathcal{M}.con \leftarrow \mathtt{x}_\mathtt{i}^0 = 1\ \forall\ i \in I$
    $\mathcal{M}.con \leftarrow \mathtt{x}_\mathtt{i}^0 = 0\ \forall\ i \notin I$
  $\mathcal{M} \leftarrow \mathtt{K}_\mathtt{i}^{*,0}\ \forall\ i \in \{0, 1, \ldots, 287\}$ as binary
  $\mathcal{M} \leftarrow \mathtt{IV}_\mathtt{i}^{*,0}\ \forall\ i \in \{0, 1, \ldots, 287\}$ as binary
  $\mathcal{M}.con \leftarrow \mathtt{k}_\mathtt{i} = \mathtt{K}_{128-\mathtt{i}}^{*,0} \vee \mathtt{s}_\mathtt{i}^0$ for $i = 0, \ldots, 92$
  $\mathcal{M}.con \leftarrow \mathtt{k}_\mathtt{i} = \mathtt{K}_{128-\mathtt{i}}^{*,0}$ for $i = 93, \ldots, 127$
  $\mathcal{M}.con \leftarrow \mathtt{x}_\mathtt{i} = \mathtt{IV}_{128-\mathtt{i}}^{*,0} \vee \mathtt{s}_{93+\mathtt{i}}^0$ for $i = 0, \ldots, 127$
  for $r = 0$ *to* $R - 1$ **do**
    $(\mathcal{M}, \mathtt{K}_0^{*,\mathtt{r}+1}, \ldots, \mathtt{K}_{127}^{*,\mathtt{r}+1}, \mathtt{a}^\mathtt{r}) \leftarrow \texttt{LFSR}(\mathcal{M}, \mathtt{K}_0^{*,\mathtt{r}}, \ldots, \mathtt{K}_{127}^{*,\mathtt{r}})$
    $(\mathcal{M}, \mathtt{IV}_0^{*,\mathtt{r}+1}, \ldots, \mathtt{IV}_{127}^{*,\mathtt{r}+1}, \mathtt{a}^\mathtt{r}) \leftarrow \texttt{LFSR}(\mathcal{M}, \mathtt{IV}_0^{*,\mathtt{r}}, \ldots, \mathtt{IV}_{127}^{*,\mathtt{r}})$
    $(\mathcal{M}, \mathtt{x}_0, \ldots, \mathtt{x}_{287}) = \texttt{TriviumCore}(\mathcal{M}, \mathtt{s}_1^\mathtt{r}, \ldots, \mathtt{s}_{288}^\mathtt{r}, 65, 170, 90, 91, 92)$
    $(\mathcal{M}, \mathtt{y}_0, \ldots, \mathtt{y}_{287}) = \texttt{TriviumCore}(\mathcal{M}, \mathtt{x}_1, \ldots, \mathtt{x}_{288}, 161, 263, 174, 175, 176)$
    $(\mathcal{M}, \mathtt{z}_0, \ldots, \mathtt{z}_{287}) = \texttt{TriviumCore}(\mathcal{M}, \mathtt{y}_1, \ldots, \mathtt{y}_{288}, 242, 68, 285, 286, 287)$
    $\mathcal{M}.var \leftarrow \mathtt{t}_1^\mathtt{r}, \mathtt{t}_3^\mathtt{r}$ as binary
    $\mathcal{M}.con \leftarrow \mathtt{t}_1^\mathtt{r} = \mathtt{z}_{92} + \mathtt{b}^\mathtt{r}$
    $\mathcal{M}.con \leftarrow \mathtt{t}_3^\mathtt{r} = \mathtt{z}_{287} + \mathtt{a}^\mathtt{r}$
    $(\mathtt{s}_0^{\mathtt{r}+1}, \ldots, \mathtt{s}_{287}^{\mathtt{r}+1}) = (\mathtt{t}_3^\mathtt{r}, \mathtt{z}_0, \ldots, \mathtt{z}_{91}, \mathtt{t}_1^\mathtt{r}, \mathtt{z}_{93}, \ldots, \mathtt{z}_{286})$
  for $i = 0$ *to* 127 **do**
    /* $\boldsymbol{t}^{(R)} = (t_0^{(R)}, \ldots, t_{543}^{(R)})$ */
    $\mathcal{M}.con \leftarrow \mathtt{K}_\mathtt{i}^{*,\mathtt{R}} = t_i^{(R)}$
    $\mathcal{M}.con \leftarrow \mathtt{IV}_\mathtt{i}^{*,\mathtt{R}} = t_{i+128}^{(R)}$
  for $i = 0$ *to* 287 **do** $\mathcal{M}.con \leftarrow \mathtt{s}_\mathtt{i}^\mathtt{R} = t_{i+256}^{(R)}$
  return $\mathcal{M}$

E Models and Functions Used for Kreyvium

Algorithm 11: `ChooseRiKreyvium` and `ChooseTiKreyvium`

Procedure `ChooseRiKreyvium`($\mathbb{S}$, r)**:**
  $r' = 1$ **while** $|\mathbb{S}'| < 100,000$ *and* $r - r' > 0$ **do**
    $\mathbb{S}' = \emptyset$
    for $s \in \mathbb{S}$ **do**
      $\mathbb{S}' = \mathbb{S}' \cup$ `Express`(s, r, r')
  return r'
Procedure `ChooseTiKreyvium`(r)**:**
  if $r \geq 600$ **then** $\tau = 60$ seconds
  else if $r \geq 500$ **then** $\tau = 120$ seconds
  else if $r \geq 400$ **then** $\tau = 180$ seconds
  else if $r \geq 300$ **then** $\tau = 360$ seconds
  else if $r \geq 200$ **then** $\tau = 1440$ second
  else if $r \geq 0$ **then** $\tau = \infty$
  return τ

F The Superpoly of 894-Round Kreyvium

$$
\begin{aligned}
p_I = {} & k_{119} + k_{118} + k_{117}k_{118} + k_{105} + k_{104} + k_{102} + k_{99} + k_{95} + k_{94} + k_{93} + k_{92} + k_{89} + \\
& k_{88}k_{96} + k_{87}k_{88} + k_{86}k_{87}k_{96} + k_{85} + k_{85}k_{86} + k_{84}k_{127} + k_{84}k_{88} + k_{84}k_{86}k_{87} + \\
& k_{83} + k_{83}k_{88} + k_{83}k_{86}k_{87} + k_{83}k_{84} + k_{82}k_{83} + k_{82}k_{83}k_{127} + k_{82}k_{83}k_{88} + \\
& k_{82}k_{83}k_{86}k_{87} + k_{81}k_{82} + k_{81}k_{82}k_{88} + k_{81}k_{82}k_{86}k_{87} + k_{78} + k_{78}k_{85} + k_{78}k_{83}k_{84} + \\
& k_{76} + k_{76}k_{77} + k_{76}k_{77}k_{85} + k_{76}k_{77}k_{83}k_{84} + k_{75}k_{76} + k_{74}k_{104} + k_{73}k_{88} + \\
& k_{73}k_{86}k_{87} + k_{73}k_{74} + k_{72}k_{107} + k_{71}k_{101} + k_{71}k_{88} + k_{71}k_{86}k_{87} + k_{71}k_{76} + k_{70}k_{71} + \\
& k_{69} + k_{69}k_{70}k_{101} + k_{69}k_{70}k_{88} + k_{69}k_{70}k_{86}k_{87} + k_{69}k_{70}k_{76} + k_{68}k_{84} + k_{68}k_{82}k_{83} + \\
& k_{67}k_{68} + k_{66}k_{127} + k_{66}k_{104} + k_{66}k_{68} + k_{64}k_{65}k_{127} + k_{64}k_{65}k_{104} + k_{64}k_{65}k_{68} + \\
& k_{61} + k_{61}k_{104} + k_{61}k_{96} + k_{61}k_{84} + k_{61}k_{83} + k_{61}k_{82}k_{83} + k_{61}k_{81}k_{82} + k_{61}k_{73} + \\
& k_{61}k_{71} + k_{61}k_{69}k_{70} + k_{60} + k_{60}k_{84} + k_{60}k_{82}k_{83} + k_{60}k_{71} + k_{60}k_{69}k_{70} + k_{59}k_{117} + \\
& k_{59}k_{102} + k_{59}k_{79} + k_{59}k_{71} + k_{59}k_{69}k_{70} + k_{59}k_{60} + k_{59}k_{60}k_{104} + k_{58} + k_{58}k_{118} + \\
& k_{58}k_{84} + k_{58}k_{82}k_{83} + k_{58}k_{78} + k_{58}k_{76}k_{77} + k_{58}k_{66} + k_{58}k_{64}k_{65} + k_{57} + k_{57}k_{127} + \\
& k_{57}k_{88} + k_{57}k_{86}k_{87} + k_{57}k_{68} + k_{57}k_{61} + k_{57}k_{60} + k_{57}k_{58}k_{102} + k_{57}k_{58}k_{79} + k_{56} + \\
& k_{56}k_{88} + k_{56}k_{86}k_{87} + k_{56}k_{61} + k_{51} + k_{51}k_{85} + k_{51}k_{83}k_{84} + k_{51}k_{58} + k_{50} + k_{48} + \\
& k_{48}k_{72} + k_{46} + k_{45}k_{74} + k_{45}k_{66} + k_{45}k_{64}k_{65} + k_{45}k_{61} + k_{45}k_{59}k_{60} + k_{44}k_{101} + \\
& k_{44}k_{88} + k_{44}k_{86}k_{87} + k_{44}k_{76} + k_{44}k_{61} + k_{44}k_{60} + k_{44}k_{59} + k_{43} + k_{43}k_{59} + \\
& k_{43}k_{57}k_{58} + k_{42} + k_{42}k_{71} + k_{42}k_{69}k_{70} + k_{42}k_{44} + k_{41} + k_{40} + k_{39}k_{127} + k_{39}k_{104} + \\
& k_{39}k_{68} + k_{39}k_{58} + k_{39}k_{45} + k_{39}k_{40} + k_{38}k_{72} + k_{37}k_{88} + k_{37}k_{86}k_{87} + k_{37}k_{61} + k_{36} + \\
& k_{35} + k_{34}k_{104} + k_{34}k_{45} + k_{33}k_{34} + k_{32} + k_{32}k_{102} + k_{32}k_{79} + k_{32}k_{43} + k_{30} + k_{28} + \\
& k_{27} + k_{27}k_{88} + k_{27}k_{86}k_{87} + k_{27}k_{61} + k_{23}k_{24} + k_{20}k_{59} + k_{20}k_{57}k_{58} + k_{20}k_{32} + k_{17} + \\
& k_{17}k_{71} + k_{17}k_{69}k_{70} + k_{17}k_{44} + k_{15}k_{104} + k_{15}k_{73} + k_{15}k_{45} + k_{14} + k_{14}k_{88} + \\
& k_{14}k_{86}k_{87} + k_{14}k_{74} + k_{14}k_{61} + k_{14}k_{15} + k_{13} + k_{13}k_{107} + k_{13}k_{48} + k_{13}k_{38} + k_3 +
\end{aligned}
$$

$$k_1k_{84} + k_1k_{82}k_{83} + k_1k_{71} + k_1k_{69}k_{70} + k_1k_{57} + k_1k_{44} + k_0k_{71} + k_0k_{69}k_{70} + k_0k_{44}$$

G Möbius Transform Using Sparse ANFs

We consider a variant of the Möbius transformation by exploiting the property of the recovered superpolies. Recovered superpolies are massive compared to previous ones [19,20,23], but they are very sparse than random Boolean functions. For example, the recovered superpoly of 845-round TRIVIUM with $19,967,968 \approx 2^{24.3}$ monomials is really sparse as random Boolean function should have 2^{79} monomials. When the number of non-zero ANF coefficients is small, almost all 1-bit XORes are $0 \oplus 0$ in small k and they are redundant computation. Algorithm12 shows the binary Möbius transformation exploiting the sparse ANF. Let U_0 be the set of the monomials in recovered superpolies, and we traverse only elements in U in the first m steps. Then, for example in the 1st step, the number of XORing is reduced from 2^{n-1} to at most $|U_0|$ XORes. Moreover, in the mth step, $|U| \leq 2^{m-1}|U_0|$. The time complexity can be roughly written as

$$\sum_{i=0}^{m} 2^i|U_0| + (n-m)2^{n-1},$$

but the unit of the complexity in the 1st step ($1 \leq k \leq m$) and the 2nd step ($m+1 \leq k \leq n$) is different. Since we suppose that the unit of the 1st step is much expensive than that of the 2nd step, the value m should be chosen such that $2^m|U_0|$ can be regarded as negligible compared to 2^{n-1}.

In the case of the recovered superpoly of 845-round Trivium with $2^{24.3}$ monomials, $\sum_{i=0}^{m} 2^i|U_0| \approx 2^{64.3}$ with $m = 40$, and it should be negligible compared to $(80-40) \times 2^{79}$. In other words, this algorithm roughly allows us to halve the required time complexity of the Möbius transformation. Practically, the number of non-zero coefficients in our recovered superpolies is significantly small. Therefore, we simply assume that the time complexity of the Möbius transformation is $n \times 2^{n-2}$ hereinafter.

Algorithm 12: Möbius transformation exploiting sparse ANF

```
1  Procedure MöbiusTransForSparseANF(U_0 ⊂ F_2^n):
2      U = U_0
3      for k = 1 to m do
4          forall u ∈ {U|(u ∧ 2^k) = 0} do
5              u' = u ⊕ 2^k
6              if u' is included in U then
7                  Remove u' from U
8              else
9                  Insert u' into U
10     forall u ∈ U do
11         a[u] = 1
12     for k = m + 1 to n do
13         for i = 0 to 2^{n-k} do
14             for j = 0 to 2^{k-1} − 1 do
15                 a[2^k i + 2^{k-1} + j] = a[2^k i + j] ⊕ a[2^k i + 2^{k-1} + j]
16     return a
```