Key Filtering in Cube Attacks from the Implementation Aspect

Hao Fan[1], Yonglin Hao[2(✉)], Qingju Wang[3], Xinxin Gong[2], and Lin Jiao[2]

[1] School of Cyber Science and Technology, Shandong University, Qingdao, China
[2] State Key Laboratory of Cryptology, Beijing, 100878, China
[3] Telecom Paris, Institut Polytechnique de Paris, France
yykdszniao@gmail.com, haoyonglin@yeah.net, qjuwang@gmail.com, xinxgong@126.com, jiaolin_jl@126.com

Abstract. In cube attacks, key filtering is a basic step of identifying the correct key candidates by referring to the truth tables of superpolies. When terms of superpolies get massive, the truth table lookup complexity of key filtering increases significantly. In this paper, we propose the concept of implementation dependency dividing all cube attacks into two categories: *implementation dependent* and *implementation independent*. The implementation dependent cube attacks can only be feasible when the assumption that *one encryption oracle query is more complicated than one table lookup* holds. On the contrary, implementation independent cube attacks remain feasible in the extreme case where encryption oracles are implemented in the full codebook manner making one encryption query equivalent to one table lookup. From this point of view, we scrutinize existing cube attack results of stream ciphers Trivium, Grain-128AEAD, Acorn and Kreyvium. As a result, many of them turn out to be implementation dependent. Combining with the degree evaluation and divide-and-conquer techniques used for superpoly recovery, we further propose new cube attack results on Kreyvium reduced to 898, 899 and 900 rounds. Such new results not only mount to the maximal number of rounds so far but also are implementation independent.

Keywords: Stream ciphers · Cube attacks · Division property · Superpoly · Key filtering

1 Introduction

Cube attack was proposed by Dinur and Shamir in [2] at EUROCRYPT 2009 and has become one of the most efficient cryptanalysis methods against primitives taking public initial values (IV) and secret key as inputs. For a cipher with public IV $\boldsymbol{v} = (v_0, v_1, \ldots, v_{m-1}) \in \mathbb{F}_2^m$ and secret key $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1}) \in \mathbb{F}_2^n$, an output bit generated by the cipher can be regarded as a polynomial of $\boldsymbol{v}, \boldsymbol{x}$ denoted as $f(\boldsymbol{x}, \boldsymbol{v})$. In cube attacks, a set of IV indices, referred to as the *cube indices*, is selected as $I = \{i_0, i_1, \ldots, i_{|I|-1}\} \subset \{0, 1, \ldots, m-1\}$. Such a set I determines a specific structure called *cube*, denoted as C_I, containing $2^{|I|}$ values: the cube variables in $\{v_{i_0}, v_{i_1}, \ldots, v_{i_{|I|-1}}\}$ take all possible combinations

of values while the key and non-cube IV variables are static. It is proved that the summation of f over the cube C_I equals a particular polynomial $p(\boldsymbol{x}, \boldsymbol{v})$, commonly referred as the *superpoly* of cube I, denoted as $p_I(\boldsymbol{x}, \boldsymbol{v})$ or $p(\boldsymbol{x}, \boldsymbol{v})$ when I is clear from the context. The superpoly p_I also defines a set $J \subseteq \{0, \ldots, n-1\}$ such that the algebraic normal form ANF of p_I is only related to the key bit variable x_j for $j \in J$.

The general process of cube attacks can be naturally summarized into the 4 phases namely *superpoly recovery*, *key filtering*, *cube sum computation* and *exhaustive search*. The *superpoly recovery* recovery phase is carried out offline for determining I and the ANF (or truth table) of the corresponding superpoly p_I. Then, the cube summation over C_I, denoted as θ, is computed by querying the targeted encryption oracle for $2^{|I|}$ times. After that, the *key filtering* phase filter the correct key candidates satisfying $p(\boldsymbol{x}, \boldsymbol{v}) = \theta$ so as to recover 1 bit of secret key information. Finally, the *exhaustive search* recovers the remaining key bits through the exhaustive search with 2^{n-1} encryption oracle queries.

The superpoly recovery phase is crucial and used to dominate the overall complexity. Originally, the superpolies in cube attacks can only be recovered with repeated cube summation experiments restricting the superpoly ANFs to linear/quadratic form and limiting the cube dimensions within practical reach [2,13]. Theoretic deduction remains infeasible until the proposal of the division property based cube attack [18]: a combination of the division property [17,16,19,22] and cube attacks. Such a new cryptanalysis method enables us to conduct cube attacks with the mixed integer linear programming (MILP)–a mature technique that have been widely in the security evaluations of symmetric primitives against differential, linear and many other cryptanalysis methods [12,14,4]. The original division property based cube attacks suffer from extremely high offline complexities and a significant loss of accuracy [20,21]. After years' development, the state-of-art three-subset division property based cube attack [6] has been combined with the divide-and-conquer model solving technique [9,15,7] enabling us to recover the accurate ANFs of superpolies within a practical complexity, even when the superpolies are massive with $|J| \approx n$ and high algebraic degrees.

Motivations. Now that the superpoly recovery is no longer the complexity dominant, researchers turn to use smaller dimensional cubes with massive superpolies so as to conduct cube attacks covering more rounds. Following such a strategy, the current best cube attacks on Trivium, Grain-128AEAD, Kreyvium and Acorn, etc. [9,7] are all using massive superpolies related to almost all key bits, i.e., $|J| \approx n$, resulting in 2^n truth table lookups in the key filtering phase. Adding the 2^{n-1} queries in the exhaustive search phase, there is an obvious challenge that the overall complexity of cube attacks using massive superpolies may have exceeded the generic complexity bound of 2^n. According to the explanations in [9,7], the feasibility of such attacks is based on the assumption that 1 query to the encryption oracle is much more complicated than 1 table lookup: for example in [7], 1 query to the 848-round Trivium encryption oracle is regarded as $848 \times 9 = 7632$ XORs while a table lookup only contains 1 XOR. However, from

the adversary's view, a query to the oracle does not take more effort than the execution of an XOR operation. Besides, such a bitwise and roundwise implementation is not the only way to realize cryptographic primitives: the selection of tags in TRIVIUM naturally supports a 64-time acceleration [1] for fast software speeds; the time for an unrolled hardware implementation of the full encryption is exactly 1 clock tick which is equal to that of a XOR. Therefore, the applicability of massive superpolies to cube attacks heavily relies on the implementations so the following 2 questions should be discussed in detail:

1. Whether the existing cube attacks are feasible for arbitrary implementations.
2. Whether there exist implementation-independent cube attacks that can reach more rounds.

Our Contributions. In this paper, we answer the above questions and scrutinize the existing cube attacks on several ciphers. Our contributions can be summarized as follows:

- We give the concept of implementation dependency and divide cube attacks into two categories, namely the *implementation dependent cube attacks* and the *implementation independent cube attacks*. Implementation dependent cube attacks can only be feasible when a query to the encryption oracle is more expensive than a table lookup while the implementation independent cube attacks remain feasible in the extreme case where the encryption oracle is implemented as the full codebook making one oracle query equivalent to one table lookup.
- Following the above new concepts for cube attacks, we revisit the latest three-subset division property based cube attacks on several symmetric primitives. According to our evaluations, many current best results using massive superpolies, such as all attacks on TRIVIUM in [9], are *implementation dependent*.
- We also propose new *implementation independent* results on 898-, 899-, 900-round KREYVIUM using the methods in [9,7]: superior to their massive-superpoly based, implementation dependent counterparts.

We list all our results in Table 1.

Organization of the Paper. Sect. 2 provides the necessary background information. Then, we describe our new three-subset division property based cube attacks on round-reduced KREYVIUM in Sect. 3. After that, we introduce the concept of implementation dependency and detail the evaluation of an existing cube attack on TRIVIUM in Sect. 4. Thorough implementation dependency evaluations of current best cube attack results for our targeted primitives are given in Sect. 5 and we conclude the paper in Sect. 6.

2 Preliminary

In this section, we first summarize the general procedure of cube attacks in Sect. 2.1. Then, we briefly review the technique details of division property based superpoly recovery (Sect. 2.2) and table-lookup based key filtering (Sect. 2.3).

Table 1. The complexity and implementation dependency of cube attacks. The complexities are evaluated with the number of instructions.

Cipher	#Rounds	Cube Attack	Exhaustive Search†	Implement. Dependency‡	Source
Trivium	843	$2^{82.99}$	2^{81}	✓	[9]
	844	$2^{82.84}$	2^{81}	✓	[9]
	845	$2^{84.92}$	2^{81}	✓	[9]
	846	$2^{84.58}$	2^{81}	✓	[7]
	847	$2^{84.58}$	2^{81}	✓	[7]
	848	$2^{84.58}$	2^{81}	✓	[7]
Grain-128AEAD	191	$2^{131.55}$	2^{129}	✓	[9]
	192	$2^{133.17}$	2^{129}	✓	[7]
Acorn	776	$2^{128.58}$	2^{129}	×	[7]
Kreyvium	894	2^{128}	2^{129}	×	[9]
	895	$2^{133.17}$	2^{129}	✓	[7]
	898	$\mathbf{2^{128.58}}$	$\mathbf{2^{129}}$	×	**Sect. 3**
	899	$\mathbf{2^{128.58}}$	$\mathbf{2^{129}}$	×	**Sect. 3**
	900	$\mathbf{2^{128.58}}$	$\mathbf{2^{129}}$	×	**Sect. 3**

† One query of the cipher considered is a table lookup and equals two instructions, then the brute force attack of the cipher needs $2^{\kappa+1}$ instructions where κ is the key size.
‡ ✓ denotes *implementation dependent* and × denotes *implementation independent*. The details of our analysis can be found in Sect. 5.

We first define some notations used in the remainder of this paper. We consider the stream ciphers with n-bit secret key $\boldsymbol{x} = (x_0, \ldots, x_{n-1})$ and m-bit public IV $\boldsymbol{v} = (v_0, \ldots, v_{m-1})$. For arbitrary positive integer $t > 1$, we denote the set of integers $\{0, \ldots, t-1\}$ as $[0, t)$ hereafter.

2.1 The Main Procedures of Cube Attacks

In cube attacks, the adversary is faced with an encryption oracle of the targeted stream cipher, denoted as E. The adversary can query E with a public IV vector $\boldsymbol{v}$ and acquire the key stream bits corresponding to $\boldsymbol{v}$ and an embedded secret key $\boldsymbol{x}_e$, denoted as $\boldsymbol{z}(\boldsymbol{v}, \boldsymbol{x}_e) = E(\boldsymbol{v})$. When queried with a key-IV pair $(\boldsymbol{x}, \boldsymbol{v})$, the oracle E outputs the corresponding key stream bits $\boldsymbol{z}\ (\boldsymbol{v}, \boldsymbol{x}) = E(\boldsymbol{v}, \boldsymbol{x})$. The target for the adversary is to retrieve the embedded key $\boldsymbol{x}_e$ within feasible complexity limits. The procedures of cube attacks for recovering $\boldsymbol{x}_e$ can be summarized as follows:

1. `Superpoly Recovery.` Recover the ANF of the superpoly $p_I(\boldsymbol{x}, \boldsymbol{IV})$ where IV is a known constant and p_I can be a simple and low-degree polynomial related to key bits $\boldsymbol{x}[J]$ where $J \subseteq [0, n-1]$. Such a superpoly can be recovered with division property based techniques that we will detail in.
2. `Cube Sum Computation.` For all $2^{|I|}$ $\boldsymbol{v} \in C_I(\boldsymbol{IV})$, query $E(\boldsymbol{v})$ and sum the output keystream bits for the exact value of superpoly $p_I(\boldsymbol{IV}, \boldsymbol{x}_e) = \theta$.

3. `Key Filtering.` For involved candidate bits: construct lookup tables for identifying the correct key candidate $\boldsymbol{x}_c$'s, s.t. $p_I(\boldsymbol{IV}, \boldsymbol{x}_c) = \theta$.
4. `Exhaustive Search.` find the only correct key $\boldsymbol{x}_e$ from the remaining keys.

2.2 Division Property based Superpoly Recoveries

In the view of Boolean function for describing division property, the monomial prediction technique is developed to evaluate the degree of Boolean functions and is soon applied to recovery target polynomials, mainly for the polynomials after many rounds of iteration in stream or block ciphers. Hu et al. proposed the monomial prediction technique in [10], then developed it to the Nested Framework, which was used to recover the exact ANFs of massive superpolies [9].

(Bit-Based) Division Property. Before giving a brief introduction to division property, we need some notations for bit-vectors. For any bitvector $\boldsymbol{x} \in \mathbb{F}_2^m$, $x[i]$ denotes the ith bit of $\boldsymbol{x}$ where $i \in \{0, 1, \ldots, m-1\}$. Given two bitvectors $\boldsymbol{x} \in \mathbb{F}_2^m$ and $\boldsymbol{u} \in \mathbb{F}_2^m$, $\pi_{\boldsymbol{u}}(\boldsymbol{x}) = \boldsymbol{x}^{\boldsymbol{u}} = \prod_{i=0}^{m-1} x[i]^{u[i]}$. Moreover, $\boldsymbol{x} \succeq \boldsymbol{u}$ denotes $x[i] \geq u[i]$ for all $i \in \{0, 1, \ldots, m-1\}$; otherwise we denote $\boldsymbol{x} \not\succeq \boldsymbol{u}$.

The (conventional) division property, a.k.a two-subset division property, was proposed at Eurocrypt 2015 [17], and it is regarded as the generalization of the integral property.

Definition 1 (Two-subset division property). *Let $\mathbb{X}$ be a multiset whose elements take a value of $\mathbb{F}_2^m$, and $\mathbb{K} = \{\boldsymbol{k} \mid \boldsymbol{k} \in \mathbb{F}_2^m\}$ be a set of m-dimension bit vectors. When the multiset $\mathbb{X}$ has the division property $\mathcal{D}_{\mathbb{K}}^{1^m}$, it fulfills the following conditions:*

$$\bigoplus_{x \in \mathbb{X}} \boldsymbol{x}^{\boldsymbol{u}} = \begin{cases} \text{unknown} & \text{if there are } \boldsymbol{k} \in \mathbb{K} \text{ s.t. } \boldsymbol{u} \succeq \boldsymbol{k}, \\ 0 & \text{otherwise.} \end{cases}$$

To improve the accuracy of the division property propagation, the three-subset division property was proposed in [19], where the number of divided subsets is extended from two to three.

Definition 2 (Three-subset division property). *Let $\mathbb{X}$ be a multiset whose elements take a value of $\mathbb{F}_2^m$, and $\mathbb{K} = \{\boldsymbol{k} \mid \boldsymbol{k} \in \mathbb{F}_2^m\}$ and $\mathbb{L} = \{\boldsymbol{\ell} \mid \boldsymbol{\ell} \in \mathbb{F}_2^m\}$ be two sets of m-dimension bit vectors. Define $\boldsymbol{x}^{\boldsymbol{u}} := \prod_{i=0}^{m-1} x_i^{u_i}$, $\boldsymbol{u} \in \mathbb{F}_2^m$. When the multiset $\mathbb{X}$ has the three-subset division property $\mathcal{D}_{\mathbb{K},\mathbb{L}}^{1^m}$, it fulfills the following conditions:*

$$\bigoplus_{\boldsymbol{x} \in \mathbb{X}} \boldsymbol{x}^{\boldsymbol{u}} = \begin{cases} \text{unknown} & \text{if there are } \boldsymbol{k} \in \mathbb{K} \text{ s.t. } \boldsymbol{u} \succeq \boldsymbol{k}, \\ 1 & \text{else if there is } \boldsymbol{\ell} \in \mathbb{L} \text{ s.t. } \boldsymbol{u} = \boldsymbol{\ell}, \\ 0 & \text{otherwise.} \end{cases}$$

Xiang et al. introduced MILP-based method to automatically search integral distinguishers (based on two-subset division property) for several block ciphers [22]. They modeled the propagation rules of basic operations such as COPY, AND,

and XOR by MILP. Later the MILP division property method was further applied to cube attacks on stream ciphers [18,20]. For the three-subset division property and the variant without unknown (removing the unknown set $\mathbb{K}$ from the Definition 2 for make cube attacks based on three-subset division property infeasible and/or practical), the detailed propagation rules and the MILP modelings can be found in [21,6].

Monomial Prediction. The monomial prediction technique [10] can be used to determine that the coefficient of an involved monomial is 0 or 1 in the ANF of a given Boolean function, which can be applied to the construction of SAT models for block ciphers taking the key schedule into consideration in order to find refined integral distinguishers [5], or to recover the ANF of the superpoly of the cube attacks. In this paper, we focus on the latter application.

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

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

where $a_{\boldsymbol{u}} \in \mathbb{F}_2$, $\pi_{\boldsymbol{u}}(\boldsymbol{x}) = \prod_{i=0}^{n-1} x_i^{u_i}$ is defined as before and is a monomial.

Let $\boldsymbol{f} : \mathbb{F}_2^n \to \mathbb{F}_2^m$ be a vectorial Boolean function with $\boldsymbol{y} = (y_0, y_1, \ldots, y_{m-1}) = \boldsymbol{f}(\boldsymbol{x}) = (f_0(\boldsymbol{x}), f_1(\boldsymbol{x}), \ldots, f_{m-1}(\boldsymbol{x}))$, where $f_i : \mathbb{F}_2^n \to \mathbb{F}_2$ is a Boolean function. For $\boldsymbol{u} \in \mathbb{F}_2^n$ and $\boldsymbol{v} \in \mathbb{F}_2^m$, we use $\boldsymbol{x}^{\boldsymbol{u}} \to \boldsymbol{y}^{\boldsymbol{v}}$ to denote that monomial $\boldsymbol{x}^{\boldsymbol{u}}$ appears in $\boldsymbol{y}^{\boldsymbol{v}}$.

We are interested in the following case: Let $\boldsymbol{f}$ be a composition of a sequence of r vectorial Boolean functions

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

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 $\boldsymbol{f}^{(i)}$. We are interested in whether a monomial of $\boldsymbol{x}^{(0)}$, say $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)})$, appears in one monomial of $\boldsymbol{x}^{(r)}$, i.e., $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \to \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$. To make it happen, for one monomial in $\pi_{\boldsymbol{u}^{(i)}}(\boldsymbol{x}^{(i)})$, there must exist at least one monomial in $\pi_{\boldsymbol{u}^{(i+1)}}(\boldsymbol{x}^{(i+1)})$, i.e., for every $0 \le i \le r-1$, a transition $\pi_{\boldsymbol{u}^{(i)}}(\boldsymbol{x}^{(i)}) \to \pi_{\boldsymbol{u}^{(i+1)}}(\boldsymbol{x}^{(i+1)})$ must be guaranteed.

Definition 3 (Monomial Trail [10]). *Let $\boldsymbol{x}^{(i+1)} = \boldsymbol{f}^{(i)}(\boldsymbol{x}^{(i)})$ for $0 \le i \le r-1$. We call a sequence of monomials $(\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}), \pi_{\boldsymbol{u}^{(1)}}(\boldsymbol{x}^{(1)}), \ldots, \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)}))$ an r-round monomial trail connecting $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)})$ and $\pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$ under the composite function $\boldsymbol{f}(x) = \boldsymbol{f}^{(r-1)} \circ \boldsymbol{f}^{(r-2)} \circ \cdots \circ \boldsymbol{f}^{(0)}$ if there exist*

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

If there exist at least one monomial trail connecting $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)})$ and $\pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$, we write $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \rightsquigarrow \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$. Otherwise, $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \not\rightsquigarrow \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$.

We describe the following theorem that is integrated from [6,8,10].

Theorem 1. *Let $\boldsymbol{f} = \boldsymbol{f}^{(r-1)} \circ \boldsymbol{f}^{(r-2)} \circ \cdots \circ \boldsymbol{f}^{(0)}$ defined as above. Denote all the trails from $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)})$ to $\pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$ by $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \bowtie \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$. Then $\pi_{\boldsymbol{u}^{(0)}}(\boldsymbol{x}^{(0)}) \rightsquigarrow \pi_{\boldsymbol{u}^{(r)}}(\boldsymbol{x}^{(r)})$ if and only if*

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

Degree Evaluation for Superpoly. The technique of the superpoly degree evaluation for cube attacks was proposed in [20], to avoid constructing the whole truth table of the superpoly using cube summations which can eventually reduce the entire complexity of cube attacks.

Based on the MILP-aided two-subset division property, the upper bound for the algebraic degree, denoted as d, of the superpoly can be derived. With the knowledge of its degree d (and J as the set of the indices of key involved), the superpoly can be completely represented with its $\binom{|J|}{\leq d}$ coefficients rather than the whole truth table, where $\binom{|J|}{\leq d} := \sum_{i=0}^{d} \binom{|J|}{i}$. If $d < |J|$, which is true for lightweight ciphers because the algebraic degrees of their round functions are usually quite low, the coefficients of the monomials with degrees higher than d are constantly 0. Thus, the complexity of superpoly recovery can be reduced from $2^{|I|+|J|}$ to $2^{|I|} \times \binom{|J|}{\leq d}$. Therefore, the degrees d are often much smaller than $|J|$, especially when high-dimensional cubes are used.

Although such degree d's are only upper bounds for the degree of superpolies, the superpolies with a lower degree are more likely to be simpler than those with higher ones. This technique is used to help us heuristically filter out superpolies with a higher degree which potentially lead to massive superpolies. The effectiveness of using this technique is verified by simple superpolies we finally obtained for more rounds of Kreyvium in Sect. 3.

Divide-and-Conquer Strategy for Recovering ANFs of Superpolies. As the number of rounds evaluated grows, the superpolies for certain cubes become increasingly complex. Many existing methods for superpoly recovery quickly hit their bottlenecks. Motivated by this, Hu et al. [9] proposed a framework with *nested monomial predictions* that scales well for massive superpoly recovery. The nested method actually is a hybrid of four popular methods in this area, namely Wang et al.'s pruning method [21], Ye and Tian's algebraic method [24], Tian's recursively-expressing method [23] and Hao et al.'s PoolSearchMode method [6]. Later, He et al. [7] improved the nested monomial prediction framework to further simplify the MILP model and speed up the model solving. Sun [15] also used a similar technique to handle the heavy search in the superpoly recovery. In this paper, we do not go deep into their respective details and uniformly called them the divide-and-conquer techniques, and we briefly describe the main idea of the strategy they follow.

In this kind of frameworks, 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 that cannot be resolved 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.

2.3 Table Lookup based Key Filtering Techniques

In order to identify the $\boldsymbol{x}_c$'s satisfying $p_I(\boldsymbol{x}, \boldsymbol{IV}) = \theta$, one has to refer to the truth table of p_I, denoted as T_I. Such T_I is of size $2^{|J|}$. It also takes $2^{|J|}$ table lookups so as to identify the $\boldsymbol{x}[J]$ candidates. However, for the massive superpolies with $J = [0, n)$, tricks can be played to avoid storing and traversing the whole T_I. In [9], *the disjoint set* was used to decompose the whole superpoly into several sub-superpolies, thus the task of constructing a huge truth table for a massive superpoly can be divided into several smaller scale tasks of constructing smaller truth tables, which reduces the entirety of the complexity. We recall the idea of superpoly recovery using a disjoint set briefly in the following.

Disjoint Set based Key Filtering. Given a polynomial $p(\boldsymbol{x})$ with n variables, if for $0 \leq i \neq j < n$, x_i and x_j are never multiplied mutually in all monomials of $p(\boldsymbol{x})$, then we say x_i and x_j are disjoint. For a subset of variables $D \subseteq \{x_0, x_1, \cdots, x_{n-1}\}$, if every pair of variables like $(x_i, x_j) \in D$ are disjoint, we call D a disjoint set. Given the disjoint set $D = \{x_0, x_1, \ldots, x_{\ell-1}\}$, denote the set of the rest of the key variables not in D as $\overline{D} = \{x_0, x_1, \ldots, x_{n-1}\} \backslash D$, the superpoly can be re-written as a linear combination:

$$p(\boldsymbol{x}) = x_0 \cdot p_0(\overline{D}) + x_1 \cdot p_1(\overline{D}) + \cdots + x_{\ell-1} \cdot p_{\ell-1}(\overline{D}) + p_\ell(\overline{D}) \,,$$

where $p(\overline{D})$ is a polynomial of the variables only in $\overline{D}$, which is usually simplier than $p(\boldsymbol{x})$. By this the huge truth table of $p(\boldsymbol{x})$ can be replaced by smaller sub-tables corresponding to $p_0(\overline{D}), p_1(\overline{D}), \ldots, p_{\ell-1}(\overline{D})$ and the residue $p_\ell(\overline{D})$. In the key filtering phase, the bits in disjoint set are guessed and refer to the corresponding sub-tables sequentially.

The key filtering procedures based on a single superpoly can easily be extended to multiple superpolies. In addition to the key filtering method based on the disjoint set, improvement was further proposed in [7] for key filtering: they choose to guess some key bits for simplifying the massive superpoly and construct truth tables on the fly for filtering keys.

Note that all truth tables are constructed using the Möbius transformation technique in [9]: for a Boolean function with n variables, the Möbius transformation algorithm can be used to construct its truth table with $n \cdot 2^{n-1}$ XOR operations.

3 New Attacks on Kreyvium

So far, cube attacks on stream ciphers are conducted the following two main strategies:

- The massive superpoly strategy uses low dimensional cubes but the superpolies are usually complicated;
- On the contrary, the conventional strategy turns to using high dimensional cubes so as to acquire low-degree superpolies related to very few key bits.

In the high-level view, our new cube attacks on 898-, 899- and 900-round KREYVIUM follow the conventional strategy. The reason we choose this strategy will be given in Sect. 4.1. In the low-level view, we propose our own specific procedures for constructing the cubes utilized in our cube attacks on KREYVIUM. We summarize them in the following:

- Since the key and IV of KREYVIUM share the same length of 128 bits, we decide to use the largest possible dimension of cubes as $|I| = m - 2 = 126$.
- The cube indices are selected so as to result in lower superpoly degrees which are evaluated naturally with the two-subset division property based degree evaluation technique [20].
- After finding cubes with low-degree superpolies, the superpolies recovery can be accomplished directly with the methods in [9,7].

In the following, we give the 898-, 899- and 900-round cube attacks on KREYVIUM, with the corresponding balanced superpolies. So far as we know, these are the best cube attacks on KREYVIUM.

3.1 New Results for 898-Round KREYVIUM

For 898-round KREYVIUM, there are plenty of 126-dimensional cubes with simple superpolies so we randomly pick several 126-dimensional cubes, run degree evaluation procedures and select the cubes with the lowest degrees. After examining several trials, we find two cubes, denoted as I_0 and I_1 respectively, with degree evaluations 2 and 3. The cube I_0 is defined as $I_0 = [0, 127]\backslash\{5, 56\}$, the superpoly $p_{I_0}(\boldsymbol{x}, \mathbf{0})$ of 898-round KREYVIUM is determined as the follows

$$p_{I_0}(\boldsymbol{x}, \mathbf{0}) = x_{11} + x_{13} + x_{28} + x_{37} + x_{38} + x_{39} + x_{53} + x_{53}x_{54} + x_{55} + x_{62}x_{63} + x_{70} + x_{72} + x_{87} + x_{97} + x_{98} + x_{112} + x_{54}x_{112} + x_{113} + x_{53}x_{113} + x_{112}x_{113} + x_{114} + x_{123}.$$

The definition of I_1 is $I_1 = [0, 127]\backslash\{38, 86\}$ and the superpoly $p_{I_1}(\boldsymbol{x}, \mathbf{0})$ is derived as Eq. (7) in App. A.2.

3.2 New Results for 899-Round KREYVIUM

Following the procedure for the 898-round case, we still hope to find a cube of dimension 126 whose superpoly has a considerably lower degree, for instance, 2 or 3. However, when we ran similar procedures directly for 899-round Kreyvium, we found that low-degree superpolies became quite rare given 126-dimensional cubes.Instead of constructing a 126-dimensional cube directly, we have to exploit new methods.

First, wes run degree evaluation procedure for all 127-dimensional cubes so as to find good indices for further exclusions. To be more specific, for all the 128 cubes $I_\lambda = [0, 127]\backslash\{\lambda\}$ with $\lambda = 0, \ldots, 127$, we acquire the degree upper bounds of their corresponding superpolies, denoted as $\deg(p_{I_\lambda})$, using the degree evaluation based on the conventional division property in [20]. We find that only 13 λ's satisfy $\deg(p_{I_\lambda}) \leq 3$ and we store such 13 λ's in the set Λ below:

$$\Lambda = \{\lambda \in [0, 127] : \deg(p_{I_\lambda}) \leq 3\} = \{28, 29, 41, 47, 48, 49, 52, 55, 60, 61, 70, 74, 75, 79\}.$$

Details of the 128 $\deg(p_{I_\lambda})$'s can be seen in Table 2 of App. A.1.

Next, we further construct the 126-dimensional cube $I = [0, 127]\backslash\{29, 47\}$ and the degree evaluation gives $\deg(p_I) = 3$. Therefore, we are able to recover p_I using the method of [9]. The ANF of $p_I(\boldsymbol{x}, \mathbf{0})$ is as follows:

$$p_I(\boldsymbol{x}, \mathbf{0}) = x_2 + x_3 + x_8 + x_{10} + x_{11} + x_{10}x_{11} + x_{15} + x_{18} + x_{19} + x_{20} + x_6x_{20} + x_{21} + x_{24} + x_{28} + x_{29} + x_6x_{30} + x_{28}x_{34} + x_{20}x_{37} + x_{30}x_{37} + x_{34}x_{37} + x_{24}x_{38} + x_{39} + x_{20}x_{40} + x_{30}x_{40} + x_{41} + x_{28}x_{44} + x_{37}x_{44} + x_{45} + x_{51} + x_{52} + x_{39}x_{52} + x_{51}x_{52} + x_{34}x_{53} + x_{44}x_{53} + x_{34}x_{54} + x_{38}x_{54} + x_{44}x_{54} + x_{52}x_{54} + x_{34}x_{53}x_{54} + x_{44}x_{53}x_{54} + x_{34}x_{55} + x_{44}x_{55} + x_{20}x_{56} + x_{30}x_{56} + x_{62} + x_{54}x_{62} + x_{61}x_{62} + x_{63} + x_{34}x_{62}x_{63} + x_{44}x_{62}x_{63} + x_{34}x_{64} + x_{44}x_{64} + x_{63}x_{64} + x_{24}x_{63}x_{64} + x_{20}x_{65} + x_{24}x_{65} + x_{30}x_{65} + x_{20}x_{66} + x_{30}x_{66} + x_{66}x_{67} + x_{68} + x_{71} + x_{70}x_{71} + x_{72} + x_{74} + x_{77} + x_{78} + x_{77}x_{78} + x_{39}x_{77}x_{78} + x_{39}x_{79} + x_{80} + x_{79}x_{80} + x_{38}x_{79}x_{80} + x_{52}x_{79}x_{80} + x_{62}x_{79}x_{80} + x_{81} + x_{38}x_{81} + x_{52}x_{81} + x_{62}x_{81} + x_{83} + x_{38}x_{83} + x_{63}x_{64}x_{83} + x_{65}x_{83} + x_{86} + x_{87} + x_{34}x_{87} + x_{44}x_{87} + x_{88} + x_{87}x_{88} + x_{89} + x_6x_{89} + x_{37}x_{89} + x_{40}x_{89} + x_{56}x_{89} + x_{65}x_{89} + x_{66}x_{89} + x_{90}x_{91} + x_{92} + x_{95} + x_{20}x_{96} + x_{30}x_{96} + x_{89}x_{96} + x_{97} + x_{54}x_{97} + x_{79}x_{80}x_{97} + x_{81}x_{97} + x_{98} + x_{52}x_{98} + x_{77}x_{78}x_{98} + x_{79}x_{98} + x_{20}x_{99} + x_{30}x_{99} + x_{89}x_{99} + x_{28}x_{103} + x_{37}x_{103} + x_{53}x_{103} + x_{54}x_{103} + x_{53}x_{54}x_{103} + x_{55}x_{103} + x_{62}x_{63}x_{103} + x_{64}x_{103} + x_{87}x_{103} + x_{111} + x_{53}x_{111} + x_{112} + x_{34}x_{112} + x_{44}x_{112} + x_{52}x_{112} + x_{34}x_{54}x_{112} + x_{44}x_{54}x_{112} + x_{103}x_{112} + x_{54}x_{103}x_{112} + x_{111}x_{112} + x_{34}x_{113} + x_{44}x_{113} + x_{34}x_{53}x_{113} + x_{44}x_{53}x_{113} + x_{103}x_{113} + x_{53}x_{103}x_{113} + x_{34}x_{112}x_{113} + x_{44}x_{112}x_{113} + x_{103}x_{112}x_{113} + x_{34}x_{114} + x_{44}x_{114} + x_{103}x_{114} + x_{120} + x_{121} + x_{54}x_{121} + x_{79}x_{80}x_{121} + x_{81}x_{121} + x_{20}x_{124} + x_{30}x_{124} + x_{89}x_{124} + x_{20}x_{125} + x_{30}x_{125} + x_{89}x_{125}.$$

3.3 New Results for 900-Round KREYVIUM

As for 900-round KREYVIUM, the cube construction follows the same steps as 899-round in Sect. 3.2. The superpoly recovery is accomplished using the method in [7]. We take $I = [0, 127]\backslash\{38, 86\}$ as the cube for 900-round KREYVIUM, and the superpoly $p_I(\boldsymbol{x}, \mathbf{0})$ is given in App. A.2.

4 Implementation Dependency

The stream cipher E in Sect. 2.1 can be implemented in many different ways. In codebook implementations, E is simply a lookup table storing all key-IV pairs $(\boldsymbol{x}, \boldsymbol{v})$'s along with the corresponding keystream $\boldsymbol{z}(\boldsymbol{x}, \boldsymbol{v})$ values. In round-wise implementations, E is simply executed by sequential assembly instructions describing the round functions of stream ciphers. In this case, E is implemented round by round so a query of E seems more complicated than a table lookup. However, for E's implemented in a codebook manner, a query of E is simply a table lookup. Therefore, for cube attacks, we propose the concept of implementation dependency revealing whether it can be feasible for both round-wise and codebook implementation oracles.

Implementation dependent cube attacks. When the number of table lookups in the key filtering phase approaches the exhaustive search complexity, the cube attack may become infeasible for codebook-implemented oracles. We refer to the cube attacks that only work for round-wise implementations as implementation dependent cube attacks.

Implementation independent cube attacks. On the contrary, those cube attacks work for both round-wise and codebook implementation attacks are therefore called the implementation independent cube attacks.

Consider a cube attack using ℓ cubes $I_0, \ldots, I_{\ell-1}$ with superpolies correlated to key bits $J_0, \ldots, J_{\ell-1}$. The cube attack in Sect. 2.1 requires $2^{|I_0|} + \ldots + 2^{|I_{\ell-1}|}$ oracle queries for `Cube Sum Computation` procedure, at least $2^{|J_0|} + \ldots + 2^{|J_\ell - 1|}$ table lookups in `Key Filtering` and another $2^{n-\ell}$ oracle queries for the last `Exhaustive Search` procedure.

In fact, a table lookup takes two assembly instructions: one addition and one comparison. We further assume that the implementation of querying E takes α instructions. Besides, there may also involve basic operations such as XOR, for constructing the lookup tables used in the `Key Filtering` phase and the number of instructions for the table construction is denoted as β. Therefore, the complexity of the cube attack in Sect. 2.1 has now become:

$$C_{\text{new}} = \sum_{j=0}^{\ell-1} 2^{|I_j|} + 2^{n-\ell} + \frac{2}{\alpha} \sum_{j=0}^{\ell-1} 2^{|J_j|} + \frac{\beta}{\alpha} \tag{1}$$

The attack can only work when $C_{\text{new}} < 2^n$.

4.1 An Implementation Dependency Analysis Example

According to the concepts of implementation dependency, we find that the cube attack on 845-round Trivium given in [9] is *implementation dependent*. We detail such an implementation dependency analysis here as an example and leave the same analysis of other cube attack results in Sect. 5.1.

Trivium [1] is a hardware oriented stream cipher. It has been selected as part of the eSTREAM portfolio [3] and specified as an International Standard under ISO/IEC 29192-3 [11]. Then key and IV of Trivium are both of 80 bits. Both key and IV are first loaded in a 288-bit internal state and run 1152-round initialization afterwards. The whole initialization process can be summarized as

follows:

$$
\begin{aligned}
&(s_0, s_1, \ldots, s_{92}) \leftarrow (K_0, K_1, \ldots, K_{79}, 0, \ldots, 0)\\
&(s_{93}, s_{95}, \ldots, s_{177}) \leftarrow (IV_0, IV_1, \ldots, IV_{79}, 0, \ldots, 0)\\
&(s_{177}, s_{179}, \ldots, s_{287}) \leftarrow (0, \ldots, 0, 1, 1, 1)\\
&\textbf{for}\ \ i = 0 \text{ to } 1151\ \ \textbf{do}\\
&\quad t_1 \leftarrow s_{65} \oplus s_{90} \cdot s_{91} \oplus s_{92} \oplus s_{170} \qquad (2)\\
&\quad t_2 \leftarrow s_{161} \oplus s_{174} \cdot s_{175} \oplus s_{176} \oplus s_{263} \qquad (3)\\
&\quad t_3 \leftarrow s_{242} \oplus s_{285} \cdot s_{286} \oplus s_{287} \oplus s_{68} \qquad (4)\\
&\quad (s_0, s_1, \ldots, s_{92}) \leftarrow (t_3, s_0, s_1, \ldots, s_{91})\\
&\quad (s_{93}, s_{95}, \ldots, s_{177} \leftarrow (t_1, s_{93}, s_{94}, \ldots, s_{175})\\
&\quad (s_{177}, s_{179}, \ldots, s_{287}) \leftarrow (t_2, s_{177}, s_{178} \ldots, s_{286})\\
&\textbf{end for}
\end{aligned}
$$

After the initialization phase, one key stream bit is generated by

$$z = s_{65} \oplus s_{92} \oplus s_{161} \oplus s_{176} \oplus s_{242} \oplus s_{287}. \qquad (5)$$

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.

If implementing Trivium bit-wisely, we can get quickly from Eqs. (2) to (5) that each round of Trivium (one initialization and one keystream bit generation) requires 14 XORs, 3 ANDs and 288 rotates instructions, thus a total of 305 instructions. In fact, using parallel computing in a hardware environment could give 64 times speed up for the iterations, which leads to about just 4.8 instructions for each round. So considering a codebook-implemented oracle, one query of the Trivium is just a table lookup.

In [9], Hu et al. found two cubes I_2 and I_3 (notations exactly follow [9]) which have the same disjoint set $D = \{k_1, k_{10}\}$, then they used 2 corresponding equations for the key recovery procedure of 845-round Trivium, where $|I_2| = 55$ and $|I_3| = 54$. Obviously, we can filter keys by half for the remaining keys once we get another equation (if the equation is balanced). So three quarters of keys will be filtered by two equations. For the 845-round attack on Trivium, the two equations are:

$$\begin{cases} p^{(2)} = k_1 \cdot p_0^{(2)} \oplus k_{10} \cdot p_1^{(2)} \oplus p_2^{(2)} \\ p^{(3)} = k_1 \cdot p_0^{(3)} \oplus k_{10} \cdot p_1^{(3)} \oplus p_2^{(3)} \end{cases} \qquad (6)$$

There are 6 truth tables in the equations and recovering them needs using Möbius transformation technique. There are 3 tables for $p^{(2)}$ and 3 tables for $p^{(3)}$. Let T_1, T_2, T_0 are truth tables for $p_1^{(2)}$, $p_2^{(2)}$, $p_0^{(2)}$, the size of them are 2^{78}, 2^{77} and 2^{78}, and the probability of $p^{(2)}$ being balanced is 0.5. And there are four tables of 2^{78} size and 2 of 2^{77} size in the 6 tables. In the table constructing phase, the number of XORs is $4 \cdot 78 \cdot 2^{76} + 2 \cdot 77 \cdot 2^{75} \approx 1.51 \cdot 2^{84}$.

In the `Cube Sum Computation` phase, totally $2^{55} + 2^{54} = 1.5 \cdot 2^{55}$ queries and XORs are used to get $p^{(2)} = \theta_2$ and $p^{(3)} = \theta_3$, and this complexity could be

ignored comparing with the `Key Filtering` and the `Exhaustive Search` phase. After we get the values of $p^{(2)}$ and $p^{(3)}$ by doing the `Cube Sum Computation` in the online phase, we can filter and search 2^{80} keys with the given equations. We simply consider the complexity of lookup for one equation (that is to say just use one single cube) first and then extend to that of two equations. There are totally four cases for the table lookup: $(k_1, k_{10}) = (0,0), (0,1), (1,0), (1,1)$. Consider three kinds of operations: lookup, XOR and judgement. Let k be a key which is filtered, and $k = *k_1 * * * k_{10} * **$, where $*$ represents a bit not belonging to the disjoint set. That is to say, we should give three lookups each for T_0, T_1 and T_2. Firstly, consider the first equation in Eq. (6). For the four (k_1, k_{10}) situations in , we should compute the results and compare them with $p^{(2)}$, and if one key leads to $p^{(2)} = \theta_2 \oplus 1$, there is no need to do anything for the other equation with this key.

So the number of instructions of the first equation consists of three parts according to three kind of operations we considered:

1. Table lookups for T_0, T_1 and T_2. Totally $2^{78} + 2^{77} + 2^{78} = 1.25 \cdot 2^{79}$ lookups, and $1.25 \cdot 2^{80}$ instructions.
2. XORs in equation evaluation calculation. The four (k_1, k_{10}) cases need 0, 1, 1 and 2 XORs respectively. In total, there are $2^{78} + 2^{78} + 2 \cdot 2^{78} = 2^{80}$instructions.
3. Judgements. After calculating $k_1 \cdot p_0^{(2)} \oplus k_{10} \cdot p_1^{(2)} \oplus p_2^{(2)}$, give a judgement to check if it equals to $p^{(2)}$ so as to filter keys. There are totally 2^{80} judgements needed.

Now consider the situation for other equations. We need only to process the keys that are filtered by the first equation, about 2^{79} keys. This leads to a half number of XORs and half number of the judgements but full table lookups for the second equation. Totally it costs about $1.25 \cdot 2^{80}$ instructions for lookups, 2^{79} instructions for XORs and 2^{79} instructions for judgements.

After key filtering, we should exhaustively search the remaining $(1/4) \cdot 2^{80}$ keys, which needs $(1/4) \cdot 2^{80}$ oracle queries, scilicet, 2^{79} instructions.

Totally, the process of key filtering and searching uses $1.5 \cdot 2^{82}$ instructions and the full attack uses $1.89 \cdot 2^{84} = 2^{84.92}$ instructions. Instead, using brute force attack for the cipher needs 2^{80} oracle queries. One query of the Trivium is a table lookup and equals 2 instructions, then the full brute force attack of Trivium needs 2^{81} instructions. It means cube attack does not work for codebook implementation Trivium over 845-round. We also find some other cases to illustrate the universality of this phenomenon, and we put them in Sect. 5.1.

5 Further Analysis for Cube Attacks

Similar to the analysis in Sect. 4.1, we further scrutinize existing cube attack results of stream ciphers Trivium, Grain-128AEAD, Acorn and Kreyvium in Sect. 5.1to see whether they are *implementation independent*. We also discuss if

using multiple cubes (superpolies) can reduce the complexity of the key recovery in Sect. 5.2. For the convenience of the explanation, we give a brief introduction to TRIVIUM in Sect. 4.1. However, due to page limits, we refer the specifications for KREYVIUM, Grain-128AEAD and ACORN to the respective design papers.

5.1 Analysis for More Cases of Cube Attacks

We recall the corresponding relationship between operations and the number of instructions. All operations considered are: oracle query, table lookup, XOR, and judgment.

As has been explicit in Sect. 2.1, there are four procedures in cube attacks, namely `Superpoly Recovery`, `Cube Sum Computation`, `Key Filtering` and `Exhaustive Search`. Now that we consider the complexity between oracle and key filtering, which is important to justify what steps should be considered.

Oracle implementation is querying a stream cipher through a table lookup. One query for one key so that total 2^L queries for a cipher with an L-bit length key. One oracle query equals m-instructions so there are total $m \cdot 2^L$ instructions for the whole search. And in this section we let one oracle query equal to one table lookup, then we get $m = 2$. So in our analysis, we keep using: 1 oracle query $\approx$ 1 table lookup $\approx$ 2 instructions.

- The step `Superpoly Recovery` uses the nest framework [9] in the offline phase so its complexity is not involved.
- The step `Cube Sum Computation` needs to query the oracle $2^{|I|}$ times where $|I|$ is the number of indices of the cube. For someone who has much smaller cube sizes than L , the time cost for cube sum can be ignored. However, for those using heavy cubes , the time cost should be considered.
- The step `Key Filtering` uses superpoly and its value to eliminate wrong keys. One superpoly can filter half keys of the remaining keys (in most cases the balancedness of a superpoly is 0.5).
  Calculating instructions of the table lookups is easy, while the number of XORs is calculated as follows: Suppose the superpoly can be re-written using the common disjoint set containing ℓ keys $k_0, \ldots, k_{\ell-1}$, it means 2^ℓ keys share the same table and just change the values of $k_0, \ldots, k_{\ell-1}$. Obviously, there are $\ell \cdot 2^\ell/2$ XORs for 2^ℓ keys, so on average $\ell/2$ XORs for each key.
- Finally, we should execute `Exhaustive Search` procedure for the remaining keys. For one equation situation, there are still half of the total keys.

Note that for TRIVIUM, we follow the same notations in [9]. The complexity of the steps of `Cube Sum Computation` in cube attacks for 843- and 848-round TRIVIUM is negligible due to the small size of cubes.

The attack on 843-round TRIVIUM in [9] uses three cubes I_0, I_2 and I_3 with sizes 56, 55, 54, and the corresponding superpolies p_0, p_2 and p_3 all have 5 sub truth tables(separated by their disjoint sets). The biggest truth table sizes of the three superpolies are 75, 74 and 75 respectively, so the number of instructions for table construction is $2 \cdot 75 \cdot 2^{73} + 74 \cdot 2^{72} = 1.46 \cdot 2^{80}$.

The `Key Filtering` uses three superpoly equations: the 1st equation involves $5 \cdot 2^{79}$ XORs, 2^{75} table lookups and 2^{80} judgements which is $3.5625 \cdot 2^{80}$ instructions in total; the 2nd involves $0.5 \cdot 5 \cdot 2^{79}$ XORs, $0.5 \cdot 2^{80}$ judgements and 2^{74} table lookups so there are $1.78125 \cdot 2^{80}$ instructions; the 3rd involves $0.25 \cdot 5 \cdot 2^{79}$ XORs, $0.25 \cdot 2^{80}$ judgements and 2^{75} table lookups, totally $0.9375 \cdot 2^{80}$ instructions.

The `Exhaustive Search` for the remaining $1/8 \cdot 2^{80}$ keys requires $1/8 \cdot 2^{80}$ encryption oracle queries which is equivalent to $1/4 \cdot 2^{80}$ instructions.

To sum up, the total amount of instructions for the whole attack is $(1.46 + 3.5625 + 1.78125 + 0.9375 + 0.25) \cdot 2^{80} = 2^{82.99}$ which is higher than that of the exhaustive search. So this is an *implementation dependent* result.

The attack on 844-Round TRIVIUM **in [9]** uses 2 cubes I_2 and I_3 with size 55, 54, and the superpolies p_2 and p_3 have the same disjoint set with size 6. The truth table sizes of the two superpolies are both 74 respectively, so the number of instructions for table construction is $2 \cdot 74 \cdot 2^{72} = 0.58 \cdot 2^{80}$.

The `Key Filtering` uses two superpoly equations : the 1st equation involves $6 \cdot 2^{79}$ XORs, 2^{74} table lookups and 2^{80} judgements which is $4.03 \cdot 2^{80}$ instructions in total; the 2nd involves $0.5 \cdot 6 \cdot 2^{79}$ XORs, $0.5 \cdot 2^{80}$ judgements and 2^{74} table lookups, totally $2.03 \cdot 2^{80}$ instructions.

The `Exhaustive Search` for the remaining $1/4 \cdot 2^{80}$ keys requires $1/4 \cdot 2^{80}$ encryption oracle queries which is equivalent to $1/2 \cdot 2^{80}$ instructions.

To sum up, the total amount of instructions for the whole attack is $(0.58 + 4.03 + 2.03 + 0.5) \cdot 2^{80} = 2^{82.84}$ which is higher than that of exhaustive search. So this is an *implementation dependent* result.

The attacks on 846-, 847- and 848-round TRIVIUM **in [7]** use the same cube I with size 53, and the sizes of the corresponding superpolies are all 80 so the number of instructions for table construction is $80 \cdot 2^{78} = 20 \cdot 2^{80}$.

The `Key Filtering` uses one superpoly equation: the equation involves 2^{80} table lookups and 2^{80} judgements, totally $3.5625 \cdot 2^{80}$ instructions.

The `Exhaustive Search` for the remaining $1/2 \cdot 2^{80}$ keys requires $1/2 \cdot 2^{80}$ encryption oracle queries which is equivalent to 2^{80} instructions.

To sum up, the total amount of instructions for the whole attack is $(20 + 3 + 1) \cdot 2^{80} = 2^{84.58}$ which is higher than that of the exhaustive search. So these are *implementation dependent* results.

The attacks on 191-round Grain-128AEAD in [9] uses 2 cubes I_0 and I_1 with size 96 and 95, and the corresponding superpolies p_0 and p_1 have the same disjoint set with size 12. The biggest truth table sizes of the two superpolies are both 116 respectively, so the number of instructions for table construction can be ignored.

The `Key Filtering` uses two superpoly equations: the 1st equation involves $12 \cdot 2^{127}$ XORs, $2 \cdot 2^{116} + 2^{115}$ table lookups and 2^{128} judgements which is $7 \cdot 2^{128}$ instructions in total; the 2nd involves $0.58 \cdot 12 \cdot 2^{127}$ XORs, $0.58 \cdot 2^{128}$ judgements and $2 \cdot 2^{116} + 2^{115}$ table lookups, totally $0.58 \cdot 7 \cdot 2^{128} = 4.06 \cdot 2^{128}$ instructions.

The `Exhaustive Search` for the remaining $(1 - 0.42)^2 \cdot 2^{128}$ keys requires $(1 - 0.42)^2 \cdot 2^{128}$ encryption oracle queries which is equivalent to $0.67 \cdot 2^{128}$ instructions.

To sum up, the total amount of instructions for the whole attack is $7 \cdot 2^{128} + 4.06 \cdot 2^{128} + 0.67 \cdot 2^{128} = 2^{131.55}$ which is higher than that of the exhaustive search. So this is an *implementation dependent* result.

Remark 1. The balancedness for p_0 is 0.31 and 0.30 for p_1, so using the knowledge of classical models of probability, we can filter $0.3 \cdot 0.7 + 0.7 \cdot 0.3 = 0.42$ of the total keys using one equation with the mathematic expectation (more details referring to [9]). And it means the best balancedness is 0.5 in this attack.

The attacks on 192-round Grain-128AEAD [7] uses one cube I with size 94, and the size of the corresponding superpoly is 128 so the number of instructions for table construction is $128 \cdot 2^{126} = 32 \cdot 2^{128}$.

The `Key Filtering` uses one superpoly equation: the equation involves 2^{128} table lookups and 2^{128} judgements, totally $3 \cdot 2^{128}$ instructions.

The `Exhaustive Search` for the remaining $1/2 \cdot 2^{128}$ keys requires $1/2 \cdot 2^{128}$ encryption oracle queries which is equivalent to 2^{128} instructions.

To sum up, the total amount of instructions for the whole attack is $(32+3+1) \cdot 2^{128} = 2^{133.17}$ which is higher than that of the exhaustive search. So this is an *implementation dependent* result.

The attacks on 776-round Acorn in [7] uses 2 cubes I_1 and I_2 with both size 126, so the complexity of the two cubes sum computation is $2 \cdot 2^{126}$ encryption oracle queries which is equivalent to 2^{128} instructions, and the corresponding superpolies are p_0 and p_1. The biggest truth table sizes of the two superpolies are 120 and 119, so the number of instructions for table construction could be ignored and so as the `Key Filtering`.

The `Exhaustive Search` for the remaining $(1/4) \cdot 2^{128}$ keys requires $(1/4) \cdot 2^{128}$ encryption oracle queries which is equivalent to $(1/2) \cdot 2^{128}$ instructions.

To sum up, the total amount of instructions for the whole attack is $2^{128} + (1/2) \cdot 2^{128} = 2^{128.58}$ which is lower than that of the exhaustive search. So this is an *implementation independent* result.

The attacks on 894-round Kreyvium in [9] uses 1 cube I with size 119, and the corresponding superpoly is p. The truth table size of the superpoly is 77, so the number of instructions for table construction could be ignored and so as the `Key Filtering`.

The `Exhaustive Search` for the remaining $1/2 \cdot 2^{128}$ keys requires $1/2 \cdot 2^{128}$ encryption oracle queries which is equivalent to 2^{128} instructions.

To sum up, the total amount of instructions for the whole attack is 2^{128} which is lower than that of the exhaustive search. So this is an *implementation independent* result.

The attacks on 895-round Kreyvium in [7] uses one cube I with size 120, and the corresponding superpoly p has a single truth table. The truth table size of the superpoly is 128, so the number of instructions for table construction is $128 \cdot 2^{126} = 32 \cdot 2^{128}$.

The `Key Filtering` uses one superpoly equation: the equation involves 2^{128} table lookups and 2^{128} judgements, totally $3 \cdot 2^{128}$ instructions.

The `Exhaustive Search` for the remaining $1/2 \cdot 2^{128}$ keys requires $1/2 \cdot 2^{128}$ encryption oracle queries which is equivalent to 2^{128} instructions.

To sum up, the total amount of instructions for the whole attack is the equation involves 2^{128} table lookups and 2^{128} judgements, totally $3 \cdot 2^{128}$ instructions, which is higher than that of the exhaustive search. So this is an *implementation dependent* result.

The attack on 898-round KREYVIUM **in this paper** uses two cubes I_0 and I_1 with sizes both 126, so the complexity of the two cubes sum computation requires $2 \cdot 2^{126}$ encryption oracle queries which is equivalent to 2^{128} instructions, and the corresponding superpolies p_0 and p_1 both have one truth table. The truth table sizes of the superpolies are far smaller than 128, so the number of instructions for table construction can be ignored and so as the `Key Filtering`.

The `Exhaustive Search` for the remaining $1/4 \cdot 2^{128}$ keys requires $1/4 \cdot 2^{128}$ encryption oracle queries which is equivalent to $(1/2) \cdot 2^{128}$ instructions.

To sum up, the total amount of instructions for the whole attack is $2^{128} + (1/2) \cdot 2^{128} = 1.5 \cdot 2^{128} = 2^{128.58}$ which is lower than that of the exhaustive search. So this is an *implementation independent* result.

The attack on 899-round KREYVIUM **in this paper** uses one cube I with size 126, so the complexity of the cube sum computation requires 2^{126} encryption oracle queries which is equivalent to 2^{127} instructions, and the corresponding superpoly p have one truth table. The truth table size of the superpoly is far smaller than 128, so the number of instructions for table construction can be ignored and so as the `Key Filtering`.

The `Exhaustive Search` for the remaining $1/2 \cdot 2^{128}$ keys requires $1/2 \cdot 2^{128}$ encryption oracle queries which is equivalent to 2^{128} instructions.

To sum up, the total amount of instructions for the whole attack is $(1/2) \cdot 2^{128} + 2^{128} = 1.5 \cdot 2^{128} = 2^{128.58}$ which is lower than that of the exhaustive search. So this is an *implementation independent* result.

The attack on 900-round KREYVIUM **in this paper** uses one cube I with size 126, so the complexity of the cube sum computation requires 2^{126} encryption oracle queries which is equivalent to 2^{127} instructions, and the corresponding superpoly p have one truth table. The truth table size of the superpoly is far smaller than 128, so the number of instructions for table construction can be ignored and so as the `Key Filtering`.

The `Exhaustive Search` for the remaining $1/2 \cdot 2^{128}$ keys requires $1/2 \cdot 2^{128}$ encryption oracle queries which is equivalent to 2^{128} instructions.

To sum up, the total amount of instructions for the whole attack is $(1/2) \cdot 2^{128} + 2^{128} = 1.5 \cdot 2^{128} = 2^{128.58}$ which is lower than that of the exhaustive search. So this is an *implementation independent* result.

A summary of all the analyzed results is given in Table 1 in the Introduction.

5.2 Multiple Cubes vs Single Cube

We find using multiple cubes may not result in more efficient key recoveries than its single-cube counterpart, and examples are 843, 844, 845-Round Trivium, 898-round Kreyvium and 776-round Acorn.

We find the interesting property firstly in studying key filtering. For a cipher with several cubes that can be exploited, such as N cubes, which correspond to N equations. Each equation could reduce half of the remaining keys. That means for an equation in the latter of the key filtering procedure, the cost of constructing its truth table and doing the cube summation might be unbearable. Though we handle corresponding fewer key bits for the equation in the back, we must pay for full time constructing its truth table and doing cube sum just like what we did for the first equation. And we give several examples.

The cube attack on 843-round Trivium uses 3 cubes, which means the third equation could only filter 1/8 keys from the remaining 1/4 keys but should pay the whole expenses of truth table constructions.The cost is $75 \cdot 2^{73} = 1.17 \cdot 2^{79}$ XORs while the exhaustive search for (1/8) keys costs only $(1/8) \cdot 2^{80}$ oracle queries, equals to $0.5 \cdot 2^{79}$ instructions. And this does not consider the cost of table lookups, XORs and judgments for the third equation.

843, 844 and 845-round Trivium are typical cases. And unexpectedly, even in the feasible cube attack on 898-round Kreyvium, though the total complexity is less than exhaustive search, its second equation corresponds to a cube of size 126, which queries 2^{126} times Kreyvium, and it just filters half of the rest keys, that is 2^{126} keys. And querying the Kreyvium 2^{126} times is just the same as the exhaustive search. So using multiple cubes may not surpass using a single cube. A similar situation happens to the attack on 776-round Acorn, as the cube sum invokes oracle as that for an exhaustive search.

6 Conclusions

In this paper, we focused on the real performance of cube attacks for ciphers with massive superpolies or heavy cubes. We analyzed a dozen recent cube attack results on Trivium, Kreyvium, Grain-128AEAD and Acorn, and found cube attacks are ineffective against some of them in the situation of code-book implementation. We also gave some new results on 898-, 899- and 900-round Kreyvium. In addition, we discussed the efficiency of cube attacks between multiple cubes and one single cube, and found sometimes the number of cubes used should be limited.

Acknowledgments. The authors thank all reviewers for their suggestions. This work is supported by the National Key Research and Development Program of China (Grant No. 2022YFA1004900), and by the National Natural Science Foundation of China (Grant No. 62002024, 62202062).

References

1. Cannière, C.D., Preneel, B.: Trivium. In: Robshaw, M.J.B., Billet, O. (eds.) New Stream Cipher Designs - The eSTREAM Finalists, LNCS, vol. 4986, pp. 244–266. Springer (2008). https://doi.org/10.1007/978-3-540-68351-3_18, https://doi.org/10.1007/978-3-540-68351-3_18
2. Dinur, I., Shamir, A.: Cube attacks on tweakable black box polynomials. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 278–299. Springer, Heidelberg (Apr 2009). https://doi.org/10.1007/978-3-642-01001-9_16
3. eSTREAM: the ECRYPT stream cipher project (2018). https://www.ecrypt.eu.org/stream/, accessed: 2021-03-23
4. Hadipour, H., Eichlseder, M.: Autoguess: A tool for finding guess-and-determine attacks and key bridges. In: Ateniese, G., Venturi, D. (eds.) ACNS 22. LNCS, vol. 13269, pp. 230–250. Springer, Heidelberg (Jun 2022). https://doi.org/10.1007/978-3-031-09234-3_12
5. Hadipour, H., Eichlseder, M.: Integral cryptanalysis of WARP based on monomial prediction. IACR Trans. Symmetric Cryptol. **2022**(2), 92–112 (2022). https://doi.org/10.46586/tosc.v2022.i2.92-112, https://doi.org/10.46586/tosc.v2022.i2.92-112
6. Hao, Y., Leander, G., Meier, W., Todo, Y., Wang, Q.: Modeling for three-subset division property without unknown subset - improved cube attacks against Trivium and Grain-128AEAD. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part I. LNCS, vol. 12105, pp. 466–495. Springer, Heidelberg (May 2020). https://doi.org/10.1007/978-3-030-45721-1_17
7. He, J., Hu, K., Preneel, B., Wang, M.: Stretching cube attacks: Improved methods to recover massive superpolies. In: ASIACRYPT 2022, Part IV. pp. 537–566. LNCS, Springer, Heidelberg (Dec 2022). https://doi.org/10.1007/978-3-031-22972-5_19
8. Hebborn, P., Lambin, B., Leander, G., Todo, Y.: Lower bounds on the degree of block ciphers. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020, Part I. LNCS, vol. 12491, pp. 537–566. Springer, Heidelberg (Dec 2020). https://doi.org/10.1007/978-3-030-64837-4_18
9. Hu, K., Sun, S., Todo, Y., Wang, M., Wang, Q.: Massive superpoly recovery with nested monomial predictions. In: Tibouchi, M., Wang, H. (eds.) ASIACRYPT 2021, Part I. LNCS, vol. 13090, pp. 392–421. Springer, Heidelberg (Dec 2021). https://doi.org/10.1007/978-3-030-92062-3_14
10. Hu, K., Sun, S., Wang, M., Wang, Q.: An algebraic formulation of the division property: Revisiting degree evaluations, cube attacks, and key-independent sums. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020, Part I. LNCS, vol. 12491, pp. 446–476. Springer, Heidelberg (Dec 2020). https://doi.org/10.1007/978-3-030-64837-4_15
11. ISO/IEC: 29192-3:2012: Information technology — Security techniques — Lightweight cryptography — part 3: Stream ciphers. https://www.iso.org/standard/56426.html
12. Mouha, N., Wang, Q., Gu, D., Preneel, B.: Differential and linear cryptanalysis using mixed-integer linear programming. In: Wu, C.K., Yung, M., Lin, D. (eds.) Information Security and Cryptology. pp. 57–76. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
13. Mroczkowski, P., Szmidt, J.: The cube attack on stream cipher trivium and quadraticity tests. Fundam. Informaticae **114**(3-4), 309–318 (2012). https://doi.org/10.3233/FI-2012-631, https://doi.org/10.3233/FI-2012-631

14. Sun, S., Hu, L., Wang, P., Qiao, K., Ma, X., Song, L.: Automatic security evaluation and (related-key) differential characteristic search: Application to SIMON, PRESENT, LBlock, DES(L) and other bit-oriented block ciphers. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014, Part I. LNCS, vol. 8873, pp. 158–178. Springer, Heidelberg (Dec 2014). https://doi.org/10.1007/978-3-662-45611-8_9
15. Sun, Y.: Cube attack against 843-round trivium. Cryptology ePrint Archive, Report 2021/547 (2021), https://eprint.iacr.org/2021/547
16. Todo, Y.: Integral cryptanalysis on full MISTY1. In: Gennaro, R., Robshaw, M.J.B. (eds.) CRYPTO 2015, Part I. LNCS, vol. 9215, pp. 413–432. Springer, Heidelberg (Aug 2015). https://doi.org/10.1007/978-3-662-47989-6_20
17. Todo, Y.: Structural evaluation by generalized integral property. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part I. LNCS, vol. 9056, pp. 287–314. Springer, Heidelberg (Apr 2015). https://doi.org/10.1007/978-3-662-46800-5_12
18. Todo, Y., Isobe, T., Hao, Y., Meier, W.: Cube attacks on non-blackbox polynomials based on division property. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part III. LNCS, vol. 10403, pp. 250–279. Springer, Heidelberg (Aug 2017). https://doi.org/10.1007/978-3-319-63697-9_9
19. Todo, Y., Morii, M.: Bit-based division property and application to simon family. In: Peyrin, T. (ed.) FSE 2016. LNCS, vol. 9783, pp. 357–377. Springer, Heidelberg (Mar 2016). https://doi.org/10.1007/978-3-662-52993-5_18
20. Wang, Q., Hao, Y., Todo, Y., Li, C., Isobe, T., Meier, W.: Improved division property based cube attacks exploiting algebraic properties of superpoly. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 275–305. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96884-1_10
21. Wang, S., Hu, B., Guan, J., Zhang, K., Shi, T.: MILP-aided method of searching division property using three subsets and applications. In: Galbraith, S.D., Moriai, S. (eds.) ASIACRYPT 2019, Part III. LNCS, vol. 11923, pp. 398–427. Springer, Heidelberg (Dec 2019). https://doi.org/10.1007/978-3-030-34618-8_14
22. Xiang, Z., Zhang, W., Bao, Z., Lin, D.: Applying MILP method to searching integral distinguishers based on division property for 6 lightweight block ciphers. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016, Part I. LNCS, vol. 10031, pp. 648–678. Springer, Heidelberg (Dec 2016). https://doi.org/10.1007/978-3-662-53887-6_24
23. Ye, C.D., Tian, T.: Revisit division property based cube attacks: Key-recovery or distinguishing attacks? IACR Trans. Symm. Cryptol. **2019**(3), 81–102 (2019). https://doi.org/10.13154/tosc.v2019.i3.81-102
24. Ye, C.D., Tian, T.: Algebraic method to recover superpolies in cube attacks. IET Information Security **14**(4), 430–441 (2020)

Appendix

A Details of Our Attacks on Kreyvium

A.1 Degree Evaluations of 899-Round Kreyvium

Table 2: The upper bound degree $\deg(p_{I_\lambda})$ of superpolies p_{I_λ} for 899-round KREYVIUM, with cube dimension 127.

λ	$\deg(p_{I_\lambda})$	λ	$\deg(p_{I_\lambda})$	λ	$\deg(p_{I_\lambda})$	λ	$\deg(p_{I_\lambda})$
0	5	32	6	64	4	96	4
1	6	33	5	65	5	97	5
2	6	34	4	66	6	98	4
3	4	35	5	67	4	99	5
4	4	36	4	68	4	100	4
5	5	37	5	69	4	101	5
6	5	38	5	**70**	**3**	102	5
7	4	39	4	71	4	103	6
8	7	40	4	72	4	104	6
9	6	**41**	**3**	73	4	105	5
10	4	42	4	**74**	**3**	106	5
11	5	43	4	**75**	**2**	107	6
12	4	44	4	76	4	108	4
13	5	45	5	77	5	109	4
14	4	46	4	78	5	110	4
15	5	**47**	**2**	**79**	**3**	111	4
16	5	**48**	**3**	80	4	112	4
17	5	**49**	**3**	81	4	113	6
18	5	50	4	82	6	114	6
19	5	51	4	83	7	115	6
20	5	**52**	3	84	5	116	6
21	6	53	4	85	4	117	5
22	5	54	4	86	4	118	5
23	4	**55**	**3**	87	4	119	5
24	6	56	4	88	5	120	4
25	6	57	4	89	4	121	5
26	6	58	4	90	5	122	5
27	4	59	4	91	5	123	6
28	**3**	**60**	2	92	5	124	5
29	**3**	**61**	**3**	93	4	125	4
30	5	62	4	94	6	126	4
31	4	63	5	95	6	127	4

A.2 The ANFs of Superpolies corresponding to Attacks on 898- and 900-Round KREYVIUM

For $I_1 = [0, 127]\backslash\{38, 86\}$, the superpoly $p_{I_1}(\boldsymbol{x}, \mathbf{0})$ for 898-round KREYVIUM is as Eq. (7)

$$
\begin{aligned}
p_{I_1}(\boldsymbol{x}, \mathbf{0}) &= x_{12} + x_{20} + x_{21} + x_{20}x_{21} + x_{23} + x_{31} + x_{36} + x_{11}x_{36} + x_{12}x_{36} + x_{26}x_{36} \\
&+ x_{37} + x_{11}x_{37} + x_{12}x_{37} + x_{26}x_{37} + x_{38} + x_{11}x_{38} + x_{12}x_{38} + x_{26}x_{38} + x_{36}x_{38} \\
&+ x_{37}x_{38} + x_{41} + x_{45} + x_{45}x_{46} + x_{47} + x_{46}x_{47} + x_{48} + x_{47}x_{48} + x_{49} + x_{48}x_{49} + x_{50} \\
&+ x_{11}x_{55} + x_{12}x_{55} + x_{26}x_{55} + x_{38}x_{55} + x_{56} + x_{11}x_{56} + x_{12}x_{56} + x_{26}x_{56} + x_{38}x_{56} \\
&+ x_{57} + x_{58} + x_{59} + x_{64}x_{65} + x_{66} + x_{67} + x_{66}x_{67} + x_{68} + x_{36}x_{70} + x_{37}x_{70} + x_{38}x_{70} \\
&+ x_{55}x_{70} + x_{56}x_{70} + x_{71} + x_{36}x_{71} + x_{37}x_{71} + x_{38}x_{71} + x_{55}x_{71} + x_{56}x_{71} + x_{80}x_{81} \\
&+ x_{11}x_{80}x_{81} + x_{12}x_{80}x_{81} + x_{26}x_{80}x_{81} + x_{38}x_{80}x_{81} + x_{70}x_{80}x_{81} + x_{71}x_{80}x_{81} + x_{82} \\
&+ x_{11}x_{82} + x_{12}x_{82} + x_{26}x_{82} + x_{38}x_{82} + x_{70}x_{82} + x_{71}x_{82} + x_{81}x_{82} + x_{11}x_{81}x_{82} \\
&+ x_{12}x_{81}x_{82} + x_{26}x_{81}x_{82} + x_{38}x_{81}x_{82} + x_{70}x_{81}x_{82} + x_{71}x_{81}x_{82} + x_{83} + x_{11}x_{83} \\
&+ x_{12}x_{83} + x_{26}x_{83} + x_{38}x_{83} + x_{70}x_{83} + x_{71}x_{83} + x_{83}x_{84} + x_{85} + x_{84}x_{85} + x_{87} + x_{90} \\
&+ x_{89}x_{90} + x_{91} + x_{95} + x_{11}x_{95} + x_{12}x_{95} + x_{26}x_{95} + x_{38}x_{95} + x_{70}x_{95} + x_{71}x_{95} + x_{96} \\
&+ x_{11}x_{96} + x_{12}x_{96} + x_{26}x_{96} + x_{38}x_{96} + x_{70}x_{96} + x_{71}x_{96} + x_{97} + x_{11}x_{97} + x_{12}x_{97} \\
&+ x_{26}x_{97} + x_{36}x_{97} + x_{37}x_{97} + x_{55}x_{97} + x_{56}x_{97} + x_{70}x_{97} + x_{71}x_{97} + x_{80}x_{81}x_{97} \\
&+ x_{82}x_{97} + x_{81}x_{82}x_{97} + x_{83}x_{97} + x_{95}x_{97} + x_{96}x_{97} + x_{98} + x_{114} + x_{123} + x_{126}. \qquad (7)
\end{aligned}
$$

For $I = [0, 127]\backslash\{38, 86\}$, the superpoly $p_I(\boldsymbol{x}, \mathbf{0})$ for 900-round KREYVIUM is as Eq. (8).

$$
\begin{aligned}
p_I(\boldsymbol{x}, \mathbf{0}) &= x_{125} + x_{122} + x_{121} + x_{116} + x_{113}x_{124} + x_{112} + x_{111} + x_{111}x_{112}x_{124} + \\
&x_{110}x_{124} + x_{110}x_{111}x_{124} + x_{106} + x_{105}x_{124} + x_{104} + x_{103} + x_{101} + x_{98}x_{125} + x_{98}x_{113} + \\
&x_{98}x_{111}x_{112} + x_{98}x_{110} + x_{98}x_{110}x_{111} + x_{98}x_{105} + x_{97}x_{124} + x_{97}x_{98} + x_{96} + x_{96}x_{120} + \\
&x_{96}x_{97} + x_{95} + x_{95}x_{123} + x_{94} + x_{92} + x_{92}x_{124} + x_{92}x_{98} + x_{91}x_{124} + x_{91}x_{98} + x_{90} + \\
&x_{90}x_{91} + x_{90}x_{91}x_{124} + x_{90}x_{91}x_{98} + x_{89}x_{121} + x_{89}x_{97} + x_{89}x_{96} + x_{89}x_{90} + x_{89}x_{90}x_{124} + \\
&x_{89}x_{90}x_{98} + x_{88} + x_{87} + x_{87}x_{88} + x_{87}x_{88}x_{121} + x_{87}x_{88}x_{97} + x_{87}x_{88}x_{96} + x_{87}x_{88}x_{95} + \\
&x_{86} + x_{86}x_{124} + x_{86}x_{98} + x_{85} + x_{85}x_{124} + x_{85}x_{98} + x_{84} + x_{83} + x_{82}x_{91} + x_{82}x_{89}x_{90} + \\
&x_{80}x_{81}x_{98} + x_{80}x_{81}x_{91} + x_{80}x_{81}x_{89}x_{90} + x_{80}x_{81}x_{83} + x_{80}x_{81}x_{82} + x_{79}x_{124} + x_{79}x_{98} + \\
&x_{79}x_{89} + x_{79}x_{88} + x_{79}x_{87}x_{88} + x_{79}x_{80} + x_{78}x_{89} + x_{77}x_{124} + x_{77}x_{98} + x_{77}x_{78} + \\
&x_{77}x_{78}x_{124} + x_{77}x_{78}x_{98} + x_{77}x_{78}x_{89} + x_{77}x_{78}x_{87}x_{88} + x_{76}x_{124} + x_{76}x_{98} + x_{76}x_{77} + \\
&x_{75}x_{76} + x_{75}x_{76}x_{78} + x_{75}x_{76}x_{77} + x_{73} + x_{72} + x_{72}x_{73} + x_{70} + x_{70}x_{89} + x_{70}x_{87}x_{88} + \\
&x_{70}x_{82} + x_{70}x_{80}x_{81} + x_{68}x_{125} + x_{68}x_{124} + x_{68}x_{121} + x_{68}x_{113}x_{124} + x_{68}x_{111}x_{112}x_{124} + \\
&x_{68}x_{110}x_{124} + x_{68}x_{110}x_{111}x_{124} + x_{68}x_{105}x_{124} + x_{68}x_{98}x_{125} + x_{68}x_{98}x_{113} + \\
&x_{68}x_{98}x_{111}x_{112} + x_{68}x_{98}x_{110} + x_{68}x_{98}x_{110}x_{111} + x_{68}x_{98}x_{105} + x_{68}x_{97} + x_{68}x_{92}x_{124} + \\
&x_{68}x_{92}x_{98} + x_{68}x_{91}x_{124} + x_{68}x_{91}x_{98} + x_{68}x_{90}x_{91}x_{124} + x_{68}x_{90}x_{91}x_{98} + \\
&x_{68}x_{89}x_{90}x_{124} + x_{68}x_{89}x_{90}x_{98} + x_{68}x_{86}x_{124} + x_{68}x_{86}x_{98} + x_{68}x_{85}x_{124} + x_{68}x_{85}x_{98} + \\
&x_{68}x_{80} + x_{68}x_{77}x_{124} + x_{68}x_{77}x_{98} + x_{68}x_{76}x_{124} + x_{68}x_{76}x_{98} + x_{67}x_{68} + x_{66} + x_{66}x_{98} + \\
&x_{66}x_{91} + x_{66}x_{89}x_{90} + x_{66}x_{88} + x_{66}x_{70} + x_{66}x_{68} + x_{66}x_{68}x_{98} + x_{65}x_{124} + x_{65}x_{113} + \\
&x_{65}x_{111}x_{112} + x_{65}x_{110} + x_{65}x_{110}x_{111} + x_{65}x_{105} + x_{65}x_{98} + x_{65}x_{97} + x_{65}x_{92} + x_{65}x_{91} +
\end{aligned}
$$

$$x_{65}x_{90}x_{91} + x_{65}x_{89}x_{90} + x_{65}x_{85} + x_{65}x_{79} + x_{65}x_{77} + x_{65}x_{77}x_{78} + x_{65}x_{76} + x_{65}x_{70} +$$
$$x_{65}x_{68}x_{124} + x_{65}x_{68}x_{113} + x_{65}x_{68}x_{111}x_{112} + x_{65}x_{68}x_{110} + x_{65}x_{68}x_{110}x_{111} +$$
$$x_{65}x_{68}x_{105} + x_{65}x_{68}x_{98} + x_{65}x_{68}x_{92} + x_{65}x_{68}x_{91} + x_{65}x_{68}x_{90}x_{91} + x_{65}x_{68}x_{89}x_{90} +$$
$$x_{65}x_{68}x_{86} + x_{65}x_{68}x_{85} + x_{65}x_{68}x_{77} + x_{65}x_{68}x_{76} + x_{64}x_{124} + x_{64}x_{98} + x_{64}x_{95} +$$
$$x_{64}x_{82} + x_{64}x_{80}x_{81} + x_{64}x_{68}x_{124} + x_{64}x_{68}x_{98} + x_{64}x_{66} + x_{64}x_{65}x_{91} + x_{64}x_{65}x_{89}x_{90} +$$
$$x_{64}x_{65}x_{88} + x_{64}x_{65}x_{70} + x_{64}x_{65}x_{68} + x_{63} + x_{63}x_{124} + x_{63}x_{98} + x_{63}x_{68}x_{124} +$$
$$x_{63}x_{68}x_{98} + x_{63}x_{65} + x_{63}x_{65}x_{68} + x_{63}x_{64}x_{86} + x_{63}x_{64}x_{70} + x_{63}x_{64}x_{66} + x_{63}x_{64}x_{65} +$$
$$x_{62} + x_{62}x_{124} + x_{62}x_{121} + x_{62}x_{98} + x_{62}x_{97} + x_{62}x_{96} + x_{62}x_{95} + x_{62}x_{89} + x_{62}x_{87}x_{88} +$$
$$x_{62}x_{79} + x_{62}x_{77}x_{78} + x_{62}x_{70} + x_{62}x_{68} + x_{62}x_{68}x_{124} + x_{62}x_{68}x_{98} + x_{62}x_{65} +$$
$$x_{62}x_{65}x_{68} + x_{62}x_{63} + x_{61}x_{96} + x_{61}x_{62}x_{124} + x_{61}x_{62}x_{98} + x_{61}x_{62}x_{68}x_{124} +$$
$$x_{61}x_{62}x_{68}x_{98} + x_{61}x_{62}x_{65} + x_{61}x_{62}x_{65}x_{68} + x_{60} + x_{60}x_{61}x_{124} + x_{60}x_{61}x_{98} +$$
$$x_{60}x_{61}x_{68}x_{124} + x_{60}x_{61}x_{68}x_{98} + x_{60}x_{61}x_{65} + x_{60}x_{61}x_{65}x_{68} + x_{58}x_{59} + x_{56}x_{98} +$$
$$x_{56}x_{82} + x_{56}x_{80}x_{81} + x_{56}x_{68} + x_{56}x_{68}x_{98} + x_{55} + x_{55}x_{113} + x_{55}x_{111}x_{112} + x_{55}x_{110} +$$
$$x_{55}x_{110}x_{111} + x_{55}x_{105} + x_{55}x_{98} + x_{55}x_{97} + x_{55}x_{92} + x_{55}x_{90}x_{91} + x_{55}x_{86} + x_{55}x_{85} +$$
$$x_{55}x_{83} + x_{55}x_{81}x_{82} + x_{55}x_{79} + x_{55}x_{77} + x_{55}x_{77}x_{78} + x_{55}x_{76} + x_{55}x_{70} + x_{55}x_{68} +$$
$$x_{55}x_{68}x_{113} + x_{55}x_{68}x_{111}x_{112} + x_{55}x_{68}x_{110} + x_{55}x_{68}x_{110}x_{111} + x_{55}x_{68}x_{105} +$$
$$x_{55}x_{68}x_{92} + x_{55}x_{68}x_{91} + x_{55}x_{68}x_{90}x_{91} + x_{55}x_{68}x_{89}x_{90} + x_{55}x_{68}x_{86} + x_{55}x_{68}x_{85} +$$
$$x_{55}x_{68}x_{77} + x_{55}x_{68}x_{76} + x_{55}x_{65} + x_{55}x_{65}x_{68} + x_{55}x_{64}x_{68} + x_{55}x_{63} + x_{55}x_{63}x_{68} +$$
$$x_{55}x_{62} + x_{55}x_{62}x_{68} + x_{55}x_{61}x_{62} + x_{55}x_{61}x_{62}x_{68} + x_{55}x_{60}x_{61} + x_{55}x_{60}x_{61}x_{68} +$$
$$x_{55}x_{56} + x_{54} + x_{54}x_{124} + x_{54}x_{98} + x_{54}x_{68}x_{124} + x_{54}x_{68}x_{98} + x_{54}x_{65} + x_{54}x_{65}x_{68} +$$
$$x_{54}x_{55} + x_{54}x_{55}x_{68} + x_{53} + x_{53}x_{111}x_{124} + x_{53}x_{98}x_{111} + x_{53}x_{68}x_{111}x_{124} +$$
$$x_{53}x_{68}x_{98}x_{111} + x_{53}x_{65}x_{111} + x_{53}x_{65}x_{68}x_{111} + x_{53}x_{55}x_{111} + x_{53}x_{55}x_{68}x_{111} + x_{52} +$$
$$x_{52}x_{124} + x_{52}x_{112}x_{124} + x_{52}x_{110}x_{124} + x_{52}x_{98} + x_{52}x_{98}x_{112} + x_{52}x_{98}x_{110} + x_{52}x_{68} +$$
$$x_{52}x_{68}x_{112}x_{124} + x_{52}x_{68}x_{110}x_{124} + x_{52}x_{68}x_{98}x_{112} + x_{52}x_{68}x_{98}x_{110} + x_{52}x_{65} +$$
$$x_{52}x_{65}x_{112} + x_{52}x_{65}x_{110} + x_{52}x_{65}x_{68}x_{112} + x_{52}x_{65}x_{68}x_{110} + x_{52}x_{55} + x_{52}x_{55}x_{112} +$$
$$x_{52}x_{55}x_{110} + x_{52}x_{55}x_{68}x_{112} + x_{52}x_{55}x_{68}x_{110} + x_{52}x_{53}x_{124} + x_{52}x_{53}x_{98} +$$
$$x_{52}x_{53}x_{68}x_{124} + x_{52}x_{53}x_{68}x_{98} + x_{52}x_{53}x_{65} + x_{52}x_{53}x_{65}x_{68} + x_{52}x_{53}x_{55} +$$
$$x_{52}x_{53}x_{55}x_{68} + x_{51}x_{124} + x_{51}x_{111}x_{124} + x_{51}x_{98} + x_{51}x_{98}x_{111} + x_{51}x_{96} + x_{51}x_{77} +$$
$$x_{51}x_{75}x_{76} + x_{51}x_{68}x_{124} + x_{51}x_{68}x_{111}x_{124} + x_{51}x_{68}x_{98} + x_{51}x_{68}x_{98}x_{111} + x_{51}x_{65} +$$
$$x_{51}x_{65}x_{111} + x_{51}x_{65}x_{68} + x_{51}x_{65}x_{68}x_{111} + x_{51}x_{55} + x_{51}x_{55}x_{111} + x_{51}x_{55}x_{68} +$$
$$x_{51}x_{55}x_{68}x_{111} + x_{51}x_{52}x_{124} + x_{51}x_{52}x_{98} + x_{51}x_{52}x_{68}x_{124} + x_{51}x_{52}x_{68}x_{98} +$$
$$x_{51}x_{52}x_{65} + x_{51}x_{52}x_{65}x_{68} + x_{51}x_{52}x_{55} + x_{51}x_{52}x_{55}x_{68} + x_{50} + x_{50}x_{78} + x_{50}x_{76}x_{77} +$$
$$x_{49} + x_{48} + x_{47}x_{48} + x_{46}x_{124} + x_{46}x_{98} + x_{46}x_{68}x_{124} + x_{46}x_{68}x_{98} + x_{46}x_{65} +$$
$$x_{46}x_{65}x_{68} + x_{46}x_{55} + x_{46}x_{55}x_{68} + x_{46}x_{47} + x_{45} + x_{44} + x_{42} + x_{40} + x_{39} + x_{39}x_{125} +$$
$$x_{39}x_{113} + x_{39}x_{111}x_{112} + x_{39}x_{110} + x_{39}x_{110}x_{111} + x_{39}x_{105} + x_{39}x_{97} + x_{39}x_{92} +$$
$$x_{39}x_{90}x_{91} + x_{39}x_{88} + x_{39}x_{86} + x_{39}x_{85} + x_{39}x_{80}x_{81} + x_{39}x_{79} + x_{39}x_{77} + x_{39}x_{77}x_{78} +$$
$$x_{39}x_{76} + x_{39}x_{70} + x_{39}x_{68}x_{125} + x_{39}x_{68}x_{113} + x_{39}x_{68}x_{111}x_{112} + x_{39}x_{68}x_{110} +$$
$$x_{39}x_{68}x_{110}x_{111} + x_{39}x_{68}x_{105} + x_{39}x_{68}x_{92} + x_{39}x_{68}x_{91} + x_{39}x_{68}x_{90}x_{91} +$$
$$x_{39}x_{68}x_{89}x_{90} + x_{39}x_{68}x_{86} + x_{39}x_{68}x_{85} + x_{39}x_{68}x_{77} + x_{39}x_{68}x_{76} + x_{39}x_{66} +$$

$$x_{39}x_{66}x_{68}+x_{39}x_{65}x_{68}+x_{39}x_{64}x_{68}+x_{39}x_{63}+x_{39}x_{63}x_{68}+x_{39}x_{63}x_{64}+x_{39}x_{62}+$$
$$x_{39}x_{62}x_{68}+x_{39}x_{61}x_{62}+x_{39}x_{61}x_{62}x_{68}+x_{39}x_{60}x_{61}+x_{39}x_{60}x_{61}x_{68}+x_{39}x_{56}+$$
$$x_{39}x_{56}x_{68}+x_{39}x_{55}+x_{39}x_{54}+x_{39}x_{54}x_{68}+x_{39}x_{53}x_{111}+x_{39}x_{53}x_{68}x_{111}+x_{39}x_{52}+$$
$$x_{39}x_{52}x_{112}+x_{39}x_{52}x_{110}+x_{39}x_{52}x_{68}x_{112}+x_{39}x_{52}x_{68}x_{110}+x_{39}x_{52}x_{53}+$$
$$x_{39}x_{52}x_{53}x_{68}+x_{39}x_{51}+x_{39}x_{51}x_{111}+x_{39}x_{51}x_{68}+x_{39}x_{51}x_{68}x_{111}+x_{39}x_{51}x_{52}+$$
$$x_{39}x_{51}x_{52}x_{68}+x_{39}x_{46}+x_{39}x_{46}x_{68}+x_{38}x_{124}+x_{38}x_{98}+x_{38}x_{96}+x_{38}x_{89}+$$
$$x_{38}x_{87}x_{88}+x_{38}x_{86}+x_{38}x_{70}+x_{38}x_{68}+x_{38}x_{66}+x_{38}x_{65}+x_{38}x_{64}x_{65}+x_{38}x_{62}+$$
$$x_{38}x_{55}+x_{37}x_{120}+x_{37}x_{97}+x_{37}x_{89}+x_{37}x_{87}x_{88}+x_{37}x_{62}+x_{37}x_{61}+x_{37}x_{51}+$$
$$x_{37}x_{38}+x_{36}+x_{36}x_{123}+x_{36}x_{87}x_{88}+x_{36}x_{64}+x_{36}x_{62}+x_{35}x_{124}+x_{35}x_{98}+$$
$$x_{35}x_{68}x_{124}+x_{35}x_{68}x_{98}+x_{35}x_{65}+x_{35}x_{65}x_{68}+x_{35}x_{55}+x_{35}x_{55}x_{68}+x_{35}x_{39}+$$
$$x_{35}x_{39}x_{68}+x_{34}+x_{33}+x_{32}+x_{31}+x_{30}x_{95}+x_{30}x_{78}+x_{30}x_{36}+x_{29}x_{79}+x_{29}x_{66}+$$
$$x_{29}x_{64}x_{65}+x_{29}x_{39}+x_{28}+x_{27}+x_{27}x_{124}+x_{27}x_{98}+x_{27}x_{68}x_{124}+x_{27}x_{68}x_{98}+$$
$$x_{27}x_{65}x_{68}+x_{27}x_{63}x_{64}+x_{27}x_{55}+x_{27}x_{55}x_{68}+x_{27}x_{39}+x_{27}x_{39}x_{68}+x_{27}x_{38}+x_{26}+$$
$$x_{26}x_{124}+x_{26}x_{98}+x_{26}x_{68}x_{124}+x_{26}x_{68}x_{98}+x_{26}x_{65}+x_{26}x_{65}x_{68}+x_{26}x_{55}+$$
$$x_{26}x_{55}x_{68}+x_{26}x_{39}+x_{26}x_{39}x_{68}+x_{25}+x_{23}+x_{23}x_{98}+x_{23}x_{39}+x_{22}+x_{21}+x_{21}x_{68}+$$
$$x_{20}x_{95}+x_{20}x_{88}+x_{20}x_{78}+x_{20}x_{36}+x_{20}x_{29}+x_{19}x_{89}+x_{19}x_{30}+x_{19}x_{20}+x_{18}x_{124}+$$
$$x_{18}x_{98}+x_{18}x_{68}x_{124}+x_{18}x_{68}x_{98}+x_{18}x_{65}+x_{18}x_{65}x_{68}+x_{18}x_{55}+x_{18}x_{55}x_{68}+$$
$$x_{18}x_{39}+x_{18}x_{39}x_{68}+x_{17}x_{124}+x_{17}x_{98}+x_{17}x_{68}x_{124}+x_{17}x_{68}x_{98}+x_{17}x_{65}+$$
$$x_{17}x_{65}x_{68}+x_{17}x_{55}+x_{17}x_{55}x_{68}+x_{17}x_{39}+x_{17}x_{39}x_{68}+x_{15}+x_{14}+x_{13}+x_{11}x_{89}+$$
$$x_{11}x_{87}x_{88}+x_{11}x_{82}+x_{11}x_{80}x_{81}+x_{11}x_{68}+x_{11}x_{66}+x_{11}x_{65}+x_{11}x_{64}x_{65}+$$
$$x_{11}x_{63}x_{64}+x_{11}x_{62}+x_{11}x_{55}+x_{11}x_{39}+x_{11}x_{38}+x_{10}x_{88}+x_{10}x_{29}+x_{9}x_{125}+x_{9}x_{124}+$$
$$x_{9}x_{121}+x_{9}x_{113}x_{124}+x_{9}x_{111}x_{112}x_{124}+x_{9}x_{110}x_{124}+x_{9}x_{110}x_{111}x_{124}+x_{9}x_{105}x_{124}+$$
$$x_{9}x_{98}x_{125}+x_{9}x_{98}x_{113}+x_{9}x_{98}x_{111}x_{112}+x_{9}x_{98}x_{110}+x_{9}x_{98}x_{110}x_{111}+x_{9}x_{98}x_{105}+$$
$$x_{9}x_{97}+x_{9}x_{92}x_{124}+x_{9}x_{92}x_{98}+x_{9}x_{91}x_{124}+x_{9}x_{91}x_{98}+x_{9}x_{90}x_{91}x_{124}+$$
$$x_{9}x_{90}x_{91}x_{98}+x_{9}x_{89}+x_{9}x_{89}x_{90}x_{124}+x_{9}x_{89}x_{90}x_{98}+x_{9}x_{86}x_{124}+x_{9}x_{86}x_{98}+$$
$$x_{9}x_{85}x_{124}+x_{9}x_{85}x_{98}+x_{9}x_{80}+x_{9}x_{77}x_{124}+x_{9}x_{77}x_{98}+x_{9}x_{76}x_{124}+x_{9}x_{76}x_{98}+$$
$$x_{9}x_{66}+x_{9}x_{66}x_{98}+x_{9}x_{65}x_{124}+x_{9}x_{65}x_{113}+x_{9}x_{65}x_{111}x_{112}+x_{9}x_{65}x_{110}+$$
$$x_{9}x_{65}x_{110}x_{111}+x_{9}x_{65}x_{105}+x_{9}x_{65}x_{98}+x_{9}x_{65}x_{92}+x_{9}x_{65}x_{91}+x_{9}x_{65}x_{90}x_{91}+$$
$$x_{9}x_{65}x_{89}x_{90}+x_{9}x_{65}x_{86}+x_{9}x_{65}x_{85}+x_{9}x_{65}x_{77}+x_{9}x_{65}x_{76}+x_{9}x_{64}x_{124}+x_{9}x_{64}x_{98}+$$
$$x_{9}x_{64}x_{65}+x_{9}x_{63}x_{124}+x_{9}x_{63}x_{98}+x_{9}x_{63}x_{65}+x_{9}x_{62}+x_{9}x_{62}x_{124}+x_{9}x_{62}x_{98}+$$
$$x_{9}x_{62}x_{65}+x_{9}x_{61}x_{62}x_{124}+x_{9}x_{61}x_{62}x_{98}+x_{9}x_{61}x_{62}x_{65}+x_{9}x_{60}x_{61}x_{124}+$$
$$x_{9}x_{60}x_{61}x_{98}+x_{9}x_{60}x_{61}x_{65}+x_{9}x_{56}+x_{9}x_{56}x_{98}+x_{9}x_{55}+x_{9}x_{55}x_{113}+$$
$$x_{9}x_{55}x_{111}x_{112}+x_{9}x_{55}x_{110}+x_{9}x_{55}x_{110}x_{111}+x_{9}x_{55}x_{105}+x_{9}x_{55}x_{92}+x_{9}x_{55}x_{91}+$$
$$x_{9}x_{55}x_{90}x_{91}+x_{9}x_{55}x_{89}x_{90}+x_{9}x_{55}x_{86}+x_{9}x_{55}x_{85}+x_{9}x_{55}x_{77}+x_{9}x_{55}x_{76}+$$
$$x_{9}x_{55}x_{65}+x_{9}x_{55}x_{64}+x_{9}x_{55}x_{63}+x_{9}x_{55}x_{62}+x_{9}x_{55}x_{61}x_{62}+x_{9}x_{55}x_{60}x_{61}+$$
$$x_{9}x_{54}x_{124}+x_{9}x_{54}x_{98}+x_{9}x_{54}x_{65}+x_{9}x_{54}x_{55}+x_{9}x_{53}x_{111}x_{124}+x_{9}x_{53}x_{98}x_{111}+$$
$$x_{9}x_{53}x_{65}x_{111}+x_{9}x_{53}x_{55}x_{111}+x_{9}x_{52}+x_{9}x_{52}x_{112}x_{124}+x_{9}x_{52}x_{110}x_{124}+$$
$$x_{9}x_{52}x_{98}x_{112}+x_{9}x_{52}x_{98}x_{110}+x_{9}x_{52}x_{65}x_{112}+x_{9}x_{52}x_{65}x_{110}+x_{9}x_{52}x_{55}x_{112}+$$
$$x_{9}x_{52}x_{55}x_{110}+x_{9}x_{52}x_{53}x_{124}+x_{9}x_{52}x_{53}x_{98}+x_{9}x_{52}x_{53}x_{65}+x_{9}x_{52}x_{53}x_{55}+$$

$$x_9x_{51}x_{124} + x_9x_{51}x_{111}x_{124} + x_9x_{51}x_{98} + x_9x_{51}x_{98}x_{111} + x_9x_{51}x_{65} + x_9x_{51}x_{65}x_{111} + x_9x_{51}x_{55} + x_9x_{51}x_{55}x_{111} + x_9x_{51}x_{52}x_{124} + x_9x_{51}x_{52}x_{98} + x_9x_{51}x_{52}x_{65} + x_9x_{51}x_{52}x_{55} + x_9x_{46}x_{124} + x_9x_{46}x_{98} + x_9x_{46}x_{65} + x_9x_{46}x_{55} + x_9x_{39}x_{125} + x_9x_{39}x_{113} + x_9x_{39}x_{111}x_{112} + x_9x_{39}x_{110} + x_9x_{39}x_{110}x_{111} + x_9x_{39}x_{105} + x_9x_{39}x_{92} + x_9x_{39}x_{91} + x_9x_{39}x_{90}x_{91} + x_9x_{39}x_{89}x_{90} + x_9x_{39}x_{86} + x_9x_{39}x_{85} + x_9x_{39}x_{77} + x_9x_{39}x_{76} + x_9x_{39}x_{66} + x_9x_{39}x_{65} + x_9x_{39}x_{64} + x_9x_{39}x_{63} + x_9x_{39}x_{62} + x_9x_{39}x_{61}x_{62} + x_9x_{39}x_{60}x_{61} + x_9x_{39}x_{56} + x_9x_{39}x_{54} + x_9x_{39}x_{53}x_{111} + x_9x_{39}x_{52}x_{112} + x_9x_{39}x_{52}x_{110} + x_9x_{39}x_{52}x_{53} + x_9x_{39}x_{51} + x_9x_{39}x_{51}x_{111} + x_9x_{39}x_{51}x_{52} + x_9x_{39}x_{46} + x_9x_{38} + x_9x_{35}x_{124} + x_9x_{35}x_{98} + x_9x_{35}x_{65} + x_9x_{35}x_{55} + x_9x_{35}x_{39} + x_9x_{30} + x_9x_{27}x_{124} + x_9x_{27}x_{98} + x_9x_{27}x_{65} + x_9x_{27}x_{55} + x_9x_{27}x_{39} + x_9x_{26}x_{124} + x_9x_{26}x_{98} + x_9x_{26}x_{65} + x_9x_{26}x_{55} + x_9x_{26}x_{39} + x_9x_{21} + x_9x_{20} + x_9x_{18}x_{124} + x_9x_{18}x_{98} + x_9x_{18}x_{65} + x_9x_{18}x_{55} + x_9x_{18}x_{39} + x_9x_{17}x_{124} + x_9x_{17}x_{98} + x_9x_{17}x_{65} + x_9x_{17}x_{55} + x_9x_{17}x_{39} + x_9x_{11} + x_8x_{124} + x_8x_{98} + x_8x_{68}x_{124} + x_8x_{68}x_{98} + x_8x_{65} + x_8x_{65}x_{68} + x_8x_{55} + x_8x_{55}x_{68} + x_8x_{39} + x_8x_{39}x_{68} + x_8x_9x_{124} + x_8x_9x_{98} + x_8x_9x_{65} + x_8x_9x_{55} + x_8x_9x_{39} + x_7 + x_7x_{124} + x_7x_{98} + x_7x_{68}x_{124} + x_7x_{68}x_{98} + x_7x_{65} + x_7x_{65}x_{68} + x_7x_{55} + x_7x_{55}x_{68} + x_7x_{39} + x_7x_{39}x_{68} + x_7x_9x_{124} + x_7x_9x_{98} + x_7x_9x_{65} + x_7x_9x_{55} + x_7x_9x_{39} + x_6 + x_5x_{95} + x_5x_{36}. \tag{8}$$