A New Method for Solving Polynomial Systems with Noise over $\mathbb{F}_2$ and Its Applications in Cold Boot Key Recovery*

Zhenyu Huang and Dongdai Lin

State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{huangzhenyu,ddlin}@iie.ac.cn

Abstract. The family of Max-PoSSo problems is about solving polynomial systems with noise, and is analogous to the well-known Max-SAT family of problems when the ground field is $\mathbb{F}_2$. In this paper, we present a new method called **ISBS** for solving the family of Max-PoSSo problems over $\mathbb{F}_2$. This method is based on the ideas of incrementally solving polynomial system and searching the values of polynomials with backtracking. The **ISBS** method can be combined with different algebraic methods for solving polynomial systems, such as the Gröbner Basis method or the Characteristic Set(CS) method. By combining with the CS method, we implement **ISBS** and apply it in Cold Boot attacks. A Cold Boot attack is a type of side channel attack in which an attacker recover cryptographic key material from DRAM relies on the data remanence property of DRAM. Cold Boot key recovery problems of block ciphers can be modeled as Max-PoSSo problems over $\mathbb{F}_2$. We apply the **ISBS** method to solve the Cold Boot key recovery problems of AES and Serpent, and obtain some experimental results which are better than the existing ones.

Keywords: polynomial system with noise, Max-PoSSo, Cold Boot attack, boolean equations, Characteristic Set method, AES, Serpent.

1 Introduction

Solving polynomial system with noise, which means finding an optimal solution from a group of polynomials with noise, is a fundamental problem in several areas of cryptography, such as algebraic attacks, side-channel attacks and the cryptanalysis of LPN/LWE-based schemes. In computation complexity field, this problem is also significant and called the maximum equation satisfying problem [7,14]. In the general case, this problem is NP-hard even when the polynomials are linear. In [1], the authors classified this kind of problems into three categories:

* This work was in part supported by National 973 Program of China under Grants No. 2013CB834203 and No. 2011CB302400, the National Natural Science Foundation of China under Grant No. 60970152 and the "Strategic Priority Research Program" of the Chinese Academy of Sciences under Grant No. XDA06010701.

L.R. Knudsen and H. Wu (Eds.): SAC 2012, LNCS 7707, pp. 16–33, 2013.

Max-PoSSo, Partial Max-PoSSo, and Partial Weighted Max-PoSSo, and called them the family of Max-PoSSo problems. Moreover, they presented a model by which they can convert the Cold Boot key recovery problems of block ciphers into the family of Max-PoSSo problems. The Cold Boot key recovery problems originated from a side channel attack which is called the Cold Boot attack[8]. In a Cold Boot attack, an attacker with physical access to a computer is able to retrieve sensitive information from a running operating system after using a cold reboot to restart the machine from a completely "off" state. The attack relies on the data remanence property of DRAM to retrieve memory contents which remain readable in the seconds to minutes after power has been removed. Furthermore, the time of retention can be potentially increased by reducing the temperature of memory. Thus, data in memory can be used to recover potentially sensitive information, such as cryptographic keys. Due to the nature of the Cold Boot attack, it is realistic to assume that only decayed image of the data in memory can be available to the attacker, which means a fraction of memory bits will be flipped. Therefore, the most important step of the Cold Boot attack is recovering the original sensitive information from the decayed data.

In the case of block cipher, the sensitive information is the original key, and the decayed data is likely to be the round keys, which are generated from the origin key by the key schedule operation. Thus the Cold Boot key recovery problem of block cipher is recovering the origin key from the decayed round keys. Intuitively, every bit of round keys corresponds to a boolean polynomial equation with the bits of origin key as its variables. Then all bits of these round keys correspond to a boolean polynomial system. However, because of the data decay this polynomial system has some noise. In general case, these polynomials can be seen as random ones, so a random assignment may satisfy about half of them. If the percentage of the decayed bits is smaller than 50%, an assignment satisfying the maximum number of these polynomials may be equal to the origin key with high probability. By this way, we can model the Cold Boot key recovery problem of block cipher as the Max-PoSSo problem over $\mathbb{F}_2$, which is finding the optimal solution of a polynomial system with noise.

As mentioned before, the general Max-PoSSo problem over $\mathbb{F}_2$ is NP-hard. A natural way of solving Max-PoSSo problems over $\mathbb{F}_2$ is converting them into their SAT equivalents and then solve them by Max-SAT solvers. However, this method has a disadvantages that the original algebraic structure is destroyed. In [1], the authors converted the Max-PoSSo problems over $\mathbb{F}_2$ into mixed integer programming problems, and used the **MIP** solver **SCIP** to solve them. They presented some experimental results about attacking AES and Serpent. Their attack result about Serpent is a new result, and they showed that comparing with generic combinatorial approach their attack is much better.

The main contribution of this paper is that we propose a new method called **ISBS** for solving the family of Max-PoSSo problems over $\mathbb{F}_2$. The basic idea of **ISBS** is searching the values of polynomials. Precisely speaking, given a polynomials system with noise $\{f_1, f_2, \ldots, f_m\}$, we try to solve polynomial systems $\{f_1 + e_2, f_2 + e_2, \ldots, f_m + e_m\}$, where $\{e_1, e_2, \ldots, e_m\}$

can be equal to $\{0,0,\ldots,0\},\{1,0,\ldots,0\},\ldots,\{1,1,\ldots,1\}$. The solution of a $\{f_1+e_2, f_2+e_2,\ldots,f_m+e_m\}$ with $\{e_1,e_2,\ldots,e_m\}$ having the smallest Hamming-weight is the solution of the Max-PoSSo problem.

In the **ISBS** method, we combine the above idea with the ideas of incrementally solving $\{f_1+e_1, f_2+e_2,\ldots,f_m+e_m\}$ and searching $\{e_1,e_2,\ldots,e_m\}$ with backtracking. By this way, we can cut off a lot of branches when searching the values of $\{e_1,e_2,\ldots,e_m\}$. Furthermore, with the incremental solving method, we can use the former results to derive the latter ones, by which we can reduce a lot of computation.

In order to further improve the efficiency of **ISBS**, we combine it with an algebraic method for solving polynomial system which is called the Characteristic Set(CS) Method. In the field of symbolic computation, the CS method is an important tool for studying polynomial, algebraic differential, and algebraic difference equation systems. Its idea is reducing equation systems in general form to equation systems in the form of triangular sets. This method was introduced by Ritt and the recent study of it was inspired by Wu's seminal work on automated geometry theorem proving [13]. In [2], the CS method was firstly extended to solve polynomial equations in boolean ring. In [6], it was further extended to solve polynomial equations in general finite fields, and an efficient variant of the CS method called the **MFCS** algorithm was proposed and systematically analyzed. **MFCS** is an algorithm for solving boolean polynomial system, and it already had some applications in cryptanalysis[9]. **MFCS** has some advantage in incrementally solving polynomial system, thus we implemented **ISBS** with the **MFCS** algorithm.

Furthermore, we used **ISBS** to solve some Cold Boot key recovery problems of AES and Serpent, and compared our experimental results with those in [1]. From these results, we showed that by solving these problems with **ISBS** the success rate of recovering the origin key is higher and the average running time is shorter.

The rest of this paper is organized as follows. In Section 2, we introduce the family of Max-PoSSo problems and its relation with Cold Boot key recovery. In Section 3, we present the **ISBS** method and simply introduce the **MFCS** algorithm. In Section 4, our experimental results of attacking AES and Serpent are shown. In Section 5, the conclusions are presented. In Appendix, we present some tricks we used in solving the symmetric noise problems.

2 The Family of Max-PoSSo Problems and the Cold Boot Problem

In this section we will introduce the family of Max-PoSSo problems and its relationship with the Cold Boot key recovery problem.

2.1 The Family of Max-PoSSo Problems

Let $\mathbb{F}$ be a field, and $\mathbb{P} = \{f_1,\ldots,f_m\} \subset \mathbb{F}[x_1,\ldots,x_n]$ is a polynomial system. The *polynomial system solving (PoSSo)* problem is finding a solution $(x_1,\ldots,x_n) \in \mathbb{F}^n$ such that $\forall f_i \in \mathbb{P}$, we have $f_i(x_1,\ldots,x_n) = 0$.

By *Max-PoSSo*, we mean the problem of finding any $(x_1, \ldots, x_n) \in \mathbb{F}^n$ that satisfies the maximum number of polynomials in $\mathbb{P}$. The name "Max-PoSSo" was first proposed in [1]. In the computational complexity field, this problem is sometimes called the maximum equation satisfying problem [7,14]. Obviously, Max-PoSSo is at least as hard as PoSSo. Moreover, whether the polynomials in $\mathbb{P}$ are linear or not, Max-PoSSo is a NP-hard problem.

Besides Max-PoSSo, in [1], the authors introduced another two similar problems: Partial Max-PoSSo and Partial Weighted Max-PoSSo.

Partial Max-PoSSo problem is finding a point $(x_1, \ldots, x_n) \in \mathbb{F}^n$ such that for two sets of polynomials $\mathcal{H}, \mathcal{S} \in \mathbb{F}[x_1, \ldots, x_n]$, we have $f(x_1, \ldots, x_n) = 0$ for all $f \in \mathcal{H}$, and the number of polynomials $f \in \mathcal{S}$ with $f(x_1, \ldots, x_n) = 0$ is maximised. It is easy to see that Max-PoSSo is a special case of Partial Max-PoSSo when $\mathcal{H} = \emptyset$.

Let $\mathcal{H}, \mathcal{S}$ are two polynomial sets in $\mathbb{F}[x_1, \ldots, x_n]$. $\mathcal{C} : \mathcal{S} \times \mathbb{F}^n \to \mathbb{R}_{\geq 0}, (f, x) \to v$ is a cost function. $v = 0$ if $f(x) = 0$ and $v > 0$ if $f(x) \neq 0$. *Partial Weighted Max-PoSSo* denotes the problem of finding a point $(x_1, \ldots, x_n) \in \mathbb{F}^n$ such that $\forall f \in \mathcal{H}$ we have $f(x) = 0$ and $\sum_{f \in \mathcal{S}} \mathcal{C}(f, x)$ is minimised. Obviously, Partial Max-PoSSo is Partial Weighted Max-PoSSo when $\mathcal{C}(f, x) = 1$ if $f(x) \neq 0$ for all f.

In the definition of the above four problems, the ground field $\mathbb{F}$ can be any field. However, in the following of this paper, we focus on the case of $\mathbb{F} = \mathbb{F}_2$, where $\mathbb{F}_2$ is the finite field with elements 0 and 1, and this is the most common case in cryptanalysis.

2.2 Cold Boot Key Recovery as Max-PoSSo

The Cold Boot key recovery problem was first proposed and discussed in the seminal work of [8]. In [1], the authors built a new mathematical model for Cold Boot key recovery problem of block cipher, by which they can convert the Cold Boot problem into the Partial Weighted Max-PoSSo problem. In the following, we extract the definition of this model from [1].

First, according to [8], we should know that bit decay in DRAM is usually asymmetric: bit flips $0 \to 1$ and $1 \to 0$ occur with different probabilities, depending on the "ground state". The Cold Boot problem of block cipher can be defined as follows.

Consider an efficiently computable vectorial Boolean function $\mathcal{KS} : \mathbb{F}_2^n \to \mathbb{F}_2^N$ where $N > n$, and two real numbers $0 \leq \delta_0, \delta_1 \leq 1$. Let $K = \mathcal{KS}(k)$ be the image for some $k \in \mathbb{F}_2^n$, and K_i be the i-th bit of K. Given K, we compute $K' = (K'_0, K'_1, \ldots, K'_{N-1}) \in \mathbb{F}_2^N$ according to the following probability distribution:

$$Pr[K'_i = 0 | K_i = 0] = 1 - \delta_1, Pr[K'_i = 1 | K_i = 0] = \delta_1,$$

$$Pr[K'_i = 1 | K_i = 1] = 1 - \delta_0, Pr[K'_i = 0 | K_i = 1] = \delta_0.$$

Then we can consider such a K' as a noisy output of $\mathcal{KS}$ for some $k \in \mathbb{F}_2^n$, with the probability of a bit 1 in K flipping to 0 is δ_0 and the probability of a bit 0 in K flipping to 1 is δ_1. It follows that a bit $K'_i = 0$ of K' is correct with probability

$$Pr[K_i = 0|K_i = 0] = \frac{Pr[K'_i = 0|K_i = 0]Pr[K_i = 0]}{Pr[K'_i = 0]} = \frac{(1-\delta_1)}{(1-\delta_1+\delta_0)}.$$

Likewise, a bit $K'_i = 1$ of K' is correct with probability $\frac{(1-\delta_0)}{(1-\delta_0+\delta_1)}$. We denote these values by Δ_0 and Δ_1 respectively. Now assume we are given a description of the function $\mathcal{KS}$ and a vector $K' \in \mathbb{F}_2^N$ obtained by the process described above. Furthermore, we are also given a control function $E : \mathbb{F}_2^n \to \{True, False\}$ which returns $True$ or $False$ for a candidate k. The task is to recover k such that $\mathcal{E}(k)$ returns $True$. For example, $\mathcal{E}$ could use the encryption of some known data to check whether k is the original key. In the context of this work, we can consider the function $\mathcal{KS}$ as the key schedule operation of a block cipher with n-bit keys. The vector K is the result of the key schedule expansion for a key k, and the noisy vector K' is obtained from K due to the process of memory bit decay.

We can consider the Cold Boot Problem as a Partial Weighted Max-PoSSo problem over $\mathbb{F}_2$. Let $F_{\mathcal{K}}$ be an equation system corresponding to $\mathcal{KS}$ such that the only pairs (k, K) that satisfy $F_{\mathcal{K}}$ are any $k \in \mathbb{F}_2^n$ and $K = \mathcal{KS}(k)$. In our task however, we need to consider $F_{\mathcal{K}}$ with k and K'. Assume that for each noisy output bit K'_i there is some $f_i \in F_{\mathcal{K}}$ of the form $g_i + K'_i$ where g_i is some polynomial. Furthermore assume that these are the only polynomials involving the output bits ($F_{\mathcal{K}}$ can always be brought into this form) and denote the set of these polynomials by $\mathcal{S}$. Denote the set of all remaining polynomials in $F_{\mathcal{K}}$ as $\mathcal{H}$, and define the cost function $\mathcal{C}$ as a function which returns

$$\frac{1}{1-\Delta_0}, \text{ for } K'_i = 0, f(x) \neq 0; \quad \frac{1}{1-\Delta_1}, \text{ for } K'_i = 1, f(x) \neq 0; \quad 0, \text{ otherwise.}$$

This cost function returns a cost proportional to the inverse of the probability that a given value is correct. Finally, let $F_{\mathcal{E}}$ be an equation system that is only satisfiable for $k \in \mathbb{F}_2^n$ for which $\mathcal{E}$ returns True. This will usually be an equation system for one or more encryptions. Add the polynomials in $F_{\mathcal{E}}$ to $\mathcal{H}$.[2] Then $\mathcal{H}, S, C$ define a Partial Weighted Max-PoSSo problem. Any optimal solution x to this problem is a candidate solution for the Cold Boot problem.

3 A New Method for Solving Max-PoSSo Problems

3.1 The Incremental Solving and Backtracking Search (ISBS) Method

In this part, we will introduce the **ISBS** method for solving the family of Max-PoSSo problems. First, let's show our idea. Almost all existing algorithms for solving Max-PoSSo problems are based on the idea of searching the values of

[2] Actually, in our following attacks on AES and Serpent, we didn't add $F_{\mathcal{E}}$ into $\mathcal{H}$. The reason is in our method we need to solve $\mathcal{H}$ first, and if $F_{\mathcal{E}}$ is added solving $\mathcal{H}$ will be extremely inefficient.

variables, such as the Max-SAT solvers. Our idea is based on another direction which is searching the values of polynomials. Now let's show it specifically.

Given a noisy polynomial set $\mathbb{P} = \{f_1, f_2, \ldots, f_m\}$. For every vector $E = [e_1, e_2, \ldots, e_m] \in \mathbb{F}_2^n$, we can solve the polynomial system $\{f_1+e_1, f_2+e_2, \ldots, f_n+e_m\}$ by an algebraic method. We can exhaustively searching E in the order of increasing Hamming weight and solve the corresponding polynomial system for each E. If the corresponding equation set of some E have a solution, then it is the solution of the Max-PoSSo problem.

To improve the solving and searching efficiencies, we combine the incremental solving method and backtracking search method with the above idea. For a polynomial set $\mathbb{P}$, we can solve it by some algebraic method, such as the Characteristic Set(CS) method [2,6] or the Gröbner Basis method [4,5]. We denote the output results of such a solving algorithm with input $\mathbb{P}$ by Result($\mathbb{P}$). From Result($\mathbb{P}$), we can deriver all the solutions of $\mathbb{P}$ easily. We remind the reader that, for different methods, Result($\mathbb{P}$) can be different. For example, if we use the CS method to solve $\mathbb{P}$, Result($\mathbb{P}$) $= \cup_i \mathcal{A}_i$ is a group of triangular sets(A triangular set $\mathcal{A}_i$ is a polynomial set which can be easily solved, and its precise definition will be given in next section). If we use the Gröbner Basis method to solve $\mathbb{P}$, Result($\mathbb{P}$) is the Gröbner Basis of idea $< \mathbb{P} >$. When this polynomial system has no solution, we set Result($\mathbb{P}$) $= \{1\}$. Obviously, given Result($\mathbb{P}$) and a polynomial g, we can achieve Result($\{$Result($\mathbb{P}$), $g\}$). For example, for the CS method, we need to compute each Result($\mathcal{A}_i, g$) and return the union of them. For the Gröbner Basis method, we need to compute the Gröbner Basis of idea generated by the new polynomial set. Therefore, given a polynomial system $\mathbb{P} = \{f_1, f_2, \ldots, f_m\}$, we can get Result($\mathbb{P}$) by computing Result($\{f_1\}$), Result($\{$Result($\{f_1\}$), $f_2\}$), ... This is the incremental solving method.

In the **ISBS** method, first we try to incrementally solve $\{f_1+e_1, f_2+e_2, \ldots, f_i+e_i\}$ for i from 1 to m with each $e_i = 0$. If Result($\{f_1+e_1, f_2+e_2, \ldots, f_i+e_i\}$) $= 1$ for some e_i, we flip e_i to 1 and solve this new $\{f_1 + e_1, f_2 + e_2, \ldots, f_i + e_i\}$. At last, we will obtain a candidate Result($\{f_1 + e_1, f_2 + e_2, \ldots, f_m + e_m\}$) where $[e_1, \ldots, e_m] = [e'_1, \ldots, e'_m]$. Then, in order to obtain a better candidate, we search all the possible values of $[e_1, \ldots, e_m]$ with backtracking based on the value $[e'_1, \ldots, e'_m]$. That is we flip the value of e'_i for i from m to 1, and try to incrementally solve all the new systems $\{f_1+e'_1, \ldots, f_i+e'_i+1, f_{i+1}+e_{i+1}, \ldots, f_m+e_m\}$. Finally, we will find the optimal solution after searching all the possible $[e_1, \ldots, e_m]$.

In the following Algorithm 1, we will present the **ISBS** method specifically.

Here we should introduce a notation. For a polynomial set $\mathbb{P} = \{f_1, \ldots, f_m\}$, We use Zero($\mathbb{P}$) to denote the common zeros of the polynomials in $\mathbb{P}$ in the affine space $\mathbb{F}_2^n$, that is,

$$\text{Zero}(\mathbb{P}) = \{(a_1, \cdots, a_n), a_i \in \mathbb{F}_2, s.t., \forall f_i \in \mathbb{P}, f_i(a_1, \cdots, a_n) = 0\}.$$

Let $\mathbb{Q} = \cup_i \mathbb{P}_i$ be the union of some polynomial sets, we define Zero($\mathbb{Q}$) to be $\cup_i$Zero($\mathbb{P}_i$).

Theorem 1. *Algorithm 1 terminates and returns a solution of the Partial Weighted Max-PoSSo problem.*

Algorithm 1. Incremental Solving and Backtracking Search(ISBS) algorithm

Input: Two boolean polynomial sets $\mathcal{H} = \{h_1, h_2, \ldots, h_r\}$,$\mathcal{S} = \{f_1, f_2, \ldots, f_m\}$.
A cost function $\mathcal{C}(f_i, x)$,
where $\mathcal{C}(f_i, x) = 0$ if $f_i(x) = 0$, $\mathcal{C}(f_i, x) = c_i$ if $f_i(x) = 1$.

Output: $(x_1, \ldots, x_n) \in \mathbb{F}_2^n$ s.t. $h_i(x) = 0$ for any i and $\sum_{f_i \in \mathcal{S}} \mathcal{C}(f_i, x)$ is minimized.

1. Let $E = [e_1, e_2, \ldots, e_m]$ be a m-dim vector.
   Solve $\mathcal{H}$ by an algebraic method and set $\mathbb{Q}_0 = \text{Result}(\mathcal{H})$.
2. For i from 1 to m do
   1.1. Set $\mathbb{P}_i = \{\mathbb{Q}_{i-1}, f_i\}$, solve $\mathbb{P}_i$ and achieve Result($\mathbb{P}_i$).
   1.2. If Result($\mathbb{P}_i$) = $\{1\}$, then set $\mathbb{Q}_i = \mathbb{Q}_{i-1}$ and $e_i = 1$.
   1.3. Else, set $\mathbb{Q}_i = \text{Result}(\mathbb{P}_i)$, $e_i = 0$.
3. Set $\mathbb{S} = \mathbb{Q}_m$ and $u_{bound} = \sum_i c_i e_i$.
   Set $u = u_{bound}$, $k = m$.
4. while $k \geq 1$ do
   4.1. If $e_k = 0$ and $u + c_k < u_{bound}$, then
      4.1.1. Set $e_k = 1$, $u = u + c_k$.
         Solve $\mathbb{P}_k = \{\mathbb{Q}_{k-1}, f_k + 1\}$ and achieve Result($\mathbb{P}_k$).
      4.1.2. If Result($\mathbb{P}_k$) = $\{1\}$, then goto Step 4.2.
      4.1.3. Else, Set $\mathbb{Q}_k = \text{Result}(\mathbb{P}_k)$.
         For i from $k + 1$ to m do
         4.1.3.1. Solve $\mathbb{P}_i = \{\mathbb{Q}_{i-1}, f_i\}$ and achieve Result($\mathbb{P}_i$).
         4.1.3.2. If Result($\mathbb{P}_i$) = $\{1\}$ then $\mathbb{Q}_i = \mathbb{Q}_{i-1}, e_i = 1$.
            $u = u + c_i$. If $u \geq u_{bound}$, then set $k = i$, and goto Step 4.2.
         4.1.3.3. Else, $\mathbb{Q}_i = \text{Result}(\mathbb{P}_i)$, $e_i = 0$.
      4.1.4. Set $k = m$, $\mathbb{S} = \mathbb{Q}_m$, $u_{bound} = u$.
   4.2. Else, $u = u - e_k c_k$, $k = k - 1$.
5. Get $(x_1, \ldots, x_n)$ from $\mathbb{S}$, and return $(x_1, \ldots, x_n)$.

Proof: The termination of Algorithm 1 is trivial, since in this algorithm we are searching all possible values of $[e_1, e_2, \ldots, e_m]$ with backtracking and the number of possible values is finite. In the following, we will explain the operations of this algorithm more specifically, from which we can show the correctness of this algorithm.

From Step 1 to Step 3, we do the following operations recursively. For k from 0 to $m - 1$, We compute Result($\{\mathbb{Q}_k, f_{k+1}\}$).

- If the result is not $\{1\}$, it means that $f_{k+1} = 0$ can be satisfied. We set $e_{k+1} = 0$ and use $\mathbb{Q}_k$ to store the result. Then we compute Result($\{\mathbb{Q}_{k+1}, f_{k+2}\}$).
- If the result is $\{1\}$, $f_{k+1} = 0$ cannot be satisfied by any point in Zero($\mathbb{Q}_k$). It implies that $f_{k+1} = 1$ holds for all these points, so we set $e_{k+1} = 1$. We set $\mathbb{Q}_{k+1} = \mathbb{Q}_k$, and it is also equal to Result($\{\mathbb{Q}_k, f_k + 1\}$). Then we compute Result($\{\mathbb{Q}_{k+1}, f_{k+2}\}$).

After Step 3, we achieve $\mathbb{S}$, and the points in Zero($\mathbb{S}$) are candidates of the problem. For any k with $e_k = 0$, the corresponding polynomial f_k vanishes on Zero($\mathbb{S}$). For any other f_k, $f_k = 0$ is unsatisfied by these points. Note that for

the points in Zero($\mathbb{S}$), the values of the corresponding cost functions are same. We use u_{bound} to store this value. Obviously, u, the value of the cost function for a better candidate, should satisfy $u < u_{bound}$.

In Step 4, we are trying to find a better candidate by backtracking to e_k. From $k = m$ to 1, we try to flip the value of e_k, and there are four cases.

1) $e_k = 1$, and Zero($\{f_1 + e_1, \ldots, f_{k-1} + e_{k-1}, f_k\}$) $= \emptyset$. We don't flip e_k, since Zero($\{f_1 + e_1, \ldots, f_{k-1} + e_{k-1}, f_k\}$) $= \emptyset$ and we can not find any candidate from Zero($\{f_1 + e_1, \ldots, f_k\}$).
2) $e_k = 1$, and e_k has already been flipped with the same $e_1, \ldots, e_{k-1}$. We don't flip e_k, since we have already considered the points in Zero($\{f_1+e_1, \ldots, f_k\}$).
3) $e_k = 0$, and $u \geq u_{bound}$, where u is the value of the cost function corresponding to $[e_1, \ldots, e_{k-1}, e_k + 1]$. We don't flip e_k, since the points in Zero($\{f_1 + e_1, \ldots, f_k + e_k + 1\}$) are worse than the stored candidate.
4) $e_k = 0$, and $u < u_{bound}$, where u is the value of the cost function corresponding to $[e_1, \ldots, e_{k-1}, e_k + 1]$. We flip e_k.

After we flipping e_k to 1, we compute Result($\{\mathbb{Q}_{k-1}, f_k+1\}$). This means that we are trying to find better candidates from the points in Zero($\{f_1 + e_1, \ldots, f_{k-1} + e_{k-1}, f_k + 1\}$). If Result($\{\mathbb{Q}_{k-1}, f_k + 1\}$) $= \{1\}$, it implies that $\{f_1 + e_1 = 0, \ldots, f_{k-1} + e_{k-1} = 0, f_k + 1 = 0\}$ is unsatisfied by any points in $\mathbb{F}_2^n$. Obviously, we cannot find better candidates in this case. When Result($\{\mathbb{Q}_{k-1}, f_k + 1\}$) $\neq \{1\}$, we execute Step 4.1.3.1-4.1.3.3 to incrementally solve the following polynomials $\{f_{k+1}, \ldots, f_m\}$ as the operations in Step 1-3. In this procedure, if the value of the cost function u corresponding to $[e_1, \ldots, e_i]$ is not smaller than u_{bound}, we have to interrupt the incrementally solving procedure, and backtrack to e_{i-1}. If we successfully complete the incrementally solving process, we will achieve a better candidate $\mathbb{Q}_n$. Then we replace the old $\mathbb{S}$ by $\mathbb{Q}_n$ and refresh the value of u_{bound}. After these operations, we return to Step 4 to backtrack and search again.

From the above statement, we can know that before Step 5 we exhaustively search all the possible values of $[e_1, e_2, \ldots, e_m]$ and solve the corresponding polynomial systems $\{f_1 + e_1, f_2 + e_2, \ldots, f_m + e_m\}$ in order to find the optimal candidate. We only cut off some branches in the following cases, and we will prove that we cannot achieve a better candidate in these cases .

(a) We obtain $\{1\}$ when incrementally solving $\{f_1 + e_1, \ldots, f_{k-1} + e_{k-1}, f_k\}$. In this case, Zero($\{f_1 + e_1, \ldots, f_{k-1} + e_{k-1}, f_k\} = \emptyset$ and we cannot find any solution from Zero($\{f_1 + e_1, \ldots, f_{k-1} + e_{k-1}, f_k, f_{k+1} + e_{k+1}, \ldots, f_m + e_m\}$, where $e_1, \ldots, e_{k-1}$ are fixed and $e_{k+1}, \ldots, e_m$ can be any values. Thus, we cut off these branches and only consider the branches with $e_k = 1$.
(b) In Step 4.1.3.2, $u \geq u_{bound}$. This implies that the candidates in Zero($\{f_1 + e_1, \ldots, f_{i-1} + e_{i-1}, f_i + 1\}$ will not be better than the stored one. Thus, we cut off the following branches and backtrack to e_{i-1}.
(c) In Step 4.1.2, Result($\mathbb{P}_k$) $= \{1\}$. This means that Zero($\{f_1 + e_1, \ldots, f_{k-1} + e_{k-1}, f_k + 1\}$) $= \emptyset$. Similarly as case (a), we cut off the following branches because we cannot find any solution from them.

(d) In Step 4.1, when we want to flip e_k from 0 to 1, we find $u + c_k \geq u_{bound}$. This case is similar as case (b). $u + c_k \geq u_{bound}$ implies that the candidates in Zero($\{f_1 + e_1, \ldots, f_{k-1} + e_{k-1}, f_k + 1\}$ will not be better than the stored one, thus we cut off the following branches and backtrack to e_{k-1}.

The loop of Step 4 ends when $k = 0$ which means that we have exhaustively searched all possible branches except the redundant ones and the candidate we stored is the best one among all the points in $\mathbb{F}_2^n$.

Finally, we obtain solutions from $\mathbb{S}$ by Step 5, and this procedure is very easy when $\mathbb{S}$ is some triangular sets or a Gröbner Basis. In most time when $m > n$, S has very simple structure which only contain several points. □

Remark 1. Step 4.1.1 can be changed in Algorithm 1. We only need to set $e_k = 1$, $u = u + c_k$, $\mathbb{Q}_k = \mathbb{Q}_{k-1}$ without solving $\{\mathbb{Q}_{k-1}, f_k + 1\}$. This means that we skip the polynomial f_k in the incremental solving process. Actually, if we achieve a better candidate from the points in Zero($\mathbb{Q}_k$) in the following process, all points in this candidate must satisfy $f_k + 1 = 0$. Suppose $f_k = 0$ for some point in this candidate, then we have $f_1 + e_1 = 0, \ldots, f_{k-1} + e_{k-1} = 0, f_k = 0, f_{k+1} + e_{k+1} = 0, \ldots, f_m + e_m = 0$ hold on this point. Since $f_k = 0$, this point should already be contained in a candidate of the previous process. However, this point is from a better candidate, which means that it is better than itself and leads to a contradiction. This proves the correctness of our modification. Intuitively, after this modification, the algorithm will be more efficient since we don't need to compute Result($\{\mathbb{Q}_{k-1}, f_k + 1\}$), but the opposite is true. From experiments we found that this modification will reduce the efficiency of our algorithm. The reason is that constraint $f_k + 1 = 0$ makes the point sets considered smaller which will accelerate the following compute. More importantly, if Result($\mathbb{P}_k$) = $\{1\}$ we can cut off this backtracking branch instantly. If we consider solving Max-PoSSo over a big finite field, this modification may have some advantage, but this is beyond the scope of this article.

Theoretically estimating the complexity of **ISBS** is very difficult. The only thing we know is that the number of paths in the whole search tree is bounded by 2^m. However, when contradictions occur in the algebraic solving process, a lot of subtree will be cut off. Thus, in our experiments the numbers of paths are much smaller than 2^m.

3.2 The Characteristic Set Method in $\mathbb{F}_2$

It is easy to see that the efficiency of the algebraic solving algorithm will significantly influence the efficiency of the whole algorithm. In our implementation of the **ISBS** method, we use the **MFCS** algorithm as the algebraic solving algorithm. The **MFCS** algorithm is an variant of the Characteristic Set(CS) method for solving the boolean polynomial systems, and it is efficient in the case of incrementally solving. In this subsection, we will simply introduce the **MFCS** algorithm. More details of it can be found in [6].

For a boolean polynomial $P \in \mathbb{F}_2[x_1, x_2, \ldots, x_n]/ < x_1^2 + x_1, x_2^2 + x_2, \ldots, x_n^2 + x_n >$, the *class* of P, denoted as $\mathrm{cls}(P)$, is the largest index c such that x_c occurs in P. If P is a constant, we set $\mathrm{cls}(P)$ to be 0. If $\mathrm{cls}(P) = c > 0$, we call x_c the *leading variable* of P, denoted as $\mathrm{lvar}(P)$. The leading coefficient of P as a univariate polynomial in $\mathrm{lvar}(P)$ is called the *initial* of P, and is denoted as $\mathrm{init}(P)$.

A sequence of nonzero polynomials

$$\mathcal{A}: \quad A_1, A_2, \ldots, A_r \tag{1}$$

is a *triangular set* if either $r = 1$ and $A_1 = 1$ or $0 < \mathrm{cls}(A_1) < \cdots < \mathrm{cls}(A_r)$. A boolean polynomial P is called *monic*, if $\mathrm{init}(P) = 1$. Moreover, if the elements of a triangular set are all monic, we call it a monic triangular set.

Algorithm 2. MFCS($\mathbb{P}$)

Input: A finite set of polynomials $\mathbb{P}$.
Output: Monic triangular sets $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_t\}$ such that $\mathrm{Zero}(\mathbb{P}) = \cup_{i=1}^{t} \mathrm{Zero}(\mathcal{A}_i)$ and $\mathrm{Zero}(\mathcal{A}_i) \cap \mathrm{Zero}(\mathcal{A}_j) = \emptyset$

1 Set $\mathbb{P}^* = \{\mathbb{P}\}$, $\mathcal{A}^* = \emptyset$ and $\mathcal{C}^* = \emptyset$.
2 While $\mathbb{P}^* \neq \emptyset$ do
  2.1 Choose a polynomial set $\mathbb{Q}$ from $\mathbb{P}^*$.
  2.2 While $\mathbb{Q} \neq \emptyset$ do
    2.2.1 If $1 \in \mathbb{Q}$, $\mathrm{Zero}(\mathbb{Q}) = \emptyset$. Goto Step 2.1.
    2.2.2 Let $\mathbb{Q}_1 \subset \mathbb{Q}$ be the polynomials with the highest class.
    2.2.3 Let $\mathbb{Q}_{monic} = \emptyset$, $\mathbb{Q}_2 = \mathbb{Q} \setminus \mathbb{Q}_1$.
    2.2.4 While $\mathbb{Q}_1 \neq \emptyset$ do
      Let $P = Ix_c + U \in \mathbb{Q}_1$, $\mathbb{Q}_1 = \mathbb{Q}_1 \setminus \{P\}$.
      $\mathbb{P}_1 = \mathbb{Q}_{monic} \cup \mathbb{Q}_2 \cup \mathbb{Q}_1 \cup \{I, U\}$.
      $\mathbb{P}^* = \mathbb{P}^* \cup \{\mathbb{P}_1\}$.
      $\mathbb{Q}_{monic} = \mathbb{Q}_{monic} \cup \{x_c + U\}$, $\mathbb{Q}_2 = \mathbb{Q}_2 \cup \{I + 1\}$.
    2.2.5 Let $Q = x_c + U$ be a polynomial with lowest degree in $\mathbb{Q}_{monic}$.
    2.2.6 $\mathcal{A} = \mathcal{A} \cup \{Q\}$.
    2.2.7 $\mathbb{Q} = \mathbb{Q}_2 \cup \{R \neq 0 | R = Q_i + Q, Q_i \in \mathbb{Q}_{monic}\}$.
  2.3 if $\mathcal{A} \neq \emptyset$, set $\mathcal{A}^* = \mathcal{A}^* \cup \{\mathcal{A}\}$.
3 Return $\mathcal{A}^*$

With the **MFCS** algorithm, we can decompose $\mathrm{Zero}(\mathbb{P})$, the common zero set of a polynomial set $\mathbb{P}$, as $\cup_i \mathrm{Zero}(\mathcal{A}_i)$, the union of the common zero sets of some monic triangular sets $\mathcal{A}_i$. We first convert all polynomials into monic ones by the following decomposition formula: $\mathrm{Zero}(Ix_c + U) = \mathrm{Zero}(Ix_c + U, I + 1) \cup \mathrm{Zero}(I, U) = \mathrm{Zero}(x_c + U, I + 1) \cup \mathrm{Zero}(I, U)$. Note that, with decomposition we will generate some new polynomial sets, and we call these new polynomial sets to be new components. Then we can choose one monic polynomial $x_c + R$ to eliminate the x_c of other polynomials by doing addition $x_c + R + x_c + R_1 = R + R_1$. Note that $R + R_1$ is a polynomial with lower class. Therefore, we can obtain the

following group of polynomial sets $\{x_c + R, \mathbb{P}'\}, \mathbb{P}_1, \ldots, \mathbb{P}_t$, where $\mathbb{P}'$ is a set of polynomials with class lower than c and each $\mathbb{P}_i$ is a new generating polynomial set. Then we can recursively apply the above operations to the polynomials with highest class in $\mathbb{P}'$. After dealing with all classes, we will obtain a monic triangular set or constant 1, and generate a group of new polynomial sets. Then we recursively apply the above operations to every new set. Finally, we will obtain the monic triangular sets we need. Obviously, for a monic triangular set $\{x_1 + c_1, x_2 + f_1(x_1), x_3 + f_2(x_2, x_1), \ldots, x_n + f_{n-1}(x_{n-1}, \ldots, x_1)\}$, we can easily solve it.

MFCS has the following properties[6]:

1. The size of polynomials occurring in the whole algorithm can be controlled by that of the input ones. The expansion of the internal result will not happen. Note that in different components most polynomials are same, and the same ones can be shared in the memory with data structure SZDD[11]. For the above reasons, the memory cost of **MFCS** is small.
2. **MFCS** can solve one component very fast. The bitwise complexity of solving one component is $O(LMn^{d+2})$, where L is the number of input polynomials, n is the number of variables, d is the highest degree of the input polynomials and M is the maximal number of terms for all input polynomials. Obviously, when d is fixed, this is a polynomial about n.

4 Experimental Results for Attacking AES and Serpent

According to Section 2.2 we can model the Cold Boot key recovery problems of AES and Serpent as Partial Weighted Max-PoSSo problems. We applied the **ISBS** method to solve these problems and compared our results with those shown in [1]. As in [1], we focused on the 128-bit versions of the two ciphers.

The benchmarks are generated the same way as those in [1]. The experimental platform is a PC with i7 2.8Ghz CPU(only one core is used), and 4G Memory. For every instance of the problem we performed 100 experiments with randomly generated keys, and we only used a reduced round of key schedule. In the first two groups of experiments, we also set δ_1 to be 0.001 as [1,8,12] and used the "aggressive" modelling in most time, where we assume $\delta_1 = 0$ instead of $\delta_1 = 0.001$. In the "aggressive" modelling, all equations with $K_i' = 1$ should be satisfied, so they should be added into the set $\mathcal{H}$ and the problem reduces to Partial Max-PoSSo. Note that, the input data in our experiments are generated with $\delta_1 > 0$, thus in "aggressive" modelling the equations in $\mathcal{H}$ with $K_i = 1$ may be incorrect.

In the following tables, the line "**ISBS**" shows the results of attacking AES and Serpent by using **ISBS**. The line "**SCIP**" shows the results in [1] where they used the **MIP** solver **SCIP** for solving these problems. The column "aggr" denotes whether we choose the aggressive ("+") or normal ("-") modelling. As in [1], we also interrupted the solver when the running time exceeded the time limit. The column "r" gives the success rate, which is the percentage of the

instances we recovered the correct key. There are two cases in which we cannot recover the correct key.

- We interrupted the solver after the time limit.
- The optimal solution we achieved from the (Partial weighted) Max-PoSSo problems is not the correct key. In "aggressive" modelling, when some polynomial in $\mathcal{H}$ is incorrect, this will always happen. When all polynomials in $\mathcal{H}$ are correct, if we added polynomials in $\mathbb{F}_{\mathcal{E}}$ which are the checking polynomials into the set $\mathcal{H}$, the optimal solution will always be the correct key. However, as mentioned before we didn't do this in order to decrease the running time. Thus, with a quite low probability, the optimal solution may not be the correct key.

Table 1. AES considering N rounds of key schedule output

δ_0	Method	N	aggr	limit t	r	min t	avg. t	max t
0.15	**ISBS**	4	+	60.0 s	75%	0.002 s	0.07 s	0.15 s
	SCIP	4	+	60.0 s	70%	1.78 s	11.77 s	59.16 s
0.30	**ISBS**	4	+	3600.0 s	70%	0.002 s	0.14 s	2.38 s
	SCIP	4	+	3600.0 s	69%	4.86 s	117.68 s	719.99 s
0.35	**ISBS**	4	+	3600.0 s	66%	0.002 s	0.27 s	7.87 s
	SCIP	4	+	3600.0 s	68%	4.45 s	207.07 s	1639.55 s
0.40	**ISBS**	4	+	3600.0 s	58%	0.002 s	0.84 s	20.30 s
	SCIP	4	+	3600.0 s	61%	4.97 s	481.99 s	3600.00 s
	SCIP	5	+	3600.0 s	62%	7.72 s	704.33 s	3600.00 s
0.50	**ISBS**	4	+	3600.0 s	23%	0.002 s	772.02 s	3600.00 s
	ISBS	5	+	3600.0 s	63%	0.003 s	1.05 s	46.32 s
	SCIP	4	+	3600.0 s	8%	6.57 s	3074.36 s	3600.00 s
	SCIP	4	+	7200.0 s	13%	6.10 s	5882.66 s	7200.00 s

Table 1 presents the results of attacking AES. For attacking AES, we didn't use the normal("-") modelling.[3] In the aggressive modelling, by adding some intermediate variables we convert the S-box polynomials with degree 7 into the quadratic polynomials[3]. Since these intermediate quadratic polynomials must be satisfied, we add them into set $\mathcal{H}$. For these problems from AES, after we solving $\mathcal{H}$, the point in Zero(Result($\mathcal{H}$)) is small, finding an optimal one from this set is not too hard. Thus, most of the running time is spent on solving $\mathcal{H}$. When we use more rounds of key schedule output, we can solve $\mathcal{H}$ faster. This explains why the result of $N = 5$ is better than that of $N = 4$ in the case of $\delta_0 = 0.5$. From the results of **SCIP**, it seems that more rounds will lead to a worse result. From the results of Table 1, we can see that, for the easy problems, with keeping the same success rate, the running time of **ISBS** is much shorter.

[3] Since the first round key of AES is its initial key, we have 128 polynomials which have form $x_i + 1$ or x_i. In this case, using **ISBS** to search the value of polynomials is equal to exhaustively searching the value of variables.

For the hard problem, the success rate of **ISBS** is higher, and the running time of it is shorter.

In the aspect of attacking cipher, our results are worse than those in [12]. Actually, as mentioned before, if we use all rounds of the key schedule, the running times of **ISBS** will be much shorter and close to the running times in [12]. However, using all rounds will make Zero(Result($\mathcal{H}$)) only contain one point, and this means that we just need to solve a PoSSo problem instead of a Partial Max-PoSSo problem. Hence, for testing the efficiency of **ISBS** for solving Partial Max-PoSSo problems, we only use 4 or 5 rounds of key schedule and achieved these poorer attack results.

Table 2. SERPENT considering $32 \cdot N$ bits of key schedule output

δ_0	Method	N	aggr	limit t	r	min t	avg. t	max t
0.05	**ISBS**	8	−	600.0 s	90%	0.41 s	58.49 s	600.00 s
	SCIP	12	−	600.0 s	37%	8.22 s	457.57s	600.00 s
0.15	**ISBS**	12	+	60.0 s	81%	1.19 s	3.82 s	60.00 s
	SCIP	12	+	60.0 s	84%	0.67 s	11.25 s	60.00 s
	SCIP	16	+	60.0 s	79%	0.88 s	13.49 s	60.00 s
0.30	**ISBS**	16	+	600.0 s	81%	4.73 s	11.66 s	58.91 s
	SCIP	16	+	600.0 s	74%	1.13s	57.05 s	425.48 s
0.50	**ISBS**	20	+	3600.0 s	55%	14.11 s	974.69 s	3600.00 s
	SCIP	16	+	3600.0 s	38%	136.54 s	2763.68 s	3600.00 s

The results of attacking Serpent are given in Table 2. In the normal modelling with $\delta_0 = 0.05$, we set $N = 8$. The reason is that in this modelling we don't have polynomials in $\mathcal{H}$, and more input polynomials will make us search more possible values of these polynomials. If $N = 4$, we can get an optimal result very fast, but it isn't the correct key in most time. The reason is that the randomness of the first round of the Serpent key schedule is poor. By setting $N = 8$, we have a good success rate and short running times. In the aggressive modelling, the situations are similar to those in AES. For **SCIP**, when N is larger, the results are worse. For **ISBS**, more bits of key schedule will make it solve $\mathcal{H}$ easier. Therefore, when δ_0 increase we set N larger.

Table 3 presents the results when considering symmetric noise(i.e., $\delta_0 = \delta_1$). This is a pure Max-PoSSo problem. As mentioned before, when $\mathcal{H} = \emptyset$, with less equations **ISBS** can be more efficient. Thus, in this group of experiments, we set $N = 8$. When solving these problems, we used some important tricks to accelerate the computation. These tricks will be introduced in the appendix detailedly. From the results, we can see that **ISBS** has higher success rates and shorter running time comparing to **SCIP**.

If we don't use the incremental solving and backtracking search method, which means that we only exhaustively search the value of polynomials in the order of increasing Hamming-weight and solve the corresponding polynomial systems, when $\delta_0 = \delta_1 = 0.05$ the running time will be about $\binom{256}{0.05 \cdot 256} \cdot T_0 \approx 2^{71} \cdot$

Table 3. SERPENT considering $32 \cdot N$ bits of key schedule output(symmetric noise)

$\delta_0 = \delta_1$	Method	N	limit t	r	min t	avg. t	max t
0.01	**ISBS**	8	3600.0 s	100%	0.78 s	9.87 s	138.19 s
	SCIP	12	3600.0 s	96%	4.60 s	256.46 s	3600.00 s
0.02	**ISBS**	8	3600.0 s	96%	0.80 s	163.56 s	3600.00 s
	SCIP	12	3600.0 s	79%	8.20 s	1139.72 s	3600.00 s
0.03	**ISBS**	8	3600.0 s	90%	1.74 s	577.60 s	3600.00 s
	SCIP	12	7200.0 s	53%	24.57 s	4205.34 s	7200.0 s
0.05	**ISBS**	8	3600.0 s	38%	12.37 s	917.05 s	3600.00 s
	SCIP	12	3600.0 s	18%	5.84 s	1921.89 s	3600.00 s

T_0, where T_0 is the time for solving a polynomial system with 128 variables and 256 equations. This polynomial system can be easily solved, because of the easy invertibility of the key schedule operations. From our experiments with 100 instances, the average value of T_0 is 0.30 seconds, then $2^{71} \cdot T_0 \approx 2^{69.3}$ seconds which is much larger than our result $917.05 \approx 2^{9.8}$ seconds.[4] This implies that by using incremental solving and backtracking search method, we avoid a lot of repeated computation and cut off a lot of redundant branches which highly improve the efficiency of searching.

5 Conclusion

In this paper, we proposed a new method called **ISBS** for solving the family of Max-PoSSo problems over $\mathbb{F}_2$, and applied the method in solving the Cold Boot key recovery problems of AES and Serpent. Our work is inspired by the work in [1], and provide a new way of solving the non-linear polynomials system with noise. Our method was combined with the Characteristic Set method, which is a powerful tool in symbolic computation and has good performances on solving the boolean polynomial systems. The main innovation of **ISBS** is the combination of incremental solving and backtracking search. By this idea, we can use the former results to reduce the repeated computations and use the conflictions to cut off a lot of redundant branches. Our experimental data shows that **ISBS** has good performances on solving the Cold Boot key recovery problems of AES and Serpent, and its results are better than the previously existing ones by using **SCIP** solver.

References

1. Albrecht, M., Cid, C.: Cold Boot Key Recovery by Solving Polynomial Systems with Noise. In: Lopez, J., Tsudik, G. (eds.) ACNS 2011. LNCS, vol. 6715, pp. 57–72. Springer, Heidelberg (2011)

[4] With the tricks introduced in appendix, for every instance in our experiments, we can search and solve all branches by **ISBS** within the time limit under the condition that the number of unsatisfied polynomials $\leq 0.05 \cdot 256 \approx 13$.

2. Chai, F., Gao, X.S., Yuan, C.: A Characteristic Set Method for Solving Boolean Equations and Applications in Cryptanalysis of Stream Ciphers. Journal of Systems Science and Complexity 21(2), 191–208 (2008)
3. Courtois, N.T., Pieprzyk, J.: Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In: Zheng, Y. (ed.) ASIACRYPT 2002. LNCS, vol. 2501, pp. 267–287. Springer, Heidelberg (2002)
4. Faugère, J.C.: A New Efficient Algorithm for Computing Gröbner Bases (F4). Journal of Pure and Applied Algebra 139(1-3), 61–88 (1999)
5. Faugère, J.C.: A New Efficient Algorithm for Computing Gröner Bases Without Reduction to Zero (F5). In: Proc. ISSAC, pp. 75–83 (2002)
6. Gao, X.S., Huang, Z.: Efficient Characteristic Set Algorithms for Equation Solving in Finite Fields. Journal of Symbolic Computation 47(6), 655–679 (2012)
7. Håstad, J.: Satisfying Degree-d Equations over $GF[2]^n$. In: Goldberg, L.A., Jansen, K., Ravi, R., Rolim, J.D.P. (eds.) APPROX/RANDOM 2011. LNCS, vol. 6845, pp. 242–253. Springer, Heidelberg (2011)
8. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest We Remember: Cold Boot Attacks on Encryption Keys. In: USENIX Security Symposium, pp. 45–60. USENIX Association (2009)
9. Huang, Z., Lin, D.: Attacking Bivium and Trivium with the Characteristic Set Method. In: Nitaj, A., Pointcheval, D. (eds.) AFRICACRYPT 2011. LNCS, vol. 6737, pp. 77–91. Springer, Heidelberg (2011)
10. Huang, Z.: Parametric Equation Solving and Quantifier Elimination in Finite Fields with the Characteristic Set Method. Jounral of Systmes Science and Complexity 25(4), 778–791 (2012)
11. Minto, S.: Zero-Sppressed BDDs for Set Manipulation in Combinatorial Problems. In: Proc. ACM/IEEE Design Automation, pp. 272–277. ACM Press (1993)
12. Kamal, A.A., Youssef, A.M.: Applications of SAT Solvers to AES key Recovery from Decayed Key Schedule Images. In: Proceedings of the Fourth International Conference on Emerging Security Information, Systems and Technologies, SECURWARE 2010, Venice/Mestre, Italy, July 18-25 (2010)
13. Wu, W.T.: Basic Principles of Mechanical Theorem-proving in Elementary Geometries. Journal Automated Reasoning 2, 221–252 (1986)
14. Zhao, S.W., Gao, X.S.: Minimal Achievable Approximation Ratio for MAX-MQ in Finite Fields. Theoretical Computer Science 410(21-23), 2285–2290 (2009)

Appendix: Some Tricks Used in ISBS for Solving Symmetric Noisy Problems

In this section, we will introduce some tricks we used in solving the problems in Table 3. For this kind of problems, these tricks can greatly improve the efficiency of **ISBS**.

(I) In **ISBS**, if u_{bound} is smaller, the branches we need to search is less, and the algorithm will end faster. When u_{min}, the value of the cost function corresponding to the optimal solution, is small, we can increasingly set the u_{bound} as $k, 2k, 3k, \ldots, u_0$. Here k is a value which is determined by the number of input polynomials, and u_0 is the value of the cost function corresponding to $\mathbb{Q}_n$ which

is generated from Step 2. Then we try to find a solution under these bounds. Precisely speaking, in Step 3, we set $\mathbb{S} = \emptyset$ and $u_{bound} = mk$, $m = 1, 2, \ldots$ If **ISBS** returns a nonempty solution when $u_{bound} = mk < u_0$ for some m, then it is the optimal solution of the problem. Otherwise, we set $u_{bound} = (m+1)k$ if $(m+1)k < u_0$; or $u_{bound} = u_0$ if $(m+1)k \geq u_0$. Then we execute Step 3-5 of **ISBS** again until we achieve a nonempty solution. If we cannot achieve a nonempty solution, $\mathbb{Q}_n$ from Step 2 will be the optimal solution. The disadvantage of this modification is that in the case of $u_{bound} = (m+1)k$ we need to search a lot of branches which have already be searched in the case of $u_{bound} = mk$. If u_{min} is big, the cost of repeated computation is big. That is why we only use this modification when u_{min} is small.

(II) Note that, in our experiments, we used 256 bits of key schedule output, while the key is 128 bits. Thus the input polynomial system has 128 variables and 256 polynomials. In the experiments, after we had incrementally solved the first 128 polynomials with any assignment of $[e_1, \ldots, e_{128}]$, we always obtained a result which only contains several points. Then the process of solving and backtracking the following 128 polynomials is very easy. For example, if $\mathrm{Zero}(\{f_1+e'_1, \ldots, f_{128}+e'_{128}\}) = \{x_0\}$ for some $[e'_1, \ldots, e'_{128}]$, where x_0 is a point in $\mathbb{F}_2^n$. Then $[e_{129}, \ldots, e_{256}]$ must be equal to $[f_{129}(x_0), \ldots, f_{256}(x_0)]$. Any flip of $e_i, i = 129, \ldots, 256$ will lead to $\mathrm{Result}(\mathrm{Result}(\{f_1+e'_1, \ldots, f_{128}+e'_{128}\}), f_i+e_i+1) = 1$, so this branch will be cut off instantly.

Based on the above observation, given a u_{bound}, the running time of **ISBS** is almost equal to the time of searching and solving $\{f_1+e_1, f_2+e_2, \ldots, f_{128}+e_{128}\}$ where $[e_1, \ldots, e_{128}]$ satisfies $u(e_1, e_2, \ldots, e_{128}) \leq u_{bound}$ and $u(e_1, e_2, \ldots, e_{128})$ is the value of the cost function corresponding to $[e_1, e_2, \ldots, e_{128}]$. The cost of the operations about the following 128 polynomials can be ignored.

For an assignment $[e_1^0, e_2^0, \ldots, e_{256}^0]$ of $[e_1, e_2, \ldots, e_{256}]$ satisfying $u(e_1^0, e_2^0, \ldots, e_{256}^0) \leq u_{bound}$, we have $u(e_1^0, \ldots, e_{128}^0) \leq \frac{1}{2}u_{bound}$ or $u(e_{129}^0, \ldots, e_{256}^0) \leq \frac{1}{2}u_{bound}$. Instead of searching all the branches satisfying $u(e_1, e_2, \ldots, e_{128}) \leq u_{bound}$, we can search the branches satisfying $u(e_1, e_2, \ldots, e_{128}) \leq \frac{1}{2}u_{bound}$ or $u(e_{129}, e_{130}, \ldots, e_{256}) \leq \frac{1}{2}u_{bound}$. To do this, we can use **ISBS** one time to solve $\{f_1, \ldots, f_{128}, f_{129}, \ldots, f_{256}\}$ under the condition of $u(e_1, e_2, \ldots, e_{128}) \leq \frac{1}{2}u_{bound}$, then use **ISBS** another time to solve $\{f_{129}, \ldots, f_{256}, f_1, \ldots, f_{128}\}$ under the condition of $u(e_{129}, e_{130}, \ldots, e_{256}) \leq \frac{1}{2}u_{bound}$. To understand why this modification can accelerate the computation, we need the following two lemmas.

Lemma 1. *Let a, b, n be positive integers, and $n > a+b$. Then* $\sum_{i=0}^{a}\binom{n}{i} + \sum_{i=0}^{b}\binom{n}{i} \geq \sum_{i=0}^{\lceil\frac{a+b}{2}\rceil}\binom{n}{i} + \sum_{i=0}^{\lfloor\frac{a+b}{2}\rfloor}\binom{n}{i}$. *The equality holds if and only if $a = b$ or $a + b = n - 1$.*

Proof: Obviously, when $a = b$ the equality holds. If $a + b = n - 1$, $\sum_{i=0}^{a}\binom{n}{i} + \sum_{i=0}^{b}\binom{n}{i} = \sum_{i=0}^{a}\binom{n}{i} + \sum_{i=n-b}^{n}\binom{n}{i} = \sum_{i=0}^{a}\binom{n}{i} + \sum_{i=a+1}^{n}\binom{n}{i} = 2^n$. $\sum_{i=0}^{\lceil\frac{a+b}{2}\rceil}\binom{n}{i} + \sum_{i=0}^{\lfloor\frac{a+b}{2}\rfloor}\binom{n}{i} =$

$\sum_{i=0}^{\lceil\frac{n-1}{2}\rceil}\binom{n}{i}+\sum_{i=n-\lfloor\frac{n-1}{2}\rfloor}^{n}\binom{n}{i}=\sum_{i=0}^{\lceil\frac{n-1}{2}\rceil}\binom{n}{i}+\sum_{i=\lceil\frac{n-1}{2}\rceil+1}^{n}\binom{n}{i}=2^n$. Thus, the equality holds when $a+b=n-1$.

If $a\neq b$ and $a+b<n-1$, we can assume $a>b$ without loss of generality. Then it is equal to prove

$$\binom{n}{b+1}+\binom{n}{b+2}+\cdots+\binom{n}{\lfloor\frac{a+b}{2}\rfloor}<\binom{n}{\lceil\frac{a+b}{2}\rceil+1}+\binom{n}{\lceil\frac{a+b}{2}\rceil+2}+\cdots+\binom{n}{a}. \tag{2}$$

Note that both sides of the inequality have the same number of terms.

- If $a\leq\lfloor\frac{n}{2}\rfloor$, obviously we have $\binom{n}{b+1}<\binom{n}{\lceil\frac{a+b}{2}\rceil+1},\binom{n}{b+2}<\binom{n}{\lceil\frac{a+b}{2}\rceil+2},\ldots,\binom{n}{\lfloor\frac{a+b}{2}\rfloor}<\binom{n}{a}$. Summing up all these inequalities, we can obtain (2).
- If $a>\lfloor\frac{n}{2}\rfloor$, then we can divide the right part of (2) into two parts

$$\binom{n}{\lceil\frac{a+b}{2}\rceil+1}+\cdots+\binom{n}{\lfloor\frac{n}{2}\rfloor},\binom{n}{\lfloor\frac{n}{2}\rfloor+1}+\cdots+\binom{n}{a}, \tag{3}$$

and also divide the left part of (2) into two parts

$$\binom{n}{a+b-\lfloor\frac{n}{2}\rfloor+1}+\cdots+\binom{n}{\lfloor\frac{a+b}{2}\rfloor},\binom{n}{b+1}+\cdots+\binom{n}{a+b-\lfloor\frac{n}{2}\rfloor}. \tag{4}$$

The first parts of (3) and (4) have the same number of terms, and the second parts of (3) and (4) also have the same number of terms. Since $\lfloor\frac{n}{2}\rfloor>\lfloor\frac{a+b}{2}\rfloor$, we have $\binom{n}{a+b-\lfloor\frac{n}{2}\rfloor+1}<\binom{n}{\lceil\frac{a+b}{2}\rceil+1},\ldots,\binom{n}{\lfloor\frac{a+b}{2}\rfloor}<\binom{n}{\lfloor\frac{n}{2}\rfloor}$. Then $\binom{n}{a+b-\lfloor\frac{n}{2}\rfloor+1}+\cdots+\binom{n}{\lfloor\frac{a+b}{2}\rfloor}<\binom{n}{\lceil\frac{a+b}{2}\rceil+1}+\cdots+\binom{n}{\lfloor\frac{n}{2}\rfloor}$. Then it is sufficient to prove that the second part of (3) is not less than that of (4). We have $\binom{n}{\lfloor\frac{n}{2}\rfloor+1}+\binom{n}{\lfloor\frac{n}{2}\rfloor+2}+\cdots+\binom{n}{a}=\binom{n}{\lceil\frac{n}{2}\rceil-1}+\binom{n}{\lceil\frac{n}{2}\rceil-2}\cdots+\binom{n}{n-a}=\binom{n}{n-a}+\binom{n}{n-a+1}+\cdots+\binom{n}{\lceil\frac{n}{2}\rceil-1}$. Since $n-a>b+1$, we have $\binom{n}{n-a}>\binom{n}{b+1},\ldots,\binom{n}{\lceil\frac{n}{2}\rceil-1}>\binom{n}{a+b-\lfloor\frac{n}{2}\rfloor}$. This proves the inequality (2). □

Furthermore, we can prove the following lemma similarly as Lemma 1,

Lemma 2. *Let a,b,n be positive integers. Assume $n>a+b$ and $a>b$. Then $\sum_{i=0}^{a}\binom{n}{i}+\sum_{i=0}^{b}\binom{n}{i}\geq\sum_{i=0}^{a-t}\binom{n}{i}+\sum_{i=0}^{b+t}\binom{n}{i}$, where t is an integer satisfying $0<t<\frac{a-b}{2}$. The equality holds if and only if $a+b=n-1$.*

According to Lemma 1, we have $\sum_{i=0}^{u_{bound}}\binom{128}{i}>\sum_{i=0}^{\frac{1}{2}u_{bound}}\binom{128}{i}+\sum_{i=0}^{\frac{1}{2}u_{bound}}\binom{128}{i}$. In the problems of Table 3, the value of the cost function is the number of unsatisfying equations. Thus, the inequality implies that the number of branches satisfying

$u(e_1, e_2, \ldots, e_{128}) \leq u_{bound}$ is larger than the number of branches satisfying $u(e_1, e_2, \ldots, e_{128}) \leq \frac{1}{2}u_{bound}$ or $u(e_{129}, e_{130}, \ldots, e_{256}) \leq \frac{1}{2}u_{bound}$ without considering cutting off branching. For example, if $u_{bound} = 10$, then $\sum_{i=0}^{u_{bound}} \binom{128}{i} \approx 2^{47.8}$ and $\sum_{i=0}^{\frac{1}{2}u_{bound}} \binom{128}{i} + \sum_{i=0}^{\frac{1}{2}u_{bound}} \binom{128}{i} \approx 2^{29.0}$. Obviously, this is a remarkable improvement.

However, in the Serpent problems, solving a branch with input $\{f_{129}, \ldots, f_{256}, f_1, \ldots, f_{128}\}$ is slower than solving a branch with input $\{f_1, \ldots, f_{128}, f_{129}, \ldots, f_{256}\}$. The reason is that $f_1, \ldots, f_{256}$ is a polynomials sequence which is sorted from the "simplest" one to the "most complex" one. In this order incremental solving can be more efficient. A better strategy is considering the branches satisfying $u(e_1, e_2, \ldots, e_{128}) \leq \frac{6}{10}u_{bound}$ or $u(e_{129}, e_{130}, \ldots, e_{256}) \leq \frac{4}{10}u_{bound}$. From Lemma 2, we know that under this strategy the branches we need to solve are still much less than those in the case of $u(e_1, e_2, \ldots, e_{128}) \leq u_{bound}$. From experiments, we found that the total running time under this strategy will be smaller than that under the strategy of considering $u(e_1, e_2, \ldots, e_{128}) \leq \frac{1}{2}u_{bound}$ or $u(e_{129}, e_{130}, \ldots, e_{256}) \leq \frac{1}{2}u_{bound}$.

In the experiments of Table 3 when $\delta_0 = 0.01, 0.02, 0.03$, we used trick (I). For the problems with $\delta_0 = 0.05$, we used both trick (I) and trick (II).