CoTree: Push the Limits of Conquerable Space in Collision-Optimized Side-Channel Attacks*

Changhai Ou[1], Debiao He[1], Zhu Wang[2], Kexin Qiao[3], Shihui Zheng[4], and Siew-Kei Lam[5]

[1] School of Cyber Science & Engineering, Wuhan University, Wuhan, China
`ouchanghai@whu.edu.cn, hedebiao@whu.edu.cn`

[2] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
`wangzhu@iie.ac.cn`

[3] School of Cyberspace Science & Technology, Beijing Institute of Technology, Beijing, China
`qiao.kexin@bit.edu.cn`

[4] School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing, China
`shihuizh@bupt.edu.cn`

[5] School of Computer Science & Engineering, Nanyang Technological University, Singapore
`assklam@ntu.edu.sg`

Abstract

By introducing collision information into side-channel distinguishers, the existing collision-optimized attacks exploit collision detection algorithm to transform the original candidate space under consideration into a significantly smaller collision chain space, thus achieving more efficient key recovery. However, collision information is detected very repeatedly since collision chains are created from the same sub-chains, i.e., with the same candidates on their first several sub-keys. This aggravates when exploiting more collision information. The existing collision detection algorithms try to alleviate this, but the problem is still very serious. In this paper, we propose a highly-efficient detection algorithm named Collision Tree (CoTree) for collision-optimized attacks. CoTree exploits tree structure to store the chains creating from the same sub-chain on the same branch. It then exploits a top-down tree building procedure and traverses each node only once when detecting their collisions with a candidate of the sub-key currently under consideration. Finally, it launches a bottom-up branch removal procedure to remove the chains unsatisfying the collision conditions from the tree after traversing all candidates (within given threshold) of this sub-key, thus avoiding the traversal of the branches satisfying the collision condition. These strategies make our CoTree significantly alleviate the repetitive collision detection, and our experiments verify that it significantly outperforms the existing works.

Contents

*Corresponding author: Changhai Ou.

1 Introduction

Secret information will unintentionally leak through side-channels such as execution time [9], power consumption [23], electromagnetic [7] and cache patterns [11] when cryptographic algorithms are executed on devices. These side-channel information can be collected by adversaries and they can launch the Side-Channel Attacks (SCAs) in two models: divide-and-conquer (e.g., Correlation Power Analysis (CPA) [4] and Template Attack (TA) [5]) and analytical (e.g., collision attacks [13]). For the former, they divide the full-key into small blocks (e.g., sub-keys in AES-128) and conquer them one by one. For the latter, they consider all sub-keys simultaneously, and by solving a system of equations. It exploits more leaky information and is more efficient than divide-and-conquer attacks, but is also more complex.

It is worth mentioning that the key recovery in the above two models is limited by the computing power of the adversaries. Benefitting from side-channel information, if we rank the candidates of each sub-key from the most possible one to the least possible one, the correct one is usually ranked the front. However, due to the limited side-channel information, it may not be the best. In this case, key enumeration strategies [8, 10, 19], which enumerate the full-key candidates from the most possible one to the least possible one, are exploited to optimize the key recovery. However, they are still limited by the computing power of the attacker, and only can be exploited in the scenarios that cryptographic implementations are "practically insecure" (for which the leakage allows for key enumeration).

Recent years, several collision-optimized attacks introduced collision information into divide-and-conquer attacks (e.g., [3, 16, 22]), thus establishing the relationship between sub-keys and breaking their independence. Different from key enumeration schemes, they only consider a part of the best candidates for each sub-key and collision value in two combined attacks (e.g. CPA and Correlation-Enhanced Collision Attack (CECA) [3]). The candidates unsatisfying the collision conditions are discarded, thus key recovery can still be feasibly applied on the remaining collision space much smaller than the one provided by divide-and-conquer attack. In other words, they transform the original candidate space into a much smaller collision chain space. Related works will be introduced in the next subsection before introducing our contributions.

1.1 Related Works

Collision-optimized attacks rank the candidates of each collision value and sub-key from the most possible one to the least possible one for the analytic collision attack and the divide-and-conquer attack to be optimized. Then, they set two thresholds for these two combined attacks and only consider the candidates within them. The first work exploiting collision information to optimize the key recovery of side-channel attacks was given in [3]. Since then, several collision-optimized attacks have been proposed and their goals can be classified into three categories: collision exploitation, fault tolerance and repetitive collision detection alleviation.

For collision exploitation, the first collision-optimized attack named Test-of-Chain (abbreviated to TOC for convenience here) was given in [3]. Take AES-128 for an example, TOC exploits the collisions between each two adjacent sub-keys, thus 15 pairs of collisions are exploited and most collisions are wasted. This also happens in [17], which exploits nearly 40 pairs of collisions without fault tolerance. To improve this, Full-Collision Chain (FCC) was given in [15] to quickly exploit all collisions. It can provide us with the smallest collision space, thus the remaining key recovery is the most efficient. However, any collision beyond the threshold will lead to the failure of key recovery and this collision condition is very strict. We have to face very huge (sometimes even unconquerable) collision threshold in this case.

For fault tolerance, the first collision-optimized attack with fault tolerance strategy named Fault-Tolerant Chain (FTC) was given in [22]. Take AES-128 for an example, it exploits the 15 collisions between the first sub-key and the remaining 15 sub-keys and discards the other 105 pairs of collisions, just like TOC. The fault tolerance here means, if there is no collision between a sub-key and the first sub-key, FTC enumerates its candidates beyond the threshold from the most possible one to the least possible one until finds a collision. It is obvious that the $2^{nd} \sim 16^{th}$ sub-keys are still independent of each other in this case, which will lead to very huge collision space. The authors in [16] stated that the collision threshold could be significantly reduced by performing fault tolerance on a small number of collisions (i.e., allowing them to be beyond the threshold of collision attack), and provided a flexible fault tolerant strategy with high performance verified by their experiments.

For repetitive collision detection alleviation, this only happens in the collision detections where the collision information between the coming sub-key and several sub-keys in the chains is utilized. For example, Full-Collision Chain (FCC) exploits the collision information between the current sub-key and all sub-keys on the chain. Since some chains are often built from the same sub-chain, i.e., the candidates of these sub-keys expect for the last one are the same, and the collisions between them and the coming sub-key will be repeatedly detected in this case. Another example is that only a pair of collision between the sub-key currently under consideration and the chains is exploited in TOC and FTC, so no repetitive collision detection happens. This disadvantage will significantly affect the detection performance and the Light-weight Collision Detection (LCD) algorithm given in [16] exploits a highly-efficient jump collision detection mechanism to optimize this. Specifically, LCD considers the longest common sub-chain in the current chain set and detects collisions between the candidates of the coming sub-key and it only once (see Section 3.3 for details). In this case, it efficiently reduces repetitive collision detection.

It is worth mentioning that our goal is not to improve the performance of single side-channel distinguishers like the improved collision attack in [2] and the genetic algorithm-based sieve built from CPA in [6], rather than exploit collisions to optimize the key recovery. There are also several algorithms like the improved CECA in [24] that try to remove a part of candidates to recover the key easier. Although they are also interesting and important, they deviate from the purpose of this paper. Therefore, we do not consider them here.

1.2 Our Contributions

This paper proposes a collision detection algorithm named Collision Tree (CoTree) for collision-optimized attacks to more efficiently alleviate the repetitive collision detections. Our main contributions are as follows:

1. CoTree exploits tree structure to store the long chains established from the same short chain on the same branch, thus facilitating the traversal in nodes insertion and removal.

2. CoTree exploits a top-down tree building procedure to significantly alleviate the repetitive collision detection, since the nodes on the tree need to be traversed only once for each candidate (within threshold) of the sub-key currently under consideration. This advantage becomes more significant under much larger thresholds.

3. CoTree launches a bottom-up branch removal procedure to remove the candidates unsatisfying the collision conditions after considering all candidates (within the threshold) of the coming sub-key, which avoids repetitive traversal of the branches satisfying the collision condition and makes the complexity smaller than the number of nodes.

Our experiments verify that compared to the existing works, our CoTree significantly alleviates the burden from repetitive collision detection, and achieves much better performance.

1.3 Organization

The rest of this paper is organized as follows: TA and CECA are introduced in Section 2. The existing detection algorithms for collision-optimized attacks such as TOC, FTC, FCC and LCD, are presented in Section 3. Our CoTree and its optimizations are detailed in Sections 4 and 5. Experiments on DPA *contest* $v4.1$ dataset and an AT89S52 micro-controller are presented in Section 6 to illustrate the superiority of our CoTree. Finally, we conclude this paper in Section 7.

2 Preliminaries

Let $\kappa^* = (\kappa_1^*, \kappa_2^*, \ldots, \kappa_{16}^*)$ denote the key of AES-128 and $x^i = \left(x_1^i, x_2^i, \ldots, x_{16}^i\right)$ denote the i^{th} encrypted plaintext. Suppose that the attacker encrypts n plaintexts $\mathbb{X} = \left(x^1, x^2, \ldots, x^n\right)$ and collects n power traces $\mathbb{T} = \left(t^1, t^2, \ldots, t^n\right)$. The classic CPA, TA and CECA are given in Sections 2.1$\sim$ 2.3.

2.1 Correlation Power Analysis

CPA [4] is a well-known model-dependent distinguisher, for which the attacker needs a hypothesis model, e.g., Hamming weight model, to exploit side-channel leakage information. Here the corresponding hypothesis power consumption of the intermediate values under a guessing sub-key κ_j can be denoted as $\mathbb{I}^{\kappa_j} = \mathsf{Hw}\left(\mathsf{Sbox}\left(\mathbb{X}_j \oplus \kappa_j\right)\right)$. CPA calculates the correlation coefficient between it and the real power consumption $\mathbb{T}$, and returns the sub-key candidate corresponding to the maximum correlation coefficient ρ:

$$\kappa_j^{'} = \arg\max_{\kappa_j} \{\rho\left(\mathbb{I}^{\kappa_j}, \mathbb{T}\right) | \kappa_j = 0, 1, \ldots, 255\}. \tag{1}$$

Here "$\mathsf{Sbox}(\cdot)$" denotes S-box operation, "$\mathsf{Hw}(\cdot)$" denotes Hamming weight operation and "$\mathbb{X}_j$" denotes the j^{th} bytes of the encrypted plaintexts.

2.2 Template Attack

The classic TA [5, 20] assumes that the attacker has the same device as the targeted one, and it includes two phases: leakage profiling and exploitation. For a cryptographic device with leakage following 9 different normal distributions for 9 Hamming weights of S-boxes outputs of AES-128, the attacker calculates the mean:

$$\mathbf{m}_j^i = \frac{1}{n}\sum_{v=1}^{n} t_j^v \tag{2}$$

and covariance matrix:

$$\mathbf{C}_j^i = \frac{1}{n}\sum_{v=1}^{n} \left(t^v - \mathbf{m}_j^i\right)\left(t^v - \mathbf{m}_j^i\right)^{\mathsf{T}}, \tag{3}$$

to profile templates $\left(\mathbf{m}_j^i, \mathbf{C}_j^i\right)$ $(1 \le i \le 9)$ for the j^{th} S-box in the profiling stage. Here the symbol "T" denotes matrix transposition. If the attacker collects a power trace t^* from the targeted device, the probability that it well matches the i^{th} template $\left(\mathbf{m}_j^i, \mathbf{C}_j^i\right)$ is:

$$p\left(t^* | \mathbf{m}_j^i, \mathbf{C}_j^i\right) = \frac{e^{-\frac{\left(\mathbf{m}_j^i - t^*\right)\cdot\left(\mathbf{C}_j^i\right)^{-1}\cdot\left(\mathbf{m}_j^i - t^*\right)^{\mathsf{T}}}{2}}}{\sqrt{(2\cdot\pi)^{|\mathbf{m}_j^i|}\,\mathbf{det}\left(\mathbf{C}_j^i\right)}}. \tag{4}$$

Here $|\cdot|$ is the size of a variable. To achieve better performance, TA is usually performed on Points-Of-Interest (POIs) with obvious leakage (see [20] for details).

2.3 Correlation-Enhanced Collision Attack

An AES-128 linear collision [3] occurs if two S-boxes in the same AES encryption or different encryptions receive the same byte value as their inputs:

$$\mathsf{Sbox}(x_{j_1}^{i_1} \oplus \kappa_{j_1}) = \mathsf{Sbox}(x_{j_2}^{i_2} \oplus \kappa_{j_2}). \tag{5}$$

This can be optimized as:

$$\delta_{j_1,j_2} = \kappa_{j_1} \oplus \kappa_{j_2} = x_{j_1}^{i_1} \oplus x_{j_2}^{i_2}. \tag{6}$$

CECA [13] divides the samples of the j_1-th and j_2-th sub-keys into 256 classes respectively according to their plaintext byte values, then computes the correlation coefficient:

$$\rho\left\{\left(\mathbf{m}_{j_1}^{\beta}, \mathbf{m}_{j_2}^{\beta\oplus\delta_{j_1,j_2}}\right) | \beta = 0, 1, 2, \ldots, 255\right\} \tag{7}$$

between them under a guessing δ_{j_1,j_2}.

3 Existing Detection Algorithms for Collision-Optimized Attacks

3.1 Collision Chain

A cryptographic system often exploits a very huge key candidate space to against exhaustion (e.g., 2^{128} in AES-128), thus we cannot traverse all candidates in attacks. This happens in key enumeration [10, 19], e.g., 2^{40} key candidates may take about one day on a desktop computer. This also happens in collision-optimized attacks. Let τ_κ and τ_d be the thresholds of two combined attacks in collision-optimized attacks, which means we only consider the best τ_κ candidates of each sub-key and the best τ_d candidates for each δ. For collision-optimized attacks, the detection algorithms mean that we will try to recover the key from them. Actually, if we detect m pairs of collisions, then a collision system with m linear equations is obtained:

$$\begin{cases} \kappa_{j_1} \oplus \kappa_{j_2} = \delta_{(\kappa_{j_1},\kappa_{j_2})}, \\ \kappa_{j_3} \oplus \kappa_{j_4} = \delta_{(\kappa_{j_3},\kappa_{j_4})}, \\ \vdots \\ \kappa_{j_{2m-1}} \oplus \kappa_{j_{2m}} = \delta_{(\kappa_{j_{2m-1}},\kappa_{j_{2m}})}. \end{cases} \tag{8}$$

It can be divided into several sub-systems with independent variables, and each of them contains collision information of several sub-keys and a free variable. They are named as collision chain.

Collision chains break the independence of sub-keys and establish the relationship among them, thus transferring the original candidate space within τ_κ into a significantly smaller collision space. Due to the high efficiency of collision-optimized attacks, the key can be quickly recovered from a very huge candidate space, which has been well illustrated in [15, 18, 22]. Therefore, both τ_κ and τ_d can be set to large values, and the space under τ_κ can be even much larger than the space acceptable in key enumeration. This conclusion is also verified in our experiments.

3.2 Test-of-Chain and Fault-Tolerant Chain

For convenience, we exploit $\kappa_{j_1} \leftrightarrow \kappa_{j_2}$ to denote the collision between κ_{j_1} and κ_{j_2}, which means these two sub-keys and their collision value $\delta_{(\kappa_{j_1},\kappa_{j_2})}$ are within thresholds simultaneously. As the first collision-optimized attack, Test-of-Chain (TOC) detects the collisions between any two adjacent sub-keys $\kappa_1 \leftrightarrow \kappa_2$, $\kappa_2 \leftrightarrow \kappa_3$, $\cdots$, $\kappa_{14} \leftrightarrow \kappa_{15}$ and $\kappa_{15} \leftrightarrow \kappa_{16}$ in turn. Finally, it obtains chains containing information of all 16 sub-keys. TOC is a milestone, since it starts a new direction of key recovery for side-channel attacks. Obviously, TOC exploits only 15 pairs of collisions and discards the other 105 pairs of collisions. Moreover, TOC does not include any fault tolerant strategy, and any collision values falling out of threshold τ_d will make it fail.

Different from TOC, another collision-optimized attack named Fault-Tolerant Chain (FTC) in [22] tries to find the collisions $\kappa_1 \leftrightarrow \kappa_2$, $\kappa_1 \leftrightarrow \kappa_3$, $\ldots$, $\kappa_1 \leftrightarrow \kappa_{15}$ and $\kappa_1 \leftrightarrow \kappa_{16}$ between the first sub-key and the remaining 15 sub-keys. If there is no collision between κ_1 and the candidates within τ_κ of κ_j $(2 \le j \le 16)$, FTC then enumerates κ_j $(2 \le j \le 16)$ beyond threshold from the most possible to the least possible until finds one. However, this only happens with a very small probability when both τ_κ and τ_d are very large, such as the ones considered in this paper. Similar to TOC, FTC also exploits only 15 pairs of collisions and wastes the other 105 pairs of collisions.

3.3 Full-Collision Chain and Light-weight Collision Detection Algorithm

Full-Collision Chain (FCC) in [15] aims to maximize the use of collision information. It detects the collisions between the i^{th} sub-key and all the first $i-1$ sub-keys, and saves the chains with all $i\cdot(i-1)/2$ pairs of collisions. However, this collision condition is very strict and the authors provided a fault tolerant strategy to allow a small part of collisions beyond the threshold τ_d. Another shortage of FCC is that it does not contain any mechanism for its repetitive collision detection, and too many collisions are repetitively detected. However, TOC and FTC only exploit the collision between the sub-key currently under consideration and one of the sub-keys on the chains, repetitive collision detection does not exist in this case.

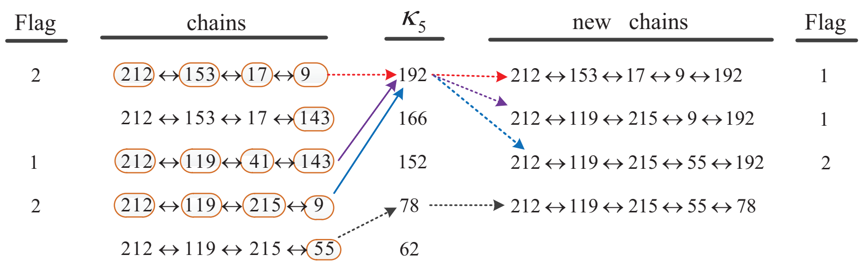

Figure 1: Jump collision detection mechanism in LCD.

Benefits from its jump detection mechanism, LCD in [16] efficiently alleviates repetitive collision detection in FCC. Suppose the current state of collision chain detection is as shown in Figure 1 and LCD exploits an array "Flag" to record the number of chains established from a collision chain. It can be seen that there are 2, 1 and 2 chains built from sub-chains $212 \leftrightarrow 153 \leftrightarrow 17$, $212 \leftrightarrow 119 \leftrightarrow 41$ and $212 \leftrightarrow 119 \leftrightarrow 215$, respectively. The collisions built from the same sub-chains need to be detected only once when considering the 5 candidates of the newly coming sub-key κ_5 within threshold τ_κ. LCD jumps over the detection of the corresponding sub-chains according to the step length in array "Flag". In other words, we only need to detect the candidates on chains with circles on the left. Obviously, the more chains built from a sub-chain, the longer the step, and the more effective the mechanism is. The authors in [16] provided a highly-efficient fault tolerant strategy for LCD, and we will provide a short introduction in Section 4.5 when exploiting it in our CoTree algorithm.

4 CoTree Algorithm

Existing solution LCD takes advantage of its jump collision detection strategy, it can effectively alleviate the repetitive collision detection. However, limited by its data structure and algorithm design, this problem is still very serious in LCD. In this section, we will give a new algorithm named **Collision Tree (CoTree)** to more efficiently alleviate the repetitive collision detection. Our CoTree is stored as an array, and each node takes a row. The five fields of each node on this tree are as follows:

- **Data**: candidate of a sub-key within threshold τ_κ.

- **Parent**: identifier (i.e., the row number) of its parent node.
- **Level**: level of this node on the tree. For example, the level of a candidate of the i^{th} sub-key is i.
- **First Child**: identifier (i.e., the row number) of the first child of this node (if exists).
- **Degree**: The number of branches of this node, i.e., the number of children of this node.

The operations related to our CoTree includes: initialization, node insertion and removal when considering candidates of a coming sub-key, and chain reading after the consideration of the 16^{th} sub-key. All these above operations in our CoTree will be introduced in detail in Sections 4.1$\sim$ 4.4.

4.1 Tree Initialization

Our CoTree is based on tree structure, and we should first consider how to exploit a collision tree to store collision information (i.e., collision chains). A simple scheme is to construct a tree for each candidate of the first sub-key within the threshold τ_κ. Specifically, we first detect the collisions between the first and second sub-keys. If one of the candidates of κ_1 has a collision, we take it as the root node to build a tree. In this case, there are at most τ_κ and at least 0 trees in our CoTree algorithm. It is worth noting that we can detect whether there are new nodes inserted into the tree after node removal. If not, it means that there is no candidate satisfying the collision condition at the current level and this tree is removed. Tree initialization will be efficient in this case.

4.2 Node Insertion

The creation of CoTree is a process of constantly detecting collisions and building taller trees. We have introduced TOC and FTC in Section 3.2, which only exploit 15 pairs of collisions, resulting in a waste of most of collision information. FCC, LCD, and our CoTree introduced in this section will aim to maximize the use of collision information. Specifically, for our CoTree, we detect the collisions between all nodes from leaf to root on the tree and the candidates of the coming sub-key within τ_κ.

Let $\mathcal{T}$ denote a CoTree stored in an array, K_i denote the candidates of the sub-key κ_i currently under consideration, Δ-s denote the collision values returned by CECA. Both sub-key candidates and collision candidates are ranked in descending order according to their possibility to be the correct sub-key and collision value. Thresholds τ_d and τ_κ are for the analytic CECA and the divide-and-conquer attack to be optimized, which means we only consider the best candidates within both of them as explained before. i is the index of the i^{th} sub-key. The corresponding node insertion operations of our CoTree are shown in Algorithm 1.

First, the positions to insert new nodes are the leaves of the tree, and we exploit an index j to point the last one of them in our algorithm (Step 1 in Algorithm 1). For each candidate in K_i of the sub-key κ_i currently under consideration, we initialize an array $Coll$ to record the number of collisions for all nodes in the tree (Step 3). Specifically, we set a collision flag for each node, and compute the location of 256 collision values loc from its level and i (Step 5). We further detect whether a collision happens, i.e., the collision value between $\mathsf{K}_i\ (ix)$ and $\mathcal{T}\,(kx, 1)$ is within the threshold τ_d located by loc in Δ. If the collision occurs, the collision flag is set to 1 (Steps 6$\sim$10).

Algorithm 1: Node Insertion Operations.

```
Input: Collision tree $\mathcal{T}$, candidates $\mathsf{K}_i$ of the sub-key $\kappa_i$ currently under consideration,
       collision values $\Delta$-s, Thresholds $\tau_d$ and $\tau_\kappa$, $i$.
Output: The updated collision tree $\mathcal{T}$.
1  $j \leftarrow$ size $(\mathcal{T})$; $index \leftarrow j$ ;
2  for $ix$ from 1 to $\tau_\kappa$ do
3      Initialize an array: $Coll\,(1 \ldots j) \leftarrow 0$;
4      for $kx$ from 2 to $j$ do
5          $flag \leftarrow 0$; $loc \leftarrow$ Location $(\mathcal{T}\,(kx, 3)\,, i)$;
6          for $mx$ from 1 to $\tau_d$ do
7              if $\mathsf{K}_i\,(ix) \oplus \mathcal{T}\,(kx, 1) = \Delta_{(mx,loc)}$ then
8                  $flag \leftarrow 1$;
9              end
10         end
11         if $flag$=1 then
12             $Coll\,(kx) \leftarrow Coll\,(\mathcal{T}\,(kx, 2)) + 1$ ;
13         else
14             $Coll\,(kx) \leftarrow Coll\,(\mathcal{T}\,(kx, 2))$ ;
15         end
16         if $\mathcal{T}\,(kx, 3) = i - 1$ and $Coll\,(kx) = i - 1$ then
17             $index \leftarrow index + 1$;
18             $\mathcal{T}\,(index, 1 : 5) \leftarrow (\mathsf{K}_i\,(ix)\,, kx, i, 0, 0)$ ;
19             $\mathcal{T}\,(kx, 5) \leftarrow \mathcal{T}\,(kx, 5) + 1$ ;
20             if $\mathcal{T}\,(kx, 5) = 1$ then
21                 $\mathcal{T}\,(kx, 4) \leftarrow index$ ;
22             end
23         end
24     end
25 end
```

Secondly, node insertion in our CoTree is a top-down procedure, which ensures each node on the tree only be traversed once when detecting the collision between them and a candidate value waiting for insertion. This effectively reduces the repetitive collision detection. It is worth noting that, to make full use of collision information, our CoTree passes the number of collisions as a parameter from top to bottom. In other words, each node inherits the number of collisions from its parent (Steps 11 $\sim$ 15).

Finally, whether the candidate value $\mathsf{K}_i\,(ix)$ could be a new leaf of a branch is decided by the collision conditions (Steps 16 $\sim$ 18). Here we exploit the collision information passed from root to current leaf as the collision condition. $Coll\,(kx) = i - 1$ means all $i \cdot (i-1)/2$ collisions are exploited, since each node inherits the number of collisions from its parent at Steps 11 $\sim$ 15, all collisions between a sub-key and its former sub-keys are considered in this case. If $\mathsf{K}_i\,(ix)$ satisfies the collision condition, the number of branches of its parent node increases by 1 (Step 19). If the new node is the first child of its parent, the corresponding information is updated (Steps 20 $\sim$ 22). The node insertion finishes after Algorithm 1.

4.3 Node Removal

Suppose that the current state of a CoTree is as shown in Figure 2 and the corresponding array is shown in Table 1. The identifiers in Table 1 are table indexes. Therefore, the array exploited in our CoTree only contains five fields as explained at the beginning of Section 4.

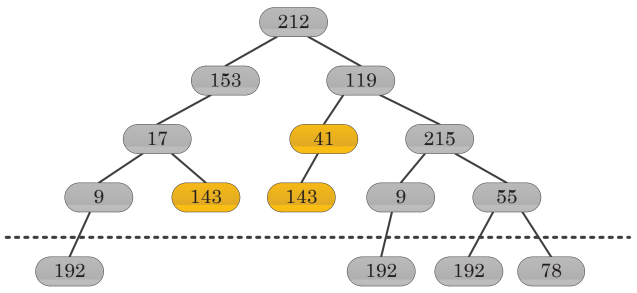

Figure 2: A CoTree containing candidates of the first 5 sub-keys.

Table 1: A CoTree containing collision information of the first 5 sub-keys.

Identifier	Candidates	Parent	Level	First child	Degree
1	212	-1	1	2	2
2	153	1	2	4	1
3	119	1	2	5	2
4	17	2	3	7	2
5	41	3	3	9	1
6	215	3	3	10	2
7	9	4	4	12	1
8	143	4	4	0	0
9	143	5	4	0	0
10	9	6	4	13	1
11	55	6	4	14	2
12	192	7	5	0	0
13	192	10	5	0	0
14	192	11	5	0	0
15	78	11	5	0	0

Node insertion is a **top-down** procedure. However, for each leaf, if we traverse the tree and remove the nodes unsatisfying collision conditions in this model, all non-leaf nodes from it to root have to be traversed once. Obviously, this pulls our CoTree back to the worst case. In this case, we launch a **bottom-up** procedure to optimize this. Specifically, we exploit τ_f to record the last node on the penultimate layer. The bottom-up node removal means that starting from the τ_f-th node, we traverse the collision tree $\mathcal{T}$ from the back to the front, and the nodes without any branch are deleted. The corresponding node removal operations are shown in Algorithm 2.

First, we record the identifier of the last node τ_f on the $n-1$ level of CoTree $\mathcal{T}$ to i, and start the bottom-top node removal. Before removing a node without any branches, we exploit symbols j and kx to record its parent's identifier and the number of nodes on the tree,

Algorithm 2: Node Removal Operations.

```
Input: Collision tree $\mathcal{T}$, the identifier of the last node $\tau_f$ in $n-1$ level.
Output: The updated collision tree $\mathcal{T}$.
1  $i \leftarrow \tau_f$ ;
2  while $i \geq 2$ do
3      if $\mathcal{T}(i,5) = 0$ then
4          $j \leftarrow \mathcal{T}(i,2)$; $kx \leftarrow$ size $(\mathcal{T})$;
5          $\mathcal{T}(j,5) \leftarrow \mathcal{T}(j,5) - 1$ ;
6          if $\mathcal{T}(j,4) = i$ and $\mathcal{T}(j,5) = 0$ then
7              $\mathcal{T}(j,4) \leftarrow -1$ ;
8          else if $\mathcal{T}(j,4) = i$ and $\mathcal{T}(j,5) > 0$ then
9              $\mathcal{T}(j,4) \leftarrow i$ ;
10         end
11         $\mathcal{T}(j+1:kx,4) \leftarrow \mathcal{T}(j+1:kx,4) - 1$ ;
12         $\mathcal{T}(i,1:5) \leftarrow []$ ;
13         $j \leftarrow \mathcal{T}(i,4)$; $k \leftarrow$ size $(\mathcal{T})$;
14         if $\mathcal{T}(i,5) > 0$ then
15             $\mathcal{T}(j:kx,2) \leftarrow \mathcal{T}(j:kx,2) - 1$;
16         end
17     end
18     $i = i - 1$ ;
19 end
```

and remove the branch information of the node from its parent (Steps 4∼5). If its parent has branches and the node is the first child, its nearest brother becomes the new first child (Steps 8∼9). It is worth noting that, when we delete the i^{th} node, its nearest brother (i.e., the $(i+1)$-th node) becomes the new i^{th} node, so we set the first child of its parent to i rather than $i-1$.

Consequently, we perform the node removal operation. The first children of all non-leaf nodes behind the parent of the removed node should be moved forward one position (Step 11). For example, all the non-leaf nodes 214, 9, 9 and 55 behind the parent 41 of the 9^{th} node 143 should be moved forward one position when it is removed (as shown in Figure 2). We update the number of nodes on the tree to parameter kx after this deletion. If the new i^{th} node has more than one branch, the parents' information of all subsequent nodes is updated accordingly. For example, the 10^{th} node 9 becomes the new 9^{th} node after removing the 9^{th} node 143, the parents from its child 192 to the last node 78 need to be moved forward one position.

The above operations guarantee the correctness of the relationship among the remaining nodes on our CoTree before and after removing nodes. Finally, the variable i always indicates the node to be considered. It only needs to move forward one position no matter whether this node is deleted or not. All nodes unsatisfying the collision conditions are removed after the iteration. For example, all yellow nodes in Figure 2 that do not meet the collision conditions are removed, and the remaining gray nodes constitute the updated collision tree in this case.

4.4 Collision Chain Extraction

All collision chain detection algorithms will eventually extract the collision chains as full-key candidates for verification. For example, the plaintext can be encrypted by using these key candidates according to certain rules (e.g., their ranked probabilities or number of collisions).

If the corresponding ciphertext is generated in some encryption, we will consider the candidate as the correct key. The existing collision detection algorithms continuously construct long chains from short ones. The collision chains need to be extracted once after considering all candidates within threshold τ_κ of the sub-key currently under consideration, and the new chains are saved in the array. It is worth mentioning that the extraction of collision chains in our CoTree is only performed once after finishing the node insertion of the 16^{th} sub-key, so this will not bring too much computation.

4.5 Complexity

The complexity of collision-optimized attacks changes dynamically, and we may achieve very different complexity under the same thresholds τ_d and τ_κ in different attacks. Here we first analyze the space complexity of CoTree in two extreme cases. The first extreme case is that there is only one chain on CoTree. CoTree not only consumes more space but also consumes more time compared to FCC and LCD in this case. However, this makes no sense because the key is usually unrecoverable. Another extreme case is that our CoTree includes:

$$\mathcal{S}(n,\tau_\kappa)=\frac{(1-\tau_\kappa^n)}{1-\tau_\kappa} \tag{9}$$

nodes, which means it is a full τ_κ tree. Here "n" is 15 for AES-128 since we create a tree for each candidate of the first sub-key within the threshold τ_κ. There will be at most τ_κ^n chains for FCC and LCD, and the space consumed is $16\cdot\tau_\kappa^n$ in this case.

Suppose that each field is with the same size, the more required space of LCD and FCC compared to CoTree is:

$$\mathcal{G}(\tau_\kappa)=16\cdot\tau_\kappa^{15}-5\cdot\frac{\left(1-\tau_\kappa^{16}\right)}{1-\tau_\kappa} \tag{10}$$

in the second extreme case, since there are 5 fields for each node on our CoTree. Obviously,

$$\mathcal{G}(\tau_\kappa)=\frac{16\cdot\tau_\kappa^{15}(\tau_\kappa-1)-5\cdot\left(\tau_\kappa^{16}-1\right)}{\tau_\kappa-1}, \tag{11}$$

and this gives a new function $\mathcal{G}'(\tau_\kappa)$ as:

$$\mathcal{G}'(\tau_\kappa)=11\cdot\tau_\kappa^{16}-16\cdot\tau_\kappa^{15}. \tag{12}$$

Obviously, "$\tau_\kappa\geq 1.45$" well guarantees that our CoTree consumes less space than FCC and LCD if it is a full τ_κ tree. Although CoTree is not always a full tree satisfying the above equations, this space advantage becomes more significant with the increase of τ_d and τ_κ.

It is noteworthy that the advantage we want to emphasize here is not space complexity, but the time complexity. It is also more intuitive and easier to compare. First, when considering the $(i+1)$-th sub-key, LCD only considers the case where several chains share the longest prefixes (i.e., their first $i-1$ sub-keys are with the same candidates). Our CoTree considers all possible prefixes thus it can be more efficiently alleviate the repetitive collision detection during each node insertion. Moreover, the bottom-up procedure guarantees that we should check all nodes only once in node removal operations at the worst case, and this only happens after all nodes are inserted. Therefore, most of the computation of our CoTree takes place in node insertion operations since it traverses all nodes on the tree to check their collisions with each candidate within τ_κ of the sub-key currently under consideration. Obviously, the larger the threshold, the less time consumption of CoTree compared to FCC and LCD, and this advantage will be further verified in our experiments in Section 6.

5 Optimization

We consider the collisions between a candidate of the sub-key currently under consideration and all the nodes on the tree, and then consider the next candidate in Algorithm 1, rather than the collisions between nodes on a branch and all the candidates of this coming sub-key. This is to maintain the ascending order of node identifiers on the tree, so as to facilitate the removal of nodes unsatisfying the collision conditions given in Section 4.3. The nodes of our CoTree will be traversed τ_κ times in this case just like FCC and LCD. Moreover, the latter removal of the nodes unsatisfying the collision conditions will be performed only once as explained in Section 4.5. It is obvious that these operations will significantly reduce the repetitive collision detection compared to FCC and LCD and accelerate the candidate space transformation. We will further optimize our CoTree in several ways to enhance its performance in Sections 5.1∼5.3 .

5.1 Branch Removal

The advantage of Algorithm 2 is that, to check whether all nodes on the tree satisfy the collision conditions, it only needs to traverse them once. This effectively alleviates the repetitive collision detection, especially for prefix sub-chains sharing the first several sub-keys candidates. Since it is unnecessary to traverse the CoTree when removing nodes unsatisfying the collision conditions. Specifically, if a node on the penultimate layer has one or several children (i.e. leaves) , this well illustrates that all nodes from its child (or children) to the root well satisfy the collision conditions. Therefore, we only need to judge the state of the penultimate nodes. If a node does not have any leaf node as its child, this branch should be removed recursively from the bottom to the root.

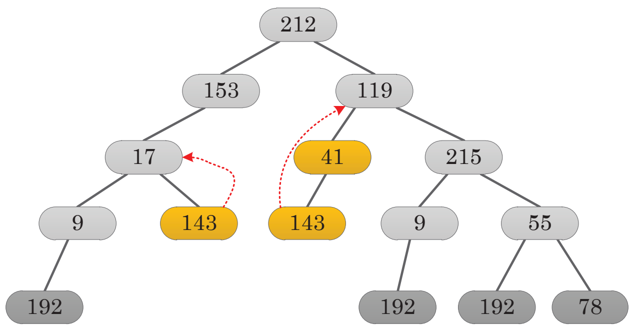

Figure 3: Recursive deletion of branches unsatisfying collision conditions.

To illustrate the above node removal optimization, we give the corresponding diagram of the tree traversal in Figure 3. Nodes 55 and 9 have children, which means that the nodes from them to the root and children all well satisfy the collision conditions, so only their states need to be detected. In other words, there is no need to traverse their corresponding branches from bottom to top (i.e., to the root node). Node 143 has no children. Before removing it, we need to determine whether its parent node 41 has only one child. If so, this parent node should be removed, too. The process proceeds recursively until node 119, since it has a child 215 and the recursion stops. It is obvious that this mechanism is more efficient than Algorithm 2 and it can avoid the traversal of some nodes, and this is especially for very large CoTree-s.

5.2 Tree Merging

We take the candidates of the first sub-key within the threshold τ_κ as the roots in Section 4.1, thus the algorithm creates at most τ_κ trees. Actually, we can add an additional root to these trees to make the candidates of the first sub-key fall to the second layer, since all operations of these trees are the same. This will facilitate the processing of all sub-keys when they are considered. It is worth noting that there is no real collision between the root and the candidates of the first sub-key in this case, we can directly insert these candidates as the leaves in initialization.

5.3 Fault Tolerance

Suppose that 120 pairs of collisions between 16 sub-keys in the first round of AES-128 returned by CECA are shown in Figure 4. Specifically, the abscissa value "$15 \cdot i + (i-1) \cdot (i-2) + (j-i)$" represents the collision information of the i^{th} and j^{th} sub-keys ($i \leq j$), while the ordinate represents the position of the collision value after ranking. For example, the abscissa of the collision value between the first two sub-keys κ_1 and κ_2 is 1, and the ordinate is its ranking position 26. It can be seen that the deepest collision value is 136, which means that, to recover the key, we need to set τ_d to at least 136 for all 120 pairs of collision values if without any fault tolerance. However, such a huge threshold is usually unacceptable when performing collision chain detection.

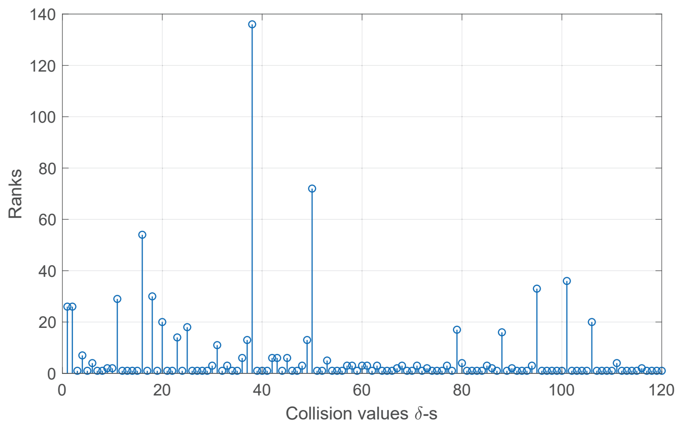

Figure 4: 120 collision values between 16 sub-keys of AES-128.

To solve the above problem, the collision detection algorithm should contain fault tolerance mechanism, thus reducing the threshold τ_d. To illustrate this, we further give collision values corresponding to Figure 4 but ranked beyond threshold 20 in Table 2. Here "$[20, 40)$" means the collision value is ranked greater than or equal to 20, but less than 40. It vividly shows that

6, 1, 1 and 1 collisions fall into the ranges: $20 \sim 40$, $40 \sim 60$, $60 \sim 80$, and ≥ 80, respectively. If we can accurately identify the positions of $\delta_{(\kappa_2,\kappa_3)}$, $\delta_{(\kappa_3,\kappa_{12})}$ and $\delta_{(\kappa_4,\kappa_{12})}$, τ_d can be reduced from 136 to 40, thus significantly reducing the collision space. Although it is usually very difficult for us to achieve such a goal, this has fully illustrated the significance of fault tolerance mechanisms in collision-optimized attacks.

Table 2: An example of collision values ranked beyond threshold $\tau_d = 20$.

Ranks	Collisions
$[20, 40)$	$\delta_{(\kappa_1,\kappa_2)}$, $\delta_{(\kappa_1,\kappa_3)}$, $\delta_{(\kappa_1,\kappa_{12})}$ $\delta_{(\kappa_2,\kappa_5)}$, $\delta_{(\kappa_9,\kappa_{12})}$, $\delta_{(\kappa_{10},\kappa_{12})}$
$[40, 60)$	$\delta_{(\kappa_2,\kappa_3)}$
$[60, 80)$	$\delta_{(\kappa_4,\kappa_{12})}$
$[80, 140)$	$\delta_{(\kappa_3,\kappa_{12})}$

The goal of this paper is to propose CoTree to significantly alleviate the repetitive collision detection, not to design a fault tolerance strategy specially for it. Therefore, we exploit the existing fault tolerance mechanism with high efficiency proposed in [15, 18] in CoTree. Specifically, we record the number of collisions of all the current chains in our CoTree, and extract the chains with a number of collisions satisfying:

$$\tau_c = \mathcal{C}_{max} - \frac{i}{\epsilon} \tag{13}$$

according to the capability of attacker. The symbol "$\mathcal{C}_{max}$" is the maximum number of collisions considering all chains, i indicates the i^{th} sub-key, and ϵ is fault tolerance coefficient. Since the later a sub-key is considered, the more collisions it contains, and the more of them should be fault tolerated.

The above fault-tolerant strategies in our CoTree are as follows: 1) On the basis of Algorithm 1, an array is initialized to record the collision information among nodes on the tree and all candidates within threshold τ_κ of the sub-key currently under consideration; 2) We add the number of collisions as the 6^{th} field of each node. The collision information of each node is transferred to its children and accumulated; 3) The candidates satisfying the threshold given in Equation 13 are extracted from the array and become new leaf nodes. It is possible that the nodes on the penultimate layer still have no children, i.e., the collision conditions may still be unsatisfied. Therefore, we still have to use the optimized Algorithm 2 introduced in Section 5.1 to remove these nodes.

6 Experimental Results

To illustrate the performance of CoTree, we first exploit CECA to optimize CPA, and perform the first experiment on DPA Contest V4.1 dataset in Section 6.1. Then, we further extend CECA to the profiled setting and exploit it to optimize TA, the corresponding experiment is performed on an AT89S52 micro-controller in Section 6.2. All above experiments are performed on MATLAB $R2016b$ on a DELL desktop computer with 6 Intel(R) Core(TM) i5-9500 CPUs, 16 GB RAM and a Windows 10 operating system. The widely exploited evaluation metric, Success Rate (SR) given in [21], is exploited to compare the performance of the existing collision-optimized attacks and our CoTree. The fault tolerance coefficient ϵ in Equation 13 is set to 3. We only perform 100 repetitions for each experiment since the evaluation is very time-consuming.

6.1 Experiments on DPA Contest $v4.1$ Dataset

The widely used DPA Contest $v4.1$ open dataset [1] was targeted on the AES-256 algorithm protected by Rotated S-boxes Masking (RSM) [14], and the algorithm was implemented on an Atmel ATMega-163 smart card connected to the side-channel attack standard evaluation board named SASEBO-W. Due to the defect of the exploited random numbers, the implementation exists first-order leakage, i.e., can be directly attacked by first-order CPA and CECA (see [12] for detail). Here we exploit CPA with Hamming weight model to exploit the leakage and further exploit traditional CECA to optimize CPA. We extract the best POI for each S-box and randomly extract 1100 power traces to perform the remaining experiments, and the time consumption of FCC, LCD and our CoTree is shown in Figure 5. Here the threshold τ_κ for CPA is set to 25, thus we optimize and explore a very huge candidate space with $\underline{25^{16} \approx 2^{74.3017}\text{ candidates}}$. FCC, LCD and our CoTree can transform very huge candidate space into a much smaller collision space quickly, and this even holds under very large τ_d considered in Figure 5.

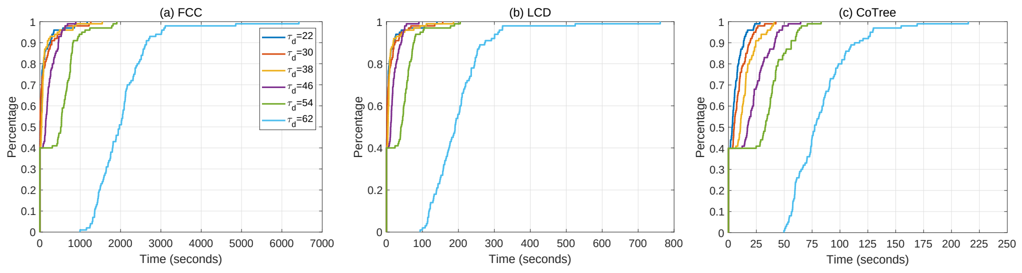

Figure 5: Time consumption under different thresholds τ_d.

FCC, LCD and CoTree here exploit the same fault tolerance strategy and consider the same fault tolerance coefficient ϵ in our experiments (see Equation 13). Therefore, we will achieve the <u>same success rate</u> in this case, and the only difference among these three schemes is their strategies of repetitive collision detection alleviation, and Figure 5 vividly shows that our CoTree achieves significantly higher performance compared to FCC and LCD. Compared to FCC, LCD and CoTree, TOC and FTC can only work under very small thresholds τ_d and τ_κ since they only exploit 15 pairs of collisions and too many collision chains will satisfy this condition. Our experiments given in Figure 6 also verify that a small increase in the threshold τ_d (τ_κ is set to 25 here) in TOC and FTC will make a much larger number of collision chains satisfying the collision condition and greatly increase the burden of collision detection.

We also exploit key rank estimation tool to enhance our comparison and the original ranks of the key in CPA is given in Figure 6(c). The remaining chains are ranked and successively verified according to their number of collisions in TOC, FTC, FCC, LCD and CoTree as exploited in the previous work [16]. It's obvious that TOC achieves a much better key rank and this collision optimization consumes very limited time. However, FTC is not suitable for the case that both thresholds τ_κ and τ_d are very large. It runs slowly, and there will often be no chain meeting the collision conditions when considering more sub-keys, the same conclusion will be drawn in Section 6.2. Therefore, FTC is unpractical in our scenarios. Although we consider much larger thresholds for FCC, LCD and CoTree, they can even spend less time on space transformation and leave a smaller collision space compared to TOC and FTC (see Figure 7).

Above all, compared to TOC and FTC, the schemes FCC, LCD and CoTree exploit all collisions between any two sub-keys, although some of them are fault tolerated, there are still

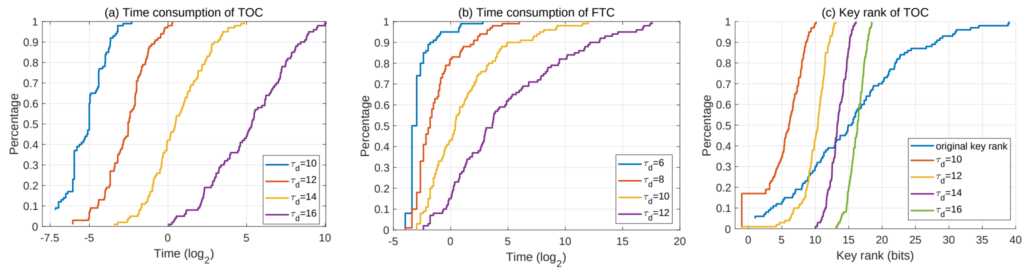

Figure 6: Time consumption of TOC and FTC, and key rank of TOC under different thresholds τ_d.

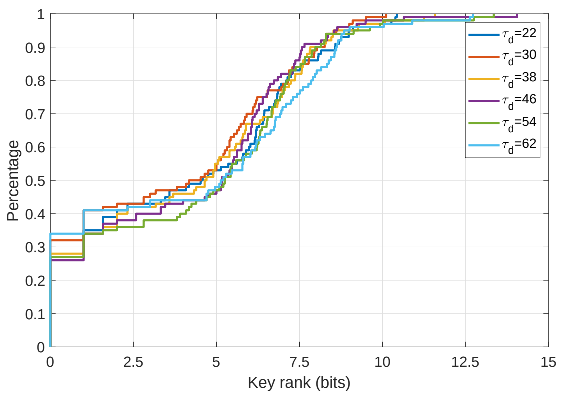

Figure 7: Key rank of CoTree under different thresholds τ_d.

many fewer chains meeting the conditions. Therefore, they achieves a much smaller collision space and make the latter key verification easier. For side-channel attacks, the widely used success rate will be a very important evaluation metric. The success rates of TOC, FTC, FCC, LCD and our CoTree are shown in Table 3, and it vividly shows that the success rate is zero in TOC and FTC. Very different from them, the schemes FCC, LCD and our CoTree achieve success rates even greater than 0.40. It's noteworthy that FCC, LCD and CoTree do not bring us a higher success rate in Table 3 if we enlarge the threshold τ_d, since the fault tolerance coefficient given in Equation 13 is flexible and the collision chains satisfying the conditions are also dynamic.

Our CoTree exploits a data structure very different from FCC and LCD, and it is a tree rather than simply an array exploited in both of them. We can conclude from the success rate, time consumption and the key rank that our CoTree's success rate is much higher than TOC and FTC's, and it also consumes much less time and achieves much better key rank. In other words, It achieves performance significantly better than the previous works FCC and LCD.

Table 3: Success rates under different thresholds τ_d.

τ_d $(\tau_\kappa = 25)$	22	30	38	46	54	62
FCC, LCD, CoTree	0.43	0.49	0.44	0.41	0.41	0.47
τ_d $(\tau_\kappa = 25)$	10	12	14	16	–	–
TOC	0.00	0.00	0.00	0.00	–	–
τ_d $(\tau_\kappa = 25)$	6	8	10	12	–	–
FTC	0.00	0.00	0.00	0.00	–	–

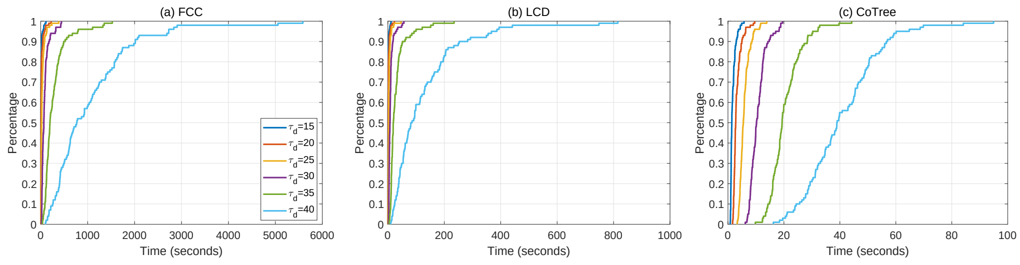

Figure 8: Time consumption under different thresholds τ_d.

6.2 Experiments on an AT89S52 Micro-controller

Our second experiment is performed on an AES-128 algorithm implemented on an AT89S52 micro-controller specially designed for SCA. The operating frequency of this chip is 12 MHz. The sampling rate of our Picoscope 3000 is set to 125 MS/s. We acquire 50, 000 power traces and perform CPA to extract the best POI for each S-box. We exploit collision information to optimize TA, but CECA's performance is far inferior to TA's, and it's a not good choice to exploit CECA's collision information to optimize TA in this case. Therefore, we extend CECA to the profiled setting. Specifically, we use the second order of Minkowski Distance (i.e., Euclidean distance) in CECA, and define the distance function $\mathcal{D}$ as:

$$\mathcal{D}_{\kappa_1,\kappa_2} = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{2}\left(t_j^i - \mathbf{m}_j^{\mathsf{Sbox}\left(x_j^i \oplus \kappa_j\right)}\right)} \tag{14}$$

as exploited in [15, 16]. Here x_j^i denotes the j^{th} byte of the i^{th} encrypted plaintext, t_j^i denotes the corresponding POI and $\mathbf{m}_j^{\mathsf{Sbox}\left(x_j^i \oplus \kappa_j\right)}$ denotes the profiled mean power consumption corresponding to intermediate value output by S-box receiving x_j^i and κ_j as its inputs (see [15, 16] for details). This profiled CECA will achieve performance close to TA, and their combination will be more meaningful in this case as explained before.

The threshold setting in collision-optimized attacks is not invariable. There are many factors affecting this (e.g., different target devices and power analysis methods), and different scenarios will consume very different time when we transform candidates within the same given thresholds into the collision space. If we set the same thresholds for FCC, LCD and CoTree as considered in Section 6.1, they will spend much time and even not work. Similar to the very huge candidate space with $\underline{25^{16} \approx 2^{74.3017}\text{ candidates}}$ we optimized and explored in Section 6.1, τ_κ here is also set to 25. We then randomly extract 260 power traces, consider τ_d from 15 to 40, and perform the remaining experiments in each repetition. The time consumption of FCC, LCD and our CoTree is shown in Figure 8, which vividly illustrates that LCD significantly alleviates the repetitive collision detection, and our CoTree achieves much higher performance than LCD due to its efficient collision chain storage structure and detection algorithm.

We also compare FCC, LCD and our CoTree with TOC and FTC in our experiments, and the results are shown in Figure 9. Time consumption of TOC grows quickly, since it exploits only 15 pairs of collisions and too many chains satisfying this collision condition as we have explained before. However, similar to the results given in Section 6.1, FTC is still not feasible. It is too time-consuming under $\tau_\kappa = 25$ and $\tau_d = 2$. Therefore, we adjust these two thresholds to $\tau_\kappa = 7$ and $\tau_d = 1, 2, 3$. FTC generates a very large number of chains satisfying the collision condition, but this situation does not last until the last sub-key since almost no full-key candidates remain in most cases (e.g., 90% of repetitions). Therefore, we do not give its results in figures.

The original key ranks in TA are given in Figure 9(c), in company with the key ranks in TOC. The key ranks in FCC, LCD and our CoTree are given in Figure 10. Key ranks in FCC, LCD and CoTree under $\tau_d = 40$ are even better than these in TOC under $\tau_d = 12$. The success rates of TOC, FTC, FCC, LCD and our CoTree under different thresholds τ_d are shown in Table 4. TOC achieves a low success rate under the considered small thresholds and no success is achieved in FTC. Different from the results given in Section 6.1, FCC, LCD and our CoTree achieve higher success rates with the growth of τ_d.

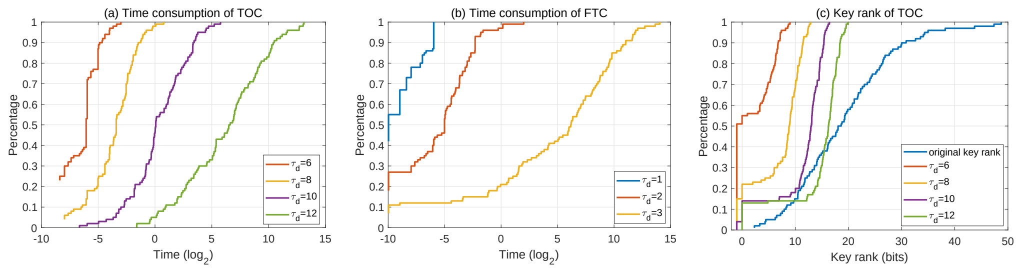

Figure 9: Time consumption of TOC and FTC, and key rank of TOC under different thresholds τ_d.

Table 4: Success rates under different thresholds τ_d.

τ_d ($\tau_\kappa = 25$)	15	20	25	30	35	40
FCC, LCD, CoTree	0.26	0.35	0.39	0.44	0.51	0.55
τ_d ($\tau_\kappa = 25$)	6	8	10	12	–	–
TOC	0.07	0.12	0.19	0.25	–	–
τ_d ($\tau_\kappa = 7$)	1	2	3	–	–	–
FTC	0.00	0.00	0.00	–	–	–

Above all, we can draw a conclusion that our CoTree achieves a much higher success rate than TOC and FTC, and its time consumption is much less than the existing schemes. This fully illustrates that our CoTree significantly outperforms these works.

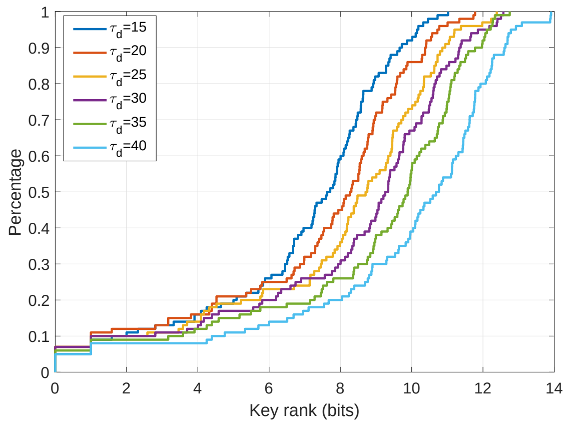

Figure 10: Key rank of CoTree under different thresholds τ_d.

7 Conclusions

To alleviate the repetitive collision detection of the existing collision-optimized attacks and further push their limits of conquerable candidate space, we proposed a highly-efficient detection algorithm named CoTree in this paper. CoTree exploits a top-down tree building procedure to guarantee that each node on the tree only needs to be traversed once. CoTree then launches a bottom-up branch removal procedure to remove the nodes unsatisfying the collision conditions after traversing all candidates of the sub-key currently under consideration. These two strategies significantly alleviate the repetitive collision detection. Our experiments verified that our CoTree significantly outperforms the existing works.

Pushing the limits of the conquerable candidate space is very important for side-channel attacks and evaluations, but we still have to face many difficulties. First, it's very meaningful to efficiently set an optimal threshold for collision attacks in theory. Furthermore, the complexity of CoTree is difficult to evaluate theoretically, just like the existing collision-optimized attacks. Here we leave them as open problems. Finally, although our CoTree achieves performance much better than the existing works, some collision information is still repetitively detected, and we will further alleviate this in our future works.

References

[1] Dpa contest v4.1. http://www.dpacontest.org/home/.

[2] Andrey Bogdanov. Multiple-Differential Side-Channel Collision Attacks on AES. In *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, pages 30–44, 2008.

[3] Andrey Bogdanov and Ilya Kizhvatov. Beyond the Limits of DPA: Combined Side-Channel Collision Attacks. *IEEE Trans. Computers*, 61(8):1153–1164, 2012.

[4] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, pages 16–29, 2004.

[5] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template Attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, pages 13–28, 2002.

[6] Yaoling Ding, Liehuang Zhu, An Wang, Yuan Li, Yongjuan Wang, Siuming Yiu, and Keke Gai. A Multiple Sieve Approach Based on Artificial Intelligent Techniques and Correlation Power Analysis. *ACM Trans. Multim. Comput. Commun. Appl.*, 17(2):71, 2021.

[7] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Side-Channel Attacks on BLISS Lattice-Based Signatures: Exploiting Branch Tracing against strongSwan and Electromagnetic Emanations in Microcontrollers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1857–1874, 2017.

[8] Vincent Grosso. Scalable Key Rank Estimation (and Key Enumeration) Algorithm for Large Keys. In *Smart Card Research and Advanced Applications, 17th International Conference, CARDIS 2018, Montpellier, France, November 12-14, 2018, Revised Selected Papers.*, pages 80–94, 2018.

[9] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, pages 104–113, 1996.

[10] Yang Li, Shuang Wang, Zhibin Wang, and Jian Wang. A Strict Key Enumeration Algorithm for Dependent Score Lists of Side-Channel Attacks. In *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*, pages 51–69, 2017.

[11] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 69–90, 2017.

[12] Amir Moradi, Sylvain Guilley, and Annelie Heuser. Detecting Hidden Leakages. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *Applied Cryptography and Network Security - 12th International Conference, ACNS 2014, Lausanne, Switzerland, June 10-13, 2014. Proceedings*, volume 8479 of *Lecture Notes in Computer Science*, pages 324–342. Springer, 2014.

[13] Amir Moradi, Oliver Mischke, and Thomas Eisenbarth. Correlation-Enhanced Power Analysis Collision Attack. In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, pages 125–139, 2010.

[14] Maxime Nassar, Youssef Souissi, Sylvain Guilley, and Jean-Luc Danger. RSM: A Small and Fast Countermeasure for AES, Secure against 1^{st} and 2^{nd}-Order Zero-Offset SCAs. In Wolfgang Rosenstiel and Lothar Thiele, editors, *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012*, pages 1173–1178. IEEE, 2012.

[15] Changhai Ou, Siew-Kei Lam, and Guiyuan Jiang. The Science of Guessing in Collision-Optimized Divide-and-Conquer Attacks. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 40(6):1039–1051, 2021.

[16] Changhai Ou, Siew-Kei Lam, Chengju Zhou, Guiyuan Jiang, and Fan Zhang. A Lightweight Detection Algorithm For Collision-Optimized Divide-and-Conquer Attacks. *IEEE Trans. Computers*, 69(11):1694–1706, 2020.

[17] Changhai Ou, Zhu Wang, Degang Sun, and Xinping Zhou. Group Collision Attack. *IEEE Trans. Information Forensics and Security*, 14(4):939–953, 2019.

[18] Changhai Ou, Chengju Zhou, Siew-Kei Lam, and Guiyuan Jiang. Multiple-Differential Mechanism for Collision-Optimized Divide-and-Conquer Attacks. *IEEE Trans. Inf. Forensics Secur.*, 16:418–430, 2021.

[19] Romain Poussier, François-Xavier Standaert, and Vincent Grosso. Simple Key Enumeration (and Rank Estimation) Using Histograms: An Integrated Approach. In *Cryptographic Hardware and*

Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings, pages 61–81, 2016.

[20] Christian Rechberger and Elisabeth Oswald. Practical Template Attacks. In *Information Security Applications, 5th International Workshop, WISA 2004, Jeju Island, Korea, August 23-25, 2004, Revised Selected Papers*, pages 440–456, 2004.

[21] François-Xavier Standaert, Tal Malkin, and Moti Yung. A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. In *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, pages 443–461, 2009.

[22] Danhui Wang, An Wang, and Xuexin Zheng. Fault-Tolerant Linear Collision Attack: A Combination with Correlation Power Analysis. In *Information Security Practice and Experience - 10th International Conference, ISPEC 2014, Fuzhou, China, May 5-8, 2014. Proceedings*, pages 232–246, 2014.

[23] Weijia Wang, Yu Yu, François-Xavier Standaert, Junrong Liu, Zheng Guo, and Dawu Gu. Ridge-Based DPA: Improvement of Differential Power Analysis For Nanoscale Chips. *IEEE Trans. Information Forensics and Security*, 13(5):1301–1316, 2018.

[24] Andreas Wiemers and Dominik Klein. Entropy Reduction for the Correlation-Enhanced Power Analysis Collision Attack. In *Advances in Information and Computer Security - 13th International Workshop on Security, IWSEC 2018, Sendai, Japan, September 3-5, 2018, Proceedings*, pages 51–67, 2018.